Dandelion: A Novel, High-Level Programming System for Software Defined Coalitions with Local State Sharing

Xin Wang†, Qiao Xiang†, Jeremy Tucker‡, Vinod Mishra*, Y. Richard Yang†,
†Yale University, ‡U.K. Defence Science and Technology Laboratory, *U.S. Army Research I
{xin.wang, qiao.xiang, yry}@cs.yale.edu, jtucker@mail.dstl.gov.uk, vinod.k.mishra.civ@mail

2019 年 9 月 16 日

摘要

² Ì ô²»º Integrating software defined networking into military coalition to realize a software defined coalition (SDC) infrastructure is non-trivial due to the stringent requirements (e.g., real-time, efficiency and reliability) of SDC applications in highly dynamic tactical network environments. Recent designed SDC datapaths improve the performance of SDC applications by offloading complex, stateful operations from the SDC control plane to data plane devices. However, they have two limitations. First, results of offloaded operations (referred to as local state) are not shared between data plane devices, resulting in substantial performance issues in SDC networks. Second, configuring low-level SDC datapaths is time-consuming and error-prone. To address these two issues, in this paper, we design Dandelion, a novel, high-level SDC programming system that allows users to specify the behavior of SDC data plane devices and the datapath configurations can be automatically generated with the local state sharing to efficiently utilize the network resources. We implement a prototype of Dandelion and demonstrate its efficiency and efficacy using experiments. Results show that with Dandelion, the total throughput is two times higher than that of without Dandelion.

1 Introduction

The recent success of software defined networking (SDN) systems [1–3] motivates the efforts to integrating SDN into military coalitions to realize an efficient, agile, and optimal software-defined coalition (SDC) infras-

tructure [4]. In SDC, autonomous coalition members operate under highly dynamic tactical network environments with resource constraints, such as limited power and processing capability, and dynamic connectivity.

One may think that the integration of SDN into SDC is straightforward, because SDN allows coalition members to realize efficient, flexible control over the coalition networks through flexible packet match-action processing on the data plane and logically centralized in the control plane [5,6]. Despite these promising features, they are insufficient for supporting SDC.

The fundamental reason behind this insufficiency is that SDC applications (e.g., flexible load-balancing and detection of DNS amplification attack) have stringent performance requirements (e.g., real-time, efficiency and reliability) under highly dynamic tactical network environments. Implementing SDC applications using the above SDN architecture in such environment would incur high communication overhead between the data and control planes of SDC applications, resulting in significant delay, efficiency and reliability issues.

Some SDC datapaths are recently proposed, which offload complex, stateful operations on packets (e.g., counting, flow security inspection and congestion control) from the control plane to data plane devices to improve the performance of SDC applications [7, 8]. Different datapath primitives, such as state counter, packet buffer and packet in-network processing block, are designed in these systems to support various SDC applications, e.g., stateful firewall, proactive routing protection, and resilient routing.

However, these systems suffer from two limitations. First, results of offloaded stateful operations at data plane devices, which we refer to as local state in the remaining of the paper, are not shared between data plane devices, resulting in substantial resource under-utilization in SDC networks. For example, a firewall middlebox is usually a bottleneck in an SDC network due to its limited processing speed. As such, once a data flow is identified as secure, non-sensitive by a firewall, all future packets of this flow should be forwarded along a path with higher reliability and throughput, without the need of passing the firewall again. However, the security state of this flow is only stored in the firewall and not shared with other devices. Without knowing that this flow is non-malicious, an upstream device (e.g., a gateway router) in the SDC network still has to forward all the packets of this flow to the firewall. Second, although some distributed update primitives provide interfaces for sharing of local state between data plane devices [9], configuring such low-level SDC datapaths still requires time-consuming and error-prone manual efforts.

Toward addressing these two limitations, in this paper, we design Dandelion, a novel, high-level SDC programming system. Specifically, Dandelion introduces a series of novel programming primitives for users to model and specify the behavior of SDC data plane devices. In addition, we design

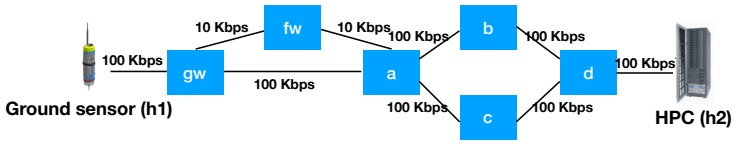
a novel data structure called decision graph and an efficient configuration framework to translate high-level SDC programs into efficient SDC datapath configurations. The translated configurations allow data plane devices in SDC to exchange local states with the goal of efficiently utilizing the resources in SDC networks (e.g., the packet processing capability of devices and the data transmitting capability between devices). We implement a prototype of Dandelion and demonstrate its efficiency and efficacy using experiments. Results show that with Dandelion, the total throughput can increase around two times higher than that of without Dandelion.

