DOC: A Simple Primitive Realizing Fast, Distributed, Consistent Data Plane Configuration

Paper ID: 103, 13 Pages.

ABSTRACT

To apply a network configuration, multiple operations need to be executed on different data plane devices. These operation executions require concurrency control to maintain data plane consistency, such as route correctness and loop freedom. Current software defined networking (SDN) heavily relies on the (logically) centralized controller to decide the order of consistent operation executions and send control messages to switches following that order, which may result in long latency of configuration completion and high control bandwidth cost. In this paper, we present Datapath Operation Container (DOC), a simple, but powerful datapath configuration primitive supporting a wide range of capabilities from spatial synchronization for network-wide update to temporal synchronization for local failover. The controller computes the operation dependencies and encodes them into a DOC for each operation. Then the data plane devices concurrently execute these operations in a distributed manner while achieving network consistency. We further propose two high-level APIs, the waypoint and contingency APIs, to automatically transform network configurations to detailed operations and DOCs. Novel algorithms are designed to realize both continuous and contingent configurations. We conduct testbed experiments and data-driven simulations, showing that our primitive significantly improves the configuration speed by up to 43.4% in network updates and up to 39.0% in local failover, while incurring small overhead.

1 INTRODUCTION

Software-defined networking (SDN) is considered a major recent advance in networking [11, 16]. It significantly simplifies network management and provides real-time network programmability by decoupling the network control plane from the data plane. In order to achieve the full potential of SDN, it is important to define appropriate datapath primitives to ensure safety, scalability and consistency. By introducing increasingly flexible matching and packet-processing-actions primitives, existing datapaths (*e.g.*, OpenFlow [25] and P4 [5]) have substantial capabilities to conduct datapath packet processing.

Despite the centralization of the control plane, the data plane, however, still needs to be regarded as a distributed system [39]. Particularly, due to asynchronous communication channels, control messages are received and executed by switches in an order different from the order sent by the controller [14, 20, 27, 28]. An inappropriate control order may violate the dependencies of operations on the datapath. As a result, there will be transient inconsistency during the data plane configuration, which may precipitate network anomalies, such as blackholes, traffic loops and congestion, resulting in packet loss and poor performance [7, 12, 19].

Existing datapaths have only configuration primitives between the controller and datapath devices, including basic primitives, such as rule modification from the controller to devices, execution confirmations from devices to the controller, and advanced primitives such as barrier messages for the controller to better control the configuration [25]. Using only centralized configuration primitives to ensure operation dependencies, however, relies on the controller too heavily. When the controller becomes a bottleneck, the network can suffer from substantial performance and reliability degradation. For example, for the in-band control plane, the control messages may have a race condition with data messages, or the connection with the controller may get lost, resulting in unreliability and huge latency.

What is missing, however, is primitives for distributed data plane configuration. The recognition of aforementioned issues of centralized configuration is not new, and multiple approaches (*e.g.*, [6, 9, 10, 16, 26, 37]) have been proposed in previous studies to scale data plane configuration. These existing approaches, however, focus on delegating certain capabilities to data plane, while neglecting the coordination between switches for consistent execution control. It is not clear whether one can design simple, unifying configuration primitives, achieving effective distributed datapath control, while still ensuring the consistency.

In this paper, we present datapath operation container (DOC), a surprisingly simple, but powerful datapath configuration primitive, that enables distributed configurations in a wide range of settings. A DOC is centrally computed and encapsulated at controller with an operation and its dependencies in a `gate` and a `release` fields. The `gate` is the logic combination that the operation depends on, whereas the `release` encodes the logic of who depends on this operation. With two simple modes, Push and Pull, the DOC can be executed by the data plane in a distributed manner, supporting capabilities from spatial synchronization for network-wide update, temporal synchronization for local failover, to stateful programming.

In order to make the primitives easier to use, we propose two high-level APIs that automatically transforms high-level configurations into low-level DOCs. (1) The first is a waypoint API used for real-time configuration, such as network updates. The challenge lies in not only the operation dependencies within one configuration, but also the interaction between configurations at different time, which enables users to continuously (re)configure the data plane on the fly. We design algorithms to completely specify them in the DOCs. (2) The second is a contingency API, that generates compact

contingent DOCs for arbitrary link failures. We design novel algorithms that addresses the exponentially many failure cases by backup rule dependencies, so that the data plane can be locally recovered.

We instantiate our primitive in a system called *Docere* and evaluate it using experiments on a modest-sized testbeds and large-scale simulations. We show that with our primitive, *Docere* effectively supports continuous data plane configuration that has not been done before. We also demonstrate that it improves network update speed by 37.1-43.4%, while incurring only 18.8% more control message overhead. Furthermore, the results validate our primitive in that it allows local failover which is up to 39.0% faster than centralized recovery approaches.

2 MOTIVATION AND BASIC IDEA

2.1 Motivation

2.1.1 Network Updates.

Network updates are among the most common data plane operations. A network update can invole multiple network devices, with operations with dependencies among them. The correct orders in which the operations are applied are important, as a violation can lead to network anomalies, such as blackholes, traffic loops and congestion.

Network Update Example. In Figure 1(a), we provide a simple example of a network update. Each link has a capacity of 10 units and each flow has a size of 5. The old configuration includes two flows $F1$ and $F2$ (labeled by dashed lines), while the updated one includes two modified flows $F1'$, $F2'$ and a new flow $F3'$ (solid lines). To accomplish this update, the data plane requires a set of operations with various dependencies between each other. For instance, switch A cannot be modified to forward $F1$ to C before the flow rule for $F1$ is installed at C; otherwise $F1$ will encounter a blackhole at C where no matched rule exists. Another instance would be like, the rule for $F3$ cannot be installed at A before neither the rule for $F1$ or $F2$ is modified; otherwise link AB (and BD) will be congested due to the capacity constraint of 10.

Current approaches of consistent network updates decompose this process into multiple steps at the SDN controller, and invokes a next step only after confirming the last one is finished [7, 12, 19, 20]. Even the two successive operations located at the same switch still need a round-trip of state exchange between the controller and the switch. The simple update in our example incurs at least 5 or 6 rounds of communications, resulting in substantial extra delays. As we can see, using only centralized primitives to configure the data planes not only slows down completion time, but also increases the controller's processing load.

2.1.2 Local Failover. In data plane configurations, some operations are not applied immediately, but are deferred to take effect until a certain time point. The representative configuration is the datapath fast failover. Upon detecting a failure, the switch can activate and use the corresponding backup forwarding plans immediately. Tunnel-based recovery approaches are proved to have great rule and tag complexity for arbitrary link failures [3]. So we consider using simple contingent operations on flow rules to locally recover routing. However, these temporal-control operations have dependencies with each other, and consistently applying them needs careful planning.

Fast Failover Example. In Figure 1(b), we present an example to show the dependencies between contingent operations. We consider D as the only destination and there are flows coming from all other nodes. Assume shortest-path routing is adopted in this network, where the cost is labelled on each link. To protect the failure of link CD, switch C pre-stores a backup rule that forwards the traffic to B for destination D. However, when CD is failed, switch B, without awareness of the failure before the controller tells it, will continue to use the ordinary flow rule that forwards the packets to C for destination D. Consequently, there will be a traffic loop for the flows from C to D when C executes the backup operation.

The traffic loop essentially stems from the dependencies between contingent operations. Particularly, if C attempts to use the backup rule for CD failure, then B has to use its own backup rule for BC failure. The two contingent operations are responsible for two different failures, but have an inherent dependency. Unfortunately, distributed protocols for applying contingent operations need complicated coordination between switches, resulting in long convergence time to recover routing [8, 17, 38].

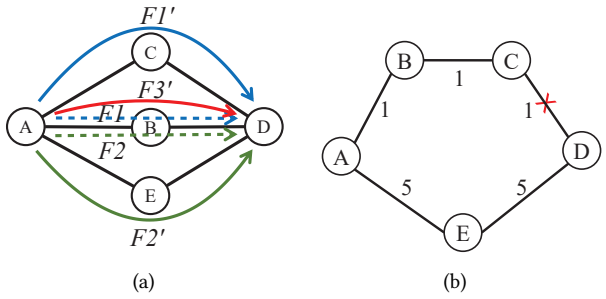


Figure 1: Data plane configuration examples. (a) A network update, with solid lines for the new configuration and dashed lines for the old one. (b) A local failover example, where link cost is labeled at each link.

2.2 Basic Idea

Given that centralized configuration primitives rely on the controller too heavily and distributed configuration needs complicated coordination for consistency, we wonder if there is an approach where the central controller with global view can pre-compute the operation dependencies and encode them into simple primitives, so that the data plane can fast, consistently execute them to complete the configurations in a distributed manner.

