DDP: Distributed Network Updates in SDN

Geng Li*, Yichen Qian†, Chenxingyu Zhao‡, Y.Richard Yang*, Tong Yang‡
*Yale University, USA, †Tongji University, China, ‡Peking University, China

***Abstract*—To quickly and consistently update a network is among the most fundamental and common challenges in software defined networking (SDN) systems. Current approaches heavily rely on the (logically) centralized controller to initiate and orchestrate the network updates, resulting in long latency of update completion. In this paper, we present DDP, a system for fast, distributed network updates while preserving various consistency properties. The key technique in DDP is a novel primitive named datapath operation container (DOC), where each DOC is encoded with an individual operation and its dependency logic. DDP adopts the simple, but powerful DOCs to configure the network, so that network updates can be triggered and executed at the data plane in a distributed and local manner. Novel algorithms are designed to compute and optimize the DOCs for consistent updates. We implement DDP to evaluate its performance in various update scenarios. Experimental results show that DDP significantly improves network update speed by up to 52.1% for the real-time updates initiated by the controller, and further improves the speed by 55.6-61.4% for the updates directly triggered at the data plane, such as failure recovery.**

I. Introduction

Software-defined networking (SDN) is considered as a major recent advance in networking [1], [2]. It significantly simplifies network management and provides real-time network programmability by decoupling the network control plane from the data plane. Network updates are among the most common data plane operations, either periodically or triggered by local events such as failures. However, quickly and consistently updating the distributed data plane poses a major and common challenge in SDN systems [3], [4]. Specifically, due to asynchronous communication channels, control messages are often received and executed by switches in an order different from the order sent by the controller. An inappropriate control order may violate the consistency properties on the datapath, resulting in network anomalies, such as blackholes, traffic loops and congestion [5]–[8].

The consistent network update problem has been widely studied in the literature [3]–[7]. However, all of the work is based on centralized initiation and orchestration to perform the updates. An update can be launched only by the control plane, where the controller decides an order in which operations must be applied. The coordination of the distributed data plane requires frequent communication between the controller and switches, which slows down the update completion time, and increases the controller's processing load. In addition, centralized updates rely on the control plane too heavily. When the controller becomes a bottleneck, the network may suffer from substantial performance and reliability degradation.

In this paper, we present Distributed Datapath (DDP), a system for fast, distributed network updates in SDN, while maintaining various consistency properties. DDP still benefits from centralized intelligence at the control plane, but develops distributed coordination abilities at the data plane. The key technique in DDP is a simple, but powerful primitive named datapath operation container (DOC), where each DOC is encoded with an individual operation and its dependency logic. For real-time updates initiated by the controller, the involved DOCs are sent to the data plane in one shot, and the switches can consistently execute them in a distributed manner. For updates directly triggered by local events, the controller pre-stores the DOCs at the data plane, and when corresponding events happen, the updates can be locally triggered and executed. We further design novel algorithms to compute and optimize the primitive DOCs for consistent updates.

We fully implement the DDP system to evaluate its performance in various update scenarios. The results demonstrate that compared to state-of-the-art centralized approaches (*e.g.*, Dionysus [5]), DDP improves real-time network update speed by 31.4-52.1%. Furthermore, we show that DDP can locally initiate updates triggered by link failures, and is up to 61.4% faster than centralized approaches to recovering routing.

II. Network Update and Motivation

We start by formalizing the network update problem and then give an example to show the limitations of centralized approaches.

A. Network Update Problem

Our focus is on flow-based traffic management applications [2], [5], where each flow is an aggregate of packets between ingress and egress switches. We let C denote a network configuration state, which is a collection of exact match rules determining each flow's datapath. A network update is defined as a transition of configuration state from C to C'. We denote the update process as $C' = update(C, O, e)$, where $O = \{o\}$ is a set of datapath operations that implement the update. Each operation o is a modification on the data plan state, *e.g.*, to insert/delete/modify a flow rule at a particular switch. e is a local event at the data plane that triggers the update, *e.g.*, a link/switch failure and link congestion. Note that e is just used for identifying the update origin. Sometimes the update is triggered by operators or applications, and then $e = \varnothing$.