The rest of this paper is organized as follows. Section 2 discusses related work and gives an motivating example. Section 3 gives an overview of Dandelion and Section 4 presents the details of how Dandelion translates a high-level SDC program into data plane configurations. We evaluate the performance of Dandelion in Section 5 before concluding the paper in Section 6.

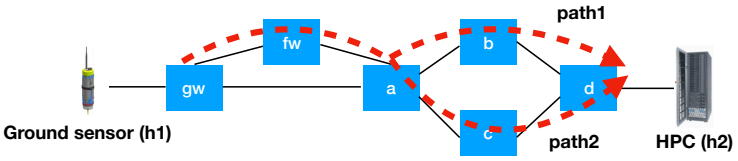
2 Related Work and Motivation

Related work: Some SDN/SDC datapaths are recently proposed to offload complex, stateful operations on packets (e.g., counting, flow security inspection and congestion control) from the control plane (e.g., a base station) to data plane devices (e.g., mobile devices) to improve the performance of SDN applications [7, 8, 10–18] under dynamic tactical environments. In these designs, results of offloaded operations are stored by each data plane device independently. We refer to them as local states. SOL [11] and Merlin [12] tackle the placement and configuration of data plane devices by solving constrained path computation problems. P4CEP [18] and OpenSDC [8] focus on expanding the capability of data plane devices from packet processing to event processing. SNAP [10] designs a high-level programming system that translates a high-level program to the configuration of stateful operations in data plane devices. Despite these substantial efforts on stateful SDC datapath, one major, common limitation of these systems is that the local states of each data plane device are not shared with others. As we will show shortly in the motivating example, this would lead to substantial resource under-utilization in SDC networks, impairing the performance of SDC applications.

To allow local state sharing between data plane devices, DDP [9] designs some primitives for distributed datapath update. Hula [14] and MP-HULA [13] also design probing mechanisms for data plane devices running load balancing applications to update their local states. However, manually configuring such low-level primitives on an application-by-application basis is time-consuming and error-prone.



(a) The network topology.



(b) The data plane configuration without local state sharing.

图 1: Motivating example: a data collection SDC application in a network with firewall.

Motivation: We give an example to demonstrate the limitation of existing systems, and the benefit of local state sharing. In particular, we consider a tactical network in Fig. 1(a), which consists of a ground sensor (h_1), a middebox firewall (fw), a computing server (h_2), a gateway switch gw, and other forwarding switches (a-d). The bandwidth of links $gw \rightarrow fw$, and $fw \rightarrow a$ is 10 Kbps, while the bandwidth of all other links is 100 Kbps. The data collected by the sensor h_1 should first be sent to the firewall fw, which identifies whether the data is sensitive or not based on the 5-tuple of the packet carrying the data (i.e., srcAddr, dstAddr, srcPort, dstPort and protocol). An SDC data collection application is running to send data collected from the sensor to the server. The network operator wants to enforce the following policy: For any data sent from h_1 to h_2, if it is sensitive, the data should be forwarded along a route passing switch b, otherwise passing switch c.

To enforce such a policy, state-of-the-art stateful datapath systems (e.g., SNAP [10]), which do not support local state sharing between data plane devices, would compute the configuration as shown in Fig. 1(b). Specifically, the gateway switch gw forwards all packets to the firewall fw. fw identifies if it is sensitive or not, and appends a tag on each packet to indicate the identification result before sending to switch a. Switch a then matches the tag of each arrival packet and forwards them to b (path1) or c (path2) based on the matching result.

Though this configuration is correct, it does not fully utilize the resources in SDC network, impairing the performance of the data collection application. Once the sensitivity of a data flow is identified by the firewall fw, no future packet of this flow needs to pass fw again. Instead, they can be forwarded along path gw, a, c, d, or gw, a, b, d with a higher transmission bandwidth. However, such a new forwarding configuration cannot be real-

ized without local state sharing between the firewall fw, the gateway gw and the switch a.

The above example demonstrates the benefits of local state sharing between data plane devices in SDC. However, manually setting up the low-level configuration for local state sharing is too time-consuming and error-prone. As such, we design Dandelion, a novel SDC programming system to automatically translate high-level SDC programs into datapath configurations with local state sharing.

3 Dandelion Overview

In this section, we first give the programming model of Dandelion, followed by its architecture, and the workflow to transform a high-level SDC program to datapath configurations with local state sharing.