In this paper, we propose the first distributed primitive for SDN data plane configuration with the following principles. (1) Simple but complete: the operation dependencies are compactly expressed in a smart way, so that consistency properties are ensured. (2) Executable in a distributed manner: with the primitive, the data plane can fully accomplish the configuration, without involving the controller any more.

(3) Synchronized for both spatial and temporal control: the primitive should ensure not only spatial consistency that enables safe transitions of distributed forwarding states, but also temporal consistency that allows localized recovery by contingent configurations.

3 DATAPATH OPERATION CONTAINERS

We introduce the concept of Datapath Operation Containers (DOCs), which is a distributed configuration primitive to represent data plane operations and their dependencies.

3.1 Specification of DOC

A Datapath Operation Container (DOC) includes one data plane operation and three fields describing the dependencies of this operation, as shown in the following:

Datapath Operation Container (DOC)			
`operation`	`gate`	`release`	`status`

To define `gate` and `release`, we first introduce the concept of an Operation Dependency Graph (ODG). An ODG is an abstraction of operation dependencies, represented by a directed graph whose vertices are operations. If an operation o_1 depends on the completion of another operation o_2, a directional edge is placed from vertex o_1 to vertex o_2. If an operation o_1 depends on a logic combination of multiple operations, then an edge is placed from o_1 to every operation in the logic combination. For example, if o_1 depends on the completion of either o_2 or o_3, then two edges are placed from o_2 to o_1 and o_3 to o_1, respectively. If an edge is placed from o_1 to o_2, we say that o_1 is an upstream operation of o_2 and o_2 is a downstream operation of o_1.

The two fields `gate` and `release` of a DOC of an operation o are explained as follows:

- `gate` is the condition to execute o, represented by a Boolean expression. The Boolean expression may consist of one or more other operations and the Boolean operators (AND (&) , OR (||) and NOT (!)). For example, if the execution of operation o depends on the completion of either o_2 or C, then o.`gate` = $o_2 || o_3$. Only upon satisfying the gate, o can be executed.
- `release` is a Boolean expression of all downstream operations of o. Upon the completion of o, all operations in `release` will be notified and freed. The semantic of the Boolean expression is the triggering condition for a contingent operation, which will be specified in Section 3.2.2.

The `status` field equals one of the following: '*Sleep*', '*Active*', '*Complete*', which will be explained in § 3.2.

Examples. Using the example in Figure 1(a), Figure 2 illustrates the ODG and all the DOCs involved in the network update. To ensure blackhole-freedom, installation of each new rule depends on the completion of the flow successor's rule installation. For example, A_{F1} depends on C_{F1} and A_{F2} depends on E_{F2}. In addition, to avoid congestion on the path *ABD*, flow $F3'$ cannot start until either flow $F1$ or flow $F2$ is moved from *ABD* to another path. Therefore A_{F3} depends on either A_{F1} and A_{F2}, and the `gate` of A_{F3} is hence $(A_{F1} || A_{F2}) \& B_{F3}$. The 'Delete' operation of a forwarding rule depends on the establishment of another path for the flow. For example, after establishing the path *ACD* for flow $F1'$, switch B can execute B_{F1} and delete the rule related to $F1$ to release the flow table space.

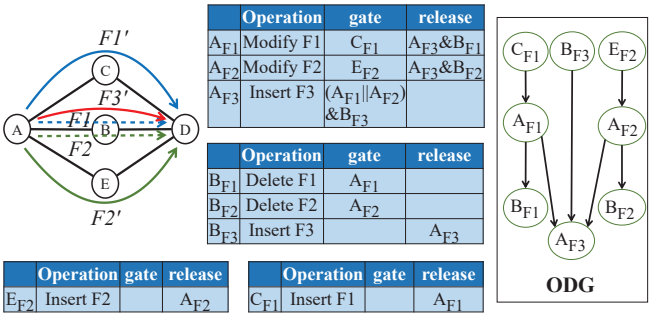
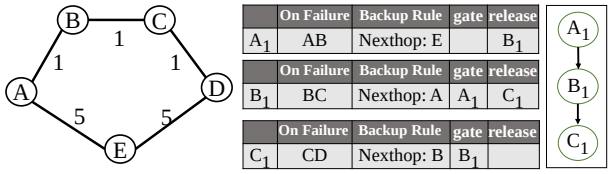


Figure 2: Network update example with the ODG and DOCs.

As another example, in the fast failover scenario of Figure 1(b). In Figure 3, we list three contingent operations A_1, B_1 and C_1 which are located at different switches and responsible for the failures of links *AB*, *BC* and *CD*, respectively. As analyzed before, if switch C wants to execute C_1 due to the failure of link *CD*, then switches B and A should have already executed their backups B_1 and A_1. Otherwise there will be traffic loops for the flows from C to D. Note that the ODG describes the global dependencies for arbitrary failures in the network, rather than just for the failure of link *CD*. For instance, if link *BC* fails, switch A also need to execute A_1 before B executing B_1.



(a) Topology (Destination: D) **(b) Contingent DOCs** **(c) ODG**

Figure 3: Fast failover example with the ODG and DOCs.

3.2 Basic Execution Modes

We define two types of execution modes for every DOC: a Push mode and a Pull mode.

3.2.1 Push Mode.

The Push mode is mainly used when the SDN controller actively modifies the data plane of switches to implement a new configuration.

Push mode work-flow. First, the SDN controller computes all update operations and their dependencies. It further constructs the DOC for each operation based on the dependency information. These DOCs are then sent to the corresponding switches for execution. We use a finite state machine to represent the changes of `status` in the Push mode, as shown in Figure 4. Initially the `status` of all DOCs are '*Active*'. If the `gate` of a DOC is empty or satisfied, the switch

will execute the operation and send PUSH messages to notify downstream DOCs in the `release`. Then the `status` becomes '*Completed*'.

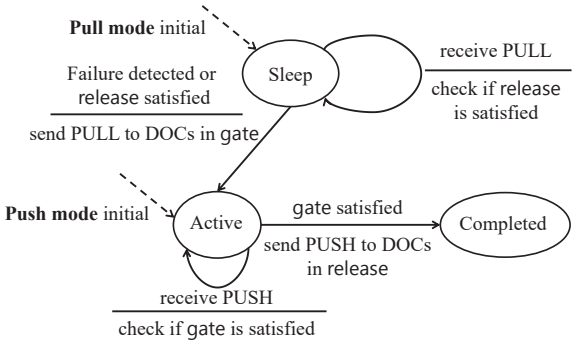


Figure 4: Finite machine for Push and Pull modes.

Push mode example. For the example in Figure 2, the SDN controller sends eight DOCs to all switches. C_{F1}, E_{F2} and B_{F3} can be executed immediately, because their `gate` are empty. A_{F1} (A_{F2}) will be executed after receiving the PUSH messages from C (E), and then sends PUSH messages to release A_{F3} and B_{F1} (B_{F2}). Note the PUSH message to trigger A_{F3} can be simplified as a notification signal within the switch. After collecting the PUSH messages from B_{F3} and either A_{F1} or A_{F2}, DOC A_{F3} will be executed. B can delete the old rules specified in B_{F1} and B_{F2} by receiving the PUSH messages from A_{F1} and A_{F2} respectively.

3.2.2 Pull Mode.

The Pull mode is mainly used when a network dynamic happens and the data plane needs to make changes to maintain functional correction, e.g, fast failover under failures. With our primitive, switches can coordinate with each other and determine the appropriate execution order of contingent operations to guarantee the data plane consistency during the failover.

Pull mode work-flow. The SDN controller first pre-computes all contingent operations, their dependencies, and the DOCs. As shown in Figure 4, all DOCs of the Pull mode, such as those for the backup rules, are initially in the status '*Sleep*'. If an event that triggers an operation, e.g.*e.g.*a link failure, happens, and detected by a switch, then the corresponding DOC on this switch will become in '*Active*' status. In the mean time, the switch will send a PULL messages to every DOC in `gate`, in order to 'wake' them. If another DOC receives the PULL message, it will evaluate whether its `release` is satisfied by one or more received PULL messages. If `release` is satisfied, it will become '*Active*' and send PULL to the DOCs in `gate`. After becoming '*Active*', the work-flow is the same to the Push mode.

Pull mode example. We use the example in Figure 3 for illustration. When link CD fails, switch C realizes this event and the DOC C_1 becomes '*Active*'. C then sends a PULL message to B_1 who is in C_1.`gate`. After B_1 receiving the PULL message, it find its `release` is satisfied and hence also become '*Active*'. Similarly A_1 will also be activated. Since A_1's `gate` is empty, it can be executed and A sends the a PUSH message to B_1. After that, B_1 and C_1 will also be executed sequentially.

