Snowcap: Synthesizing Network-Wide Configuration Updates

Tibor Schneider
ETH Zurich, Switzerland
sctibor@ethz.ch

Rüdiger Birkner
ETH Zurich, Switzerland
rbirkner@ethz.ch

Laurent Vanbever
ETH Zurich, Switzerland
lvanbever@ethz.ch

ABSTRACT

Large-scale reconfiguration campaigns tend to be nerve-racking for network operators as they can lead to significant network downtimes, decreased performance, and policy violations. Unfortunately, existing reconfiguration frameworks often fall short in practice as they either only support a small set of reconfiguration scenarios or simply do not scale.

We address these problems with Snowcap, the first network reconfiguration framework which can synthesize configuration updates that comply with arbitrary hard and soft specifications, and involve arbitrary routing protocols. Our key contribution is an efficient search procedure which leverages counter-examples to efficiently navigate the space of configuration updates. Given a reconfiguration ordering which violates the desired specifications, our algorithm automatically identifies the problematic commands so that it can avoid this particular order in the next iteration.

We fully implemented Snowcap and extensively evaluated its scalability and effectiveness on real-world topologies and typical, large-scale reconfiguration scenarios. Even for large topologies, Snowcap finds a valid reconfiguration ordering with minimal side-effects (i.e., traffic shifts) within a few seconds at most.

CCS CONCEPTS

• **Networks** → **Network management**; **Network reliability**; *Network simulations*; • **Theory of computation** → Modal and temporal logics; Logic and verification;

KEYWORDS

Network analysis, Configuration, Migration

ACM Reference Format:
Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. **Snowcap: Synthesizing Network-Wide Configuration Updates. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3452296.3472915**

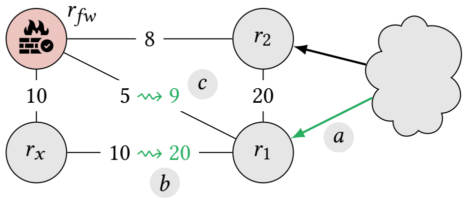


Figure 1: This scenario consists of adding an eBGP session a and adapting two link weights: b and c, while: *(i)* ensuring traffic from r_x always flows via r_{fw}; and *(ii)* minimizing traffic shifts. Two orderings achieve both goals: b c a and c b a.

1 INTRODUCTION

Network operators reconfigure their network literally every day [17, 27, 39, 40, 45]. In a Tier-1 ISP for example, network operators modify their BGP configurations up to ≈20 times per day on average [45].

While most of these reconfigurations are small (e.g., adding a new BGP session), a non-negligible fraction is large-scale. Common examples include switching routing protocols (e.g., from OSPF to IS-IS [19]), adopting a more scalable routing organization (e.g., route reflection [37]), or absorbing another network [23]. As an illustration, Google's data center networks have undergone no less than 5 large-scale configuration changes within the last decade [36].

Small or large, network reconfigurations consist in modifying the configuration of one or more network devices. Due to the distributed nature of networks, applying all reconfiguration commands atomically—on all devices—is impossible. Instead, the network necessarily transitions through a series of intermediate configurations, each of which inducing possibly distinct routing and forwarding states. Doing so the network might temporarily violate important invariants or suffer from performance drops *even if* both the initial and the final configuration are perfectly correct and verified.

While such reconfiguration issues are transient, they are also disruptive. Alibaba revealed that the majority of their network incidents (56%) resulted from operators updating configurations [29]. Our case studies (§2) confirm this: even when following best practices, reconfiguring a network often causes numerous forwarding anomalies (e.g., loops or blackholes) and unnecessary traffic shifts.

Take the scenario in Fig. 1 as an example. The operators wish to increase their capacity by establishing a new eBGP session on r_1 while, for security reasons, ensuring traffic from r_x keeps flowing through r_{fw}. For performance reasons, they also want to avoid any unnecessary traffic shifts during the reconfiguration. The first requirement is *hard*: it has to be maintained throughout the reconfiguration. In contrast, the second requirement is *soft*: it should be

SIGCOMM '21, August 23–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08...$15.00
https://doi.org/10.1145/3452296.3472915

violated as little as possible. The scenario involves applying 3 commands: adding the BGP session *a* and adapting two link weights *b* and *c*.[1] For simplicity, we assume that there are no failures.

In this example, both the initial *and* the final configurations comply with the hard requirement. This is however not the case for most intermediate forwarding states. Indeed, applying *a* first makes traffic from r_x bypass the firewall as r_x would then forward traffic to r_1 via its direct link. Applying *b* before *a* avoids this bypass. The same holds for traffic shifts: most intermediate states exhibit some. For example, applying *a* before *c* leads to a shift in which the traffic from r_x (transiently) leaves via r_1 instead of r_2.

Only two orderings *b c a* and *c b a* out of the 3! = 6 possible comply with the hard requirement while optimizing for the soft one. Finding those manually is generally hard.

Given its relevance, researchers have developed multiple tools to seamlessly reconfigure networks. We can broadly classify them in two categories depending on whether they modify the configurations *in-place* (one command at a time, as in our example above) or rely on a technique commonly known as *Ships-In-The-Night* (SITN), where routers are running multiple configurations in parallel.

While useful, both categories suffer from limitations in terms of *(i)* the reconfiguration scenarios they can support; *(ii)* the guarantees they can provide; and *(iii)* the overhead they impose. In particular, while "in-place" reconfiguration tools do not impose any extra overhead on the network, they only support a restricted set of scenarios and properties such as preserving reachability when changing an IGP link weight [13] or when removing a BGP session [12]. A bigger problem though is that their restricted model makes them unsafe to use in multi-protocol environments [42]. In contrast, SITN-based reconfiguration tools can support a larger set of scenarios and properties, at the price of duplicating the routing and forwarding table on all routers [43–45]. Besides, not all routers support running duplicated control planes in the first place [9].

Several works address the problem of safely updating Software-Defined Networks (SDN) [24, 31, 32] from one forwarding state to another. These techniques, however, do not apply to distributed routing protocols found in the vast majority of networks [6].

A fundamental research question is still open: *Is it possible to automatically and safely reconfigure a network running arbitrary protocols without imposing any extra overhead on the network?*

Snowcap. We answer positively and present Snowcap, a reconfiguration framework which can synthesize and deploy safe configuration updates, for arbitrary protocols and arbitrary correctness properties. Given *(i)* the initial and the final configurations; and *(ii)* hard and soft specifications (expressed as a linear temporal logic (LTL) formula and as an objective function, respectively), Snowcap automatically generates an ordering of the reconfiguration commands which satisfies the hard specifications, while optimizing for the soft ones. Snowcap's runtime controller then applies these commands one-by-one to the live network, appropriately waiting for network convergence in-between them. Doing so enables Snowcap to tightly control the intermediate states.

Snowcap's specification language supports path-based properties and resilience to link failure. Path-based properties limit forwarding to only use paths matching a regular expression. This allows, for example, enforcing security properties, such as requiring traffic to pass through a middlebox. In addition, Snowcap allows requiring that properties are satisfied under any link failures. This makes Snowcap practical for networks which have to provide high-percentile reachability guarantees even during reconfigurations and link failures (e.g., five nines).

Key challenges and insights. The main technical challenge we face in designing Snowcap is to efficiently navigate the space of possible reconfiguration orderings. This is hard as, besides its size (there are n! orderings given n commands to apply), the search space is typically sparse (very few orderings adhere to the specification, cf. our example above). Taken together, these characteristics make simple search strategies like random sampling extremely inefficient.

We address that by designing an efficient counter-example-guided search procedure. More specifically, Snowcap greedily builds a reconfiguration ordering leveraging the hard and soft specifications to guide the search. Upon encountering an ordering which violates the hard constraints, Snowcap uses this counter-example to identify the (minimal) ordering constraint (which we call *dependencies*). Snowcap then restarts its exploration taking these constraints into account, effectively pruning the search space in a divide-and-conquer fashion. As we show, this counter-example-guided approach tends to work particularly well in practice as it neutrally adapts to different reconfiguration scenarios.

System & results. We demonstrate a prototype of Snowcap[2] which currently supports Border Gateway Protocol (BGP) and link-state Interior Gateway Protocols (IGPs), and can easily be extended to other protocols. Our prototype not only synthesizes a "good" reconfiguration plan, but also applies it to the live network automatically. Our evaluation shows that Snowcap scales to large network topologies and reconfiguration scenarios: it finds compliant reconfiguration orderings within a few seconds. Snowcap also finds significantly more optimized orderings than the baselines.

Contributions. In summary, our main contributions are:

- A framework which poses safe reconfiguration as an optimization problem with hard and soft constraints.
- A specification language based on LTL.
- A generic search procedure which uses counter-examples to efficiently find optimized reconfiguration orderings.
- An implementation of our approach, together with an evaluation on real network topologies and scenarios.

Limitations. Snowcap guarantees that *all* properties are satisfied when the network has converged. However, it cannot always guarantee that during convergence as transient anomalies (like forwarding loops and blackholes) are inherently part of the convergence process of distributed protocols and thus outside of Snowcap's control. These effects can (and do) occur even during normal operation of the network. While this means that Snowcap cannot guarantee general reachability properties in-between updates, we prove that it *can* guarantee path-based properties *during convergence.*

[1] Adapting the weights allows operators to preserve r_x's original paths in the final configuration, while allowing other routers to use the new session. Note that lowering the local preference on the new session would prevent it from being used at all, nullifying the goal of increasing capacity.

[2] Available at https://github.com/nsg-ethz/snowcap

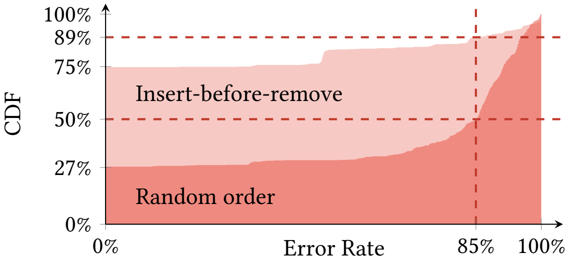


Figure 2: The reconfiguration plans based on both random order and best practice ("insert-before-remove") often violate reachability when moving from an iBGP full mesh to a route reflector configuration.

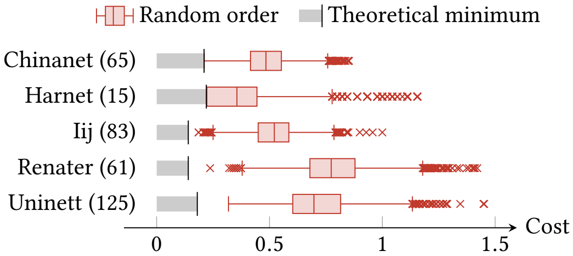


Figure 3: Different reconfiguration usually introduce unnecessary traffic shifts The y axis lists 5 different network topologies from Topology Zoo. The x axis shows the cost, i.e., number of traffic shifts (§5.3).

2 MOTIVATION

We motivate that careful planning is necessary to reconfigure a network safely and with minimal side-effects by considering two common and typical reconfiguration scenarios, which are given priority in the popular "Network Mergers and Migrations" book for Junos [23]. First, we show that forwarding anomalies (e.g., forwarding loops and blackholes) can easily occur during an iBGP reconfiguration, even when following best practices. Then, we show the difficulty of finding a reconfiguration with minimal side-effects (e.g., traffic shifts) in a network acquisition scenario.