3.1 Programming Model

Dandelion adopts an omnipotent programming model proposed in high-level SDN programming systems (e.g., Maple [19]), which logically programs every single packet with an onPacket function. In addition to the primitives that read and test on packet headers, Dandelion designs the following primitives for users to model and specify middleboxes in SDC network. The first is to specify a middlebox in the network (m = middlebox(name, property)) where the property indicates the middlebox is stateless or not; The second is to invoke the packet handling of a middlebox (m.handle(pkt)). Specifically, we consider a middlebox as a packet handling function that can return the result for the incoming packet. For the stateless middlebox, if two packets have the same 5-tuple match fields, they always have the same results. For the stateful middlebox, this cannot be guaranteed as the packet processing depends on the state in the middlebox.

In Dandelion, the interaction between switch and middlebox is bi-directional. A switch can send a packet to a middlebox for the processing through a tunnel. After the processing, the middlebox can send the packet back and share its local state of the packet with switches. In Dandelion, such interaction is transparent to the user so that the user does not need to manually configure the local state sharing between data plane devices. Instead, as we will show in the next section, Dandelion automatically translates high-level SDC programs into data plane configurations with local state sharing.

In addition to the middlebox primitives, Dandelion also adopts the route algebra primitive [20] for the user to specify path constraints for packets forwarding in SDC network. The abstract syntax of the Dandelion programming model is shown in Fig 2. Specifically, for the route algebra

expressions, we adopt the same in [20]. For the middlebox, programmers can specify its property, i.e., stateless or stateful.

p	::=	onPacket(pkt) $\{I\}$, d_1, ..., d_n	(program)
d	::=	$x^r = r$	(route algebra decl)
		$\mid x^m = middlebox(n, s)$	(middlebox decl)
I	::=	$x = e$	
		$\mid I; I$	(sequencing)
		$\mid x = x^m.handle(pkt)$	(middlebox operation)
		$\mid$ if $e^b : I$ then : I	(conditional)
		$\mid$ return $x^r \mid$ return r	(func return)
e	::=	$c \mid x$	(consts, vars)
		$\mid x^m$	(middlebox vars)
		$\mid pkt.a$	(packet fields)
e^r	::=	$e == e \mid e \leq e \ldots$	(relational)
e^b	::=	$e^r \mid e^r$ & $e^r \mid e^r \ldots$	(boolean)
$(r$	$\in$	route algebra expressions)	
$(c$	$\in$	strings)	(consts)
$(a$	$\in$	$\{$macSrc, ipDst ... $\}$)	(packet fields)
$(n$	$\in$	strings)	(middlebox names)
$(s$	$\in$	$\{$stateless, stateful$\}$)	(properties)
$(x$	$\in$	$\{x_1, x_2, \ldots, y_1, \ldots\}$)	(variables)

图 2: Dandelion abstract syntax.

Example Dandelion program: Revisit the motivating example in Section 2, the network operator can specify the policy to forward packets of sensitive and non-sensitive data along different paths using the Dandelion program in Fig. 3.

```
L1: mFW = middlebox("firewall", "stateless")
L2: PATH1 = h1 -> b -> h2 //b: waypoint
L3: PATH2 = h1 -> c -> h2 //c: waypoint
L4: //any: picking any path
L5: def onPacket(pkt):
L6:   if pkt.srcAddr == h1 & pkt.dsrAddr == h2:
L7:     if mFW.handle(pkt) == SENSITIVE:
L8:       return any(SPC.stable + PATH1)
L9:     else:
L10:      return any(SPC.stable + PATH2)
L11:  else: return DROP
```

图 3: The Dandelion program for the policy in the motivating example. For line 8 (9), the correct versions are in their comments.

Specifically, line 2 (3) specifies a path constraint that starts from h_1 to h_2 and must pass through b (c) (Note that it does not represent a concrete path but a path constraint). And the return statements use any function (defined in route algebra [20]) that picks any path satisfying the constraint. Although the programming model is quite simple, as involving middleboxes which physically map to nodes in the network, the system needs to guarantee the correctness for the program.

Correctness: We denote the correctness as the following: For any packet pkt, the real forwarding path for pkt in the network must comply with the returned path for pkt in the program. For example, if the programmer uses opt function to pick the shortest hop count path with the constraint (i.e., opt(PATH1)), then the returned path is gw, a, b, d. However, as the (first)

packet must access the firewall, the real path must include fw which does not comply with the returned path in the program.

One simple solution to guarantee the correctness is to let the programmer explicitly include the necessary middlebox nodes in the path constraint. In this case, it should be PATH1 = h1 -> fw -> b -> h2. However, this can add extra burden to programmers that they should make sure the path constraints comply with the traces of packets in the network.