3.3 Discussion

PUSH and PULL messages. The PUSH and PULL messages are logically transmitted between DOCs, but physically transmitted between switches. If the sending and receiving DOCs are located at a same switch, the message becomes a logical notification signal in memory, and no real message is sent. The PUSH and PULL messages can be sent to a non-adjacent switch. A PUSH message works as an acknowledgment between two DOCs. The message format is a tuple $\{PUSH, Dst, Src\}$, where Src is the sending DOC, and Dst is the receiving DOC. A PULL message is formatted similarly.

Failure behaviors. The basic, tentative behavior on protocol failures, such as connection/message loss, is fail-stop. Each DOC can set timers for Push and Pull behaviors. Upon failures or time-outs, related switches will stop execution and notify the controller to take over the remaining process, including recomputing DOCs or global orchestration for centralized execution. The detailed analysis of our primitive can be found in Appendix A.

4 HIGH-LEVEL APIS

It may not be easy for a network operator to directly specify datapath operations to (re)configure the network. Hence we provide a set of high-level APIs that automatically transform high-level configurations into low-level DOCs. One is called the *waypoint API*, and the other is called the *contingency API*. They cover most situations of network configurations.

Work-flow. With the high-level APIs, a network configuration consists of two phases: centralized computation at the controller and distributed executions at switches. (1) In the computation phase, the network operator submits new configurations by high-level APIs. The controller computes the operations, their dependencies, and corresponding DOCs, which are then sent to the switches. (2) In the execution phase, switches apply the operations concurrently according to DOCs, without the intervention of the controller.

4.1 Waypoint API: set_target_configuration

The waypoint API allows the network operator to continuously input high-level configurations for network updates. It transforms the high-level configurations to low-level DOCs while resolving the dependency and consistency problems within one configuration and among multiple consequence configurations. The waypoint API is defined in Figure 5.

The input C_t is a sequence of high-level instructions made at time t, each of which is to insert/delete/modify a flow route. This API includes a number of steps of calling different functions. We explain these functions as follows.

4.1.1 Function I: GetOps.

This function transforms high-level instructions on flow routes C_t into low-level operations on flow rules. The output $O_t = \{o_t(i)\}$ is a set of operations to realize the high-level

```
set_target_configuration(C_t)
{
1:  O_t = GetOps(C_t)// get target operations
2:  D_t^0 = CompDOCs(O_t)//compute DOCs of target operations
3:  Ō_t = GetExeOps(t)//get executed operations
4:  D_t = CompIntDOCs(D_t^0, Ō_t)//compute final DOCs
}
```

Figure 5: The waypoint API.

configuration C_t, *e.g.*, insert/delete/modify of rule at a particular datapath device. This function can be done by a number of existing work [32, 35] and is a mature technique. Hence we skip the details.

4.1.2 Function II: CompDOCs.

This function finds the operation dependencies within the new configuration. It takes the set of operations O_t as the input and computes the corresponding DOCs D_t^0. The function consists of three steps: (1) ODG construction, (2) Computing primitives, and (3) Completing fine-grained dependencies.

Step 1: ODG construction. To construct an ODG, we use similar ideas of existing work on consistent updates [12, 23]. To ensure *Blackhole- and loop-freedom*, 'Insert' and 'Modify' operations on a switch depend on the successor of this switch on the flow route. A 'Delete' operation on a switch depend on the predecessor of the switch on the route. To ensure *Congestion-freedom*, the current remaining resources should be enough for a resource-consuming operation. Otherwise this operation will depend on the resource-freeing operations to get enough resources. We also use the same priority-criteria in [12] to schedule resource-consuming operations to avoid deadlocks. Since this part is not our main contribution, we skip the detailed description of ODG construction due to space limit.

Step 2: Computing primitives. Algorithm 1 is a function to compute the boolean logic of `gate` and `release` of each operation $o_t(i)$. The intuition is that by constructing the ODG in Step 1, the elements in each DOC's `gate` and `release` fields are determined, while in Step 2, logic operators are further inserted to obtain the final Boolean expressions. In the waypoint API, the `release` field includes only the AND logic, because an operation has to notify all neighboring downstream nodes in the ODG after completion. So `release` is the AND of all downstream operations. For `gate`, upstream operations that are not located at the same switch of $o_t(i)$ are responsible for blackhole freedom. Hence all upstream operations are joined by & operators. Otherwise, we use another function *FindFeaibleScheduling* to compute the logic, *e.g.*, the logic among A_{F1}, A_{F2} and A_{F3} in Figure 2.

Specifically, *FindFeaibleScheduling* is a function to find all possible resource-freeing conditions that can make $o_t(i)$ scheduled. Different operation combinations in $o_t(i).flow^-$ provide many resource conditions. If the available resource is enough for $o_t(i)$, the combination becomes a feasible plan. Two feasible plans are separated by OR operator $\|$. In the example of Figure 2, A_{F1} and A_{F2} are two feasible scheduling planes for operation A_{F3}. So A_{F3}.`gate` includes $A_{F1}\|A_{F2}$.

Algorithm 1 CompPrim($o_t(i)$,ODG)

```
1:  for each o_t(j) ∈ downstream(ODG, o_t(i)) do
2:      o_t(i).release ← o_t(i).release&o_t j
3:  end for
4:  o_t(i).flow^- ← ∅
5:  for each o_t(k) ∈ upstream(ODG, o_t(i)) do
6:      if o_t(k) and o_t(i) at same switch then
7:          o_t(i).flow^- ← o_t(i).flow^- ∪ o_t(k)
8:      else
9:          o_t(i).gate ← o_t(i).gate&o_t(k)
10:     end if
11: end for
12: FS ← FindFeaibleScheduling(o_t(i).flow^-)
13: o_t(i).gate ← o_t(i).gate&FS
```

Step 3: Completing fine-grained dependencies. In this step, we add the dependencies between operations applied in one switch. We summarize two criteria that all operations should refer to in our algorithm.

1. Priority. In single-table matching, wildcards and priorities are used together to achieve better scalability, and only the highest-priority flow rule that matches the packet is selected. Therefore, to avoid matching mistakes, the higher priority a rule has, the earlier it must be installed.
2. Table jump. In a multi-table pipeline, each rule depends on the tables that packets will jump to. So in order to prevent table jump blackholes and errors, all successive tables lying on the common data path should be operated before the first-table operations.

4.1.3 Function III: GetExeOps.

According to the OpenFlow specification, switches must send to the controller all Asynchronous messages generated by OpenFlow state changes. Then the controller can be aware of the operation confirmations. This function allows the controller to know which operations are confirmed to have been executed at t. Note that $\overline{O}_t$ does not include new operations, i.e., $\overline{O}_t \cap O_t = \varnothing$

4.1.4 Function IV: CompIntDOCs. This function finds the interaction between new configuration and previous configurations. The interaction is classified into two types: cross-dependency and short-cut.

- Cross-dependency: The dependency between operations for a different configurations.
- Short-cut: If an operation in a new configuration can replace an operation in previous configurations, there will be a short-cut on execution.

Dynamic operation dependency graph. In order to find the interaction, we need a new data structure: dynamic operation dependency graph (D-ODG), which is a modification and extension of the ODG defined earlier. The program maintains a D-ODG as a global variable to keep track of the continuous operation dependencies, which interprets a

Algorithm 2 CompIntDOCs($D_t^0, \overline{O}_t$)

```
1: ∀o_t'(j) ∈ Ō_t
2: G_t' ← G_t' − o_t'(j)
      //Update D-ODG
3: for each o_t(i) ∈ D_t^0 do
4:     for each o_t'(j) ∈ G_t' do
5:         if o_t(i) depends on o_t'(j) then
      //Cross-dependency
6:             o_t(i).gate ← o_t(i).gate&o_t'(j)
7:         else if o_t(i) conflicts with o_t'(j) then
      //Short-cut
8:             o_t^e(i) ← equ_table(o_t'(j), o_t(i))
9:             o_t^e(i).release ← o_t(i).release&o_t'(j).release
10:            o_t^e(i).gate ← o_t(i).gate&!o_t'(j)
11:            o_t(i).gate ← o_t(i).gate&o_t'(j)
12:            D_t ← D_t^0 ∪ o_t^e(i)
13:        end if
14:    end for
15: end for
16: G_t ← G_t' ∪ D_t
```

trace to determine the correct order to arrange and interact new operations with ongoing counterparts. Let G_t denote the up-to-date D-ODG at time t. Intuitively, the objective of Function 4 is to merge an old D-ODG $G_{t'}$ and the ODG of new configuration C_t together into a new D-ODG G_t, where $t' < t$ represents the generic term of previous time points. To unify expression, we denote the operation related to the previous configurations as $o'_t(j)$.