A network update can involve multiple unsynchronized devices at the data plane, so achieving the consistency is challenging during the updates. The consistency usually implies three properties: 1) *blackhole-*, 2) *loop-* and 3) *congestion-freedom*, and the detailed definitions can be found in [5], [8]. To prevent a violation of the consistency properties in any

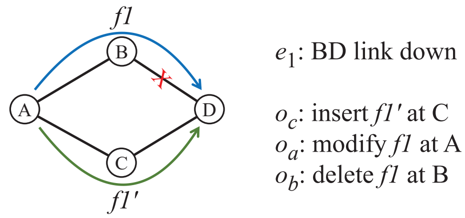


Fig. 1. A consistent network update example that incurs 3 rounds of controller-to-switch communication to orderly apply the operations.

intermediate states from C to C', the datapath operations with various dependencies are constrained in a correct processing order. Assume an update $C' = update(C, O, e)$ is given, our work is to quickly apply the operations O after e happens, while in a correct order.

B. Motivation Example

Consider an example shown in Fig. 1. There is a flow $f1$ in the network with path ABD. Assume that the link BD happens to be down, triggering an update from $f1$ to $f1'$ with 3 operations shown in the figure. So the update can be expressed as $C' = update(C, \{o_c, o_a, o_b\}, e_1)$. To ensure consistency, the 3 operations have to be processed orderly. o_c has to be applied before o_a; otherwise the flow $f1'$ will encounter a blackhole at C where no matched rule exists. Similarly, o_b has to be applied after o_a to avoid the blackhole at B. The ordered processing can be solely coordinated by the controller in centralized update approaches: the controller sends o_c to C, waits for its processing confirmation, and then sends o_a and o_b by the same token. As a result, the simple network update incurs at least 3 rounds of communication between the controller and switches, leading to substantial extra delays.

An insight we can extract from the example is that the 3 datapath operations will be applied at adjacent switches, and if the switches themselves can coordinate with each other to orderly apply the operations, the update time as well as the controller's processing load will be greatly reduced. Further more, if we can pre-store the operations at the data plane, and the link-down event can directly trigger them to apply, then the network update will be executed in a fully local manner. In this way, the update speed will be further improved.

III. DDP Design

DDP is proposed as a system to achieve fast, distributed, consistent updates in SDN. We first explore the dependencies among datapath operations and local events to ensure the consistency properties, and such dependency information is then encapsulated in a novel primitive DOC for each operation. DDP configures the network by the simple, but powerful primitive, so that network updates can be triggered and executed at the data plane in a distributed and local manner.

A. Operation Dependency Graph (ODG)

We first introduce the concept of an Operation Dependency Graph (ODG) that captures the data plane dependencies. An

(a) Type 1 connection **(b) Type 2 connection**

Fig. 2. Two types of connection in an ODG. (a) Type 1 connection: operation o_2 depends on the completion of operation o_1. (b) Type 2 connection: event e can trigger operation o.

ODG is a directed acyclic graph (DAG) where the nodes are the operations O and the trigger event e for an update $C' = update(C, O, e)$. The edge in an ODG reflects a timed order in a broad sense: upstream nodes have to happen before downstream nodes. There are two types of connection in an ODG. The first type as in Fig. 2(a) denotes that o_2 depends on the completion of another operation o_1, while the second type as in Fig. 2(b) denotes that event e can trigger o to handle this event. Note that there is no incoming edge connected to an event e, and Type 2 connection is dispensable since e can be $\varnothing$. An ODG well describes a network update, where Type 1 connection implies the correct order in which the operations are applied to ensure consistency, and Type 2 connection identifies the update trigger.

Properties of the ODG. The ODG hold several nice properties. First, the dependency is unidirectional, resulting in no cycles in the graph. Second, the dependency relations are transitive, *e.g.*, if an event e can trigger both o_1 and o_2, and o_2 depends on o_1, then e will be connected with only o_1 whose child node is o_2. Therefore, the ODG expresses an optimized result of the whole dependency relations. Third, connectivity is dispensable in the ODG. For the operations without dependencies, they will form into isolated pieces with no connections. Lastly, the ODGs are composable. Multiple ODGs for different update events can be composed together, so that the data plane can locally handle more events in DDP. The ODG composition algorithm will be introduced in Sec. IV-B

B. Datapath Operation Container (DOC) Specification

```
structure DOC
{
    operation o
    boolean expression gate
    operation set release
}
```

Fig. 3. Illustration of the primitive DOC.

To enable distributed and local network updates in DDP, we propose a novel primitive named DOC. The DOC is a structure as shown in Fig. 3, including 3 members as follows.

- ***o*** is an ordinary datapath operation.
- `gate` is the condition to apply o, represented by a Boolean expression of o's parent nodes in the ODG.
- `release` is a set of operations that depend on o, *i.e.*, the set of o's child nodes in the ODG.