2.1 Case Study: iBGP Reconfiguration

We study the prevalence of forwarding anomalies by reconfiguring 80 networks from the Topology Zoo collection [28] from an iBGP full mesh to a route reflector topology.[3]

For every topology, we randomly generate 10 sets of IGP weights and choose the router with the highest degree as the designated route reflector (following best practices [20]). We consider two reconfiguration strategies: First, we simulate a careless operator blindly reconfiguring the network by randomly choosing the order of routers in which to apply the changes. Second, we simulate an operator following the recommended reconfiguration strategy for this scenario: "insert-before-remove" [23]. That is, we (randomly) add all the iBGP route reflector sessions before removing the old ones. For each topology, set of IGP weights, and reconfiguration strategy, we simulate 10 000 different orderings.

Fig. 2 shows a CDF of the percentage of orderings that led to a blackhole or forwarding loop (i.e., the error rate) across our experiments. For 50% of the networks, we see that a "careless" operator introduced forwarding anomalies 85% of the time. In addition, a "careless" operator would create at least one forwarding anomaly in 73% of the networks (only 27% of the networks exhibited no issues in all orderings). In contrast, we see that even "best practice" operators still introduced forwarding anomalies in 25% of the topologies, and for more than 10% of the topologies, their error rate is over 85%. "Best practice" is arguably better than the careless operator, but still far from zero—hence the need for Snowcap, which performs all these reconfigurations without disruptions.

2.2 Case Study: Network Acquisition

In addition to forwarding anomalies, a reconfiguration can also lead to unnecessary traffic shifts, causing congestion or jitter. It is therefore crucial to minimize their occurrences whenever possible.

We study the prevalence of traffic shifts in a merging scenario. More specifically, we take 42 networks from the Topology Zoo collection [28][4] which we randomly partition in two distinct connected components, and assign one router in each partition to be a reflector. We then merge the two networks, during which we add several links between the two networks, generate an iBGP session between the two route reflectors, and rescale all link weights in one network to match the other's.

Fig. 3 compares the number of traffic shifts (cost) triggered during the entire reconfiguration process (cf. §5.3) between a random order to the theoretical minimum. We compute the ideal costs by assuming that the entire reconfiguration could be performed in a single step. Again, we see that random reconfigurations introduce far more traffic shifts than the theoretical minimum—hence, justifying once more the need for a tool like Snowcap, which merges the two networks with significantly lower costs (cf. §6.2). The extended figure with all tested topologies can be found in App. A.

3 OVERVIEW

We now provide an overview of Snowcap and how it computes reconfiguration orderings using a running example.

Sequence notation. Throughout this paper, we denote an ordered sequence of commands a, b, c, and d as $(a\,b\,c\,d)$. When simulating such a sequence, we see whether it satisfies or violates the hard specification ϕ. When it satisfies the specification, we color it green: $(d\,c\,a\,b) \models \phi$. When it violates the specification, we split it in a green and a red part: $(a\,b \mid c\,d) \not\models \phi$. The green part shows all commands that still satisfy the specification, while the red part starts with the first command that violates it. For example, the sequence $(a\,b \mid c\,d)$ does not satisfy the specification ϕ and the first command that violates it, is c, which we call the *problematic command*.

Example. Consider the network in Fig. 4 which consists of seven routers organized in a route reflection hierarchy with *rr* acting

[3] A route reflector distributes BGP routes to its clients, eliminating the need for establishing an iBGP full-mesh which scales poorly [4].

[4] We only consider a subset of the networks from §2.1 as not all networks contained enough devices, see App. A for more details.

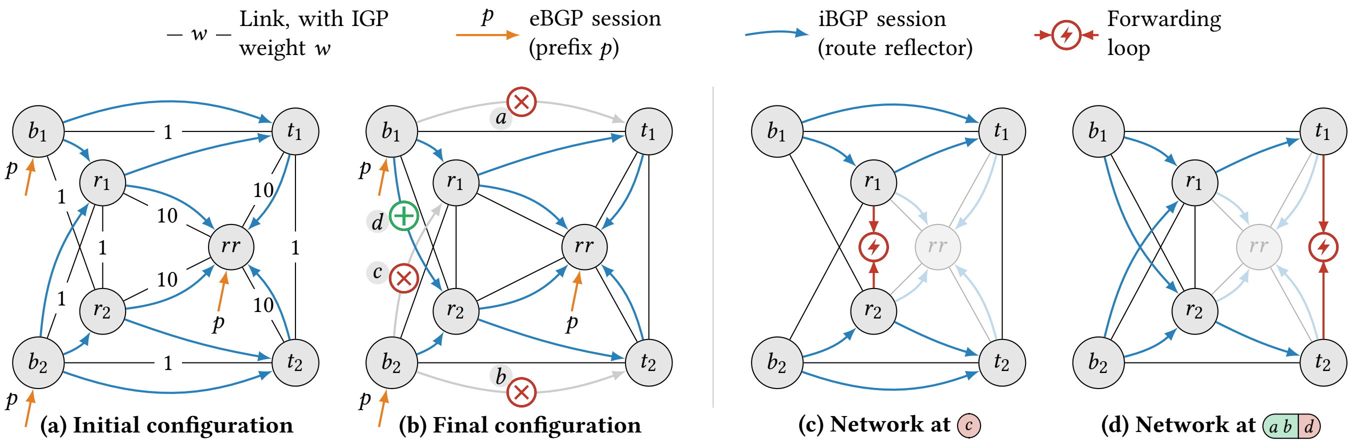

— w — Link, with IGP weight w
p eBGP session (prefix p)
iBGP session (route reflector)
Forwarding loop
b1
t1
r1
rr
r2
b2
t2
1
10
p
a
b
c
d
(a) Initial configuration
(b) Final configuration
(c) Network at c
(d) Network at a b d

Figure 4: Running example with 7 routers, where routers b_1, b_2 and rr receive an advertisement for the same prefix p. During the reconfiguration, three sessions must be removed (a , b , and c), and one added (d).

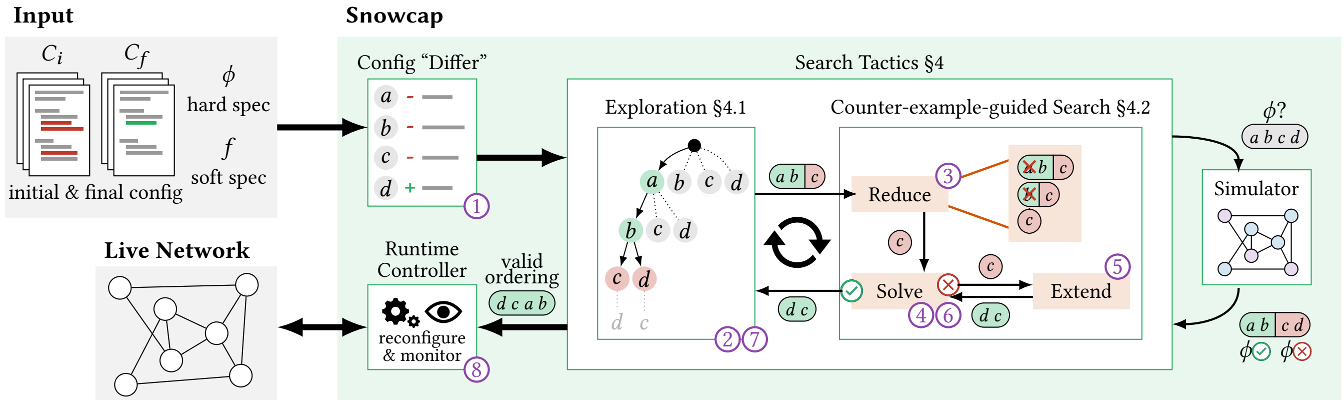


Figure 5: Snowcap finds a command ordering to safely transition from configuration C_i to C_f. The commands (a , b , c , d) correspond to the example in Figure 4.

as the root. Route reflection is used as an alternative to an iBGP full-mesh topology, which scales poorly. Any router in the iBGP topology only advertises routes to its route reflectors if they are learned from an external peer, or a route reflector client. A route reflector will always advertise its selected route to all of its clients [4, 22]. Three routers (b_1, b_2 and rr) receive the same external route for a prefix p (Fig. 4a). The reconfiguration scenario modifies the hierarchy by removing three iBGP sessions (a , b , and c) and adding one (d) (Fig. 4b). Initially, both r_1, r_2 and t_2 select the route from b_2 for prefix p (where r_2 reaches b_2 via r_1), while t_1 selects b_1. After the reconfiguration, all routers will select b_1 to reach p.

For simplicity, we assume that the operator is only interested in preserving reachability during the reconfiguration. Doing so is not easy and requires to add and remove sessions following a precise order. For example, applying c first (i.e., removing the session between b_2 and r_1) results in a loop between r_1 and r_2 (Fig. 4c). Indeed, removing the session causes r_1 to select b_1 instead of b_2 to reach p. Doing so it starts forwarding traffic to r_2; A loop is created as r_2 still learns (and prefers) b_2 to reach p and uses r_1 as next hop. Similarly, by applying d (i.e., adding the session between b_1 and r_2) after a and b causes a forwarding loop between t_1 and t_2, as t_1 only learns the route via b_2 from r_1, and t_2 only learns b_1 from r_2. Hence, t_1 will forward traffic to t_2, which itself will loop the traffic back to t_1 (Fig. 4d). Snowcap automatically computes the ordering $d\,c\,a\,b$ which preserves reachability.

Inputs. Snowcap takes 4 inputs: the initial and final configuration, C_i and C_f; together with the hard and soft specification, ϕ and f.

The hard specification ϕ is an LTL formula that has to be satisfied throughout the reconfiguration campaign. LTL allows operators to precisely specify policy transitions, as required during reconfigurations. For example, an operator can mandate traffic to first pass through the old firewall and then switch to the new one. In our running example, the reachability requirement is expressed with the formula $\phi = \mathbf{G} \bigwedge_r V_{(r,p,*)}$, which globally ($\mathbf{G}$) mandates a valid path (V) for traffic from any router r to reach a specific prefix p.

The soft specification is a function f that maps a sequence of network states to a cost: $f : \mathcal{S}^n \mapsto \mathbb{R}$, which Snowcap uses to guide its search towards better orderings from an operational perspective. As an example, to reduce the number of traffic shifts, the cost function f should be chosen to sum up the number of paths that change from every state to the next one (cf. §5.3).

Workflow (Fig. 5). Starting from the inputs, Snowcap reconfigures the network in three consecutive steps:

(1) the **Config "Differ"** first computes the set of commands to transition from C_i to C_f (*a* , *b* , *c* , *d* in our example);
(2) the **Search Tactics** then find a valid ordering of these commands satisfying the hard specification ϕ, while greedily optimizing for the soft specification f. The tactics use a simulator to compute the network state at each step;[5]
(3) the **Runtime Controller** finally applies the valid command ordering (here, *d c a b*) to the live network.

Search tactics (§4). The core of Snowcap is its search tactics which leverage counter-examples to efficiently navigate and prune the search space. At a high-level, our tactics explore different orderings until they hit a counter-example, i.e., an ordering violating the hard specification ϕ. It then aims to learn why this ordering does not work by finding dependencies among commands (e.g., *c* must *always* be applied after *d*) to iteratively prune the search space.

The strength of the search tactics lies in the interaction of two phases, which allow Snowcap to adapt to the characteristics of every reconfiguration scenario: *Exploration* (②, ⑦) and *Counter-example-guided Search* (③ — ⑥).