As shown in Algorithm 2 this function first updates the D-ODG $G_{t'}$ by removing executed operations away, then addresses cross-dependency and short-cut in turn, and at last yields a new D-ODG G_t.

Update D-ODG. If an operation is confirmed to have been executed, we remove it from the D-ODG $G_{t'}$.

Cross-dependency. If a new operation $o_t(i)$ depends on a previous one $o_{t'}(j)$, we need to add the previous operation $o_{t'}(j)$ in $o_t(i)$.gate. This dependency is found in a same fashion as Function II. However, the previous operation does not know this new dependency, hence it is not expected to send PUSH message after validity by default. To cope with this, in the modified-Push mode, some activated operations also need to initiate a Pull like in Pull mode and all operations have to send PUSH messages to its releases as well as the operations pulling from it. When $o_t(i)$ is received by the switch, it will take the initiate to pull the acknowledgement from $o_{t'}(j)$. Although $o_{t'}(j)$'s release does not include $o_t(i)$, $o_{t'}(j)$ is still liable to respond after completed. If it has been executed before receiving the PULL message, it still needs to send back a PUSH message.

In the example of Figure 2, we assume that, during the update, a new flow $F4'$ with path ABC is coming to be configured. In static systems, the new configuration has to wait until the last update is finished. But our approach can immediately take the job, and the two new operations A_{F4} and B_{F4} are put into Algorithm 2. In the algorithm, A_{F4} is found to depend on two previous operations A_{F1} and A_{F2} on account of the bandwidth constraint. So after adding $A_{F1}\&A_{F2}$ in its gate, A_{F4} encapsulated in a DOC, will be sent to the data plane. By adopting the modified-Push mode, A_{F4} will spontaneously pull the responses from A_{F1} and A_{F2}, so that $F4'$ can join the network as soon as $F1$ and $F2$ have changed their paths.

Short-cut. Short-cut applies when new operations have conflict with previous ones. Here a conflict means the two operations are applied at the same switch and works for the same flow. For example, the old operation attempts to insert a rule but is not applied yet, and the new one wants to delete this rule. In this case, the old one becomes useless, because it will be replaced by the new one eventually. A naive method is to let the new one wait for the old one to complete, which is not efficient. For optimization, the new operation can take effect directly by short cut. Assume the old one is not applied, the new operation can be transformed into an equivalent one according to the equ_table in Table 1. The equivalent operation will inherit both gate and release as the new one, and inherit the release of the old one. Note if $o_t^e(i) = equ\_table(Insert, Modify)$, it will not inherent $o_t(i)$.release.

$d_{t'}(j) \setminus d_t(i)$	**Insert**	**Delete**	**Modify**
Insert	NA	∅	Insert
Delete	∅	NA	NA
Modify	NA	Delete	Modify

Table 1: equ_table

Short-cut safety. Although the old operation is not confirmed before the computation, it may be executed during or after this progress. To guarantee short-cut safety, we must prepare for both cases. We put the old operation $o_{t'}(j)$ in the new operation $o_t(i)$.gate, and put $!o_{t'}(j)$ in the equivalent operation $o_t^e(i)$.gate. Both $o_t(i)$ and $o_t^e(i)$ will be sent to the switch. In the modified Push mode, they will pull $o_{t'}(j_2)$ to ask the status. If $o_{t'}(j)$ has been executed, it will send back a PUSH to apply $o_t(i)$, and there is no short-cut in this case. Contrarily, if $o_{t'}(j)$ is not executed, it will send back a !PUSH to apply $o_t^e(i)$, and the old $o_{t'}(j)$ and $o_t(i)$ will be deleted for optimization.

4.1.5 *Example.*

We now provide an example in Figure 6 to illustrate the algorithm execution of the waypoint API. Given a topology in Figure 7(a), a user continuously submits two configurations. The first configuration C_1 requests inserting a route $a-b-e-d$ and is transformed to three DOCs $DOC1$, $DOC2$, and $DOC3$, whose dependency is also computed and shown in Figure7(b). The second configuration C_2 changes the flow route to $a-c-e-d$, resulting three target DOCs $DOC4$, $DOC5$ and $DOC6$. At the time of inputting C_2, $DOC2$ is reported to have been executed. Our algorithm finds the cross-dependency and update the D-ODG to that in Figure7(c).

$DOC3$ is added into DOC4.gate because the two operations at different time have a cross-dependency. Since $DOC4$

At time 1: A user submits C_1: $set\_target\_config(C_1)$
C_1 = insert, match=< a, d >; route=< a, b, e, d >
1: $D_1 = D_1^0 = CompDOCs(O_1)$
DOC1: sw a: insert $a \rightarrow b$; gate: DOC2&DOC 3; release: ∅
DOC2: sw b: insert $b \rightarrow e$; gate: ∅; release: DOC1
DOC3: sw e: insert $e \rightarrow d$; gate: ∅; release: DOC1
At time 2: The user submits C_2 immediately: $set\_target\_config(C_2)$
C_2 = modify, route=< a, b, e, d >←< a, c, e, d >
1: $D_2^0 = CompDOCs(O_2)$
DOC4: sw a: modify, flow_match, ouput port=c; gate: DOC5; release: DOC6
DOC5: sw c: insert $c \rightarrow e$; gate: ∅; release: DOC4
DOC6: sw b: delete flow_match; gate: DOC4; release: ∅
2: Assumption: DOC2 is executed in $\overline{O}_2$
3: $D_2 = CompIntDOCs(D_2^0, \overline{O}_2)$
DOC4: sw a: modify, flow_match, ouput port=c; gate: DOC5&DOC3&DOC1; release: DOC6
DOC5: sw c: insert $c \rightarrow e$; gate: ∅; release: DOC4
DOC6: sw b: delete flow_match; gate: DOC4; release: ∅
DOC7: sw a: insert $a \rightarrow c$; gate:DOC5&DOC3&!DOC1;release:∅

Figure 6: A waypoint API example

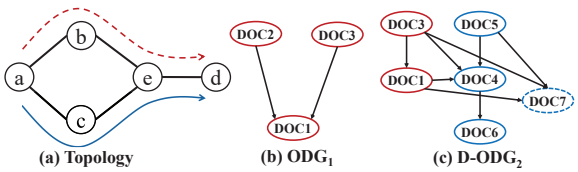


Figure 7: The topology and ODGs in the waypoint API example.

and *DOC*1 conflict, short-cut can apply and the algorithm generates an equivalent *DOC*7 according to Table 1. Both *DOC*4 and *DOC*7 are sent to switch a. If *DOC*1 is executed already, *DOC*4 will be used. Otherwise *DOC*1 will be deleted, and *DOC*7 will be executed. Hence *DOC*4.gate includes *DOC*1, whereas *DOC*7.gate includes !*DOC*1.

4.2 High-Level Contingency API: set_contingent_configuration

Besides real-time configurations executed based on the current state, some contingent configurations can be also triggered by local states, such as on detecting switch/link failures. This API is used for encapsulating backup operations (rules) into DOCs to ensure the temporal consistency. When contingencies (link failures) happen, the data plane can locally recover new routes according to the contingent DOCs. The API is defined as in Figure 8

set_contingent_configuration(*policy*)
{
1: $\mathcal{R}$ = *GetBackup*(*policy*) //get backup rules
2: D = *CompContDOCs*($\mathcal{R}$) //compute the contingent DOCs
}

Figure 8: The contingency API.

4.2.1 *Function I: GetBackup.*

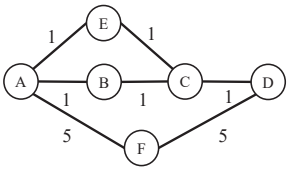
Existing work on backup traffic engineering (TE) can generate contingent plans according to different policies, such as shortest-path reachablity or congestion-free guarantee [18, 36, 41]. Here a set of backup rules are obtained by this function, and we further build their dependencies to allow them being locally activated for failover.

4.2.2 *Function II: CompContDOCs.*

This function is complementary to the backup TE, and we consider only the re-routable failures that retain connectivity between the source and the destination. Taking the backup rules as inputs, we compute the contingent DOCs. Here the operation in a DOC becomes replacing the flow table with a certain backup rule. When some links fail, the related contingent DOCs will be locally activated by the PULL messages and consistently applied to recover the routes. We present an example of shortest-path TE and the backup rules in Figure 9.