Semantics. The semantics of DOC execution is simple. For each DOC d, 1) the inside operation $d.o$ is not applied until the gate logic d.`gate` is fully satisfied; 2) After $d.o$ is applied, all operations in d.`release` will be notified. The Boolean expression in `gate` consists of either a single operation or several operations joined by the Boolean operators AND (&) and OR (||). For example, if the execution of operation o_1 depends on the completion of both o_2 and o_3, then $d_1.\texttt{gate} = o_2 \& o_3$, and $d_2.\texttt{release} = d_3.\texttt{release} = o_1$. The content in `gate` and `release` of each DOC is computed by the algorithm in Sec. IV-A.

In the DDP system, the SDN controller adopts DOCs to configure the data plane, rather than directly sending operations as in traditional SDN. The switches then coordinate with each other to execute the update at right time.

C. Execution Behaviors

In DDP, we define two types of execution behaviors at the data plane for every DOC: Push and Pull.

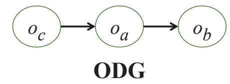
- **Push:** Upon a DOC is executed at the data plane (*i.e.*, the inside o is applied in the switch), it will send Push messages to the DOCs of all `release` operations. Hence the direction of Push is downward along with the ODG.
- **Pull:** Upon a DOC is received at the data plane, a Pull message will be sent to the DOC of every operation in its `gate`. So the Pull direction is upward. In addition, a DOC also has responsibility to Push back after receiving a Pull.

Correctness. The two behaviors guarantee the safety and liveness of distributed execution in DDP. The correctness intuition is that DOCs will be eventually executed as planned in the ODG regardless of arriving order. Suppose there are two operations with a dependency $o_1 \rightarrow o_2$, which means o_1 should be applied before o_2. The SDN controller sends out d_1 and d_2 at the same time. Case 1: d_2 arrives at the data plane first. According to the semantics, o_2 will not be applied until d_1 arrives at the data plane and sends d_2 a Push after execution. Case 2: d_1 arrives first. Then it is executed immediately and sends d_2 a Push which is useless because d_2 has yet to arrive. When d_2 arrives, it will Pull d_1 to Push back again, so that o_2 can be applied. In summary, Push and Pull are complementary to each other, and with the two behaviors, all operations will be consistently applied in a correct order. The detailed proof of the correctness can be found in Appendix A.

D. Examples

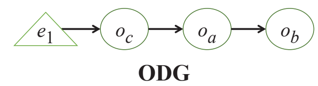
Now we use the example in Fig. 1 to see how the network update is executed in DDP.

Real-time update. Assume after detecting the link-down event e_1, the SDN controller decides to update $f1$ to $f1'$. In this case, $e = \varnothing$ because the update is initiated by the controller. Fig. 4(a) illustrates the ODG and all the DOCs involved in this real-time update. The 3 DOCs are sent to the data plane in one shot. First, d_c is executed upon arriving at C owing to the empty `gate` which is satisfied naturally. Then, a Push message is sent to d_a to apply o_a at A. Lastly, B can apply o_b after receiving the Push message from d_a. Compared with the centralized update which incurs 3 round-trip delays between the remote controller and the switches, DDP needs only 2



	operation	gate	release
d_c	o_c: insert $f1'$		o_a
d_a	o_a: modify $f1$	o_c	o_b
d_b	o_b: delete $f1$	o_a	

(a) Real-time update



	operation	gate	release
d_c	o_c: insert $f1'$	e_1	o_a
d_a	o_a: modify $f1$	o_c	o_b
d_b	o_b: delete $f1$	o_a	

(b) Local update

Fig. 4. Illustration of the ODGs and DOCs for the example in Fig. 1. (a) Real-time update which is initiated by the controller. (b) Local update which is directly triggered at the data plane.

one-way delays between adjacent switches (for coordination), therefore reduces the update completion time.

Local update. DDP can perform the update even better by pre-sending the DOCs d_a, d_b and d_c as shown in Fig. 4(b) to A, B and C respectively. Because of their non-empty `gate`s, the three operations will not take effect on the datapath at first. But when link BD is down, and C detects this event e_1 (we assume B will flood the link-down event), d_c.`gate` will become true and o_c is applied accordingly. After that, d_a and d_b will also be executed sequentially. As a result, with the powerful DOCs in DDP, the network update can be both triggered and executed in a fully local manner, further improving the update speed while maintaining the consistency.

