Master Thesis

Optimizing SDN Updates With Transient Consistency Guarantees
Quick Updates Under Loop-Freedom and Waypoint Enforcement Constraints

Damien Foucard

Supervised by:
Anja Feldmann

Assistant supervisors:
Dr. Stefan Schmid
Arne Ludwig

March 21, 2015
Erklärung der Urheberschaft

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.
Berlin, the November 2, 2016.

Unterschrift
Abstract

Software-defined networking (SDN) is a paradigm rich of opportunities. The administrator defines a simple global policy in the control plane and the controller automatically translates it into routing tables of each switch, thus shielding the administrator from the complexity of vendor-specific switch configurations. Hence, the administrator simply needs to request the controller to enforce a particular constraint, and it will automatically compute a new policy in keeping with this constraint, and updates the network to enforce this policy.

Unfortunately, while each singular policy is consistent with specified constraints, the network can remain for an arbitrary large amount of time in a transient state violating them if no specific measure is taken. To avoid this problem, the controller should make sure that the network can only go through transient states conforming to specified constraints, which requires scheduling some updates before others. However, this possibly implies that many switches must wait for others to implement the new policy. A shortest possible update schedule – reducing the latency before the new policy is enforced to the minimum while respecting specified constraints– appears then desirable but computing it induces a high computational cost, potentially not affordable in real time.

We consider two constraints: loop-freedom (no packet should loop) and waypoint enforcement (all packets must traverse a middlebox). Loop-freedom is desirable because, although packets caught in loops are usually automatically removed (e.g., with TTL), they represent an undesirable load. Waypoint enforcement is a key feature for Network Function Virtualization and is of particular relevance in security-critical environment, where allowing packets to bypass a middlebox even for a short time already represents an unacceptable risk.

An ideal algorithm for computing an update schedule would be easy to compute (minimized computational cost), update the network quickly (routers implementing new policy in minimal time), and offer strong consistency guarantees (maximized number of constraints ensured). Unfortunately, as we show in this work, it appears that this Graal does not exist, and favoring one dimension comes at the expense of another. Hence, balances need to be struck. Accordingly, we consider a variety of settings, favoring in turn one dimension or the other, and in each case present optimized algorithms and highlight caveats. In addition, we implement a framework in which we simulate our algorithms and measure their performance, and present our results, the framework’s modular design allowing for an easy expansion.
## Contents

1 Introduction ........................................... 1

2 Motivations, Simplifications, and Overview ............ 3
   2.1 Path Differentiation ........................................ 3
      2.1.1 Focus Reduction ....................................... 3
      2.1.2 Removing Dependencies with Path Differentiation ...... 6
      2.1.3 Problem Partitioning .................................... 6
      2.1.4 Minimal Number of Rounds for Single Update .......... 7
      2.1.5 Lower bound of the number of rounds ................ 11
   2.2 Loop-Freedom Enforcement (LF) .................. 12
      2.2.1 Multiple Visits ....................................... 12
      2.2.2 Discrepancy between round minimization and update maximization ........ 13
      2.2.3 Losing the benefit of source differentiation: independent nodes .... 14
   2.3 Waypoint Enforcement (WE) .................... 16
      2.3.1 Topological Approach to WE .............. 16
      2.3.2 Prerequisite for WE-compliant Policy Updates .... 17
      2.3.3 Pausing ............................................. 18
   2.4 Combined Loop-Freedom and Waypoint Enforcement (LFWE) .... 18
      2.4.1 Unsolvable Policy Updates .................... 19
      2.4.2 Always Solvable Policy updates ................ 19
      2.4.3 Ambiguous Policy Updates .................... 19
      2.4.4 Pausing as a speed-up ....................... 20

3 Model ................................................. 21
   3.1 Formalism ............................................. 21
      3.1.1 Policies ............................................. 21
      3.1.2 Updates ............................................. 21
      3.1.3 Rounds .............................................. 22
      3.1.4 Algorithms ......................................... 22
   3.2 Assumptions ........................................... 23
      3.2.1 Clean Start .......................................... 23
      3.2.2 Eventual Consistency ............................ 23
   3.3 Constraints ............................................. 23
      3.3.1 Loop-Freedom (LF) .................................. 24
      3.3.2 Waypoint Enforcement (WE) .................... 24
   3.4 Objectives ............................................. 25
      3.4.1 Main Objective: Minimizing the Number of Rounds .... 25
      3.4.2 Further Objectives .................................. 26

4 Algorithms ............................................. 29
   4.1 Dependencies ........................................... 29
      4.1.1 Update Dependencies ............................ 29
      4.1.2 Topological Dependencies .................... 32
      4.1.3 Dependency Map .................................... 34
# 4.2 Algorithmic Model

- **4.2.1 Algorithmic Definitions**

- **4.2.2 Algorithmic Constraints**

- **4.2.3 Update Management on \( p_2 \setminus p_1 \): Touchability Enforcement**

# 4.3 Letting the Dependency Forest Evolve

- **4.3.1 Embedding WE**

- **4.3.2 Updating the Dependency Forest**

- **4.3.3 Weight System**

# 4.4 Loop-Freedom (LF)

- **4.4.1 \( \text{MAXU-LF Algorithms} \)**

- **4.4.2 Lightweight \( \text{MINR-LF Algorithms} \)**

- **4.4.3 Middleweight \( \text{MINR-LF Algorithms} \)**

- **4.4.4 Heavyweight \( \text{MINR-LF Algorithms} \)**

# 4.5 Waypoint Enforcement (WE)

- **4.5.1 \( \text{WAYUP: MINR-WE with pausing} \)**

- **4.5.2 \( \text{WPWITHOUTPAUSES: MINR-WE without pausing} \)**

# 4.6 Combined Loop-Freedom and Waypoint Enforcement (LFWE)

- **4.6.1 \( \text{LFWPWITHPAUSES: MINR-LFWE with Pausing} \)**

- **4.6.2 \( \text{LFWPWITHOUTPAUSES: MINR-LFWE without Pausing} \)**

# 5 Experimental Results

- **5.1 Optimal Algorithm**

- **5.2 Suboptimal Algorithms**

# 6 Simulation Environment

- **6.1 Design Overview**

- **6.2 Comparator**

- **6.3 Policy Generator**

- **6.4 Heuristic**

- **6.5 Optimal Algorithm**

- **6.6 Future Implementations**

# 7 Discussion

- **7.1 Minimizing Pauses: Cleaning**

- **7.1.1 Local Cleaning**

- **7.1.2 Global Cleaning**

- **7.1.3 Cleaning and Loop-Freedom**

- **7.1.4 Limitations of Cleaning**

- **7.2 Imprecision of the dependency forest’s Lower Bound**

- **7.3 Cost of Cautiousness**

- **7.4 Optimized Hybrid Approximation Scheme**

- **7.4.1 Formal Approximation Process**

- **7.4.2 Avoiding Delays**

- **7.4.3 Approximations with Event-Based Updates**

- **7.4.4 Algorithms with various Precision Levels**

- **7.5 Event-Based Updates**
1 Introduction

Software-Defined Networking (SDN) is a powerful paradigm. Replacing classic destination-dependent routing, SDN allows forwarding rules to depend on various parameters such as the source of the traffic [1] or its protocol, rendering it possible to define forwarding policies with a high precision [2, 3]. In the same time, the simplicity is ensured by the fact that a network administrator only needs to communicate with a single entity, the controller, that translates all their wishes, expressed in a simple language, into a global network policy. The controller then assigns specific forwarding rules at each switch so as to accommodate not only the administrator’s desires but also a range of other network parameters, such as load-balancing or link utilization.