Phase 1: Exploration. The exploration phase quickly analyzes different command orderings by traversing the search space in a Depth First Search (DFS) manner, greedily following the direction that minimizes f. Once Snowcap finds itself in a dead end, with no commands satisfying ϕ, it switches to the second phase to resolve the violations it found.

In our example, Snowcap first applies *a* , followed by *b* . In this state however, neither command *c* nor *d* can be applied as both induce a forwarding loop. Before backtracking, Snowcap attempts to identify and resolve the hidden dependency.

Phase 2: Counter-example-guided search. In this phase, Snowcap finds dependencies by examining the command orderings violating the specification and resolving them. It works in three steps: *(i) Reduce*; *(ii) Solve*; and *(iii) Extend.*

In *Reduce*, Snowcap looks for the minimal set of commands that still violate the hard specification. In *Solve*, Snowcap then tries to find a valid ordering of the reduced set of commands, using the same approach as in the *Exploration* phase, and returns a valid ordering of these commands (if it exists) as one group, which we call a dependency group. If *Solve* does not find a solution, Snowcap performs *Extend*, introducing yet unconsidered commands, which might resolve the reduced problem. Once a dependency is found, Snowcap remembers it and continues the exploration.

Coming back to our example, Snowcap ③ removes both command *a* and *b* when considering the input ordering *a b|c*. This is because applying *c* before *d* always results in a forwarding loop between r_1 and r_2. Skipping the *Solve* phase ④, Snowcap tries to extend ⑤ the current problem. In fact, Snowcap notices that applying *d* before *c* preserves reachability. Treating *c d* as a single command, Snowcap switches back to the exploration phase ⑦ and finds the valid reconfiguration ordering *a b c d*.

Runtime controller. Finally, the runtime controller performs the reconfiguration by applying one command after the other according to the synthesized ordering until the network transitioned from the initial to the final configuration. After each command, the controller monitors the network state and waits for it to converge. Only then, it proceeds to apply the next command.

4 SEARCH TACTICS

We now explain how Snowcap's search tactics find a safe reconfiguration plan. First, we show how Snowcap explores the search space with a simple, yet effective DFS traversal (§4.1). Then, we explain how Snowcap speeds up the search by learning and resolving command dependencies, effectively pruning the space (§4.2). Finally, we present how Snowcap finds an *optimal*, valid ordering (§4.3).

In this section, we assume that we are given an oracle that determines whether a specific ordering o of the commands satisfies the hard specification: $o \models \phi$. It identifies the problematic command violating the specification, and the "reason" $\epsilon_\phi(o)$ for that, i.e., the violated part of the specification.

4.1 Simple Exploration

In the following, we present how Snowcap navigates the search space of all possible command orderings. We motivate Snowcap's DFS traversal based on an intuitive example.

The search space of all orderings is large and sparse, i.e., most orderings are invalid, rendering a random sampling approach useless. However, we can improve on the random approach by analyzing previous samples and adapting the search accordingly. If a certain sequence of commands, such as *c* in our running example (Fig. 4), violates the specification, there is no point in trying orderings which start with that invalid sequence (e.g., *c a b d*, *c a d b*, etc.).

Hence, we can approach the search for a valid ordering as a traversal of a tree, in which nodes represent orderings (only leaves are complete orderings), and traversing an edge means applying one of the remaining commands. Snowcap traverses this tree in a Depth First Search (DFS) manner, only exploring valid options by backtracking whenever a command violates the specification. Note, this exploration is complete, i.e., it finds a valid ordering, if and only if such an ordering exists.

Intuitively, the exploration prunes orderings which start with a known, invalid sequence. While this approach works well in many cases, it does not yet understand the underlying problem of these sequences. In fact, the exploration algorithm quickly reaches its limits if applying one command early on leads to problems towards the end of the reconfiguration. This means a command early in the sequence depends on one that appears only several steps later. Such dependencies have no immediate effect.

Dependencies without immediate effect. These dependencies are groups of commands that need to be applied in a specific order. If that order is not met, the specification is not violated immediately, but at a later command (see App. B for a formal definition). The

[5] Snowcap could also use any other network analyzer such as ARC [16], Batfish [10], Crystalnet [30], C-BGP [33], Minesweeper [5], or NV [18].

Algorithm 1: Counter-example-guided Search

```
  Input  : Groups $\mathcal{G}$, an ordering $o$ of $\mathcal{G}$ and the spec. $\phi$
1 $\mathcal{R} \leftarrow$ Set of groups not in $o$
2 $o_r \leftarrow Reduce(\mathcal{G}, o, \phi)$ using Alg. 2
3 loop
4     $o_s \leftarrow Solve(\mathcal{G}, o_r, \phi)$ using §4.1
5     if $o_s \neq \emptyset$ then return $o_s$
6     $o_r \leftarrow Extend(\mathcal{G}, o_r, \mathcal{R}, \phi)$ using Alg. 3
7     if $o_r = \emptyset$ then return $\emptyset$
8     $\mathcal{R} \leftarrow \mathcal{R}$ without all groups in $o_r$
```

exploration approach struggles with such situations as we show on our running example (Fig. 4): The sequence (a b) is valid, but in this state, both remaining options (a b|c) and (a b|d) violate the specification as they result in a forwarding loop. Imagine that additional commands { x_1 , . . . , x_n } exist, which are independent from the dependency group (e.g., increasing the link weights to router rr). Our simple exploration tries all possible permutations of x_1 , . . . , x_n , before finally backtracking to solve the actual problem.

4.2 Finding Dependencies

In the following, we explain how Snowcap overcomes the limitation of the simple exploration by actively searching for dependencies without immediate effect and learning how to resolve them. By finding and resolving these dependencies in a divide-and-conquer fashion, Snowcap prunes the search space even further. During exploration, once Snowcap encounters a dependency without immediate effect, it will perform a counter-example-guided search.

Counter-example-guided search. Snowcap's second search tactic is based on a divide-and-conquer approach; we split the problem of finding a valid ordering into smaller sub-problems of finding valid orderings within the dependency groups. Whenever the exploration hits a dead end, Snowcap uses that counter-example to identify the dependency groups and solve them individually, instead of backtracking (cf. Alg. 1). This works in three main phases *Reduce*, *Solve*, and *Extend*, which we explain in detail below.

In the following, we use the running example (Fig. 4) and assume that the input to Alg. 1 is the ordering o = (a b|c), like in Fig. 5. All three phases modify an incomplete ordering o of the commands, which is valid up to its final command. We say that this final command is *problematic* (c in the example).

Reduction phase (Alg. 2). The reduction phase aims to find the minimal set of commands that cause the problem, borrowing ideas from Delta Debugging [46] and Test Case Reduction [35]. During *Reduce*, we remove single commands (except the problematic one) to check whether they change the outcome. If removing a single command causes the ordering to become valid, we declare this command relevant for the current group. On the contrary, if removing the command does not change the outcome of the oracle, the command is removed; there does not seem to be a dependency. *Example:* Using our running example and the invalid ordering (a b|c), Snowcap first removes command a . Since (b|c) still causes the same forwarding loop between r_1 and r_2, a remains removed. The same applies to b , leaving us with the ordering (c).

Algorithm 2: Reduce

```
   Input  : Groups $\mathcal{G}$, an ordering $o$ of $\mathcal{G}$ and the spec. $\phi$
1  $i \leftarrow 0$
2  while $i + 1 < o.length$ do
3      $o_i \leftarrow o$ with group at position $i$ removed
4      $pos \leftarrow$ problematic command of $E_i$
5      if $pos + 1 < o_i.length$ then
6          $o'_i \leftarrow Reduce(\mathcal{G}, o_i$ up to position $pos)$
7          if Recursion depth $= 1$ then $o'_i \leftarrow o'_i.insert(0, i)$
8          return $o'_i$
9      else if $o_i \models \phi \lor \epsilon_\phi(o_i) \neq \epsilon_\phi(o)$ then $i \leftarrow i + 1$
10     else Remove group at position $i$ from $o$
11 return $o$
```

Algorithm 3: Extend

```
  Input  : Groups $\mathcal{G}$, an ordering $o$ of $\mathcal{G}$, a set of remaining groups
           $\mathcal{R}$ and the spec. $\phi$
1 for $g \in \mathcal{R}$ do
2     for $j \in \{0, 1, \ldots, o.length - 1\}$ do
3         $o_g \leftarrow o$ with group $g$ inserted at position $j$
4         $pos \leftarrow$ problematic command of $o_g$
5         if $o_g \not\models \phi \land pos + 1 < o_g.length$ then
6             return $Reduce(\mathcal{G}, o_g)$
7         if $o_g \models \phi \lor \epsilon_\phi(o_g) \neq \epsilon_\phi(o)$ then return $o_g$
8 return $\emptyset$
```

Solving phase. After Snowcap reduces the sequence to just the relevant commands, it tries to find a valid ordering of them. To this end, it uses a DFS exploration as described in §4.1. It returns the resulting ordering as a single group if it succeeds. Otherwise, it continues with the *extension phase*. *Example:* Snowcap realizes that there exists no valid ordering for (c) and continues with Alg. 3.

Extension phase (Alg. 3). If Snowcap cannot find a solution for the reduced ordering, it is a sign that the dependency group is not yet complete. Hence, it tries to extend the group with a single command, which it has not yet considered. Alg. 3 goes through every remaining command and inserts it at every possible position in the sequence. If the sequence becomes valid or the error changes (as described in the next paragraph), the algorithm returns the extended sequence. *Example:* Snowcap tries to extend the reduced sequence (c), with the yet unconsidered command d . Alg. 3 inserts d before c , resulting in (d c). Hence, the algorithm returns the extended sequence (d c) as a dependency group and continues.

Comparing errors. To find the minimal set of commands responsible for a problem, our approach has to determine whether a command is independent of the current problem or not. During the reduction phase (Line 9 of Alg. 2, for example), we compare the outcome of a sequence with and without a specific command. If the outcome is the same, the command is considered to be independent of the current problem. To compare the outcome of two different sequences, it does not suffice to check whether both of them satisfy the specification or not. One also has to check whether the same part of the specification is violated, i.e., whether the violation is due to the same reason $\epsilon_\phi(o)$ (cf. App. C.2 for a formal definition). This comparison of the oracle's result is not just used in Alg. 2, but also on Line 7 of Alg. 3. Often, removing commands does not solve the problem but shifts it to a different position. Hence, comparing the validity and the *reason* outperforms simple comparison on different networks and scenarios by several orders of magnitude.

4.3 Optimization

So far, we were only concerned with finding a valid ordering based on the hard specification ϕ. In the following, we explain how we can extend Snowcap's approach to find an optimal ordering by incorporating the soft specification f.

There exist $n!$ different command orderings with n being the total number of reconfiguration commands.[6] Due to this immensely large search space, Snowcap does not attempt to find the global optimum. Instead, it greedily minimizes each step, optimizing only locally, and hence, finding a local optimum. Instead of taking the first command that seems to work during the DFS exploration, Snowcap computes the cost of applying each remaining, valid command and continues with the one of lowest cost. While not necessarily finding the optimal solution, we show in our evaluation (§6.2) that this strategy works well in practice.