System Path Constraint: To resolve the issue, we introduce the System Path Constraint (SPC) which is a global variable storing path constraints that must be complied with when computing a path at any location in the program. The intuition is that when a packet going through the program, some statements can add path constraints for the packet. For example, in Fig. 3, line 6 adds a constraint that the path should start from h_1 to h_2; Line 7 adds a constraint that the path must include the firewall. Then, after line 7, the SPC variable includes constraints that the path starts from h_1 to h_2 and includes fw. If the line 8 uses the opt function, then it should be return opt(SPC + PATH1) where the "+" means connecting the waypoints constraints in SPC and PATH1, and its result is gw, fw, a, b, d. (Note that the definition of SPC.stable will be given in the next section.)

3.2 Architecture and Workflow

Fig. 4 presents the architecture and workflow on how Dandelion translates high-level Dandelion SDC programs to low-level configurations of data plane devices with local state sharing. Given a high-level Dandelion SDC program, Dandelion first computes a decision graph that captures the forwarding decisions process of packets and stateful operations on packets. Second, Dandelion computes optimal paths for packet forwarding in the network with a system defined objective (e.g., maximizing total throughput). Note that the system computes paths when the returned paths are not concrete (e.g., using *any* function), and in the remaining part of the paper, we assume these paths are not concrete to introduce the path computation part. Third, the device-level data plane configurations are generated, which include local state sharing configurations, so that devices can exchange local state to forward the packets along the computed optimal paths. Dandelion leverages the stateful switch for local state sharing which includes two match-action tables: one is a state table (i.e., $match \rightarrow state$) and the other is a match table (i.e., $match + state \rightarrow action$). Given a switch, a packet first enters the state table to get its corresponding state (by matching its packet fields) and then enters the match table to get the action.

Example configuration with local state sharing: Fig. 5 gives the data plane configuration with local state sharing translated from the SDC program in

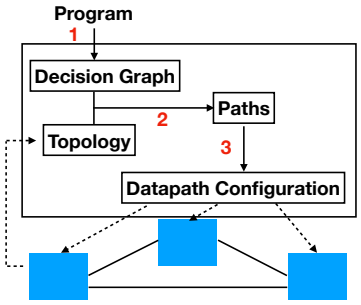


图 4: The architecture and workflow of Dandelion.

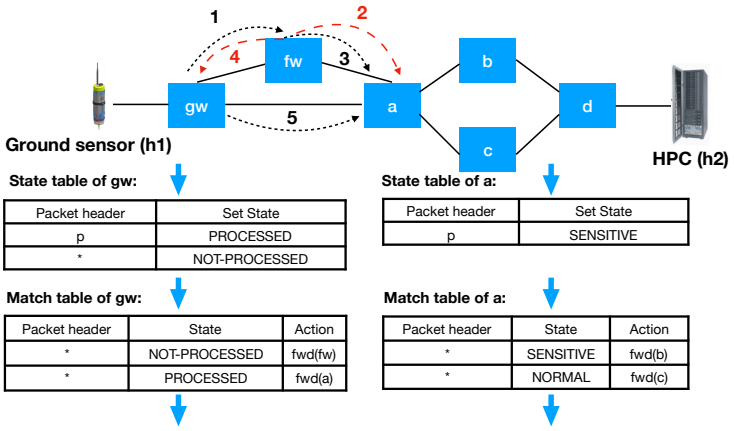


图 5: The configuration translated by Dandelion from the program in Fig. 3.

Fig. 3. Specifically, by allowing the local state sharing, the firewall fw can share its local identification state for a flow with the gateway switch gw. For example, it can send the state: $srcAddr = h1, dstAddr = h2, srcPort = 12345, dstPort = 22, protocol = tcp \rightarrow$ PROCESSED to gw, which can insert it into its state table.

With this configuration, the first packet p arrives at the gateway gw is forwarded to fw by getting a state NOT-PROCESSED at gw (Step 1). Assume fw identifies p as SENSITIVE. It first sends this local state $p \rightarrow SENSITIVE$ to a (Step 2). Next, fw forwards p to a (Step 3), where p is forwarded to b because its state is SENSITIVE. Then fw sends the local state $p \rightarrow PROCESSED$ to gw (Step 4). As such, all future packets with the same match fields as p entering gw will be forwarded directly to a (Step 5).

4 Design Details

In this section, we present the details of Dandelion. We first give the details of decision graph, followed by the path computation and the data path generation.

4.1 Decision Graph

The Decision Graph (DG) is used to capture the forwarding decisions process of packets and stateful operations on packets. We define a DG as a directed acyclic graph (with a single root) that has three types of nodes: packet-test nodes, middlebox-operation nodes, and action nodes (mapping to three basic primitives, packet-test, middlebox, and route algebra). The first two nodes are the internal nodes of DG while the last one is the leaf node. An out-edge of a node can specify a range of packets (if the node is a packet-test node) or a result from a middlebox (if the node is a middlebox-operation node). For example, the DG of the motivating program is shown in Fig. 6. Note that we do not focus on the computation of DG from a program which can be achieved by existing (compiler) work [10] [21].