The SDN’s expressiveness also gives a good handle on policy updates. This has made possible the recent emergence of various techniques to ensure consistency properties during a network update [4, 5, 6, 7].

[7] designed techniques to ensure that packets were either routed according to the old policy or to the new policy, but never a mix of both (per-packet consistency). This was realized by tagging incoming packets to signal switches that they should forward them according to the new policy, which implied that if a single switch took more time to update its forwarding table, it delayed the update of all others, arguably an undesirable property.

Tagging requires a computational and temporal overhead and is not always possible in middlebox applications. Hence later work investigated weaker forms of consistency such as ensuring that no packet can stay in a forwarding loop (loop-freedom) [6]. They did not resort to tagging anymore, but scheduled node updates so that loop-freedom was ensured, and were thus able to update some nodes before others.

In this work, we investigate fast network updates in the context of policies that integrate the source in the forwarding rules, resulting in independent routing paths, which allows a significant simplification and speedup of policy updates (see Subsection 2.1).

We consider two constraints, loop-freedom (LF) and waypoint enforcement (WE). Routers remove automatically packets caught in loops, e.g., using TTL, but on networks of switches, the Spanning Tree Protocol (STP) needs to be expressively configured to avoid loops. Thus if STP is not configured, packets caught in loops would stay in the network forever. Moreover, even when mechanisms exist to prevent packets from looping forever, packets caught in loops represent an undesirable load both for the links –consuming most wanted bandwidth [8]– as well as for the switches –consuming computation power–, and packet losses result in a decrease of the delivery performance. If any transient configuration contains a loop, it can last for an arbitrary long time in an asynchronous environment, hence notably damaging the network performance. Thus especially for performance-critical infrastructures, no transient configuration should contain a loop.

Conversely, waypoint enforcement (WE) indicates that no packet should bypass a waypoint during the policy update such as a checkpoint ensuring access control functions (e.g., realized by a middlebox [9], a switch [10], or a NFV [11]). This feature is of particular relevance in security-critical environments.

This work aims at minimizing the update time when an entity with a global view (e.g., a controller) coordinates routing updates of single switches throughout a network when loop-freedom (LF), waypoint enforcement (WE) or both combined (LFWE) are required. To model the speed of a policy update, we introduce the round complexity, where a round is a period of time starting by the controller sending a batch of updates concurrently and finishing when it has received all corresponding update acknowledgments. Minimizing the round complexity is a natural objective given the time necessary to update a single switch [5], and doing so while enforcing specified constraints is a challenge for policy updates [12, 13]. Our key contributions are the following.

1. We highlight and characterize problems and impossibility results regarding LF- and WE-compliant
policy updates, e.g., that under specific circumstances, the controller has no choice but to wait for packets to traverse the whole network before issuing a new update order if WE is to be observed. We also present a trade-off between, on the one hand, waiting a long time between broadcasts of update orders and sending many update orders at a time and, on the other hand, waiting little between broadcasts of update orders but sending only few update orders at a time. Our hope is that future work can lean on these results and make informed decisions.

2. We construct a theoretic framework for making updates quick and consistent. We introduce several forms of dependencies and discuss their potential and computational cost. Moreover, we introduce a wide range of algorithms minimizing the time to update the network, from very simple heuristics with little performance guarantees to algorithms updating the network in a minimal number of rounds. These tools can be reused and expanded in future work.

3. We present an experimental framework that does not only allow for an easy implementation of algorithms but also embed a performance evaluation mechanism. We have already implemented several algorithms, whose tests we present, but the modular architecture of the experimental framework allows for an easy development and testing of further solutions.

4. We sketch out some possibilities to make the model more flexible. We abstract from the notion of rounds and introduce algorithms updating the network at an optimal speed by sending updates as soon as they can, instead of waiting for all acknowledgments. At times, relaxing some of the specified constraints can be a good solution, e.g., if switches cannot be updated in a timely manner and it has acceptable consequences.

The remainder of this work is organized as follows. In Section 2, we introduce the reader to the problems of updating a network under LF and WE, and highlight the benefits of a source-based routing. Then, in Section 3, we define our working model, namely the objectives, the most important one being the minimization of the update duration, and constraints (LF, WE, LFWE) on which we focus. Subsequently, Section 4 provides the reader with some algorithms updating any network under one of the three constraints and optimizing speed at various levels, after which Section 5 presents the performance of some of them with regard to round minimization. The experimental design used to realize the tests is described in Section 6. Finally, we present further insights and improvements in Section 7, give an overview of related work in Section 8 before providing concluding remarks in Section 9.
2 Motivations, Simplifications, and Overview

This section gives an overview of the problems related to synchronizing routing updates of individual switches, e.g., when a controller changes the policy in a software-defined network.

Three objectives are considered, round minimization (MINR), maximization of the number of updates per round (MAXU), and round minimization for updating a specific node (MINRSINGU). The focus is laid on MINR, since the number of rounds is directly correlated with the duration of the update, a most critical aspect in a policy update. Reversely, this section reviews in turn three constraints, loop-freedom (LF), waypoint enforcement (WE), and the combination of both (LFWE).

2.1 Path Differentiation

While some models differentiate only on the final destination of packets, we take additionally into account the source information. Thus we consider policies in function of paths (source and destination). This allows for essential simplifications.

2.1.1 Focus Reduction

Standard routing is destination-based. For a given flow (entry in the flow table), a policy update could then for example look as shown in Figure 1. The full line is the old policy $p_1$, the dashed line is the new policy $p_2$, dashed circles represent legacy (non-SDN) switches, and full circles are SDN switches. This update seems complex and hard to coordinate, but can be simplified without any information loss in six steps.

![Figure 1: Example of a policy update in a network. solid line: old policy $p_1$, dashed line: new policy $p_2$, full circles: SDN switches (nodes), dashed circles: legacy (non-SDN) switches](image)

1. Any switch that is not touched by the policy update needs not be considered, resulting in Figure 2.
2. Additionally, as standardly done in literature (see Section 3), the network view is abstracted into the controller view. Legacy switches –not SDN switches– cannot be updated by the controller and
Figure 2: Policy update after removing unchanged switches. *solid* line: old policy $p_1$, *dashed* line: new policy $p_2$, *full* circles: SDN switches (nodes), *dashed* circles: legacy (non-SDN) switches

physical links between switches should be abstracted in logical links between SDN switches. With this abstraction, the network looks as shown in Figure 3.

Figure 3: Reduction of the node set to SDN-enabled switches

3. Going further into simplifications, the network can be split into forwarding trees where a node represents a switch and it points at its father if it has a forwarding rule to it. The focus is in turn laid on a specific destination and switches not forwarding packets to it are kept out of scope. In the example, $v_2$ is selected, simplifying the network as showed in Figure 4.