5 HARD & SOFT SPECIFICATION

This section discusses the specification guiding Snowcap's search for a valid and good ordering of reconfiguration commands in more detail. The specification consists of two parts: *(i)* the hard specification, which comprises the policies that must not be violated during the reconfiguration, and *(ii)* the soft specification, which assigns a cost to every command ordering and guides Snowcap towards an optimal one. In the following, we first introduce the language underlying the hard specification and then discuss techniques to evaluate the specification. Finally, we present the soft specification.

5.1 Specification Language

In the following, we present our hard specification language and explain how it differs from traditional specification languages for networks. The main building blocks consist of the well-known policies from the verification and synthesis literature: reachability, isolation, waypointing, and path redundancy. In contrast to prior work, the specification is dynamic: operators typically reconfigure the network due to policy changes. Our specification language (Fig. 6) is therefore based on LTL.

Logical Operators

ϕ	::=	$true$	*true*
	|	$\neg\phi$	*negation*
	|	$\phi_1 \wedge \phi_2$	*conjunction*
	|	$\phi_1 \vee \phi_2$	*disjunction*
	|	$\phi_1 \bigoplus \phi_2$	*xor*
	|	$\phi_1 \Rightarrow \phi_2$	*implication*
	|	$\phi_1 \Leftrightarrow \phi_2$	*if and only if*

Temporal Modal Operators

ϕ	::=	ϕ_1	*now*
	|	$\boldsymbol{X}\,\phi_1$	*next*
	|	$\boldsymbol{F}\,\phi_1$	*finally*
	|	$\boldsymbol{G}\,\phi_1$	*globally*
	|	$\phi_1\boldsymbol{U}\,\phi_2$	*until*
	|	$\phi_1\boldsymbol{R}\phi_2$	*release*
	|	$\phi_1\boldsymbol{W}\,\phi_2$	*weak until*
	|	$\phi_1\boldsymbol{M}\,\phi_2$	*strong release*

Propositional Variables

ϕ	::=	$V_{(r,p,c)}$	*valid path*
	|	$I_{(r,p)}$	*isolation*
	|	$V^{+}_{(r,p,c)}$	*redundancy*
	|	$C_{(r,p,c)}$	*convergence behavior*

Path Condition

c	::=	$\neg c$	*negation*
	|	$c_1 \wedge c_2$	*conjunction*
	|	$c_1 \vee c_2$	*disjunction*
	|	x	*path*
x	::=	xx	*sequence*
	|	r_i	*router r_i*
	|	$?$	*any router*
	|	$*$	*any number of routers*

Figure 6: Definition of the LTL specification language.

Basic policies. Snowcap supports a set of four basic policies on the forwarding behavior of the network, which can be combined according to the operator's requirements. We model the policies as predicates defined over a router r, a prefix p, and, except for isolation, a path condition c, which can hold on a forwarding state s. For example, $V_{(r,p,c)}$ holds on s if the path from router r to prefix p in s satisfies the path condition c. Similarly, $V^{+}_{(r,p,c)}$ holds if the path satisfies the condition under any single link failure. Path conditions are expressed as restricted regular expressions; $(*\,r_1\,*) \vee (*\,r_2\,*)$ requires packets to traverse either router r_1 or r_2, while $(*\,r_1 * r_2\,*)$ requires packets to traverse r_1 before r_2.

Temporal dimension. During reconfiguration, the behavior of the network is changing, and so is the network policy. Therefore, it is not enough to support a static specification defined over a single network state. The specification has to be defined over a sequence of them and needs to reflect configuration changes (e.g., to move traffic "gracefully" from an old to a new firewall). Hence, Snowcap's specification language is based on LTL (see App. C.1). The following examples highlight the benefits of using LTL as a specification language:

- *Reachability and redundancy*: During the entire reconfiguration process, every router should be able to reach every prefix, even under single link failures.

$$\boldsymbol{G} \bigwedge_{(r,p)\in\mathcal{F}} V_{(r,p,*)} \wedge V^{+}_{(r,p,*)}$$

  $\boldsymbol{G}\,\phi$ (globally) requires the expression ϕ to hold in every single state during the reconfiguration.

- *Firewall migration*: All traffic should be migrated from the old firewall at r_{old} to the new r_{new}, i.e., traffic should initially go via r_{old} and switch at one point over to r_{new}.

$$\bigwedge_{(r,p)\in\mathcal{F}} V_{(r,p,(*r_{old}*))}\ \boldsymbol{UG}\ V_{(r,p,(*r_{new}*))}$$

  In LTL, $\phi_1\ \boldsymbol{UG}\ \phi_2$ requires that ϕ_1 holds initially, and in all states until ϕ_2 holds for the remaining states.

- *Rerouting of a critical flow*: The flow (r, p) is critical and has to be migrated from path c^- to c^+. It is never allowed to take any other path (not even during convergence).

$$V_{(r,p,c^-)}\ \boldsymbol{U}\ \left(\left(V_{(r,p,c^+)} \wedge C_{(r,p,c^-\vee c^+)}\right) \wedge \boldsymbol{XG}\ V_{(r,p,c^+)}\right)$$

  In this expression, $\phi_1\ \boldsymbol{U}\ (\phi_2 \wedge \boldsymbol{XG}\ \phi_3)$ requires ϕ_1 to hold until ϕ_2 holds for a single, and ϕ_3 in all remaining states.

[6] Due to potential BGP Wedgies [21], the ordering of commands might change the resulting forwarding state, even if the resulting configuration is the same.

5.2 Evaluating the Hard Specification

The specification's dynamic nature brings three additional challenges: first, evaluating incomplete orderings; second, identifying the cause for the specification violation; and third, providing guarantees during convergence. In the following, we discuss each of the three points in more detail.

Partial evaluation. As the specification ϕ is defined over a sequence of network states, every single one of them must be present to check whether an ordering satisfies ϕ. It can happen that early network states violate parts of the specification, which makes it impossible for any future to be valid. To speed up the search, one would like to identify and dismiss these cases as soon as possible. To this end, we rely on partial evaluation of the LTL expression: we evaluate whether the expression *holds weakly* on a truncated sequence [8], i.e., if there exists a possible future in which the expression holds.

Error comparison. For Algs. 2 and 3, Snowcap needs to understand the exact reason for the specification violation, in order to determine if two problematic command orderings o_1 and o_2 violate ϕ due to the same reason ϵ_ϕ (cf. App. C.2). To this end, we apply two steps: First, we extract the set of propositional variables, which need to change in the last state of the sequence o_1, in order to make the LTL expression hold weakly (explained in detail in App. C.2). Second, for each of these propositional variables, we compare the actual forwarding path of its corresponding flow (r, p) in the last state of the two sequences s_1 and s_2.

Convergence behavior. A network operator is not only concerned with the sequence of converged states after every single reconfiguration step, but might also require properties during convergence. Therefore, the hard specification can contain convergence policies $C_{(r,p,c)}$ that must not be violated in any possible intermediate network state.

Transient effects have already been discussed in the literature for specific protocols. More precisely, Francois et al. has shown a method [13] for seamless reconfiguration of link-state IGP protocols like OSPF. However, no approach has yet been proposed which solves this problem in the general case. Enumerating all possible forwarding states that might occur during reconfiguration is nearly impossible, even if only BGP is considered. Reordering a single BGP message can result in a completely different convergence process.

Snowcap verifies convergence properties by computing, intuitively speaking, the union over all possible forwarding states during convergence, which we call the forwarding supergraph. This supergraph overapproximates the set of all possible forwarding states during convergence.This presents a sufficient but not necessary condition: our approach guarantees that path conditions are satisfied, but it cannot guarantee the existence of a problematic message ordering. We provide a full proof of correctness in App. D.2. While this approach provides guarantees about the paths traffic takes, it cannot guarantee the absence of blackholes. Our approach can be combined with the work of Francois et al. in the special case of IGP reconfiguration.

To compute the forwarding supergraph G_{fsg}, Snowcap only analyzes the network state before and after convergence: s^- and s^+. For each route x, our system computes the set of nodes $rri(x)$ which might learn the route x in either s^- or s^+ (App. D.1 describes how to construct $rri(x)$ for BGP). Next, we build the forwarding supergraph G_{fsg} by looking at all possibly considered routes for router v: $pcr(v) = \{x \mid v \in rri^+(x) \vee v \in rri^-(x)\}$. The final graph $G_{fsg} = (\mathcal{V}, E_{fsg})$ has an edge $(u, v) \in E_{fsg}$ only if there exists a route $x \in pcr(u)$, for which $v \in nh^-(u, x) \cup nh^+(u, x)$ (with $nh(u, x)$ being the next hop at router u towards the target advertised in x). Finally, to provide the convergence guarantees, Snowcap checks that the condition is satisfied in every possible path in G_{fsg} from the source to the target. The process of computing the forwarding supergraph G_{fsg} for the combination of BGP and IGP takes $O(|\mathcal{P}| \cdot |\mathcal{R}|^2)$, where $|\mathcal{P}|$ is the number of prefixes, and $|\mathcal{R}|$ is the number of internal routers in the network. The derivation of this can be found in App. D.3.

5.3 Soft Specification

There may exist multiple orderings satisfying the hard specification, which have varying side-effects, as shown in §2.2. Therefore, Snowcap also considers a soft specification that guides its search tactics towards a "good" solution.

The soft specification consists of a cost function $f : \mathcal{S}^n \mapsto \mathbb{R}$ that maps a sequence of converged network states to a cost. The sequence of states is given by applying the configuration commands of the ordering one-by-one. Currently, Snowcap supports traffic shifts as a cost function, penalizing changes in the forwarding state. One can easily add any cost function, as long as it is monotonically increasing (i.e., $f([s_0, \ldots, s_{n-1}]) \leq f([s_0, \ldots, s_n])$). Other examples include, e.g., minimizing the number of routes maintained in the routing table or preferring orderings with a faster transition. Snowcap uses a greedy approach to find a good reconfiguration ordering with respect to the provided cost function, see §4.3.

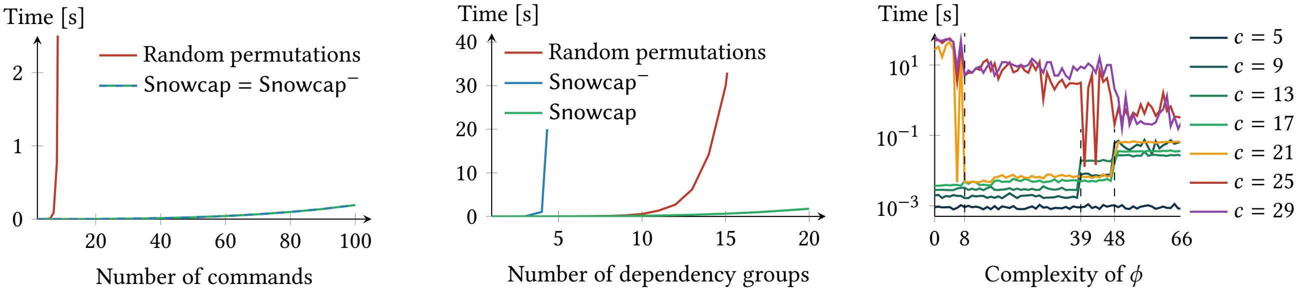
Example: Minimize Traffic Shifts. In the following, we highlight one example of a cost function that penalizes unnecessary traffic shifts during migration. First, we look at the costs associated with applying a single command. Then, we combine them to compute the costs of an entire command ordering. The cost associated with applying command with index i can be computed using the forwarding graph nh_{i-1} of the previous network state s_{i-1} and the graph nh_i of the current state s_i as follows:

$$f_i = \frac{1}{|\mathcal{R}| \cdot |\mathcal{P}|} \sum_{r \in \mathcal{R}} \sum_{p \in \mathcal{P}} \begin{cases} 1 & \text{if } nh_{i-1}(r,p) \neq nh_i(r,p) \\ 0 & \text{otherwise} \end{cases}$$

Here, $\mathcal{R}$ is the set of all internal routers, and $\mathcal{P}$ is the set of all externally advertised prefixes. The function $nh_i(r, p)$ represents the next hop for prefix $p \in \mathcal{P}$ chosen by the router $r \in \mathcal{R}$ in the state s_i. The final cost, associated with the entire ordering of length n, is computed by

$$F_C([f_1, \ldots, f_n]) = \sum_{1 \leq i \leq n} f_i$$

A cost of 0 means, that no router has changed its next hop during the reconfiguration process, i.e., no traffic shift occurred. If, during the entire reconfiguration process, the next hop of every prefix on every router changes exactly once, the cost is 1.