Given a DG, we denote the sequence of nodes and edges from the root to an action node (i.e., path constraint pc) as the trace of pc ($T(pc)$). Then, given a trace $T(pc)$, we denote the middlebox-operation nodes of $T(pc)$ as $M(pc)$ (i.e., a sequence of nodes). As middlebox-operation nodes represent the packet handling of middleboxes, to not form loops, every middlebox-operation node can only appear once in $M(pc)$. Also, by extracting middlebox-operation nodes that represent stateful middleboxes from $M(pc)$, we have a subsequence of $M(pc)$ that every node is for stateful middlebox operations (denoted by $M^s(pc)$). Given a $T(pc)$, the SPC for

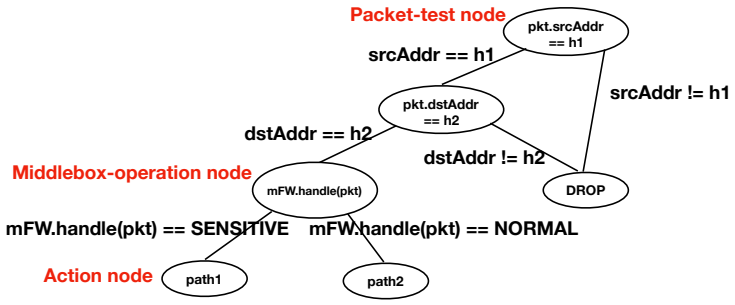


图 6: Example of DG.

its action node can be easily computed as the $T(pc)$ gives the trace of the packet enters the action node.

4.2 Path Computation

As the leaf nodes in DG are path constraints, path computation targets to compute the concrete paths for these leaf nodes with a system performance objective. (And if leaves are concrete paths, the path computation just skips them.)

Packet forwarding model: Before path computation, we first give a packet forwarding model with DG in the network as the following. As shown in Fig. 7, given a path (gw, a, b) in the motivating example, logically we consider every switch in the path has the DG of the example program where P indicates a packet-test node, M indicates a middlebox-operation node, and A indicates an action node. The A node with red color represents the corresponding constraints for the path. The first packet of the flow (arriving at gw) starts to traverse the DG as the red line in the figure. When the packet meets a middlebox-operation node, it will be sent to the corresponding middlebox through the tunnel. After the processing, the middlebox will send the packet back and share its local state (i.e., by installing/modifying rules in the state table) for the flow with the corresponding middlebox-operation nodes (each node can be viewed as a pair of state table and match table) at switches along the path. (If the middlebox is stateful, then it does not set any state but makes the packet carry the state.) After the traversal of the DG (i.e., arriving at the leaf node along the red line), the packet will get the path and should be forwarded along the path. As local states have been shared with corresponding middlebox-operation nodes (or carried by the packet for stateful middleboxes) at other switches, the packet does not need to be sent to the middlebox again at the next switch. We denote a forwarding of a packet as stable forwarding if the packet does not access any stateless middlebox (compared with the first packet that should access them). Based on the stable forwarding, we can extend original SPC variable to SPC.stable which means the system path constraints only for stable

Variable	Description
u_i, v_i	The source and destination nodes of path constraint pc_i
E	All edges (an edge e: $(e.src, e.dst)$) in the network
m_e	The maximum bandwidth of e
W_i	A set of node pairs of pc_i
z_e^i	The edge e is selected by pc_i
b_i	The bandwidth of flows using the path for pc_i

表 1: Notation.

Constraint	Description
$\forall i, \forall (x, y) \in W_i, Path(x, y, E)$	Path constraints
$\forall i, Path(u_i, v_i, E)$	Path exists in E
$\forall e \in E, \sum_i b_i * z_e^i \leq m_e$	Bandwidth constraints for edges

表 2: Constraints.

forwarding paths (i.e., remove the stateless middleboxes). Then, to allow the state sharing, SPC.stable should be used as SPC includes the whole constraints for all packets including the first packet that must pass through all the middleboxes.

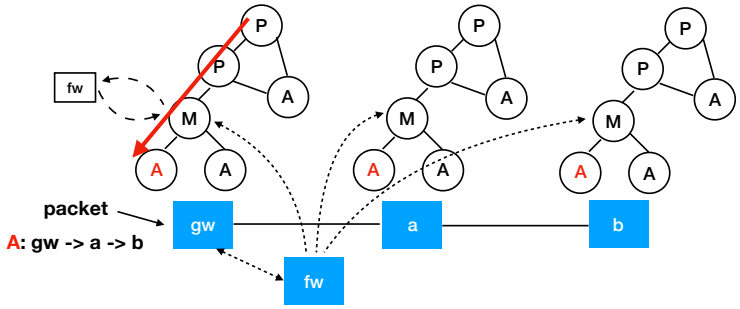


图 7: Packet forwarding model with DG.