4. The next step in this filtering process consists in not considering a destination-based routing anymore, but a routing mindful both of the source and the destination, forward rules specify then not only the destination but also the source for which they apply. This work makes this differentiation despite the minor computational overhead it implies because it simplifies the updating problem. The result is displayed in Figure 5.
Figure 4: Reduction of the node set to a specific destination (here, $v_1$), with a single old policy and a single new one

Figure 5: Reduction of the node set to a specific source (here, $v_1$), with a single old path and a single new one

5. Furthermore, the nodes either on the old path to the destination or on the new one but not on both are easy to handle. They cannot influence loop freedom or waypoint enforcement: they either implement a rule, or don’t, and then drop any incoming packet. Only the nodes on both paths can play a role regarding these constraints, focusing on them results in Figure [6]

6. Finally, for readability, the nodes on both the old and the new paths are represented in their order along the old one, resulting in Figure [7]. For simplicity, in the remaining of this work, this representation will be used.

To conclude, a problem that appeared complex from a switch perspective, and seemed to involve a high number of variables, could be put into a much more comfortable and manageable form. The result of these successive simplifications is called the line model.
2.1.2 Removing Dependencies with Path Differentiation

Differentiating both on sources and destinations can remove dependencies compared to a simple destination differentiation. Consider the policy update in Figure 8. Without differentiating sources, updating all switches concurrently would create a loop, to ensure that no loops appear in a transient network state, the controller would need to synchronize update orders carefully.

On the contrary, the same update with source differentiation is trivial. No interaction exists, and all nodes can be updated concurrently.

2.1.3 Problem Partitioning

The line model allows enforcing specific constraints to be easily split into independent subproblems, drastically simplifying the computation. Solving optimally the problem as a whole is then equivalent to solving it optimally separately on several sections, independent clusters.

Definition 1. An independent cluster is a set of nodes such that their update schedule does not constrain the update schedule of nodes outside of the independent cluster.

Figure 9 illustrates a partition of a network into independent clusters in the frame of a particular policy update.

Without source differentiation, computing independent clusters is expensive. As many nodes can point to a unique one, all possible subsets must be checked.

On the contrary, source differentiation allows to detect independent clusters simply by looking at every node and testing whether it precedes a cut (independent cluster boundary, see Definition 2). All independent clusters...
Figure 8: Example of a policy update for which a loop can appear with destination differentiation but not with path differentiation. **solid** line: old policy, **dashed** line: new policy. Old path for source $v_4$: $[v_4, v_3, v_2, v_5]$, old path for source $v_1$: $[v_1, v_5]$, new path for source $v_4$: $[v_4, v_5]$, new path for source $v_1$: $[v_1, v_2, v_4, v_5]$.

Figure 9: Policy update exhibiting several independent clusters. **rectangle**: independent cluster, **bold line**: cut

clusters are between two *cuts*. Only $n$ pairs of subsets $[1, i]$ - $[i + 1, n]$ for $i$ in $[1, n]$ – need to be considered, which is asymptotically done in $O(n)$ time.

**Definition 2.** A cut exists between two successive nodes $v_i, v_{i+1}$ if for all $j, j'$, $j \leq i$ and $j' \geq i + 1$, $v_j$ is before $v_j'$ both on the old and the new path.

2.1.4 Minimal Number of Rounds for Single Update

In addition to the advantages presented above, the line model enables an easy computation of a shortest round sequence to update a specific node under the loop-freedom (LF) constraint.

**Backward Dependency** To understand the methods presented in this work, the notion of *backward dependency* needs to be introduced. A node $v$ *backward*-depends on a node $v'$ if updating $v'$ singularly would make $v$'s update safe (not creating loops), but the backward dependency has a broader meaning, as can be seen in Definition 3 below.

**Definition 3.** Let $d_o(v)$ and $d_n(v)$ denote the nodes to which the old, respectively new edges of $v$ point. A node $v$ backward-depends on $v'$ if and only if $v$ is backward, $v'$ is between $v$ and $d_o(v)$ on the old path, and $d_n(v')$ is not between $v$ and $d_o(v)$ on the old path.
For simplicity, a new edge is called \textit{forward} if its destination is after its source on the old path (i.e., on the right), else \textit{backward}. Similarly, a node \(v\) is \textit{forward} if its new outgoing edge is \textit{forward}, respectively \textit{backward}. With these definitions in mind, the updatability when all nodes still implement the old policy is characterized as follows.

\textbf{Lemma 1.} \textit{At the beginning of the first round, any node} \(v\) \textit{is either forward, and immediately updatable, or backward, and then }\(v'\) \textit{depends on at least one node.}

\textit{Proof.} If a node is \textit{forward}, then updating it singularly cannot create a loop. If a node is \textit{backward}, then it cannot be updated, but it then \textit{depends} on another node, as otherwise the new path would violate LF, and the problem would be unsolvable.

Moreover, a dependency forest (\(DF\)) can be constructed based on the backward dependency, as described below in Definition \(4\). As a side note, leaves of \(DF\) depend on no other nodes, and are thus immediately updatable (and forward). \(DF\) can be built for other types of dependency than the backward dependency, as seen in Section \(4.1\). However, in the remaining of this work, if not stated otherwise, the dependency considered is always the backward dependency.

\textbf{Definition 4.} A dependency forest (\(DF\)) is a set of nodes and dependency relationship (pairs of nodes). A node’s children are the nodes on which it depends.

\textbf{Updating a single node} \quad \textit{Unfortunately, updating a single node in each round along any path from a leaf to a given node is not automatically loop-free. As an illustration, consider Figure 10, the excerpt of a policy update shown in its integrity in Figure 12. The corresponding dependency trees are displayed in Figures 11 and 13 respectively. The triangle represents the waypoint. Counter-intuitively, following the single path \([v_8, v_{10}, v_{12}]\) on the dependency forest violates LF. The shortest path to a node on the dependency forest (\(DF\)) is thus not simply a shortest round sequence.}

![Figure 10: Excerpt of a policy update in which following a dependency path ([v_8, v_{10}, v_{12}]) violates LF. Corresponding (initial) dependency forest in Figure 11](image)

However, although both round sequences \([v_8, v_{10}, v_{12}]\) and \([v_5, v_6, v_7, v_9, v_{12}]\) create loops, \([v_2, v_3, v_4, v_{11}, v_{12}]\) is loop-free. More generally, \(DF\) presents a very valuable property: For updating a single node \(v\) in a loop-free manner, there is always a valid loop-free round sequence that is a path on the \(DF\) from a leaf to \(v\) (Lemma 2). Accordingly, one only needs to find the shortest path on \(DF\) to have a shortest round sequence for updating a specific switch.
Figure 11: Excerpt of dependency forest in which a dependency path \([v_8, v_{10}, v_{12}]\) is not reliable (corresponds to the policy update of Figure 10), a node points at another node if it depends on it (according to one of the dependencies defined in section 4.1).

Figure 12: Policy update displaying a dependency path that cannot be followed from bottom to top: updating in turn \(v_8, v_{10}\) or \(v_5, v_6, v_7, v_9\) and \(v_{12}\) violates LF, although they are parents of one another. Corresponding initial dependency forest in Figure 13.