(a) Snowcap scales well in problem size, i.e., number of commands with straightforward dependencies.

(b) Snowcap's divide-and-conquer approach shines when faced with many complicated dependency groups.

(c) Specification complexity greatly impacts the shape of the search space, and hence Snowcap's runtime.

Figure 7: Runtime of Snowcap with respect to three different aspects; number of commands, number of dependency groups, and complexity of the hard specification.

6 EVALUATION

We now evaluate Snowcap along three dimensions using a prototype implementation by comparing it against multiple baselines. First, we analyze Snowcap's *performance* (§6.1) and show that, even for large reconfigurations, Snowcap finds a valid ordering within few seconds, which is orders of magnitude faster than the (random) baseline. We also show that Snowcap's runtime depends heavily on the shape of the search space, and only indirectly on the input size or the complexity of the reconfiguration scenario. Second, we analyze Snowcap's *effectiveness* (§6.2) at optimizing for soft objectives and the overhead incurred by the greedy optimization. We show that, in the vast majority of the networks and scenarios, Snowcap finds orderings that heavily reduce the number of traffic shifts while suffering from a predictable and acceptable overhead. Third, we analyze Snowcap's *accuracy* (§6.3) in evaluating properties during convergence. We show that Snowcap's analysis is sometimes overly cautious, but never deems a reconfiguration command safe when it is unsafe.

Implementation. Our implementation consists of ≈ 40 k lines of Rust code and currently supports: *(i)* static routes, link-state IGP protocols like OSPF, and BGP; *(ii)* the LTL-based hard specification language as shown in Fig. 6; and *(iii)* soft specifications to reduce traffic shifts. Our implementation can easily be extended to support additional protocols and specification properties. We run all experiments on a server with 64 cores clocked at 2.25 GHz and 512 GB of memory. One instance of Snowcap is always assigned a single thread. In all experiments, we use our own simulator as an oracle, which is able to verify around 50k states per second on average.

6.1 Scalability of Snowcap

In this section, we look at how Snowcap scales and how its runtime depends on: *(a)* the size of the reconfiguration problem, i.e., the number of reconfiguration commands; *(b)* the complexity of the reconfiguration problem, i.e., the number of dependency groups without immediate effect; and *(c)* the complexity of the hard specification ϕ.

Methodology. We compare Snowcap to a random baseline and Snowcap⁻ which only performs the exploration phase (§4.1) without learning dependencies (§4.2). We run each approach on each reconfiguration scenario 1 000 times and report the median execution time.

Reconfiguration size. In the first experiment, we analyze the performance of Snowcap with respect to the reconfiguration size, i.e., the number of reconfiguration commands. To this end, we use the chain gadget (see App. E.1), a variable-size synthetic topology consisting of n routers arranged in a chain, each of which is modified once during the reconfiguration to change its next hop. There exists exactly one valid ordering of these n commands, and any mistake will immediately cause a forwarding loop.

Fig. 7a shows the runtime incurred by the three approaches when checking for reachability. Snowcap clearly outperforms the random baseline as only one of the $n!$ orderings is valid. Snowcap⁻ performs identically to Snowcap, as all dependencies can be resolved using the exploration algorithm.

Reconfiguration complexity. In the second experiment, we inspect Snowcap's performance with respect to the number of dependency groups, i.e., the complexity of the reconfiguration problem. We use the Bipartite Gadget (see App. E.2), a synthetic topology built by replicating a small network. The reconfiguration for each sub-network involves three reconfiguration commands, forming a dependency group with no immediate effect. In each group, three orderings out of the possible $3! = 6$ are valid. The problem associated with each replicated network is independent of the others and can be solved in isolation.

Fig. 7b shows the runtime incurred by the three approaches when checking for reachability. Here, Snowcap's divide-and-conquer approach shines; by identifying and solving the dependency groups independently, Snowcap clearly outperforms the other approaches. While the random baseline takes more than 30 seconds to find a valid ordering for a problem with 15 dependency groups, Snowcap solves it in less than one second. Snowcap⁻ quickly reaches its limits as it has to backtrack frequently while solving the entire problem at once.

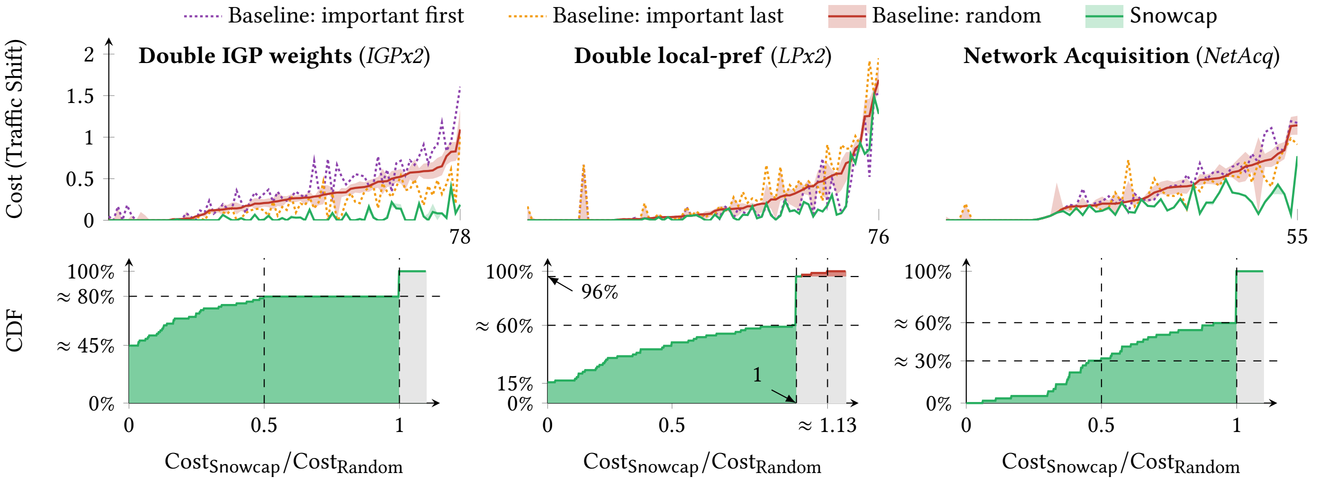


Figure 8: In almost all cases, Snowcap finds an ordering with significantly lower cost (less traffic shifts) than all three baselines, with higher consistency. The upper plots compare Snowcap with the baselines for different topologies. Below is the CDF of the median cost ratio between Snowcap and the random baseline.

Specification complexity. In the third experiment, we analyze the impact of specification complexity on Snowcap's runtime. We use the Abilene Network from Topology Zoo with six Forwarding Equivalence Classes (FECs)[7], each of which is advertised by two different eBGP peers. The network consists of 11 routers, which we reconfigure from a two-level route reflector topology to a single level, in addition to modifying several link weights. We start with a simple hard specification which requires reachability for all 66 flows. We then gradually increase the complexity of the specification by mandating an increasing number of flows to immediately switch from the initial to the final path and "stay there". We run the experiment for a varying number of commands c: we always use the 5 commands required for the BGP migration to which we add a variable number of link weight changes. For example, $c = 29$ represents the scenario in which 24 link weights are changed.

Fig. 7c shows the runtime of Snowcap (on a logarithmic scale) with respect to the number of restricted flows and commands. For simple cases ($c \leq 17$), increasing complexity leads to new dependencies, making it harder to find a solution. Especially condition 39 and 48 add complex dependencies. However, for larger scenarios ($c \geq 25$), increasing complexity might reduce the runtime, as it significantly restricts the search space. The scenario $c = 21$ exhibits both: adding condition 8 reduces the runtime by several orders of magnitude, but adding condition 48 brings new dependencies.

As it is the case for SAT solvers, Fig. 7c shows that Snowcap's runtime depends more on the shape of the search space rather than the complexity of the reconfiguration scenario.

6.2 Effectiveness of Snowcap

We measure Snowcap's effectiveness to optimize a soft specification by comparing its reconfiguration cost with three baselines and by analyzing the incurred overhead. We show that Snowcap consistently finds good reconfiguration orderings, outperforming the baselines in almost all experiments and we show that the overhead for the optimization remains bounded.

Methodology. We use a set of 80 topologies from Topology Zoo[8] (each containing between 5 and 82 routers, 34 on average) and consider four reconfiguration scenarios: doubling all IGP weights (*IGPx2*), doubling all local-preferences (*LPx2*), performing a network acquisition (*NetAcq*) (cf. §2.2), and moving from a full mesh iBGP topology to a single route reflector (*FM2RR*) (cf. §2.1). In all scenarios, we choose the IGP configuration at random. We always select the router with the most links as route reflector (following best practices [20]). Both *IGPx2* and *LPx2*, as well as the two merging networks in *NetAcq*, are configured to use a single route reflector. We then compare Snowcap's reconfiguration plans with *random* orderings, alongside with two importance-based orderings in which we order the commands according to the number of flows they affect in *increasing* or *decreasing* order.

Reconfiguration Costs. In Fig. 8, we compare the cost (number of traffic shifts, cf. §5.3) of Snowcap's reconfiguration ordering to those of the three baselines by performing 10 000 runs each. We show two plots for each scenario: First, we compare the median cost, along with the 25th and 75th percentile on each topology. Second, we show the CDF of the ratio between the median cost of Snowcap and the random approach. Intuitively speaking, the green area represents how often and by how much Snowcap outperforms the baseline, and the red area the opposite.

As Fig. 8 clearly highlights, Snowcap outperforms the baselines in terms of reconfiguration cost except for 3 out of the 209 topologies. Snowcap performs especially well for *IGPx2*, where in 80% of the topologies, it finds a solution at least twice as good as the random baseline. But also for *LPx2* and *NetAcq*, Snowcap outperforms the

[7] An FEC is a group of prefixes with identical forwarding behavior (cf. [26])

[8] Not every topology can be used for every scenario, as they have different topological requirements. The network acquisition scenario could only be evaluated on 55 of the 80 topologies. Also, few topologies could not be used with the other scenarios.