IV. Algorithms

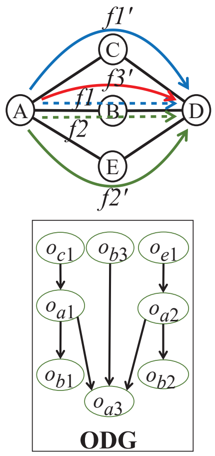
We design two algorithms in DDP: 1) computing the basic DOCs for individual updates, and 2) an optimization for multiple local updates by ODG composition.

A. Computing DOC

This algorithm computes the DOC of each datapath operation to ensure the consistency during an update. It takes the set of operations O and the event e as the input and outputs the corresponding DOCs. The algorithm consists of two steps: (1) ODG construction, and (2) Computing `gate` and `release`.

Example. To illustrate the algorithm, we provide a real-time update example in Fig. 5. Each link has a capacity of 10 units and each flow has a size of 5. The old configuration includes two flows $f1$ and $f2$ with same path ABD (labeled by dashed lines), while the updated one includes two modified flows $f1'$ with path ACD, $f2'$ with path AED and a new flow $f3'$ with path ACD (solid lines). This update is initiated by the controller, so $e = \varnothing$. We can use the 2-step algorithm in our paper to compute the corresponding DOCs.

Step 1: ODG construction. To construct an ODG, we use similar ideas of existing work on consistent updates [5], [8]. To ensure *blackhole- and loop-freedom*, 'insert' and 'modify' operations on the source switch depend on the successors on the flow route. For example, o_{a1} depends on o_{c1}, and o_{a2} depends on o_{e1} in Fig. 5. A 'delete' operation on the last switch depends on the predecessors on the route, *e.g.*, $o_{a1} \rightarrow o_{b1}$ and $o_{a2} \rightarrow o_{b2}$. To ensure *Congestion-freedom*,



Step 1: ODG construction

	operation at A	gate	release
d_{a1}	o_{a1}: modify *f1*	o_{c1}	o_{a3}, o_{b1}
d_{a2}	o_{a2}: modify *f2*	o_{e1}	o_{a3}, o_{b2}
d_{a3}	o_{a3}: insert *f3'*	$(o_{a1}\|\|o_{a2})$ $\&o_{b3}$	

	operation at B	gate	release
d_{b1}	o_{b1}: delete *f1*	o_{a1}	
d_{b2}	o_{b2}: delete *f2*	o_{a2}	
d_{b3}	o_{b3}: insert *f3'*		o_{a3}

	operation at C	gate	release
d_{c1}	o_{c1}: insert *f1'*		o_{a1}

	operation at D	gate	release
d_{e1}	o_{e1}: insert *f2'*		o_{a2}

Step 2: Computing gate and release

Fig. 5. An example of the algorithm computing the basic DOCs.

current remaining resources should be enough for a resource-consuming operation. Otherwise this operation will depend on resource-freeing operations on the same link to get enough resources, *e.g.*, o_{a3} (resource-consuming operation) depends on both o_{a1} and o_{a2} (resource-freeing operations). We use the same priority-criteria in [5] to schedule resource-consuming operations to avoid deadlocks. Lastly, if the update is triggered by a local event e, directional edges will be placed from e to the root operations in the ODG.

Step 2: Computing gate and release. Algorithm 1 is a function to compute the boolean logic in `gate` and the operation set in `release` of each operation o_i. The intuition is that by constructing the ODG in Step 1, the elements in each DOC's `gate` and `release` are determined, while in Step 2, logic operators are further inserted to obtain the final Boolean expressions in `gate`. There is no logic operator in `release`, so d_i.`release` is the operation set of o_i's children. For `gate`, the parent operations not located at the same switch as o_i are responsible for *blackhole-* and *loop-freedom*. Hence these parent operations are joined by $\&$ operators, *e.g.*, o_{b3} in d_{a3}.`gate`. Otherwise, we use another function *FindFeasibleScheduling* to compute the logic.

Specifically, *FindFeasibleScheduling* is a function to find all possible resource-freeing conditions that can make o_i scheduled. We let $o_i.flow^-$ denote the set of resource-freeing operations for o_i, which are identified as the operations on the same switch as o_i. For example, o_{a1} and o_{a2} are in $o_{a3}.flow^-$. If a resource-freeing condition is enough to schedule o_i, the combination becomes a feasible plan (f). Different feasible plans are separated by OR operators. For the example in Fig. 5, o_{a1} and o_{a2} are two feasible scheduling planes for operation o_{a3}. So d_{a3}.`gate` includes $o_{a1}||o_{a2}$. The details of this function can be found in the Appendix B.