Virtual path. We denote an ordinary or backup rule at switch X as X_i, $i \in [0, K-1]$, where K is the number of neighboring links. X_0 represents the ordinary rule while others sorted by priorities represent the backups. Each rule X_i can indicate a "virtual path" (VP_i) to the destination. By "virtual", we mean that there may be branches at successor switches, and they are all regarded as one path. Given the concept of VPs, we have two insights. First, if failures break only one VP_i, they are equivalent to one failure of X_i's next hop. Here breaking a VP means the destination is unreachable via this path. For instance in Figure 9, E_0's VP can be broken by EC failure, but not by CD failure. Second, as long as there is no failure breaking a higher-priority VP, the failures on lower-priority VPs have no impact on switch X. We have the following Theorem 1 (Proof included in Appendix B).

Theorem 1. *X_i will be used iff $\forall i' < i$, $X_{i'}$ can not be used, i.e., there are failures breaking each $VP_{i'}$.*



(a) **Topology (Destination: D)**

	On Failure	Backup Rule	gate	release
A_2	AB	Nexthop: E		B_1& !C_1
A_2	AB&AE	Nexthop: F		B_1&C_1||E_0^o

	On Failure	Backup Rule	gate	release
B_1	BC	Nexthop: A	A_1||A_2	C_1

	On Failure	Backup Rule	gate	release
C_1	CD	Nexthop: B	B_1&A_2	
C_2	CD&CB	Nexthop: E	E_1&A_2	B_1^o

(b) **Contingent DOCs**

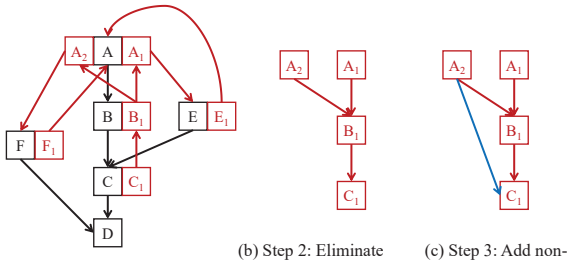
Figure 9: Contingent configuration example. (a) A topology where cost is labeled on each link. (b) Contingent DOCs where red and blue part handles 1-failure, while green part handle arbitrary failures.

Algorithm for single failure. We start with a single-failure case. This algorithm running for each destination, consists of 3 steps. In Step 1, based on the TE results, we build a routing graph, where nodes are flow rules, including ordinary and backup ones. Edges with direction are flow paths.

In Step 2, we eliminate all ordinary rules from the routing graph, together with their in and out edges. Then we reverse the remaining edges. After this process, the routing graph becomes an ODG of backup rules which only includes dependencies between adjacent switches. The connections between backup rules reflect their usage dependency.

In Step 3, we add dependencies between non-adjacent switches. For each X_i, $i \in [2, K-1]$, if there exists a common link failure that breaks $\forall X_{i'}$, $i' < i$, we find the one nearest to X. Assuming this link failure is associated with a backup rule Y_j, then $X_i.\texttt{release} \leftarrow X_i.\texttt{release}\&Y_j$ and $\forall i' < i, X_{i'}.\texttt{release} \leftarrow X_{i'}.\texttt{release}\&!Y_j$. The gate of Y_j is modified as well. The rationale of this step is the basis of our theorem and insights. For necessity, since the failure breaks all previous-priority rules' VP, X_i will be used based on Theorem 1. For sufficiency, based on our first insight, successor failures are equivalent to the first, so using the nearest Y_j is sufficient.

Single-failure example. We illustrate our single-failure algorithm in Figure 10 for the example in Figure 9. In Step 1, the routing graph for destination D is constructed as in Figure 10(a). Black blocks and edges represent ordinary flow rules and paths respectively, while red blocks and edges represent backup flow rules and paths respectively. In Step 2, all black blocks and their neighboring edges are removed, and the remaining edges are reversed as shown in Figure 10(b). In Step 3, we find the non-adjacent dependency existing only at A, because CD failure can break both A_0's VP ($ABCD$) and A_1's VP ($AECD$). Therefore, A_2 and C_1 are dependent as shown in Figure 10(c). The contingent DOCs for 1-failure are encoded in Figure 9 (b), where the red part comes from Step 2 and the blue part comes from Step 3.



(a) Step 1: Routing Graph (b) Step 2: Eliminate black boxes, then reverse (c) Step 3: Add non-adjacent dependencies

Figure 10: Illustration of the three steps in the 1-failure algorithm for the example in Figure 9.

Handling multiple failures. Recovery from multiple failures is one of the most difficult networking problems due to the association between exponential many failure scenarios and backup rules. We novelly address the problem by exploring their dependencies, and therefore enable local recovery of arbitrary numbers of failures. We induce two small changes in the Pull mode to support multiple/arbitary failures recovery. We denote the next hop of X_i to be switch Z. If X_i depends on a backup rule Z_j ($j > 0$) in Z (like C_1 depends on B_1 in the example), X_i will pull Z_j when X_i becomes active. Here the first change is that if the Pull action is blocked because of failures, a denied PULL message Z_j° will be sent back to X. We have a second change that if X_i depends on no backups, X_i also needs to pull the ordinary rule Z_0 to check the failures on $Z_0.VP$. Similarly, if there incurs failures that break the VP, Z will return X a denied PULL message Z_0°.

Algorithm 3 multi-failure_algorithm($\{X_i\}$)

```
for each X_i i ∈ [2, K − 1] do
    if X_{i−1} depends on Z_j (j > 0) then
        X_i.release ← X_i.release||Z_j^∘
    else
        X_i.release ← X_i.release||Z_0^∘
    end if
end for
```

Algorithm 3 completes the dependencies for multiple failures in switch X. In short, a backup rule is activated by the denied message of the previous-priority one. By the modified Pull mode, the failure of each rule is associated with a denied message, so the backup activation is transitive along the priorities based on Theorem 1. The correctness and finite convergence of our approach to handle multi-failure scenarios is proved in Appendix C. One may notice that checking failures on a VP and getting the denied message may take some time if the VP set is huge. Indeed, given hardness of the arbitrary-failure problem, our approach may suffer a long recovery time. But we emphasize that it happens only in multi-failure case, using our primitives to locally recover 1 failure is super quick.

Multi-failure example. For the example in Figure 9, to recover from arbitrary failures, additional dependencies (green part) are added in the DOCs. Note that $C_2.\texttt{gate}$ is computed in the same fashion as in the single-failure algorithm. For a flow from A, assume both BC and EC fail. First, B_1 will pull A_1 to be activated, and then A_1 pulls E_0. Because of the EB failure, E_0° is sent back to A, and A_2 will be used to recover routing. For a flow from C, assume both CD and CB are failed. C_1 first pulls B_1 but is denied, so B_1° will eventually wake up C_2 which reroutes the traffic to E. The contingent DOCs in E and F are easy to obtain.

5 EXTENSIONS

5.1 Speeding up configuration by pre-starting operations.

Our distributed datapath primitive bypasses the many-round communication with the SDN controller, so it addresses the bottleneck of control channel and reduces controller orchestrating latency. However, it may be argued that the latency of applying some of the data plane operations in switches, *e.g.*, insert/modify rules in TCAM, is also a bottleneck. The major cause of inconsistency is the completion of configuration instead of the start. Therefore, to achieve shorter configuration time, we consider *whether we can start applying an operation first, but somehow make it not to take effect until its* gate *is satisfied.*

To eliminate this bottleneck, we propose an enhanced method to pre-start Insert and Modify operations in switches. If a switch wants to execute a modification operation, while still waiting for the `gate` being satisfied, the modification operation can be split into an operation of inserting the new rule with lower priority and an operation of deleting the old one. The Insert operation, which takes longer time, can be performed first without waiting for any conditions, *i.e.*, the dependency fields in its container is empty, but it won't take effect after completion due to the lower priority. The Delete operation inherits the `gate` and `release` of the original Modify, so once the dependency condition is satisfied, the old rule is deleted and the new one takes effect, which is equivalent to the original Modify result, but faster. An Insert operation can be pre-started in a similar fashion.

5.2 Stateful Programming

In stateful programming, some programs may generate operations to read and write to global, persistent variables [2, 13, 31]. We extend our primitive to support operations on states, so that stateful programs can be implemented in a distributed way, which helps the compiler reconfigure the data plane with higher degree of parallelism.