**Lemma 2.** For any node \(v\), there is a path in the backward dependency forest that is a loop-free round sequence to update \(v\).

**Proof.** Let \(v\) be a node.

If \(v\) is forward (e.g., if it is the source), it is a leaf and \(\{v\}\) is trivially a valid dependency path.

If \(v\) is backward, let us assume there is no dependency path from a leaf to \(v\) on the backward-dependency forest, namely that the chain of all backward edges contains an interruption (a hole). Equivalently, there is a node \(v_e\) such that no backward edge starting between \(v_e\) and \(v\) points before \(v_e\) and no forward edge starting between \(v_e\) and \(v\) points after \(v\). The new path is then so that by starting at \(v\), there is no way to go beyond \(v\), which is a contradiction since the new path is supposed to go from the first node of the line to the last node of the line going over \(v\).

**Theorem 1.** In the line model, the problem of finding a round sequence to update a single node \(v\) can be solved in \(O(n^2)\).

**Proof.** Let \(v\) be a node to update. In a first step, we compute the dependency forest (DF), which is done by going over the line from right to left and each time hanging as children of \(v_i\) the nodes between \(d_o(v_i)\)
Figure 13: Dependency forest corresponding to the policy update of Figure 12.

and $v_i$ whose destinations are not between $d_o(v_i)$ and $v_i$. This has a computational cost of $O(n^2)$. Then, we operate a breadth-first search starting from $v$ to determine all dependency paths of $v$ ($O(n^2)$). Subsequently, we select the first valid dependency paths ($O(n)$). 

As an illustration, take a look at Figures 14 and 15. In the first graph, we differentiate policies on sources and destinations, and in the second on destinations only. In both cases, $v_{13}$ is backward and requires previous updates to become updatable. As stated in Theorem 1, the line model finds a solution to $\text{MINRSINGU}$ in $O(n^2)$. On the contrary, a solution to $\text{MINRSINGU}$ is not trivially found, since it requires testing combinations of updates.

Figure 14: Policy updates differentiating on sources and destinations
Initial Concurrent Updates Initially, all nodes implement the old policy, and updatable nodes are forward. Updating all of them concurrently cannot violate LF because no backward edge is created. This is expressed in the Lemma 3 below.

**Lemma 3.** *Updating all forward nodes in the first round does not violate LF.*

*Proof.* A cycle can only appear if a node \( v \) starts pointing at a node \( v' \) that is currently forwarding packets to \( v \). This case cannot happen if only forward nodes are updated. \( \square \)

### 2.1.5 Lower bound of the number of rounds

The dependency forest also has another valuable property, once more regarding LF. As seen above (Section 2.1.4), any node \( v \) can be updated singularly in a minimal number of rounds by following a dependency path from a leaf up to \( v \). Thus if \( h(v) \) (height of \( v \)) denotes the length of the shortest dependency path of \( v \) —the height of a leaf being 1—, then the controller needs at least \( h(v) \) rounds to update \( v \). Accordingly, to update the whole network without creating loops, the controller needs at least a number of rounds equal to the maximal height of the dependency forest. In conclusion, the maximal height of the dependency forest (\( DF \)) is a lower bound of the minimal number of rounds (\( minR \)), i.e., \( minR \geq \max_{v \in DF} h(v) \). Interestingly, this lower bound is sometimes reached, e.g., in configurations as shown on Figure 16 but the minimal number of rounds can also be much greater than the maximal height, as discussed in Section 7.2.

![Figure 15: Policy updates differentiating on destinations only](image1)

While the source differentiation allows the computation of such a lower bound, it has no equivalent
without it. This is due to the inability of the backward dependency to capture all relevant LF-relevant information as soon as updates start, an issue discussed in Section 2.2.3.

2.2 Loop-Freedom Enforcement (LF)

Updating a policy under the loop-freedom constraint, LF, is always feasible. In the worst-case, the controller updates one node at a time from the sink (last node of the path) to the source (first node of the path) along the new path. However, beyond this apparent simplicity, LF hides intriguing non-trivial properties.

2.2.1 Multiple Visits

LF means that no transient network configuration contains a loop. Naturally, one could think that LF is then equivalent to no packet passing more than once through a single node. However, counter-intuitively, this intuition does not hold.

Consider for example an old path \([v_1, v_2, v_3, v_4]\), with respective new destinations \([v_3, v_4, v_2]\) (\(v_4\) is the sink and has thus no destination). Let us assume the controller updates \(v_1\) and \(v_4\) in the first round and \(v_2\) in the second, such that no transient configuration contains a loop. If a packet enters the link \([v_2, v_3]\) before \(v_1\) becomes new but is received by \(v_3\) after its update, it is sent back to \(v_2\), thus visiting \(v_2\) twice—without any loop existing in the network at any time. Going further, the Lemma 4 holds.

**Lemma 4.** In a loop-free round sequence, a single packet can visit up to \(n/4\) times the same node.

**Proof.** As an illustration, consider Figure 17. Only relevant edges are represented, namely long-ranging backward edges that contain each other and forward edges starting each right before the source of the long-ranging backward edges. Edges starting at the destinations of long-ranging backward, respectively forward edges, are not represented, but they are assumed to be such that the new policy comprises a path. A packet leaving the source and being forwarded only according to the new policy should thus go through all links exactly once.

![Figure 17: Excerpt of a policy update where a node (here, \(v_4\)) is visited \(n/4\) times](image)

Let us assume that the controller firstly ignores the unrepresented edges, and updates (ordering from right to left) in the \(2i - 1\)th round the \(i\)th forward edge, and in the \(2i\)th round the \(i\)th backward edge. After all represented edges have been updated, the controller updates the unrepresented ones in a loop-free manner. This round sequence clearly respects LF.

However, counter-intuitively, a packet can then reach the \(i\)th forward edge when it is old, the \(i\)th backward edge when it is new, and hence go through the middle point \((v_4)\) \(r\) times, where \(r\) is the number of backward edges. Finally, since the backward edges represent a fourth of the edges of the new policy, \(r = n/4\). \(\square\)
2.2.2 Discrepancy between round minimization and update maximization

Interestingly, performing well regarding the maximization of node updates per round, \( \text{MAXU} \), does not necessarily go along with performing well with regards to the minimization of the number of rounds, \( \text{MINR} \):

**Lemma 5.** *In the worst case, maximizing the number of updates in each round leads to a performance rate of \( n/10 \) for \( \text{MINR} \).*

*Proof.* This worst case holds for very specific policy updates, repetition of the *unitary pattern* represented in Figure 18. Let nodes pointing at nodes two hops away on the right be called *bridge*, \( r \) is the number of *unitary patterns* in a network.

![Figure 18: Unit pattern used in the Lemma 5](image)

The global network looks as indicated in Figure 19.

![Figure 19: Policy Update of Lemma 5](image)

A round sequence maximizing at each round the number of updates would update in the first round all forward nodes, including the bridges. In the second round, the line would simply be equivalent to a sequence.
of 2r backward edges pointing at each other, which is why 2r more rounds would be necessary to update the whole network.