B. ODG Composition

As discussed earlier, an ODG corresponds to only one update. If we want the data plane to locally handle one of multiple updates $C_1 = update(C, O_1, e_1), C_2 = update(C, O_2, e_2), \ldots$, we need to prepare multiple ODGs. Here we assume one update is executed at a time, because all of the updates are based on the current configuration C. For the local update example in Fig. 1, if we hope the data plane can locally handle another event $e_2 = AB$ link down, then we need another ODG with a new set of DOCs. Different updates may share common operations, so an ODG composition is required to rewrite the DOCs. We let G_i denote an ODG and $\bigoplus$ denote the composing operator. Since the composing operator $\bigoplus$ is associative, *i.e.*, $(G_1 \bigoplus G_2) \bigoplus G_3 = G_1 \bigoplus (G_2 \bigoplus G_3)$, therefore the composition of arbitrary ODGs can be derived by the algorithm for composing two ODGs in Algorithm 2.

In the algorithm, we combine `gates` and `releases` for the common operations in both G_1 and G_2. The `gates` are joined by an OR ($||$) operator to make sure the common operations can be triggered in both updates. In addition, we have to distinguish the child nodes of a common operation in

Algorithm 1 ComputeGate&Release(o_i)

for each $o_j \in o_i.children$ **do**
  d_i.`release` $\leftarrow d_i$.`release` $\cup\, o_j$
end for
$o_i.flow^- \leftarrow \varnothing$
for each $o_k \in o_i.parents$ **do**
  if o_k and o_i at same switch **then**
    $o_i.flow^- \leftarrow o_i.flow^- \cup o_k$
  else
    d_i.`gate` $\leftarrow d_i$.`gate`$\& o_k$
  end if
end for
$F \leftarrow \varnothing$
for each $f \in FindFeasibleScheduling(o_i.flow^-)$ **do**
  $F \leftarrow F || f$
end for
d_i.`gate` $\leftarrow d_i$.`gate`$\& F$

Algorithm 2 ODGComposition(G_1, G_2)

for each $o_{i1} = o_{i2}, o_{i1} \in G_1, o_{i2} \in G_2$ **do**
  if d_{i1}.`gate` $\neq d_{i2}$.`gate` **then**
    $d_{i1,i2}$.`gate` $\leftarrow d_{i1}$.`gate`$||d_{i2}$.`gate`
  // $d_{i1,i2}$ is the DOC after composing
  end if
  if d_{i1}.`release` $\neq d_{i2}$.`release` **then**
    $d_{i1,i2}$.`release` $\leftarrow d_{i1}$.`release` $\cup\, d_{i2}$.`release`
    $P_1 \leftarrow o_{i1}.parents$
    $P_2 \leftarrow o_{i2}.parents$
    while $P_1 \subseteq P_2 || P_2 \subseteq P_1$ **do**
  // iteratively find the nearest different ancestors
      $P_1 \leftarrow P_1.parents$
      $P_2 \leftarrow P_2.parents$
    end while
    Find $t_1 | t_1 \in P_1 \& t_1 \notin P_2$
    Find $t_2 | t_2 \in P_2 \& t_2 \notin P_1$
  // t_1, t_2 is either an operation or an event
    for each $o_{j1} \in d_{i1}$.`release`$, o_{j2} \in d_{i2}$.`release` **do**
      d_{j1}.`gate` $\leftarrow d_{j1}$.`gate`$\& t_1$
      d_{j2}.`gate` $\leftarrow d_{j2}$.`gate`$\& t_2$
    end for
  end if
end for

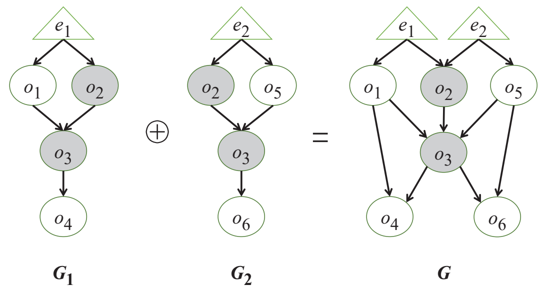


Fig. 6. An example of ODG composition where $G = G_1 \bigoplus G_2$.