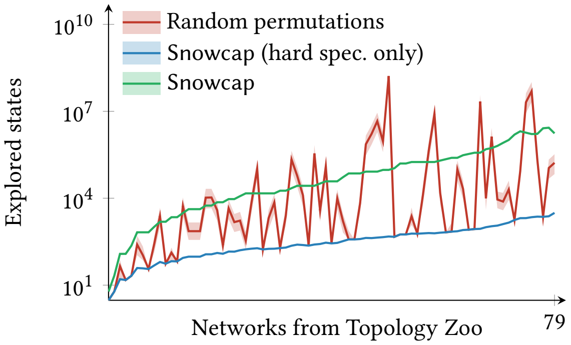


Figure 9: Optimizing soft specification comes at the cost of exploring roughly x^2 states as opposed to x.

random baseline in 60% of the cases each. In 4% of the topologies for *LPx2*, however, Snowcap incurs 13% higher cost than the random approach. "Most important first" outperforms "most important last" in scenario *IGPx2*, while it is the other way around in scenario *LPx2*, highlighting the benefits of using an adaptive approach like Snowcap's to accommodate the different scenarios.

Optimization overhead. While Snowcap effectively finds reconfiguration orderings with low costs, it also matters what overhead this incurs. Therefore, we measure the number of states explored by the random baseline and Snowcap with and without soft specification. To this end, we consider the scenario *FM2RR* and measure the median number of states, along with the 25th and 75th percentile, over 10 000 runs on 79 different topologies from Topology Zoo.

Fig. 9 shows the number of explored states for all three approaches on a logarithmic scale. The topologies are ordered by number of explored states of Snowcap's "hard spec. only" approach. The figure clearly shows that the soft specification comes at a cost: Snowcap explores roughly x^2 states compared to x without optimization, which is expected as Snowcap has to explore all possible commands at every step to find the optimal one. Nevertheless, Snowcap outperforms the random approach for some topologies by orders of magnitude while finding better solutions.

6.3 Accuracy of Snowcap

Snowcap relies on a sufficient, but not necessary condition (cf. §5.2) to provide guarantees on the network state during convergence. Since it is "only" sufficient, it can be overly careful and deem a safe convergence process to be unsafe. In the following, we analyze the accuracy of these guarantees.

Methodology. We use the Switch network from Topology Zoo, a topology with 30 routers and 61 edges. We select three route reflectors and choose 1000 different IGP configurations. The network has three neighbors e_x, e_1, and e_2, which all advertise the same FEC. The route from e_x is the least preferred. During reconfiguration, we remove the eBGP session towards e_2. As hard specification, we require all routers to forward traffic either towards e_1 or e_2 during convergence. To evaluate the accuracy, we check the convergence guarantees (*prediction*) and simulate 10 000 different, random convergence sequences to see if any sequence violates the requirement (*simulation*), resulting in 28 000 data points.

The results show that our prediction matches the simulation in 78.5% of the scenarios: for 57.6% of them Snowcap correctly assesses their safety; for 20.9% of them Snowcap correctly detects a violation. For the remaining 21.5% scenarios, Snowcap is too conservative: it sees a potential violation, even though no simulated convergence procedure violates the condition. This experiment shows that our condition is effective and most importantly, never considers a convergence process to be safe, when it is not.

7 CASE STUDY

We demonstrate Snowcap's practicality using an end-to-end implementation interfacing with a virtualized network using GNS3 [14] and FRRouting [41]. We use the Hibernia Ireland topology from Topology Zoo, a network with six routers. The reconfiguration involves moving from an iBGP full mesh to a route reflector topology (*FM2RR*), involving 15 commands.

For this case study, every router has a client connected to it, which continually sends packets towards all five external networks. We measure the number of packets lost during the reconfiguration. As a baseline, we apply all commands in random order and wait two seconds between each command, regardless of convergence. In total, around 50% of the traffic is lost during the 40 seconds it takes to apply all 15 commands. In comparison, when using the full pipeline of Snowcap without human intervention (i.e., Snowcap's runtime controller applies the sequence synthesized by our search tactics), the reconfiguration takes around 80 seconds, dropping only around 2% of the total traffic. These blackholes are caused by a specific behavior of FRR routers, which close a BGP session upon tagging a neighbor as a route reflector client. Without additional temporary safeguards (e.g., static routes), these problems cannot be avoided on FRRouting.

8 DISCUSSION

We now discuss some operational aspects of Snowcap including its complexity and completeness; and how to deal with impossible scenarios or failures during the reconfiguration.

Complete exploration. Snowcap is able to quickly find a valid solution for the vast majority of reconfiguration scenarios. This is achieved by aggressively pruning the search space using the counter-example-guided approach (§4.2). In few cases though, this approach might rule out valid solutions, potentially requiring Snowcap to exhaustively explore the search space (§4.1) instead. One example in which exhaustive exploration happens is when the configuration exhibits more than one stable state (i.e., the network contains one or more "BGP Wedgies" [21]); *and (ii)* any invalid ordering produces exactly the same error, preventing the *Reduce* phase (Alg. 2) to remove any commands.

While possible (we provide a theoretical example of the situation in App. F), we argue that these conditions are not practical—especially because they entail an incorrect BGP configuration to start with—and also did not manifest themselves in any of the practical reconfiguration scenarios we considered. Also, we stress that even in these unlikely scenarios, Snowcap works (it is complete)—albeit more slowly.

Impossible reconfigurations. It may be impossible to directly transition from the initial to the final configuration without violating the hard specification. In such situations, Snowcap is not able to find a safe ordering. To overcome the critical steps during the reconfiguration, one can introduce temporary configurations such as static routes. Finding the right temporary configurations is a difficult problem as one also needs to keep the network's resiliency (e.g., link failures) in mind. We plan to address this in future work.

Outages during reconfiguration. Networks are constantly faced with the possibility of unexpected outages, which can also happen during reconfiguration. Our specification language allows operators to express redundancy, i.e., that conditions still apply even if links in the network fail. This inherently solves the problem, without the need for control.

9 RELATED WORK

Network management automation. To reduce operator-induced downtimes, several systems have been proposed to automate network management [29, 38, 40]. These systems automate configuration generation and deployment for network operators. In addition, they monitor the network state during updates to react upon anomalies. Snowcap can extend these systems by providing a safe reconfiguration ordering, eliminating potential anomalies and human interventions during the updates.

Network migrations. Researchers have put extensive focus on the special case of IGP migrations. Francois et al. [11, 13] have shown how to avoid transient forwarding loops in link-state protocols, such as OSPF or IS-IS, by updating the routers in a specific order and progressively changing the link weights. Raza et al. [34] extended this approach by allowing to optimize for certain metrics during the reconfiguration (e.g., minimize link utilization).

Several systems build upon the technique known as Ships-In-The-Night [3, 25, 42–45], where each router is running two separate configurations in parallel. The new configuration runs in the background and the transition happens once it has converged. All these approaches pose particular requirements to the hard- and software of network devices as they need to support multiple routing and forwarding tables at the same time.

SDN updates. Several works looked at safe transitions from one configuration to another in the context of SDN [24, 31, 32]. While the problem is similar, the solution differs vastly as reconfiguration in SDN means updating the forwarding state directly. The work of McClurg et al. [32] takes a similar approach as Snowcap: it finds an ordering of data plane updates using counter-examples.

The Routing Control Platform (RCP) [7] combines ideas from SDN with traditional, distributed networking to solve the problem of network-wide configuration updates. It does so by logically-centralizing the routing information and performing the route selection on behalf of the routers. Approaches like RCP require drastic changes to the network-wide configuration and topology, and have several side-effects. Snowcap, however, can be used with traditional networks without the need for any adaptation of the network.

Network configuration repair. CPR [15] and AED [1] synthesize configuration repairs for a given configuration such that it meets the operator's specification. AED can also take management and operational objectives into account, such as minimizing the number of devices affected by the repair or the total number of configuration changes. Snowcap complements these systems as it can *apply* their repairs without violating the specification during the reconfiguration. Concretely, one could use AED to synthesize the final configuration, and then Snowcap to safely transition to it.

Abstract control plane representation. Tiramisu [2] and ARC [16] use an abstract graph representation of the control plane to analyze network configurations. Snowcap's convergence guarantees (§5.2) rely on a similar approach using a graph representation of the transient forwarding state. Its accuracy might be improved by incorporating control plane information like ARC does.

10 CONCLUSION

We presented Snowcap, the first protocol-agnostic system to synthesize *safe* network-wide configuration updates in distributed control planes by phrasing the problem as an optimization problem under constraints. We introduced a precise and dynamic specification language based on LTL to allow operators to specify the transition from the old to the new high-level policy. We further proposed search tactics which leverage counter-examples to isolate command dependencies and resolve them independently. Finally, we demonstrated Snowcap's scalability and effectiveness: Snowcap finds good reconfiguration plans for realistic network topologies and reconfiguration scenarios in few seconds.

Ethical issues. This work does not raise any ethical issues

ACKNOWLEDGEMENTS

We thank our shepherd Lixin Gao and the anonymous reviewers for their insightful comments and helpful feedback. The research leading to these results was supported by an ERC Starting Grant (SyNET) 851809.

REFERENCES

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. AED: Incrementally Synthesizing Policy-Compliant and Manageable Configurations. In *ACM CoNEXT*. Barcelona, Spain.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *USENIX NSDI*. Santa Clara, CA.

[3] Richard Alimi, Ye Wang, and Y. Richard Yang. 2008. Shadow Configuration as a Network Management Primitive. In *ACM SIGCOMM*. Seattle, WA, USA.

[4] T. Bates, E. Chen, and R. Chandra. 2006. *RFC 4456: BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP)*. Technical Report.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*. Los Angeles, CA, USA.

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2019. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations: Brief Reflections on Abstractions for Network Programming. *ACM SIGCOMM CCR* 49, 5 (2019), 104–106.

[7] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2005. Design and Implementation of a Routing Control Platform. In *USENIX NSDI*. Boston, MA, USA.

[8] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. 2003. Reasoning with Temporal Logic on Truncated Paths. In *CAV*. Boulder, CO, USA.

[9] Klaus-Tycho Foerster, Stefan Schmid, and Stefano Vissicchio. 2018. Survey of Consistent Software-Defined Network Updates. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1435–1461.

[10] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. 2015. A General Approach to Network Configuration Analysis.. In *USENIX NSDI*. Oakland, CA, USA.

[11] Pierre Francois and Olivier Bonaventure. 2007. Avoiding Transient Loops During the Convergence of Link-State Routing Protocols. *IEEE/ACM Transactions on Networking* 15, 6 (2007), 1280–1292.

[12] P. Francois, O. Bonaventure, B. Decraene, and P. Coste. 2007. Avoiding Disruptions during Maintenance Operations on BGP Sessions. *IEEE Transactions on Network and Service Management* 4, 3 (2007), 1–11.

[13] Pierre Francois, Mike Shand, and Olivier Bonaventure. 2007. Disruption Free Topology Reconfiguration in OSPF Networks. In *IEEE INFOCOM*. Barcelona, Spain.

[14] Galaxy Technologies, LLC. [n. d.]. GNS3 | The software that empowers network professionals. https://www.gns3.com/. ([n. d.]). Accessed: 2021-01-23.

[15] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically Repairing Network Control Planes Using an Abstract Representation. In *ACM SOSP*. Shanghai, China.

[16] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *ACM SIGCOMM*. Florianopolis, Brazil.

[17] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. 2015. Management Plane Analytics. In *ACM IMC*. Tokyo, Japan.

[18] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *ACM PLDI*. London, UK.

[19] V. Gill and J. Mitchell. 2003. AOL Backbone OSPF-ISIS Migration. NANOG29 Presentation. (2003).

[20] Barry Raveendran Greene and Philip Smith. 2002. *Cisco ISP Essentials*. Cisco Press.