Algorithm 4 translates the three basic execution models (sequential, parallel and atomic) into our primitive, where the `release` field is omitted for space constraints. $\{o_i\}$ denotes a set of operations on states in the program. Besides the three basic models that can be expressed by simple semantics, there are complicated read/write dependencies related to multi-state or multi-variable. We complete these dependencies in Algorithm 4, where $v.write$ and $v.read$ represent operation set that writes and reads to variable v, respectively. The algorithm is schematic, and various-variable cases can be extended from it. To avoid write conflicts (there is no read conflict), a variable can only be written by one operation at a time. Here we insert ! operator to block each other. If a variable can be written by multiple operations, any one of them can generate an outcome for the read operations. If an operation reads multiple variables' merging results (*e.g.*, $(v_1 + v_2).read$), it has to wait for all the write operations.

Algorithm 4 StatefulDependency$\{o_i\}$

Basic Execution Models:

Sequential Model: $\forall o_i, o_j, o_i \rightarrow o_j$
1: o_j.gate $\leftarrow o_j$.gate&o_i
Parallel Model: $o_i, o_j \in ParallelBundle$
2: continue
Atomic Model: $o_1, o_2, ..., o_N \in ATM(AtomicBundle)$
3: **if** $\exists o_i \in ATM, o' \notin ATM, o' \rightarrow o_i$ **then**
4: $\forall o_k \in ATM$, o_k.gate $\leftarrow o_k$.gate&o'
5: **else if** $\exists o_j \in ATM, o'' \notin ATM, o_j \rightarrow o''$ **then**
6: o''.gate $\leftarrow o''$.gate&$(o_1 \& o_2 \& ... \& o_N)$
7: **end if**

Complicated Read/write Dependencies:

1: **if** $o_i, o_j \in v_1.write$ **then**
// **Write Conflict**
2: o_i.gate $\leftarrow o_i$.gate&$(!o_j)$
3: o_j.gate $\leftarrow o_j$.gate&$(!o_i)$
4: **if** $o_k \in v_1.read$ **then**
// **Multi-write**
5: o_k.gate $\leftarrow o_k$.gate&$(o_i || o_j)$
6: **end if**
7: **else if** $o_i \in v_1.write, o_j \in v_2.write, o_k \in (v_1 + v_2).read$ **then**
// **Multi-read**
8: o_k.gate $\leftarrow o_k$.gate&$(o_i \& o_j)$
9: **end if**

6 IMPLEMENTATION

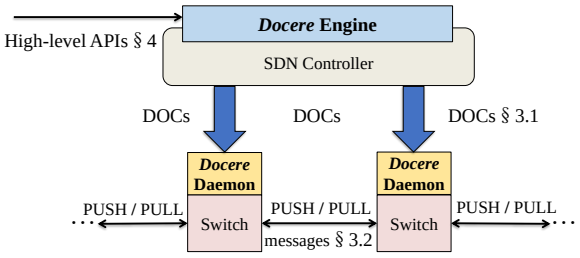


Figure 11: Prototype implementation.

We implement a system called *Docere* to realize our primitive. The key components and workflow are shown in Figure 11. When a user wants to (re)configure the data plane, she will submit a high-level configuration to the *Docere* engine via the waypoint API set_target_configuration. A set of operations encapsulated in DOCs are automatically computed and sent to *Docere* daemons through the SDN controller. The engine is able to handle interactions between DOCs at different times, so the user can constantly submit other configurations without worrying about the previous progress. She can also perform contingent configurations via the API set_contingent_configuration.

The *Docere* daemon is running at a custom software agent on each switch. When receiving basic OpenFlow messages, such as Handshakes or Flow Monitoring, the daemon transparently forwards them to the switch. For DOCs, the daemon creates a log for each of them, which records the unique ID and execution status. The daemon monitors and deals with the DOCs in parallel. Once receiving a PUSH/PULL message, the daemon modifies related DOCs' logs, and applies the operations on the switch according to the semantics of our primitive.

The engine realizing two APIs consists of 3000+ lines of Python code, and the daemon is written in 600+ lines of Python code. We use OpenDaylight 3.0.3 [24] as the OpenFlow controller and switches which support OpenFlow 1.3 [25]. Since OpenFlow does not include DOCs, the communication protocol between the controller and the daemons is implemented by an extension version of OpenFlow. There is no change for uplink, but the data in `gate` and `realse` is carried in the payload of OpenFlow messages for downlink. The communication between daemons is via UDP messages.

Table 2: Configuration time in Figure 2 example.

Approach	Time (ms)
Dionysus (centralized design)	303(±51)
Docere (distributed primitive)	217(±49)

7 TESTBED EVALUATION

We now conduct two testbed experiments that perform the configuration examples in our paper. The first one is an individual update configuration as in Figure 2 to show the benefit of distributed execution by our primitive. The second includes two continuous configurations as in § 4.1.5 to demonstrate the optimization effect of our high-level API.

7.1 Experimental Methodology

In our testbed, we deploy 5 Open vSwitches instantiated at different computers. A daemon running on each computer coordinates with other daemons to control operation execution and logs operation confirmation time. We run an SDN controller at a server directly connected with each switch. All link conditions are set to be the same. We compare *Docere* with *Dionysus* [12], a centralized approach that adopts the basic primitive confirm/do-next to realize consistent updates.

7.2 Experiment 1: Individual Configuration

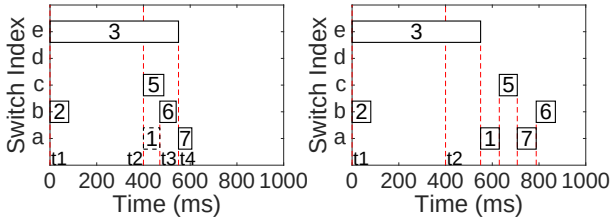
In our first experiment, we perform a network update as shown in Figure 2. We measure the configuration time of accomplishing the 8 operations that realize the update. The time is defined as the interval between when the the controller starts to send out the first operation until when the last operation is confirmed in the daemon's log. In *Dionysus*, the controller sends the 8 operations step by step according to the confirmations from the daemons.

Our distributed primitive outperforms the centralized design. We repeat this experiment 100 times and compute the mean completion time as shown in Table 2. The update configuration is completed 39.6% faster by adopting our primitive. This gain comes from the fact that our primitive allows a confirmation to be sent directly to next switch instead of relayed by the controller, hence time is reduced from round-trip time to one-way time. In particular, the confirmation from A_{F1} or A_{F2} to A_{F3} is sent within a same switch, which saves more time. It should be noted that we have adopted the best conditions for *Dionysus*. In a practical network, the controller is not guaranteed to have a direct connection with every switch, and the link condition between controller to a switch is always worse than that between two adjacent switches.

7.3 Experiment 2: Continuous Configuration

Our second experiment implements another topology as in Figure 7. Starting from an empty network state without any flows, we continuously set two configurations at $t1$ and $t2$ as the example in § 4.1.5. We continue to compare *Docere* with *Dionysus*, which has to handle different configurations separately. In order to emulate the execution progress in our example, we let switch e be a straggler switch and inject 500 *ms* execution latency on it.

Figure 12 shows the time series of this experiment for *Docere* and *Dionysus*. The x-axis is the time, and the y-axis is the switch index. A rectangle labeled by the DOC ID represents an operation applied on a switch (y-axis) for some period (x-axis). The period begins at the time when the operation can be applied, and ends at the time when the daemon receives its confirmation. So the confirmation/PUSH transmission and computation time is counted in the period. In this way, there will be no time gap between two neighboring rectangles, making it easier to visualize the time series graph.



(a) ***Docere*** (b) ***Dionysus***

Figure 12: Time series for the experiment of the example in Figure 7.

From the figure we can see, for completing the same configurations, our approach takes only 608 *ms*, while *Dionysus* takes 880 *ms*. Reducing the controller involvements demonstrated in Experiment 1 is indeed beneficial, but this is not essential due to the execution straggler. The major gain in this experiment is from two aspect of our distributed primitive: pipelining and short-cut.

Pipelining gain. As discussed in § 4.1.4, some new operations are independent from previous configurations, so they can be executed without waiting for previous confirmations, like *DOC*5. Therefore, in *Docere* as shown in Figure 12(a), *DOC*5 starts execution as soon as C_2 arrives at $t2$; *DOC*3 and *DOC*5 for different configurations are processed in parallel. But in *Dionysus* as shown in Figure 12(b), *DOC*5 is executed after all three operations in $C1$ are finished. Our primitive can guarantee transition safety with respect to cross-dependencies. As in Figure 12(a), *DOC*7 depending on *DOC*3 is executed after the latter's completion at $t4$.