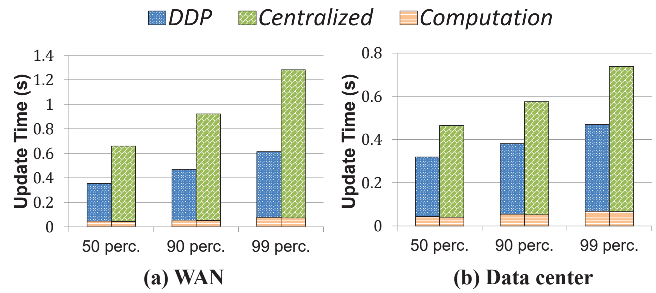


Fig. 7. Update completion time, broken down by the amount of computation and execution.

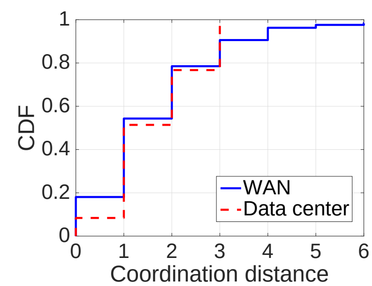


Fig. 8. CDFs of the coordination distance between DOCs, specified by the hop-count of switches between $d.o$ and $d.$`release` (if any), *e.g.*, the distance between d_{a1} (d_{a2}) and d_{a3} in Fig. 5 is 0.

different graphs; otherwise all of them will be released after the common operation. To cope with this, we iteratively find the nearest different ancestors to identify the two ODGs. First, we find P_1 and P_2 as the two non-containment ancestor sets for the common operation o_{i1} (o_{i2}). Then we pick out only one item t_1 (t_2) as a representative for each set P_1 (P_2). Here t_1 and t_2 are any of the elements in the ODG, *i.e.*, either an operation or an event. At last, we add t_1 into every child's `gate` in G_1, and t_2 into every child's `gate` in G_2, with the operator $\&$. The rewrites of `releases` for t_1 and t_2 are omitted for space constraints. As a result, for the example in Fig. 1, $d_c.$`gate` $= e_1 \| e_2$ after composition, so that no mater AB or BD down, the network can locally recover routing.

Example. We give another example in Fig. 6, where G_1 and G_2 are the ODGs for two local updates prepared in DDP (e_1 and e_2 have not happened yet). After composing, $d_2.$`gate` becomes $e_1 \| e_2$, and $d_2.$`release` $= o_3$ keeps unchanged. For o_3, both the `gates` and `releases` are different in the two ODGs, so they are rewritten as $d_3.$`gate` $= o_2 \& (o_1 \| o_5)$ and $d_3.$`release` $= \{o_4, o_6\}$. In addition, $P_1 = \{o_1, o_2\}$ and $P_2 = \{o_2, o_5\}$ are found as the nearest non-containment ancestor sets, and $t_1 = o_1$ and $t_2 = o_5$ are picked out to represent P_1 and P_2 respectively. In the end, o_4 and o_6 are distinguished by $d_4.$`gate` $= o_1 \& o_3$ and $d_6.$`gate` $= o_3 \& o_5$.

V. Performance Evaluation

We fully implement the DDP system with 3000+ lines of Python code to evaluate its performance in various update scenarios. Experimental results show that DDP can significantly speed up network updates.

A. Experimental Methodology

We conduct all experiments on real topologies consisting of Open vSwitches in both WAN and data center scenarios. For WAN, we choose 5 topologies from the Topology Zoo [10] that interconnect $O(30)$ sites with link capacities between 10 and 100 $Gbps$. For data center, we emulate a 3-tier datacenter network topology with $O(60)$ switches, where each edge link is of $10Gbps$ capacity, and aggregated link is of $100Gbps$ capacity. A custom software agent is running on each switch to coordinate with each other and create execution logs. The communication protocol between the controller and the agents is implemented by an extension version of OpenFlow, and the communication between the agents is via UDP messages. In our experiments, we compare DDP against a Centralized update system based on Dionysus [5].

B. Experimental Results

Real-time updates. We measure the update completion time of 30 real-time updates in both WAN and data center scenarios. We break down the overall time by the amount of computation at the controller and the execution at the data plane. First, Fig. 7 shows that computing the updates is not a bottleneck in both schemes, and the DDP system requires a little longer computation time than Centralized because the scheduling plans are pre-computed and encoded in the DOCs.