On the contrary, a round sequence of minimal length would discard the bridges in the first round and only update the forward long-range edges. In the second round, it updates all backward edges entering a unitary pattern. Then, the remaining backward edges follow in the third round. Finally, in the fourth round, all bridges are updated. Thus, five rounds are required to update the network.

$r$ now remains to be expressed in function of $n$. Excluding the source ($v_1$), the node following it ($v_2$), and the sink (last node), the line contains $r$ unitary patterns, one of them missing a long-range backward edge. Since a unitary pattern consists of 5 nodes, $n - 3 = 5(r - 1) + 4r$, which can be rewritten for large $n$ as $r = \frac{n}{5}$. Accordingly, maximizing the number of updates at each round increases the number of rounds by a factor of $\frac{2r}{r} = \frac{n}{10}$ in comparison with the minimum. \square

**Definition 5.** Let $v$ be a backward node. The ways out for $v$ are the nodes $v'$ meeting the following two conditions simultaneously:

1. $s_n(v) < v' < v$
2. $s_n(v') < s_n(v)$ or $v' < s_n(v')$

This problem is one of the numerous manifestations of the difficulty of selecting at each round the "good" nodes when we don’t have the possibility to come back on our decisions. Indeed, if whenever we select a node $v$, we can keep computing a round sequence, and when we notice that it is going to be too long, come back to the point where we selected $v$ and select a node $v'$, we would surely obtain at the end a shorter round sequence, but we do not allow ourselves such trial-and-error techniques because they would incur an exponential cost.

Unfortunately, as the example presented in this subsection showed, computing an optimized round sequence based solely on the information currently available is a difficult problem. Indeed, every time ways out are available, it is hard to decide whether they are “good” ones, i.e., discarding them would automatically cost more rounds afterward, from "bad" ones which are detrimental if they are updated. Moreover, while updating too many nodes can have negative consequences, so does taking too few of them. For example, Algorithm LFSIMPDEP can take $n/6$ times more rounds than necessary (cf. Section 7.3).

### 2.2.3 Losing the benefit of source differentiation: independent nodes

As discussed so far, source differentiation offers valuable information before the network update begins. Valuable dependencies are easily computable, can be used to update singular nodes quickly, and a lower bound can be computed. Most importantly, all updatable nodes –forward nodes– can actually be updated concurrently. However, these convenient characteristics unfortunately do not last after the first round.

Indeed, after the first update acknowledgment is received by the controller, the network is not a line anymore. The original path $p_1$ splits into a number of subpaths, each having a different source and the same sink.

Interestingly, if two subpaths merge at a node $v$, any node of one subpath before $v$ whose new outgoing edge points at a node on the other subpath before $v$ can always be updated loop-free singularly. Accordingly, such nodes are called independent. It should be noted that any node to which no other node points is automatically independent.
Unfortunately, a node's status may evolve. If it was independent, it may become forward or backward in the next round. For example, consider two subpaths \( p_a, p_b \) and a node \( v_1 \) on \( p_a \) such that \( d_n(v_1) \leq p_b \). If, within the current round, \( v_1 \) remains old but a single other node \( v' \) is updated, two interesting cases emerge:

1. \( v' \in p_b, d_n(v') \leq_{p_a} v_1 \)
2. \( v' \in p_a, v' >_{p_a} v_1 \) and \( d_n(v') \leq_{p_b} v_2 \)

In the first case, \( v_1 \) automatically becomes backward (and thus not updatable anymore) in the next round. In the second case, \( v_1 \) automatically becomes forward (and thus still updatable) in the next round.

Likewise, forward or backward nodes can also become independent. Consider \( v_1, v'_1 \) on \( p_i \), with \( v_1 \) forward or backward. Assume \( v'_1 \) on \( p_i \) is between \( v_1 \) and \( d_n(v_1) \) is not backward with \( d_n(v'_1) \) not between \( v_1 \) and \( d_n(v_1) \). In this configuration, updating \( v'_1 \) makes \( v_1 \) independent.

**Definition 6.** Let \( p_a, p_b \) be two subpaths with a point of junction \( v_{12} \). \( p_a \) and \( p_b \) are said to display a **x-structure** if two nodes \( v_1, v_2 \) on \( p_a, \) respectively \( p_b, \) exist such that the following hold:

\[
\begin{align*}
d_n(v_1) &\leq_{p_b} v_2 <_{p_b} v_{12} \text{ and } d_n(v_2) \leq_{p_a} v_1 <_{p_a} v_{12}
\end{align*}
\]

However, this freedom is short-lived: an independent update is always loop-free, but this does not hold for the combination of two independent updates, when they form the structure x-structure (Definition 6, example shown in Figures 20 and 21). Indeed, updating \( v_1 \) and \( v_2 \) simultaneously would then violate LF.

Additionally, one independent update can even prevent more than a single other independent update. For example, consider \( v_1..v_c \) are at the beginning of a given round so that \( v_1 \) displays a x-structure with each \( v_l, \ l \in [2, c] \) (note that on the current policy, a single path thus goes through all \( v_l, \ l \in [2, c] \)). Updating \( v_1 \) in this round would then prevent all \( v_l, \ l \in [2, c] \) from updating in this round as well. To capture dependencies between independent updates, one may analyze path dependencies, but this is left for future work.

It should be noted that after the updates begin, the network architecture is very much as if no source differentiation would exist, but differs in the fact that the in-coming degree of any node is zero, one, or two, while it is unbounded in the **destination model**.

---

**Figure 20:** Policy update showing a **x-structure**, before any update. **solid** line: old policy \( p_1 \), **dashed** line: new policy \( p_2 \), **full** circles: SDN switches (nodes)
2.3 Waypoint Enforcement (WE)

Contrary to LF, the waypoint enforcement constraint, WE, is not always enforceable. For some policy updates, no scheduling can enforce WE.

2.3.1 Topological Approach to WE

The waypoint enforcement constraint, WE, states that no packet in the network should be able to reach the sink without having passed through the waypoint at least once. Unfortunately, this constraint cannot be verified by simply looking at all possible transient policies separately and making sure the waypoint is always on the path from the source to the sink. As an example to illustrate this problem, consider the policy update in Figure 22. The round sequence \([v_1, v_3, v_2]\) — updating \(v_1\), then \(v_3\), and finally \(v_3\) — creates two transient states, both having the waypoint on the path from the source to the sink. However, this round sequence would violate WE: a packet could go from the source to \(v_1\) when the network enforces \(p_1\), and then from \(v_1\) to sink. Accordingly, one should always be mindful of the past transient policies when selecting WE-consistent updates.

To express formally the condition that makes a round-sequence WE-consistent, the following sets are introduced.

1. The infected set \(I\) is the set of nodes that have potentially been on a path from the source to the waypoint.

Figure 22: Example of a policy update violating WE. solid line: old policy \(p_1\), dashed line: new policy \(p_2\), full circles: SDN switches (nodes), triangle: waypoint

To express formally the condition that makes a round-sequence WE-consistent, the following sets are introduced.

1. The infected set \(I\) is the set of nodes that have potentially been on a path from the source to the waypoint.
2. The safe set $S$ is the set of nodes that have certainly never been on a path from the source to the waypoint.