Short-cut gain. For the short-cut in this experiment, DOC1 can be replaced by *DOC*4 and become an equivalent operation *DOC*7. As shown in Figure 12 (a), *DOC*4 and *DOC*7 learn that *DOC*1 is not applied by sending PULL messages upon arrival. Then *DOC*4 releases the downward dependency of *DOC*6 to start its execution at time $t3$. In the mean time, once receiving a PULL, the useless *DOC*1 is deleted accordingly. However, as in Figure 12(b), without awarding the short cut, all operations have to be applied, including the useless *DOC*1. Our approach can discover short-cuts and direct the data plane to the optimized paths using the distributed primitive, thereby reducing configuration time.

8 LARGE-SCALE SIMULATIONS

We now turn to large-scale simulations to show that our simple primitive can significantly speed up updates and local failover, while incurring small overhead.

8.1 Simulation Methodology

We run all simulations on a dedicated server. We evaluate configuration performance on real topologies in both WAN and data center TE scenarios. For WAN, we choose 5 topologies from the Topology Zoo [34] that interconnect $O(50)$ sites, where link capacities are set between 1 and 100 *Gbps*. For data center, we emulate a 3-tier datacenter network topology with 4:1 oversubscription. The topology contains 64 servers, whereby each edge link is of 1*Gbps* capacity, and aggregated link is of 10*Gbps* capacity. 800-1500 flows are generated by selecting nonadjacent source and destination switch pairs at random and assigning them a traffic volume generated according to the gravity model [29]. We generate the updates by simulating link failures. In the simulations, *Docere* is compared with *Dionysus* [12] and *ez-Segway* [23], a decentralized solution with distributed computing at each switch.

8.2 Simulation Results

Update completion time. We measure the average completion time of 60 update configurations in both WAN and data center TE scenarios. Here we use a static model, where each update is individually conducted and measured. We break down the overall time by the amount of computation at the controller and the execution in the data plane. Since *Dionysus* dynamically computes next step according to execution progress, its computation time is accumulated from each step. In *ez-Segway* and our approach, the controller computes only one time, so the computation time at switches is counted in the execution part. Figure 13 shows the 50^{th}, 90^{th}, and 99^{th} percentile update time.

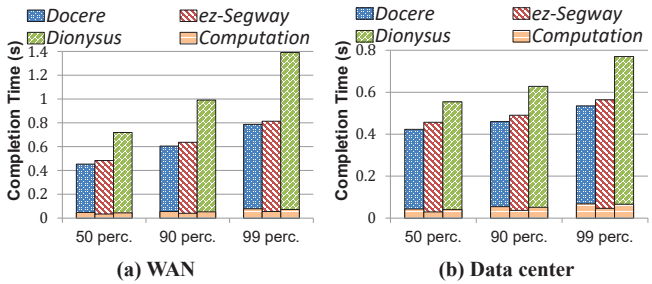


Figure 13: Update completion time which is broken down by the amount of computation and execution.

First, we observe that computing update configuration is not a bottleneck across all schemes. *ez-Segway* achieves the smallest computation time because the controller only computes a dependency graph and scheduling computation is offloaded by switches. Our approach requires the longest computation time because scheduling plans are precomputed in the engine to encode in the DOCs.

Second, we observe that *Docere* achieves the lowest execution time. *Dionysus* utilizes centralized orchestration, where the coordinating time is a sum of many-round of communication between the controller and switches. Hence *Dionysus* takes the longest execution time. In contrast, most of the coordination in *ez-Segway* and our approach is within one switch or between adjacent switches, therefore reducing the total execution time. Our approach is faster than *ez-Segway* because scheduling plans are computed in advance and our simple logic matching at switches is considerably simpler than the distributed scheduler in *ez-Segway* which computes scheduling based on a number of resource constraints.

Overall, *Docere* outperforms *Dionysus* and *ez-Segway* in both WAN and data center TE scenarios, as shown in Figure 13. For WAN TE, the 50^{th}, 90^{th}, and 99^{th} of *Docere* is 37.1%, 39.0%, and 43.4% faster than *Dionysus*, respectively, and 6.2%, 5.0%, and 3.3% faster than *ez-Segway*, respectively. For data center TE, the corresponding numbers are 23.7%, 26.8%, and 30.5% faster than *Dionysus*, respectively, and 7.3%, 6.2%, and 5.2% faster than *ez-Segway*, respectively. The WAN topologies are more complex than the data center, resulting in longer completion time. *Docere* performs better distributed execution based on the simple primitive, thereby improving individual update configurations.

Control Message Overhead. In the simulations, we also record control message overhead for each update configuration. Figure 14 shows the average overhead in both WAN and data center TE scenarios. The control message overhead in *Dionysus* includes only configuration-related messages (FLOW_MOD) and operation confirmations (asynchronous messages) [25]. In *ez-Segway*, besides the basic messages, the controller has to send every switch a dependency graph that reflects the partial network topology and resources. In *Docere*, the additional overhead comes from the Boolean logic (`gate` and `release`) in DOCs. In addition, in both *ez-Segway* and *Docere*, control messages are transmitted between switches for coordination (PUSH/PULL messages in *Docere*). For fairness, we include this part in, though this coordination overhead is extremely small.

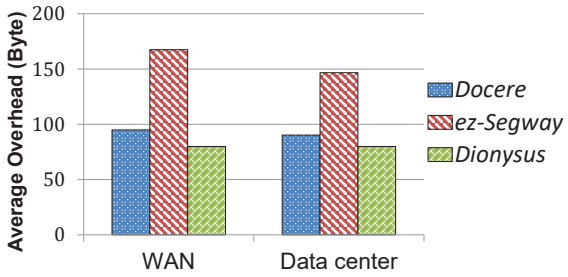


Figure 14: Control message overhead for different approaches.

From Figure 14 we can see, *Dionysus* utilizes only standard OpenFlow control messages, therefore has the lowest overhead. In *ez-Segway*, every switch needs partial knowledge of the whole network to conduct distributed computation,

which generates much additional overhead. In particular, as shown in (a), since the WAN topology is relatively large, more network information are sent out more times, which leads to 109.6% more overhead than *Dionysus*. On the contrary, our super simple DOCs carry only necessary dependency information, so the overhead is much lower than *ez-Segway*. Furthermore, our approach performs close to *Dionysus*, with at most 18.8% more overhead (13.0% more in data center TE). Therefore, we conclude *our primitive significantly speed up data plane configuration while incurring small overhead.*

Failover time. To show the benefit of our primitive on local failover, we simulate different failure scenarios in WAN TE and compare *Docere* with the controller-involved alternatives. Here we consider one destination for simplicity. First, we compute the contingent DOCs via the contingency API in § 4.2. Next, we randomly fail a certain number of links, and then measure the recovery time for re-routable flows as shown in Figure 15. In *Docere*, these victim flows are recovered locally by contingent DOCs, while in *Dionysus* and *ez-Segway*, the controller computes new routes to update the network. For fairness, their computing time is not counted in the recovery. From Figure 15 we can observe, for 1-failure, our primitive encoded with backup-rule dependencies can significantly decrease the failover time, by 39.0% at most. For multi-failure cases, our algorithm relies on a modified Pull mode that checks the connectivity of each "virtual path". As analyzed in § 4.2.2, the recovery may suffer long convergence time. Therefore, the gain declines. But our local recovery still achieves comparable failover time as the other two centralized recovery approaches.

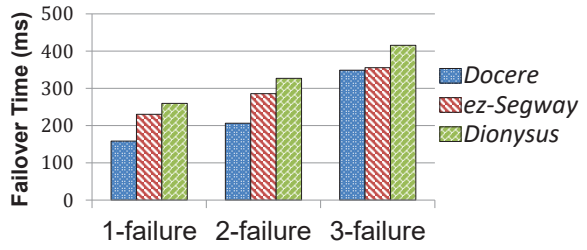


Figure 15: Failover time on different numbers of failures.

9 RELATED WORK

Datapath configuration primitives in SDN. To ensure operation dependencies in SDN, OpenFlow supports some advanced primitives that control ordered-processing [15, 25, 30]. Barrier messages prevent later operations from being applied first, but other subsequent operations will be unnecessarily affected and delayed. An ordered bundle makes operations applied in a strict order, but can not express unordered dependencies and has to lock switch states. In addition, they are used just for switch-wide operations. Our primitives enable distributed datapath configurations in a wide range of settings, thereby improving scalability and efficiency.

Centralized updates. The consistent network updates problem has been widely studied in the literature [7, 12, 14, 19, 20, 27, 28]. Like Dionysus, this line of work relies on centralized orchestration, where the SDN controller decomposes an update into steps and schedule execution step by step. The requirement of many-round of communication between the controller and switches leads to high configuration latency and low reliability, which are addressed by our distributed primitives.