From Fig. 7, we can observe that DDP achieves a much lower execution time than Centralized. This major gain is from the distributed manner of update execution in DDP. Centralized approach relies on the controller to orchestrating the updates, so the coordinating time is a sum of many-round of communication between the controller and switches. However in DDP, switches directly coordinate with each other at the data plane, therefore reducing the total execution time. From the CDFs shown in Fig. 8, we can see most of the coordination in DDP is within the same switch (up to 18.4%) or between adjacent switches (up to 43.0%), so it's more efficient for switches to coordinate with each other rather than communicating with the remote controller.

Overall, as shown in Fig. 7, DDP outperforms Centralized in real-time updates under both WAN and data center settings. For WAN, DDP is 46.4%, 49.1%, and 52.1% faster than

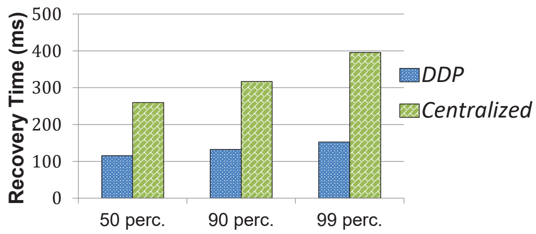


Fig. 9. Recovery time from a random link failure.

Centralized in the 50^{th}, 90^{th}, and 99^{th} percentile, respectively. For data center, the corresponding numbers are 31.4%, 33.7%, and 36.4% faster than Centralized,. The WAN topologies are more complex than the data center ones, resulting in longer completion time.

Local updates. We conduct another experiment to show the benefit of DDP in local updates. We choose one configuration in each WAN topology, and pre-compute the DOCs for any link-down events. For simplicity, we consider only 1-failure case. The ODG Composition algorithm in Sec. IV-B is used to compute the final DOCs, and the DOC number is reduced by 60.3% after composition. In the experiment, we randomly fail one link and measure the recovery time for all re-routable flows in both approaches. In Centralized approach, the link-down event has to be reported to the controller, who will then compute new routes to update the network. But in DDP, when the link-down event happens, pre-stored DOCs are locally trigged at the data plane and directly take effect to recover the routing. Note that the consistency properties are preserved during all updates. As shown in Fig. 9, DDP is 55.6%, 58.2%, and 61.4% faster than Centralized in the 50^{th}, 90^{th}, and 99^{th} percentile, respectively. By locally initiating the updates at the data plane, DDP avoids reporting time and real-time computation time at the controller, therefore speeds up the updates further more.

VI. Related work

De-centralized update. ez-Segway is proposed to address the network update problem in a de-centralized manner, where switches receive partial knowledge of the network from the controller and conduct distributed computing to execute the update [8]. DDP's improvement over it is a much more powerful primitive DOC, leading to much lower overhead and computation complexity at switches, while enabling local updates. A timed update is another decentralized approach that uses synchronized clocks to coordinate the update [11]–[13]. However, due to imprecise clock synchronization and time prediction, the consistency and efficiency of network updates are not guaranteed as in DDP. Our primitive DOC can have a wide range of extension, *e.g.*, the Boolean expression in `gate` may support time variant in the future work.

Local Recovery. Prior art on failure recovery in SDN relies on Openflow local fast failover mechanisms [14]–[16]. The controller pre-installs backup rules (tunnels) in switches' group tables, so that the backups can be immediately activated upon a link failure. DDP transforms the backup rules into compact pending DOCs, therefore saves the scarce table resources without performance loss. In addition, this line of work deals with only link-down events, whereas DDP is capable of handling any happenings that can be detected by switches, such as link congestion or unbalanced load. Our novelty lies in allowing local events to directly trigger the network-wide update, which to our knowledge has not been done before.

VII. Conclusion

This paper presents DDP, a system for fast, distributed, consistent network updates in SDN. The configuration in DDP is based on a novel primitive named DOC. The DOCs can be executed by the switches following the dependencies among different datapath operations, while achieving data plane consistency. We also design two algorithms to compute and optimize the primitive DOCs respectively. Evaluation results show that DDP significantly improves network update speed in various update scenarios. Developing some high-level APIs in DDP to automatically generate the DOCs will be the next step of this research.

References

[1] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[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 Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[3] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, 2013, p. 20.

[4] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 323–334, 2012.

[5] 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 CCR*, vol. 44, no. 4, 2014, pp. 539–550.

[6] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," in *ACM SIGCOMM CCR*, vol. 43, no. 4, 2013, pp. 411–422.