3. The end set $E$ is the set of nodes potentially currently on the path from waypoint to the sink.

**Theorem 2.** A round sequence is WE if and only if $E \in I$

**Proof.** If a node is in the end set, it has the possibility to send packets directly to the sink—without them visiting the waypoint. If a node is in the infected set, it can receive unchecked packets. Thus, if a node is in both sets, it can send an unchecked packet directly to sink, the waypoint can be bypassed. \(\square\)

To keep these sets up to date, at each round, all nodes that can be between the source and the waypoint according to a transient policy are added to the infected set ($I$), namely taken away from the safe set ($S$).

### 2.3.2 Prerequisite for WE-compliant Policy Updates

WE cannot always be guaranteed. In trivial cases, WE is even ensured by updating all nodes of the network concurrently. For example, if no node before (after) the waypoint has a new edge to a node after (before) the waypoint, updating all nodes concurrently will only make packets stay on the section of the old path before the waypoint then eventually go through the waypoint, stay for some time in the section of the old path after the waypoint and finally reach the end. It should be noted that a packet may then meet loops but this is out of the scope of this section (see Section 2.4 for LF and WE combined).

However, if one considers a setting as simple as that of Figure 23, WE is already impossible to ensure: No matter how much time is left between $v_1$’s update and $v_2$’s, a packet having left $v_1$ before its update can always theoretically stay long enough on the link $[v_1, v_2]$ to arrive at $v_2$ after its update. It would thus directly reach $v_4$, bypassing the waypoint ($v_3$). The same impossibility result can be seen from a topological perspective: $v_2$ is infected right from the beginning, but it needs to be safe at the end, which is impossible, since no infected node can become safe.

![Figure 23: Policy update where waypoint enforcement is impossible](image)

**Theorem 3.** A WE-compliant policy update is possible if and only if $p_2^{\text{wp}} \cap p_1^{\text{wp}} = \emptyset$

**Proof.** Let us first prove that the above condition is necessary.

For convenience, the notations $p_1^{\text{wp}}$ ($p_1^{\text{wp}}$) are used to indicate the section of the old path before (after) the waypoint. $p_2^{\text{wp}}$ and $p_2^{\text{wp}}$ are defined likewise on the new path.

Initially, the infected set $I$ is equal to $p_1^{\text{wp}}$ since this is the section before the waypoint. At the end, $p_2^{\text{wp}}$ must be in the safe set for WE. Since any node in $I$ at some point remains infected throughout the update, if any node of $p_2^{\text{wp}}$ is in $p_1^{\text{wp}}$, it cannot be safe at the end. Thus, the condition is necessary for WE.
Now, let us prove that this condition is sufficient, i.e., that a WE-consistent solution always exists if it is fulfilled.

Assuming the condition, let us justify that a round sequence updating in the first round $p_2^{wp}$ and the waypoint and in the second $p_2^{<wp}$ is indeed WE-consistent. In the first round, all nodes from the end to the waypoint are updated, including the waypoint itself, and since they are all on $p_1^{wp} \cap p_2^{>wp}$, they are all safe nodes.

Accordingly, both the safe set $S$ and the infected set $I$ remain unchanged, and only safe nodes are added to the end set $E$, which is safe according to the second formulation of Theorem 2. Subsequently, we update the nodes of $p_2^{<wp}$, thus making the nodes $p_2^{<wp} \cap p_1^{wp}$ migrate from the safe set to the infected set, however this is safe since $E = p_2^{wp}$ and $I = p_1^{wp} \cup (p_1^{wp} \cap p_2^{<wp})$, leading to $E \cap I = (p_2^{wp} \cap (p_1^{wp} \cup (p_1^{wp} \cap p_2^{<wp}))) = p_2^{wp} = \emptyset$. Thus, the condition is sufficient for WE.

2.3.3 Pausing

As discussed in the previous section (Section 2.3.2), under specific circumstances, no update schedule can be found to enforce WE. However, WE can still be rendered possible by letting the controller wait a specified amount of time, long enough to ensure no packets remain on links that are not used anymore. We call this technique pausing: after having received all node acknowledgments of a round $r_i$, the controller first waits a duration $\Theta$ and only then sends the update orders of round $r_{i+1}$. $\Theta$ is a given duration large enough so that no packet having entered the network before the controller starts waiting remains therein when it stops doing it. Pausing "cleans" the network. After this waiting, the infected set only contains the nodes currently on the path from the source to the sink, and all nodes previously infected but not on this path do not risk anymore to receive an unchecked packet, and are thus deemed safe.

For example, in a policy update as shown in Figure 24, $v_1$ can be updated in a first round, then the controller should do a pause, and $v_2$ and $v_3$ can then be updated concurrently. This schedule ensures WE by making use of pausing, and no schedule could update the network in a WE-consistent manner without it, since the condition of Theorem 3 is violated.

[Figure 24: Policy update requiring a pause]

Mechanisms to realize pausing and other forms of network cleaning are formally described and discussed in Section 7.1.

2.4 Combined Loop-Freedom and Waypoint Enforcement (LFWE)

This section investigates the combined constraint of loop-freedom and waypoint enforcement, LFWE. Naturally, since LFWE is a stronger constraint than WE alone, any policy update impossible to realize under the WE constraint also forbids LFWE, but some policy updates that are WE-consistent are not LFWE-consistent.
2.4.1 Unsolvable Policy Updates

Unfortunately, in some cases, no LFWE solution exists at all. For any succession of valid rounds and pauses, the controller will always reach a dead end, and not be able to implement the new policy. These configurations are called unsolvable. As an illustration of a policy update forbidding LFWE, consider Figure 25. Updating a backward node would violate LF, while going for a forward node would violate WE. No single update is possible, and, by extension, the controller cannot update the whole policy.

![Figure 25: Policy update impossible to realize without a packet either bypassing a waypoint or taking a loop](image)

Unfortunately, unsolvable cases are hard to characterize. For instance, Figure 26 does not display the same structure, but the controller can only update $v_1$ and $v_2$, and then $v_3$, after which the same structure as in Figure 25 appears, and the controller cannot update any node anymore. This case is thus also unsolvable.

![Figure 26: Policy update impossible to realize without a packet either bypassing a waypoint or taking a loop](image)

2.4.2 Always Solvable policy updates

LFWE is not always hard to maintain. For some policy updates, as long as the controller executes a succession of valid rounds and pauses, the network will always eventually reach the new policy. These configurations are called always solvable. For example, if $p_1^{<wp} \cap p_2^{<wp} = p_1^{>wp} \cap p_2^{<wp} = \emptyset$, then the problem can be split into the two independent subproblems of updating $p_1^{<wp}$ and $p_1^{>wp}$ LF, without any care necessary to enforce WE. Since LF is always enforceable, LFWE does not pose any problem then.

always solvable cases do not always exhibit such a trivial structure. For example, Figure 27 is clearly also an always solvable case but $p_1^{>wp} \cap p_2^{<wp}$ is not empty. In the first round, only $v_2$ can be updated (under the LFWE constraint), so that the controller will update it. Then, $v_3$ can be updated, and finally $v_1$.