System objective: Based on the model, as the forwarding of the first few packets has little influence on the performance of the flow, a simple path computation (that only considers the stable forwarding) would be the following. We consider the objective of the network is to maximize the total throughput, i.e., maximize $\sum_i b_i$ where i is the index of the path constraint pc_i and b_i is its throughput. The definition of variables can be found at Table 1. And the constraints can be found at Table 2 (where $Path(x, y, E)$ means there exists a simple path starting from x to y in E). In this paper we focus on the waypoints constraints for the path constraints (as they may complicate the path computation) and we consider the waypoints constraint as a set of node pairs where a pair has the form (x, y) which indicates packets must pass through x before y. (Note that x and y can be source/destination nodes.)

As the system path constraints have been added into the path constraints (by using the SPC.stable), the correctness can be guaranteed. And it enforces a packet must get results of stateful middleboxes and then select the correct path.

However, the current path constraints may cause excessive constraints for paths. For example, a stateful middlebox-operation node (M) has two path constraints A_1 and A_2 as its children in a DG. And, A_1 specifies a path must pass through switches s_1, s_2; A_2 specifies a path must pass through switches s_1, s_3. Then, the path constraints have (M, s_1) for both A_1 and A_2 as it simply connects the waypoints constraint in SPC.stable and s_1, s_2 (also s_1, s_3). However, this is only a sufficient condition as some other conforming paths are excluded (e.g., $s_1 \rightarrow M \rightarrow s_2$ (s_3)).

Waypoints constraints computation: Instead of simply connecting two waypoints constraints, now we give an algorithm to compute the waypoints constraints that enforces all the paths are correct and no conforming paths can be excluded. The high-level structure of the algorithm is to do a depth-first traversal for a DG. The traversal only considers the stateful middlebox-operation nodes and action nodes (and traverses a stateful middle operation node only after all its children are finished). When traversing a stateful middlebox-operation node, we can get all possible waypoints constraints under its decision. Then, the processing of the node is to update these constraints (each of which can be modeled as a directed acyclic graph, DAG). As discussed before, we do not want to add excessive constraints. And the solution is simple: When processing a middlebox node M, if all the no-incoming-edge nodes of M's possible constraints (i.e., DAGs) are the same, then skip these nodes recursively until these nodes are not the same and then add edges (M, x) to each of DAGs where x is the node of the location that the skipping process stopped at. The insight is that, skipping a node is safe if and only if the node is the next waypoint node for all the possible paths. For the example in Fig. 8, the node M has four possible path constraints (DAGs) and a should be skipped when updating them.

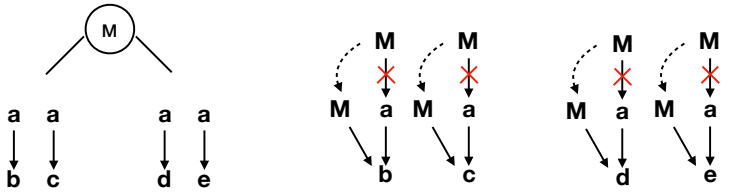


图 8: Skip the same nodes.

4.3 Datapath Generation

Remove redundant nodes of DG: After the path computation, there are concrete paths for all the leaf nodes in DG. We follow the packet forwarding model described previously that all the switches in the network have the same DG. It is easy to observe that if a switch does not belong to a path, then the corresponding leaf node of the path can be removed (and then the no-out-edge internal nodes). Also, by converting the path in a leaf node to

the next hop in the network, we may still remove redundant nodes. Still consider the firewall example. After we convert the paths to next hops, we find that the two leaf nodes of the middlebox node are the same, i.e., switch a in the network. This means the middlebox-operation node has no meaning for the gateway gw since whatever the result of the node is, the next hop does not change. Therefore, we can remove the node and replace with the leaf node with the next hop a. Fig. 9 illustrates the process.

Then, for the switch side, based on the updated DG, it can generate tables and flow rules easily with the multi-table pipeline structure. Note that a middlebox-operation node can be viewed as a state table following a match table. As for the middlebox side, if it is a stateless middlebox, then for the datapath it only needs to care about is switch-middlebox tunnels which also can be easily generated. For the stateful middleboxes, as they belong to computed paths, they can be viewed as switch nodes with stateful functions that can embed the results to the packets. When a packet meets a stateful middlebox node in DG of a switch, and the node cannot be replaced with a leaf node, then any next hop under the node is acceptable since it must eventually arrive at the middlebox and any path from the switch to the middlebox must follow the path constraints.