[21] T. Griffin and G. Huston. 2005. *RFC 4264: BGP Wedgies*. Technical Report.

[22] Timothy G Griffin and Gordon Wilfong. 2002. On the Correctness of IBGP Configuration. In *ACM SIGCOMM*. Pittsburgh, PA, USA.

[23] Gonzalo Gomez Herrero and Jan Antón Bernal Van der Ven. 2011. *Network Mergers and Migrations: Junos Design and Implementation*. Vol. 45. John Wiley & Sons.

[24] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates. In *ACM SIGCOMM*. Chicago, IL, USA.

[25] John P John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. 2008. Consensus Routing: The Internet as a Distributed System. In *USENIX NSDI*. San Francisco, CA, USA.

[26] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*. San Jose, CA, USA.

[27] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. 2011. The Evolution of Network Configuration: A Tale of Two Campuses. In *ACM IMC*. Berlin, Germany.

[28] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *IEEE JSAC* 29, 9 (2011), 1765 –1775.

[29] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic Life Cycle Management of Network Configurations. In *ACM SelfDN*. Budapest, Hungary.

[30] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully Emulating Large Production Networks. In *ACM SOSP*. Shanghai, China.

[31] Ratul Mahajan and Roger Wattenhofer. 2013. On Consistent Updates in Software Defined Networks. In *ACM HotNets*. College Park, MD, USA.

[32] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. 2015. Efficient Synthesis of Network Updates. In *ACM PLDI*. Portland, OR, USA.

[33] Bruno Quoitin and Steve Uhlig. 2005. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network* 19, 6 (2005), 12–19.

[34] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. 2011. Graceful Network State Migrations. *IEEE/ACM Transactions on Networking* 19, 4 (2011), 1097–1110.

[35] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *ACM PLDI*. Beijing, China.

[36] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*. London, United Kingdom.

[37] P. Smith. 2010. BGP Techniques for Internet Service Providers. NANOG50 Presentation. (2010).

[38] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. 2014. A Network-State Management Service. In *ACM SIGCOMM*. Chicago, IL, USA.

[39] Yu-Wei Eric Sung, Sanjay Rao, Subhabrata Sen, and Stephen Leggett. 2009. Extracting Network-wide Correlated Changes from Longitudinal Configuration Data. In *PAM*. Seoul, Korea.

[40] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *ACM SIGCOMM*. Florianopolis, Brazil.

[41] The Linux Foundation. [n. d.]. FRRouting. https://frrouting.org/. ([n. d.]). Accessed: 2021-01-23.

[42] Laurent Vanbever, Stefano Vissicchio, Luca Cittadini, and Olivier Bonaventure. 2013. When the Cure is Worse than the Disease: The Impact of Graceful IGP Operations on BGP. In *IEEE INFOCOM*. Turin, Italy.

[43] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. 2011. Seamless Network-Wide IGP Migrations. In *ACM SIGCOMM*. Toronto, Ontario, Canada.

[44] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. 2012. Lossless Migrations of Link-State IGPs. *IEEE/ACM Transactions on Networking* 20, 6 (2012), 1842–1855.

[45] Stefano Vissicchio, Laurent Vanbever, Cristel Pelsser, Luca Cittadini, Pierre Francois, and Olivier Bonaventure. 2012. Improving Network Agility with Seamless BGP Reconfigurations. *IEEE/ACM Transactions on Networking* 21, 3 (2012), 990–1002.

[46] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A NETWORK ACQUISITION CASE STUDY

To study the impact of different reconfiguration plans on the reconfiguration costs and to evaluate Snowcap, we studied the network acquisition scenario [23], in which two networks are merged. For this scenario, we automatically and randomly partition the network into two distinct connected components, where both are connected to at least one external device. For each of these components, we choose a single route reflector based on the router with the highest degree, and choose all link weights randomly. All external routers advertise each of the 5 different FECs with probability 50%. Every generation is seeded, such that statistics correspond to the same configuration and can be compared.

Initial configuration. Initially, every router has an iBGP session with the route reflector of its component. In addition, all links connecting the two components are disabled. All link weights of the second network are scaled down by a factor of 10.

Reconfiguration. During the reconfiguration, we enable the links connecting the two networks and scale up the link weights of the one network by a factor of 10, to match the range of the other. Additionally, we connect the two route reflectors as iBGP peers.

Results. Fig. 10 shows the full results of performing the reconfiguration on 42 networks from Topology Zoo. Since not all networks in the Topology Zoo collection contain two external devices, connected to different internal routers, we could not use every network.

B DEPENDENCIES WITHOUT IMMEDIATE EFFECT

Intuitively, dependencies without immediate effect are violations caused by configuration commands early in the reconfiguration process that do not manifest themselves until several commands later. Formally, we define them as follows:

Definition B.1 (Dependencies without immediate effect). let Q be a set of commands, and ϕ be the specification. Then, $G \subseteq Q$ contains dependencies with no immediate effect, iff there exists a subset $g \subset G$ and an ordering $o_g \in P(g)$ (where $P(\cdot)$ is the set of all possible permutations), for which the following two conditions hold:

(1) $o_{g \models \phi}$,
(2) $\forall o'_g \in P(G \setminus g) : (o_g + o'_g \not\models \phi$,
(3) $\exists o \in P(G) : o \models \phi$.

Based on the Definition B.1, we can see that if there exists such a dependency G, and if the ordering o_g is explored first, then the simple exploration algorithm (§4.1) needs to backtrack until the dependency is solved. The example reconfiguration, depicted in Fig. 4, contains a dependency without immediate effect. The sequence $(a\,b) \models \phi$ is valid, but for both options: $(a\,b\,|\,c) \not\models \phi$ and $(a\,b\,|\,d) \not\models \phi$. Hence, the simple exploration algorithm needs to backtrack, until the problem is solved (which is the case for the sequence $(d\,c\,a\,b)$ $\models \phi$).

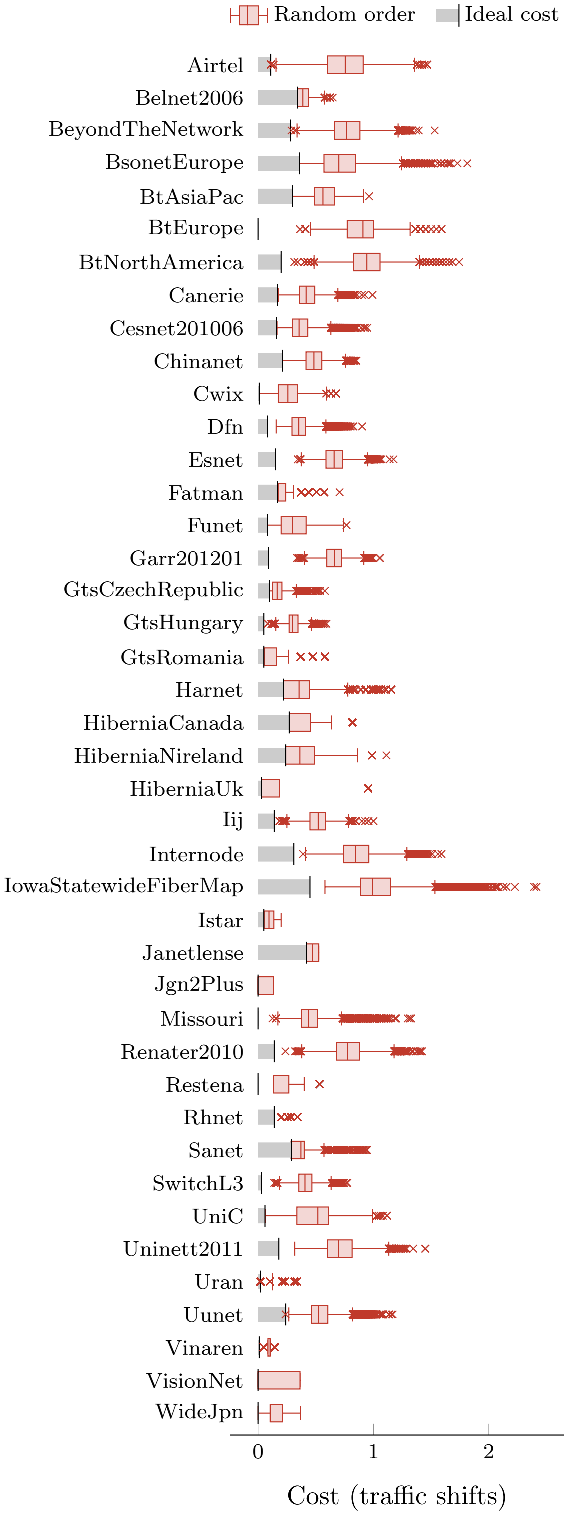


Figure 10: The reconfiguration costs of a random order compared to the ideal costs. This is an extended version of Figure 3.

C SPECIFICATION

C.1 Linear Temporal Logic

In the following, we describe all modal operators in LTL, and how to evaluate them on a sequence of states w:

- ϕ: The expression ϕ holds in the current state.
- $\boldsymbol{X}\phi$ (ne**X**t): The expression ϕ holds in the next state.
- $\boldsymbol{F}\phi$ (**F**inally): The expression ϕ must hold in either the current state, or any of the future states.
- $\boldsymbol{G}\phi$ (**G**lobally): The expression ϕ must hold in the current state and all future states.
- $\psi\ \boldsymbol{U}\ \phi$ (**U**ntil): The expression ψ must hold, until the expression ϕ holds. ϕ *must* hold eventually. In the state, where ϕ holds, ψ is not required to hold too.
- $\psi\ \boldsymbol{R}\ \phi$ (**R**elease): The expression ϕ must hold, until the expression ψ holds. The expression still evaluates to *true*, if ψ never holds, but ϕ holds indefinitely. In the state, where ϕ holds, ψ *must* hold too.
- $\psi\ \boldsymbol{W}\ \phi$ (**W**eak until): The expression ψ must hold, until the expression ϕ holds. The expression still evaluates to *true*, if ϕ never holds, but ψ holds indefinitely. In the state, where ϕ holds, ψ is not required to hold too.
- $\psi\ \boldsymbol{M}\ \phi$ (strong release): The expression ϕ must hold, until the expression ψ holds. ψ *must* hold eventually. In the state, where ϕ holds, ψ *must* hold too.

C.2 Error Comparison with LTL

For comparing errors, we wish to extract a *reason* for why an LTL expression ϕ does not hold weakly for a given sequence of states w. In the following, we will denote a (partial) sequence of states $w = (s_1, s_2, \ldots, s_n)$, which has a finite length $|w| = n$. $w \models \phi$ denotes that ϕ holds weakly on w. We denote $w^i = (s_i)$ to be the state of w at position i, and $w^{i..j} = (s_i, s_{i+1} \ldots, s_j)$ to be the partial sequence of w.

Assume we are given a sequence w, with $|w| = n$, where $w^{..n-1} \models \phi$, but $w \not\models \phi$. We define the *reason* for a sequence w to be a set of propositional variables, which, if changed in the last state w^n of w to form w_*, $w_* \models \phi$. Note, there must exist at least one set of propositional variables, for which the statement before holds, since $w^{..n-1} \models \phi$. More formally:

Definition C.1 (reason). Given ϕ, and a sequence w with $|w| = n$, where $w^{..n-1} \models \phi$, but $w \not\models \phi$. The reason $\epsilon_\phi(w)$ for $w \not\models \phi$ is given by:

$$\epsilon_\phi(w) = \bigcup \left\{p \subseteq w^n \mid (w^{..n-1}, w^n_p) \models \phi\right\},$$

where w^n_p represents a state, similar to w^n, where the value of all propositional variables in p have changed.