[7] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *IFIP Networking Conference*, 2016, pp. 1–9.

[8] T. D. Nguyen, M. Chiesa, and M. Canini, "Decentralized consistent updates in sdn," in *Proceedings of the SOSR*, 2017, pp. 21–33.

[9] "Anoymous Technical Report," https://github.com/technical-report-2018/ICDCS2018.

[10] "The Internet Topology Zoo," http://www.topology-zoo.org.

[11] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *Proceedings of the 1st ACM SIGCOMM SOSR*, 2015, p. 21.

[12] J. Zheng, G. Chen, S. Schmid, H. Dai, J. Wu, and Q. Ni, "Scheduling congestion-and loop-free network update in timed sdns," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, 2017.

[13] J. Zheng, G. Chen, S. Schmid, H. Dai, and J. Wu, "Chronus: Consistent data plane updates in timed sdns," in *Distributed Computing Systems (ICDCS), IEEE 37th International Conference on*, 2017, pp. 319–327.

[14] J. Zheng, H. Xu, X. Zhu, G. Chen, and Y. Geng, "We've got you covered: Failure recovery with backup tunnels in traffic engineering," in *Network Protocols (ICNP), International Conference on*, 2016, pp. 1–10.

[15] M. Borokhovich and S. Schmid, "How (not) to shoot in your foot with sdn local fast failover," in *International Conference On Principles Of Distributed Systems*, 2013, pp. 68–82.

[16] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proceedings of HotSDN*, 2014, pp. 121–126.

APPENDIX A
PROOF OF CORRECTNESS

Proof: Suppose o_1 and o_2 are two arbitrary operations in the ODG G. The dependency of these two operations in the ODG has following five cases:

Case 1: operation o_1 is operation o_2's parent node, i.e., $o_1 \rightarrow o_2$. In this case, o_1 should be executed before o_2. Suppose o_2 arrives at the data plane first. According to the API, o_1 will be included in o_2's gate, so o_2 will not be executed until o_1 arrives at the data plane and finishes the execution. Suppose o_1 arrives first, it can be executed immediately if the gate is satisfied. When o_2 arrives later, it will send a pull message to o_1 to check whehter o_1 has finished. If o_1 has finished, o_1 will send a push message to o_2, if not, o_1 will send the push message when it finishes. So no matter which operation arrives first, these two operations will follow the logic order and eventually be executed.

Case 2: operation o_1 is operation o_2's child node, i.e., $o_2 \rightarrow o_1$. Same to Case 1, the execution order can be guaranteed.

Case 3: operation o_1 is operation o_2's ancestor node. Then there must be a path from o_1 to o_2 in the ODG, denote as $o_1 \rightarrow o_{a1} \rightarrow o_{a2} \rightarrow ... \rightarrow o_2$. We use the solution in Case 1 on each parent-child pair in the path. Notice that the exection order has transitivity, the finial exection order of o_1 and o_2 can be guaranteed.

Case 4:operation o_1 is operation o_2's descendant node. The proof is same to Case 3.

Case 5:operation o_1 and o_2 have no dependency. Then there is no constraint on their exectuion order.

Above all, the execution order of arbitrary two operations will follow the logic order in the ODG in all cases. ■

APPENDIX B
FIND FEASIBLE SCHEDULING

Algorithm 3 shows the detailed processing of function *FindFeasibleScheduling* (FFS) which finds the feasible plan to schedule o_i. The input $flow^-$ is the resource-freeing operations obtained by Alogrithm 1, $avres$ is the current available resource, $conres$ indecates the resource that o_i consumed, f is one feasible scheduling of $o_i.flow^-$ and FS stores all the feasible plan f.

If the available resource is enough for o_i to consume then the feasible scheduling plan f is stored in the feasbile scheduling set FS (Lines 1-4). The alogrithm iteratively traverse all the combinations in $o_i.flow^-$ (Lines 5-9) and obtains the set of all feasible plans FS.

Algorithm 3 FFS($flow^-$,$avres$,$conres$,FS,f)

1: **if** $avres \geq conres$ **then**
2: $\quad FS \leftarrow FS \cup f$
3: $\quad$ **return**
4: **end if**
5: **for** each operation $o \in flow^-$ **do**
6: $\quad f \leftarrow f\&o$
7: $\quad$ FFS($flow^- \setminus o$,$avres + o.relres$,$conres$,FS,f)
8: $\quad f \leftarrow f \setminus o$
9: **end for**