De-centralized update. To avoid centralized orchestration by the controller, ez-Segway is proposed to address the network update problem in a de-centralized manner [23]. Relying on a similar dependency graph and scheduling algorithms as Dionysus, ez-Segway delegates some computation tasks from the controller to switches. The switches receive partial knowledge of the network, some of which may be redundant, and require some computation capabilities. In contrast, our primitives are much simpler and more powerful. Compact but completed information is efficiently encoded in containers, resulting in much lower overhead and computation complexity at switches. As another decentralized primitive, a timed-update is proposed, which uses synchronized clocks as a way to coordinate the updates [21, 22, 40]. However, due to imprecise clock synchronization and time prediction, the consistency and efficiency of network updates are not guaranteed [23]. Our primitives can have a wide range of extension, and an operation gate may also be considered to support time variant in the future work.

Failure Recovery. Prior art on failure recovery in SDN preinstalls backup rules in group tables and relies on Openflow local fast failover mechanisms to take effect [3, 4, 33, 41]. Tunnel-based approaches need considerable flow rule and tag complexity for exponentially many failure cases [3, 18, 36]. In addition, remote failures are transmitted to the ingress switch either by flooding, leading to high overhead, or via the controller to relay, which increases recovery time. Sentinel proposes to re-route victim flows from the failing switch rather than the ingress switch, leading to faster recovery but also longer packet lengthes [41]. These tunnel-based approaches are complementary to *Docere*. Our novelty lies in using transitive dependencies to locally guide contingent operations to take effect on compact flow rules, which to our knowledge has not been done before.

10 CONCLUSION

This paper presents Datapath Operation Container (DOC), a distributed configuration primitive for SDNs that supports concurrency control among data plane devices, *e.g.*, switches, without incurring high cost on the controller. The DOCs can be executed on the switches following the dependencies among different data plane operations, while ensuring data plane consistency. We also design two APIs to transform high-level network configurations to data plane operations and the DOCs. Evaluation results show that our method significantly reduces the configuration completion time and incurs small message overhead.

REFERENCES

[1] Anoymous Technical Report. https://github.com/docere2018/DOC.

[2] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

[3] M. Borokhovich, L. Schiff, and S. Schmid. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *Proceedings of the third workshop on Hot topics in software defined networking*.

[4] 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*.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3).

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41.

[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 (IFIP Networking) and Workshops, 2016*.

[8] J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking (TON)*, 1(1).

[9] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*.

[10] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43.

[11] 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*, volume 43.

[12] 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 Computer Communication Review*, volume 44.

[13] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, pages 103–115, 2015.

[14] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*.

[15] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan. Towards efficient implementation of packet classifiers in sdn/openflow. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*.

[16] 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*, 103(1).

[17] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4).

[18] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *ACM SIGCOMM Computer Communication Review*, volume 44.

[19] 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 Computer Communication Review*, volume 43.

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

[21] T. Mizrahi and Y. Moses. Software defined networks: It's about time. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*.

[22] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*.

[23] T. D. Nguyen, M. Chiesa, and M. Canini. Decentralized consistent updates in sdn. In *Proceedings of the Symposium on SDN Research*.

[24] OpenDaylight. http://www.opendaylight.org.

[25] B. Pfaff, B. Lantz, B. Heller, et al. Openflow switch specification, version 1.3. 0. *Open Networking Foundation*, 2012.

[26] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simplefying middlebox policy enforcement using sdn. In *ACM SIGCOMM computer communication review*, volume 43.

[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4).

[28] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*.

[29] M. Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35(5).

[30] S. A. Shah, J. Faiz, M. Farooq, A. Shafi, and S. A. Mehdi. An architectural evaluation of sdn controllers. In *Communications (ICC), 2013 IEEE International Conference on*.

[31] M. Shahbaz and N. Feamster. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*.

[32] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28. ACM, 2016.

[33] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*.

[34] The Internet Topology Zoo. http://www.topology-zoo.org.

[35] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: simplifying sdn programming using algorithmic policies. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 87–98. ACM, 2013.

[36] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang. R3: resilient routing reconfiguration. In *ACM SIGCOMM Computer Communication Review*, volume 40.

[37] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 51(2).

[38] W. T. Zaumen and J. Garcia-Luna-Aceves. Loop-free multipath routing using generalized diffusing computations. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3.

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

[40] 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*, 35(11).

[41] 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), 2016 IEEE 24th International Conference on*.

Appendices

A BENEFIT OF USING DOCS

Using DOCs for network updates. The main advantage of using DOCs for network updates is saving data plane configuration time. We compare the controller-centric update in existing work with the concurrent update using the DOC Push mode. The completion time in the controller-centric update can be estimated as

$$T_C = \frac{RTT_{CS}}{2} + RTT_{CS} * (L-1) + t * L \quad (1)$$

and the the completion time in a distributed update is

$$T_D = \frac{RTT_{CS}}{2} + \frac{RTT_{SS}}{2} * (L-1) + t * L \quad (2)$$

where RTT_{CS} and RTT_{SS} are the round-trip time of controller-to-switch and switch-to-switch, respectively. L is the length of the longest path in the ODG, and T is the average processing time of an operation.

Suppose network latencies are much bigger than the average processing time of an operation. Then the fraction of the completion time that using DOCs can reduce equals to $\frac{T_C - T_D}{T_C} \approx \frac{RTT_{CS} - RTT_{SS}/2}{RTT_{CS}}$. Note that most of PUSH messages are transmitted between adjacent switches or within one switch, so $RTT_{SS} < RTT_{CS}$, resulting in a huge benefit gain. Even if we consider RTT_{CS} approximates RTT_{SS}, we can still save 50% of the completion time of data plane updates.

In practical, the processing time t sometimes becomes large. In this situation using DOCs still achieves time efficiency, which will be discussed in §5.1.

Using DOCs for local failover. Unlike existing work, such as Sentinel [], that relies on the controller to inform other switches of failures, *Docere* can complete failover operations in the data plane without the help of the controller, while ensuring consistency. Using the similar analysis model for network updates, the failover completion time is reduced by $\frac{RTT\_CS - RTT\_SS}{RTT\_CS}$ in *Docere* (details in the extended version). Here the numerator is different from the Push mode, because in the Pull mode, inter-switch message transmission needs one RTT (a PULL and a PUSH).

Docere may also reduce switch memory space required to store backup rules in each switch. In Sentinel [], each switch needs to store backup rules for its neighboring links as well as other backup tunnels that traverse it. In *Docere*, the deferred dependencies are reflected by the primitives. Hence if we consider only 1-failure case, the space complexity can be reduced from $O(K)$ to $O(1)$, where K is the number of backup tunnels that use the switch. For more than one failures, Sentinel does not work because it is impossible to map exponentially many possibilities to backup tunnels. On the other hand *Docere* can handle arbitrary numbers of failures, and the space complexity is only $O(d)$, where d is the number of immediate neighbors of the switch. In *Docere*, all failure possibilities can be reflected by the primitives for the deferred dependencies.

B PROOF OF THEOREM 1

Sufficiency: Since $\forall i' < i$, $X_{i'}$ can not be used, the switch X has to use the current highest-priority rule, which is X_i.
Necessity: If $\exists i' < i, s.t.\ X_{i'}$ can be used, which means there exists at least a VP $VP_{i'}$ to the destination, then the current highest-priority rule which can be used (can reach the destination) is $X_{i'}$. As a result, X_i will not be used.

Combined the sufficiency and necessity, we achieve Theorem 1. Q.E.D.

C HANDLING MULTIPLE FAILURES

Consider a given switch X and its backup rules $X_i, i \in [0, K-1]$. It's easy to see that X_1 will be used if and only if X's original flow rule X_0 can't be used. According to Algorithm 3, $\forall i \in [2, K-1]$, X_i will be used if and only if X_{i-1}'s nexthop can't be used, which means X_{i-1}'s VP can't be used. By using mathematical induction, we can get that Algorithm 3 satisfies Therom 1. To illustrate the correctness of our algorithm, we give the therom below.

Theorem 2. *If there is a failure can't be recovered by alogrithm 3,i.e., there exists a source S that doesn't have a path to the destination D after processing alogrithm 3, then S and D is unreachable in the network, which means alogrithm 3 can recover arbitrary failures if it can be recovered.*

We prove therom 2 by contradiction. Suppose there is still a path from S to D, then each switch on that path has at least a VP that can be used. According to Therom 1, there is at least a backup rule can be used and found by Algorithm 3 which is contrary to the assumption. Q.E.D.

Additionally, according to Therom 1, each switch will only use the highest-priority backup rule whose VP can reach the destination. And when a lower-priority rule has been used, the higer one can't be used again. So Algorithm 3 will finally converge to a unified backup solution in finite steps. Above all, the correctness and convergence of our algorithm has been proved.