As an example, assume $w = ((x_1, x_2), (x_1, \neg x_2))$. and $\phi = x_1 \wedge x_2$. In this case, $w^{..1} \models \phi$, but $w \not\models \phi$. Then, $\epsilon_\phi(w) = \{x_2\}$, since $w^2_{x_2} = (x_1, \neg\neg x_2)$ causes $(w^{..1}, w^2_{x_2}) \models \phi$.

D CONVERGENCE GUARANTEES

D.1 Generating $rri(x)$ for BGP

The condition, presented in §5.2 requires the generation of the route reachability information $rri(x)$ for each route x in the network. In the following, we describe how to compute $rri(x)$ for BGP, but it can easily be generalized to other protocols. We first build the two directed graphs $G^\pm_{bgp} = (\mathcal{V}, E^\pm_{bgp})$ for both the network state s^- before the reconfiguration step, and s^+ after the step, where the edges $e \in E^\pm_{bgp}$ are labelled $e \in \{U, O, D\}$, corresponding to different BGP sessions, as described by [22]. Then, for each BGP route x, we traverse both forwarding graphs $G^\pm_{bgp}$ by following BGP forwarding rules. For each matching BGP route map, we generate a new route x', which is traversed separately. Then, $rri^\pm(x)$ is the set of nodes that are reached by x during this traversal.

D.2 Proof of Sufficiency

In the following, we proof that the condition presented in §5.2 is sufficient, i.e., if there exists a convergence process that result in an invalid transient network state, then our condition is necessarily violated.

Definition D.1 (Similar Network States). Two network states s^- and s^+ are similar if the following conditions are satisfied:

(1) All routes, that can exist during convergence, are also present in s^+ or s^-.
(2) For all routes x, no router $v \notin rri(x)$ can ever learn x.

With careful construction of $rri(\cdot)$, as described for BGP in App. D.1, the two states s^+ and s^- are always similar, if s^+ can be reached from s^- by applying a single command.

Lemma D.2. *If the two states s^- and s^+ are similar, then G_{fsg} contains every possible path in the network during convergence.*

Proof of Lemma D.2. In G_{fsg}, a node u has an edge to a neighboring node v if there exists a route which might reach u, and where the next hop is v. Since s^- and s^+ are similar, during construction of G_{fsg}, we have considered every route in the network by analyzing only the converged states s^- and s^+. Hence, there cannot exist a route, which might reach u during convergence, but is not present in G_{fsg}. □

Theorem D.3. *If the two states s^- and s^+ are similar, then our algorithm for checking convergence guarantees is sufficient.*

Proof of Theorem D.3. Due to Lemma D.2, the set of all paths in G_{fsg} contains every possible path in the network during convergence. Hence, if the conditions are satisfied on all paths, then there cannot exist an ordering of messages during convergence, which violates the condition. □

D.3 Complexity for Convergence Guarantees

To provide the convergence guarantees, Snowcap must first generate the forwarding supergraph G_{fsg}, and then enumerate all paths in G_{fsg}. We traverse both BGP graphs $G^\pm_{bgp}$ in a DFS manner to generate $rri^\pm(x)$, which takes $O(|E_{bgp}|) = O(|\mathcal{R}|^2)$. Then, constructing G_{fsg} for any given prefix p takes $O(|\mathcal{R}|)$ time. Finally, enumerating all simple paths in the G_{fsg} takes $O(|E_{fsg}|) = O(|\mathcal{R}|^2)$ time. Taking everything together, we can perform the convergence behavior analysis in $O(|\mathcal{P}| \cdot |\mathcal{R}|^2)$ time.

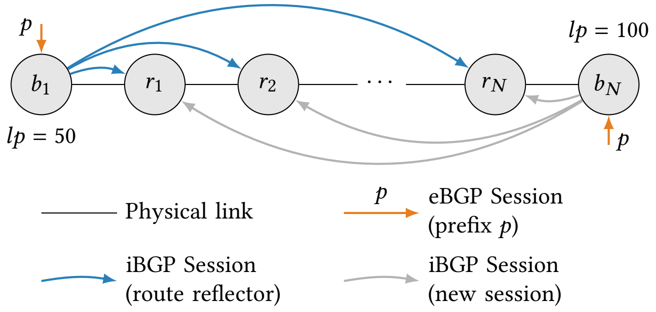

p
$lp = 100$
b_1
r_1
r_2
· · ·
r_N
b_N
$lp = 50$
p
Physical link
p
eBGP Session (prefix p)
iBGP Session (route reflector)
iBGP Session (new session)

Figure 11: Chain gadget with N routers in the chain.

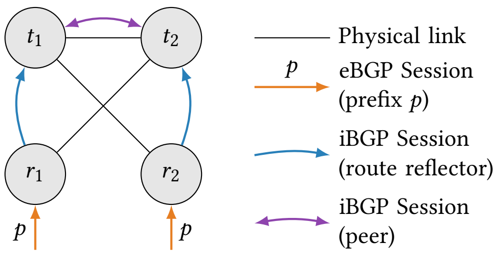


Figure 12: Unstable Gadget, where the order of advertisement changes the final forwarding state.

E SYNTHETIC GADGETS

E.1 Chain Gadget

The chain gadget (see Fig. 11) consists of $N + 2$ routers, connected in a single chain. The two routers at the end are border routers, receiving the same external prefix. The router b_1 sets the local-pref to 50, while b_N sets the local-pref to 100. In the initial configuration, all routers only have an iBGP session with b_1 and consequently use it as egress router. During the reconfiguration, we add an iBGP session from every router to b_N. Since b_N announces a route with higher preference, all routers will choose it as egress in the end.

The only valid command ordering is adding the BGP sessions from right to left, first adding the session $b_N \rightarrow r_N$, followed by $b_N \rightarrow r_{N-1}$, etc. If r_i is reconfigured before r_{i+1}, then there will be a forwarding loop between r_i and r_{i+1}.

E.2 Bipartite Gadget

The Bipartite Gadget, depicted in Fig. 13, is based on replicating a smaller gadget, the Unstable Gadget (see Fig. 12), multiple times. The Unstable Gadget represents one dependency group and consists of three reconfiguration commands: a adds the iBGP session $u_1 \rightarrow t_1$, b adds $u_2 \rightarrow t_2$, and c removes $b_2 \rightarrow r_2$.

If command a is executed first, both t_1 and t_2 will choose b_1 as egress, but if b is applied before a, then both t_1 and t_2 will choose b_2. Then, applying c will force r_2 to choose b_x as an egress, and hence, cause a forwarding loop between r_2 and t_1 if and only if b is executed before a. Hence, the following three sequences $a\,b\,c$, $a\,c\,b$ and $c\,a\,b$ don't cause a forwarding loop. However, $b\,a \mid c$, $b \mid c\,a$ and $c \mid b\,a$ cause forwarding loops.

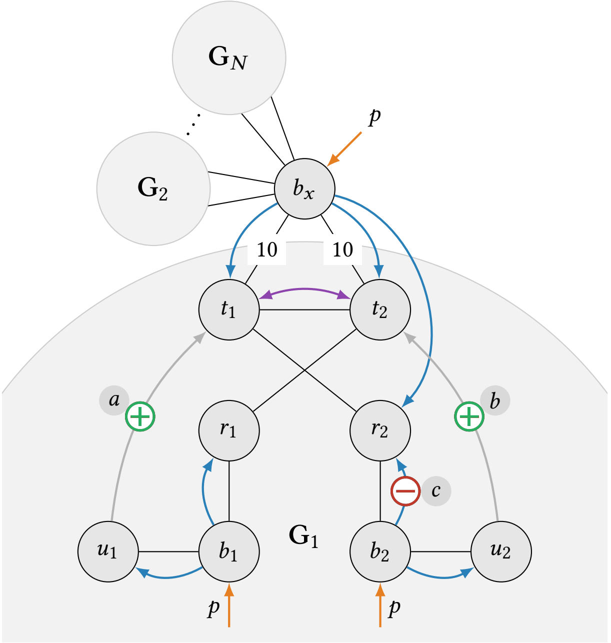


Figure 13: Bipartite Gadget with N groups of size 2. Two sessions a and b are added during reconfiguration, and the session c is removed.

F EXHAUSTIVENESS OF SNOWCAP

In the following, we highlight the conditions under which the divide-and-conquer approach (cf. §4.2) cannot find a solution. In these cases, Snowcap falls back to the exploration tactic (cf. §4.1), such that Snowcap remains exhaustive. First, we list all necessary conditions for such a case, and construct a theoretical example.

Since the exploration phase is exhaustive as long as no dependency group has been learned, the bad scenario needs to contain at least two dependencies. The group, that is learned first (called g_1), must be included in the second one (called g_2). Next, an already learned group is never split up into different groups, it may only be reordered during the *Solve* phase. A bad scenario must therefore prevent the system from entering the *Solve* phase with the complete set of commands required for the dependency to be solved. Hence, any invalid ordering must produce the exact same error, such that critical commands are removed during *Reduce*. Also, since the exploration phase may try every possible ordering of the learned groups, the group g_2 must contain the commands from g_1, but in a different order than g_1. Hence, a BGP Wedgie [21] needs to be present.

An example of such a case consists of three commands a, b and c, with the only valid solution being $a\,c\,b$. There exists a dependency with immediate effect, namely that a needs to happen before b. Then, the sequences c and $a\,b \mid c$ need to produce the exact same error (as described in App. C.2). Snowcap will either find the valid ordering initially (with probability 1/6), or learn the dependency $a\,b$ first, in which case, our system will not be able to find a valid solution. Notice, that a BGP Wedgie is present, since $a\,b \mid c$ results in a different state than $a\,c\,b$, even though the exact same configuration is running.

G ARTIFACT APPENDIX

Abstract

Snowcap's artifact includes its full source code and documentation, which is available on GitHub. In addition, we provide instructions on how to build Snowcap, alongside a script to run all experiments and reproduce the results described in this paper.

Scope

The artifact allows to reproduce all the results and claims made in this paper. With the provided source code and documentation, one can also run Snowcap in their own network and extend it.

Contents

The artifact consists of the full source code and all scripts necessary to run Snowcap, conduct every experiment in this paper and reproduce all the results. In addition, the artifact includes all the data we used to generate the plots in this paper.

Hosting

Snowcap is publicly available on GitHub[9]. The artifact documentation can be found in the main branch (commit `b486620`) in the directory `eval_sigcomm2021`.

Requirements

Software. Snowcap is built using Rust and requires:

- a stable Rust toolchain (1.49 or higher);
- Python 3.8 or higher (with the packages `numpy`, `pandas` and `matplotlib` installed);
- a Latex build environment (including `pdflatex`);
- and GNS3 (`gns3-server` and `gns3-gui`).

We provide detailed instructions on how to setup and run the artifact in the artifact documentation on GitHub. In addition, we provide a pre-configured virtual machine.

Hardware. We ran all experiments on a server with 64 cores (128 threads). It took about a week (approximately 20 000 CPU hours) to run all experiments. We provide an option to speedup the experiments by reducing the number of iterations, which leads to less precise data and less accurate statistics (more details can be found in the artifact documentation).

[9] https://github.com/nsg-ethz/snowcap