2.4.3 Ambiguous Policy Updates

Finally, in some cases, depending on the controller’s actions, it may or may not succeed to implement the new policy. Two cases can occur:
1. If the controller makes wrong choices, it will end up in a dead end, i.e., no further nodes will be updatable.

2. If it executes the right updates, it can fully update the network.

As an illustration, consider Figure 28. If the controller starts by updating concurrently $v_1$ and $v_2$, then as soon as $v_1$ and $v_2$ are new, the network exhibits the same structure as Figure 25 so that the controller has reached a dead end. On the contrary, $\{v_2\}, \text{pause}, \{v_3, v_4\}, \{v_5, v_6\}, \{v_1\}$ is a valid round sequence.

2.4.4 Pausing as a speed-up

Pausing not only makes otherwise unsolvable problems solvable, but it can even allow for quicker policy updates. Consider Figure 29 where the new edge of the source points at the waypoint. Updating this policy under the LFWE constraint takes only three rounds with pausing. $v_2$ is updated (first round), a pause is then made, after which nodes after $v_2$ are updated (second round), and $v_1$ is updated (third round). Some loops are created in the second round, but no packet can take them thanks to the pause. Thus they are innocuous. Reversely, without pausing, the controller updates $v_2$ (round 1), then all nodes following $v_2$ one by one from left to right, and finally $v_1$. This takes $n - 1$ rounds.

However, pausing has limitations, and depending on the network, $n - 3$ rounds with no pauses may be quicker than three with a pause. Pausing requires that the time for an update to traverse a link is bounded, and since this hypothesis can be hard to ensure in practice, pausing is only used when necessary in the remainder of this work. Considerations on implementations of pausing, improvements and limitations can be found in Section 7.1.4.
Figure 29: Policy update illustrating the potential benefit of making a pause. With a pause, policy update done in three rounds, without a pause, in $n - 1$ rounds.

3 Model

This section formally defines the model considered in this work, as well as the constraints and the objectives analyzed.

3.1 Formalism

This section defines formally the different entities considered in this work.

3.1.1 Policies

Definition 7. A forwarding rule $r$ for a flow $f$ attached to a node $v$ indicates to which node ($v'$) $v$ should forward packets of the flow $f$. It is identified to the tuple $(f, v, v')$.

Each node $v$ of the network is assigned a set of forwarding rules, making up the forwarding table of $v$.

Definition 8. A path $\pi$ is an ordered set of nodes. The node following (preceding) $v$ on $\pi$ is noted $d_p(v)$ ($s_p(v)$).

Node intervals on paths include both boundaries, a node between $v_1$ and $v_2$ is $v_1$, $v_2$ or is after $v_1$ and before $v_2$.

Definition 9. A policy $p$ is a set of forwarding rules.

The controller makes the network migrate from an old policy $p_1$ to a new policy $p_2$, going through various transient policies. When the network is in a transient policy $\tilde{p} \subset (p_1 \cup p_2)$, old rules are the rules in $\tilde{p} \cap p_1$, and new rules are those in $\tilde{p} \cap p_2$.

3.1.2 Updates

Definition 10. For an old policy $p_1$ and a new policy $p_2$, the update $u$ of the node $v$ is the function that maps any (transient) policy $\tilde{p}$ to a policy $\tilde{p}'$. $u$ can be of three types.

1. **rule deletion.** The forwarding rule $r$ in $\tilde{p}$ attached to $v$ is such that $\tilde{p} \setminus r = \tilde{p}'$.

2. **rule addition.** The forwarding rule $r'$ in $\tilde{p}'$ attached to $v$ is such that $\tilde{p}' \setminus r' = \tilde{p}$.

3. **rule replacement.** The forwarding rules $r, r'$ in $\tilde{p}', \tilde{p}'$ attached to $v$ are such that $\tilde{p} \setminus r = \tilde{p}' \setminus r'$.
3.1.3 Rounds

Whenever the controller sends an update order to a node \( v \), it is only sure that \( v \) implements the new policy when it receives \( v \)'s update acknowledgment. During the time between the emission of the update order and the reception of the acknowledgment, the update time of \( v \), \( v \) can be either old (implement the old policy) or new (implement the new policy), we call this state the limbo state.

**Definition 11.** A round is a period of time during which a subset of nodes gets updated concurrently.

During a round, the controller issues simultaneously a batch of update orders and waits for all their acknowledgments. All the nodes to which updates are sent during a round are in limbo, resulting in many possible transient policies (\( 2^n r \), where \( n_r \) is the number of nodes updated during the round). Thus to be sure that the network observes constraints at all times, the controller must choose nodes updated in a round so that no possible transient policy violates them.

**Definition 12.** Let \( p_1, p_2 \) be an old and a new policy.

A round sequence from \( p_1 \) to \( p_2 \) is an ordered set of rounds such that if the network implements initially \( p_1 \) and the controller executes the rounds in turn, the network implements \( p_2 \).

3.1.4 Algorithms

Algorithms considered in this work are qualified as follows.

**Definition 13.** For a set of constraints \( C \), an old and a new policy \( p_1 \) and \( p_2 \) are \( C \)-compliant if there exists a round sequence from \( p_1 \) to \( p_2 \) obeying \( C \). An algorithm is valid with respect to \( C \) if for any \( C \)-compliant old and new policies, it computes a round sequence from \( p_1 \) to \( p_2 \) obeying \( C \).

**Definition 14.** An algorithm is optimal with respect to a set of constraints \( C \) and an objective \( o \) if, for any two (old and new) policies \( p_1, p_2 \) such that a round sequence from \( p_1 \) to \( p_2 \) obeying \( C \) exists, it computes a round sequence from \( p_1 \) to \( p_2 \) obeying \( C \) and fulfilling \( o \).

It should be noted that optimality is a property stronger than validity. Any valid algorithm non-optimal is suboptimal. Constraints and objectives are presented in Sections 3.3 and 3.4.

**Definition 15.** Let \( v \) be a node and \( \{ r_j, 1 < j \leq i - 1 \} \) a round sequence. \( v \) is said to be updatable in round \( r_i \) if there exists a round sequence \( R = \{ r_j, 1 < j \leq l \} \) with \( i \leq l \) also from an old policy \( p_1 \) to a new \( p_2 \) fulfilling a set of constraints \( C \) but such that:

1. For any \( i \leq i, \) the same nodes are updated in \( r_i \) and in \( r'_i \).
2. \( v \) is updated in \( r'_i \).

Finally, the terms backward and forward are introduced to describe a relationship between an update rule and the previous policy:

**Definition 16.** A backward rule is a rule from a node \( v \) to a node \( v' \) such that \( v' \) is after \( v \) on both the old and the new paths. A node having a backward new rule is thereafter called a backward node.

**Definition 17.** Similarly, a forward rule is a rule from a node \( v \) to a node \( v' \) such that \( v' \) is after \( v \) on both the new path but before \( v \) on the old path. A node having a forward new rule is thereafter called a forward node.
3.2 Assumptions

This section formulates the set of assumptions on which the problem considered in this work relies.

3.2.1 Clean Start

For simplicity, we assume that no forwarding rules remain from a previous policy not completely updated, and that no packet is being processed according to a previous policy, clean start (Assumption 1).