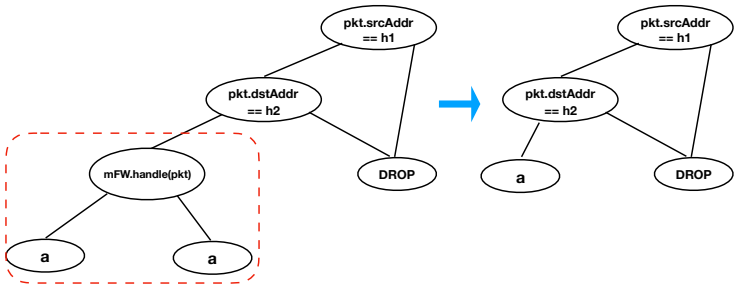


图 9: Remove redundant middlebox nodes.

Order of messages: The next issue is how a stateless middlebox share local states (i.e., update states) in other switches. It needs to guarantee that for any targeting switch, the update message should arrive earlier than the packet. For example, in the firewall example, only the step 2 is finished, step 3 can be executed. Then a simple solution can be making the packet carry the update message. Different with the state carrying for stateful middleboxes, this message can install/modify rules in the state table of the corresponding middlebox-operation node.

5 Performance Evaluation

In this section, we will first demonstrate the benefits of Dandelion from two aspects: latency and total throughput, and then give the evaluation for

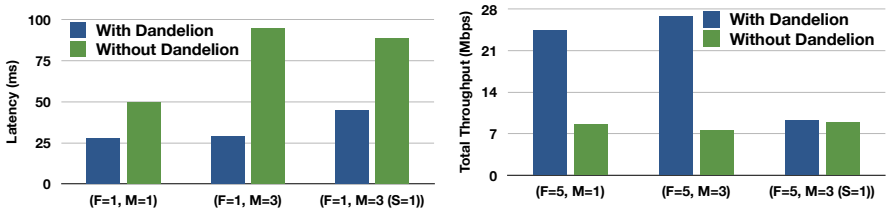
the waypoints constraints computation part. All evaluations are run on an 3.5 GHz Intel i7 processor with 16 GB of RAM running Mac OSX 10.13.
Methodology: First we generate a random topology with 25 nodes and 50 edges. For every edge, we set two random values as its latency (5 - 10 ms) and bandwidth (5 - 10 Mbps). To model a flow in the topology, we randomly choose two nodes from the topology as the source and destination nodes of the flow. And a flow can have a sequence of nodes (other than its source and destination nodes) in the topology as its required ordered middleboxes for packet processing. (We add a constraint to the selection for these middlebox nodes that the number of neighbors of a middlebox node must equal to two as typically a middlebox does not have route selection capability, i.e., for any packet, it only has one output interface.) As a comparison of Dandelion, the traditional approach does not distinguish whether a middlebox is stateful or stateless. Therefore, when computing a path for a flow in the tradition way (i.e., do not apply Dandelion), it requires the path must pass through all the middleboxes in a correct order. And when computing a path for a flow in the Dandelion approach, we random choose a subset of its required middlebox nodes as stateful middleboxes since for Dandelion approach, stateful and stateless middleboxes are handled in different ways.
Latency: To show the benefits for the latency aspect, we consider a single flow and differentiate its number of required middleboxes. And the target is to find the optimal path to minimize the latency for the flow. Then, we compare the results (i.e., the minimal latency) between applying Dandelion and not.
Total throughput: To show the benefits for the total throughput aspect, we consider multiple flows and all flows have the same required middleboxes. We also differentiate the number of middleboxes. And the target is to find optimal paths that have maximum total throughput. Then, we compare the results (i.e., the maximum total throughput) between applying Dandelion and not.
Execution time: To evaluate the performance of Dandelion, we compare the execution time of the path computation to maximize total throughput for both approaches (i.e., applying Dandelion and not).
Waypoints constraints computation: To show the benefits of the waypoints constraints computation, we set a sequence of nodes in the topology as a flow's waypoints constraint. Then, we consider the minimal latency as system's objective and compare the results between applying the waypoints constraints computation and not (i.e., leading to excessive constraints). In the excessive constraints, all the middlebox nodes should be passed through before flow's waypoints.
Results: The results in Fig. 10(a) demonstrate that by using Dandelion, the latency can be reduced. Specifically, F specifies the number of flows; M specifies the number of required middleboxes for flows; S specifies the

	(F=5, M=1)	(F=5, M=3)	(F=5, M=3 (S=1))
With Dandelion	1.3 (s)	1.4 (s)	4.2 (s)
Without Dandelion	4.5 (s)	10.8 (s)	11.5 (s)

表 3: The execution time of path computation to maximize total throughput for different scenarios.

number of stateful middleboxes for flows. As minimizing latency for a flow does not affect the results of other flows, the experiment only considers one flow scenario. From the results, we can see that without Dandelion, the latency of the flow grows up when the number of stateless middleboxes increases but if Dandelion is applied, the latency grows up only when the number of stateful middleboxes increases.

The results in Fig. 10(b) demonstrate that by using Dandelion, the total throughput can be increased. Specifically, when there are 5 flows and 3 middleboxes, the total throughput with Dandelion is around 3 times compared with that without Dandelion.



(a) The latency for different scenarios.

(b) The total throughput for different scenarios.

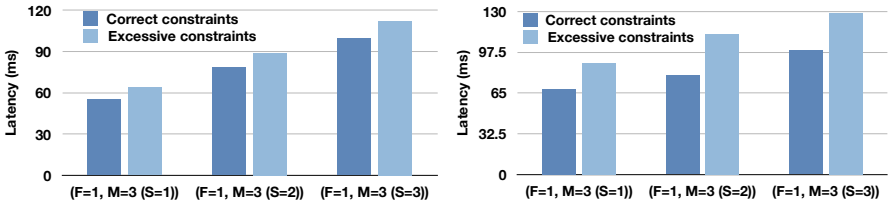
図 10: The benefits of Dandelion for latency and total throughput.

The results in Table 3 show the execution time of the path computation part to maximize total throughput. Since with Dandelion, the number of constraints is smaller than that without Dandelion, the execution time is also reduced (from 10.8 to 1.4 seconds when F=5 and M=3).

Fig. 11 shows the latency with correct constraints (i.e., by applying waypoints constraints computation) and with excessive constraints. Specifically, the results in Fig. 11(a) consider that the waypoints only have one node while Fig. 11(b) are for three nodes. From the results, we can see the excessive constraints increase the latency, i.e., lead to non-optimal path computation result. As the number of nodes increases in the waypoints constraint, the latency with excessive constraints becomes larger.

6 Conclusion

We design Dandelion, a novel, high-level SDC programming system providing novel high-level primitives for users to specify the behavior of SDC data plane devices, and automatically translates high-level SDC programs into efficient SDC datapath configurations with local state sharing.



(a) One node in waypoints constraint. (b) Three nodes in waypoints constraint.

图 11: The latency with different waypoints constraints.

Experiment results demonstrate its efficiency and efficacy.

参考文献

[1] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in SIGCOMM'13.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu et al., "B4: Experience with a globally-deployed software defined wan," in ACM SIGCOMM'13.

[3] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: steering oceans of content to the world," in SIGCOMM'17.

[4] V. Mishra, D. Verma, C. Williams, and K. Marcus, "Comparing software defined architectures for coalition operations," in ICMCIS'17.

[5] 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.'14.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in SIGCOMM'14.

[7] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," ACM SIGCOMM'14.

[8] Q. Xiang, F. Le, Y.-S. Lim, V. K. Mishra, C. Williams, Y. R. Yang, and H. Zhang, "Opensdc: A novel, generic datapath for software defined coalitions," in IEEE MILCOM'18.

[9] G. Li, Y. Qian, C. Zhao, Y. R. Yang, and T. Yang, "Ddp: Distributed network updates in sdn," in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2018, pp. 1468–1473.

[10] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in SIGCOMM'16.

[11] V. Heorhiadi, M. K. Reiter, and V. Sekar, "Simplifying software-defined network optimization using sol," in NSDI'16.

[12] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for provisioning network resources," in ACM CoNEXT'14.

[13] C. H. Benet, A. J. Kassler, T. Benson, and G. Pongracz, "Mp-hula: Multipath transport aware load balancing using programmable data planes," in NET-COMPUTE'18.

[14] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in Proceedings of the Symposium on SDN Research. ACM, 2016, p. 10.

[15] A. Gember, A. Akella, A. Anand, T. Benson, and R. Grandl, "Stratos: Virtual middleboxes as first-class entities," 2012.

[16] B. Anwer, T. Benson, N. Feamster, and D. Levin, "Programming slick network functions," in SOSR'15.

[17] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," ACM SIGPLAN Notices, vol. 47, no. 1, pp. 217–230, 2012.

[18] T. Kohler, R. Mayer, F. Dürr, M. Maaß, S. Bhowmik, and K. Rothermel, "P4cep: Towards in-network complex event processing," arXiv preprint arXiv:1806.04385, 2018.

[19] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," SIGCOMM'13.

[20] K. Gao, T. Nojima, and Y. R. Yang, "Trident: toward a unified sdn programming framework with automatic updates," in ACM SIGCOMM'18.

[21] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha, "A fast compiler for netkat," ACM SIGPLAN Notices, vol. 50, no. 9, pp. 328–341, 2015.