Assumption 1. No packet on the network is being forwarded according to a forwarding rule not in \( p_1 \). Additionally, no forwarding rule belonging to a policy prior to \( p_1 \) is implemented by any switch in the network.

Thanks to these two assumptions, the network routing consists at the beginning of the policy update and at its end in a single path defined by \( p_1 \) and \( p_2 \), respectively, both from the same source to the same sink. In the latter, only the scheduling of the updates of nodes in \( p_1 \cap p_2 \) will be considered. Updates of nodes of \( p_1 \setminus p_2 \cup p_2 \setminus p_1 \) are automatically handled by the TOUCHENF component, (see Section 4.2.3).

3.2.2 Eventual Consistency

If some nodes never answer to update orders, a round may last an infinite amount of time. For simplicity, we assume that this is not the case (Assumption 2), hence updating the whole network is supposed to be done in a finite time. Not that this is a prerequisite for Assumption 1, i.e., the initial policy is made of a single path and no forwarding rules outside of this path exist.

Unfortunately, the reality is an asynchronous environment, rendering this assumption invalid. Other techniques have then to be deployed, as discussed in Section 7.6.1. In the worst case, the controller might have to waive some constraints because otherwise no round sequence can update the network.

Assumption 2. The time necessary for updating a single node (update time) has a finite upper bound (\( \lambda \)).

Assumption 3 is a prerequisite for methods enforcing waypoint traversal, as seen in Section 7.1. If this assumption does not hold, provided the policy update does not fulfill the condition defined in Section 2.3.2 pausing (Section 2.3.3) is impossible to realize, so that WE cannot be satisfied.

Assumption 3. The time necessary for a packet to traverse a link is upper-bounded by a constant \( \theta \).

Assumption 4 is taken so that algorithms presented always eventually update the policy. If network elements do experience outages –which clearly happens in practice, then it is impossible for the controller to enforce the new policy, and another new policy should be designed.

Assumption 4. The controller, switches, links, and control connections, do not fail permanently.

3.3 Constraints

This section introduces the three constraints considered in this work.

1. Loop-Freedom (LF): The round sequence is such that no transient policy can occur that would contain a loop.

2. Waypoint Enforcement (WE): The round sequence is such that no packet can bypass the specified waypoint.
3. **Loop-Freedom and Waypoint Enforcement (LFWE)**: The round sequence is such that no transient policy can occur that would contain a loop and no packet can bypass the specified waypoint.

Formalized in Definition 15, the notion of updatability is used in the latter with the following meaning: a node \( v \) is updatable at the beginning of a round \( r \) if updating only \( v \) in \( r \) would not violate the specified constraint (LF, WE or LFWE). Unfortunately, what makes the computation of round sequences non-trivial is that updating concurrently all updatable nodes does not usually ensure the required constraint. In general, only subsets of the updatable nodes can be concurrently updated without violating the specified constraint.

Interestingly, LF and WE have an essential difference: LF is a policy-based constraint, that is, a round sequence is simply LF if all the possible transient policies are LF, and can thus be tested in a stateless manner by looking at all transient policies and verifying that they are all LF. On the contrary, WE is a global property to be tested in a stateful manner: one needs to consider all rounds \( \{r_j, j < i\} \) to assert whether a round \( r_i \) is WE.

### 3.3.1 Loop-Freedom (LF)

Loop-freedom (LF) is enforced when a round sequence does not allow the appearance of any transient policy containing a loop, which is formalized in Definition 18. As discussed in Section 2.2.1, even when LF is ensured, a packet can still go an arbitrary number of times (up to \( \frac{3}{4} \)) through the same node.

**Definition 18.** A policy \( p \) is loop-free if it does not present any cycle (sequence of nodes pointing at each other containing a node twice). A round sequence \( rs \) if all transient policies it could create are loop-free.

### 3.3.2 Waypoint Enforcement (WE)

Waypoint enforcement (WE) is said to be enforced if any packet in the network during the policy update goes through the node considered as the waypoint at least once, and only afterwards reach the sink. Definition 19 formalizes sets used. As a side note, \( p^{\leq \text{wp}} \), \( p^{> \text{wp}} \), \( p^{\neq} \) form a partition of the set of nodes on which \( p \) applies.

**Definition 19.** For a policy \( p \),

1. \( p^{\leq \text{wp}} \) is the set of nodes of \( p \) between the source and the waypoint.
2. \( p^{> \text{wp}} \) is the set of nodes of \( p \) between the waypoint and the sink.
3. \( p^{\neq} \) is the set of nodes of \( p \) not between the source and the sink.

Infected, safe and end sets are then defined in line with Section 2.3.1. It should be noted that they are defined at the end of a round.

**Definition 20.** For a given round \( r_i \) of a round sequence,

1. \( I_i \), the infected set of \( r_i \), is defined as \( I_i = \bigcup_{j \leq i} \bigcup_{p \in r_j^i} p^{\leq \text{wp}}. \)
2. \( S_i \), the safe set of \( r_i \), is defined as \( S_i = \bigcap_{j \leq i} \bigcap_{p \in r_j^i} p^{> \text{wp}}. \)
3. \( E_i \), the end set of \( r_i \), is defined as \( E_i = r f^{> \text{wp}}. \)

Finally, the constraint WE can be formally described in Definition 21.

**Definition 21.** A round sequence \( rs \) is WE if the following condition is ensured for each round \( r_i \) of \( rs \).

\[ E_i \cap I_i = \emptyset \]

It is worth noting that WE is equivalent to \( E_i \subset I_i \).
3.4 Objectives

This section presents the objectives that policy updates should fulfill, focusing in particular on minimizing the number of rounds, the main objective analyzed in this work as it directly impacts the update duration, but also sketching out other objectives that could be desirable.

3.4.1 Main Objective: Minimizing the Number of Rounds

The controller implements the new policy by updating the network in successive rounds. The time necessary for enforcing $p_2$ increases with the number of rounds, $n_r$, hence for time-critical applications, e.g., failure recovery, $n_r$ should be minimized.

In the best case, the controller can update all nodes concurrently without violating constraints set by the administrator, $n_r = 1$, but in the worst case, it updates only one node at a time, $n_r = n$. Thus, if a round sequence of length 2 exists and an algorithm needs $n$ rounds, a lot of time is wasted. At best, an algorithm should compute round sequences of minimal length, i.e. fulfill the objective MINR (Definition 22).

**Definition 22.** An algorithm fulfills MINR for a specified set of constraints $C$ if for any policy update that can be realized in keeping with $C$, it computes a round sequence of minimal length.

**Definition of the Worst-Case Performance Ratio**  Fulfiling MINR can come at a very high computational cost. Whenever algorithms compute a round sequence, the closer its length is to the minimum, the better. In order to assess the performance of an algorithm $a$ with respect to MINR for a given number of nodes $n$, the worst-case performance rate, WCPR, is introduced.

**Definition 23.** Let $a$ be a specified exact algorithm and $C$ a given set of constraints. For a policy update $p_u$ that can be realized in keeping with $C$, $min_r(p_u)$ denotes the minimal number of rounds to execute $p_u$ and $n_r(p_u)$ the number of rounds $a$ needs. The worst-case performance rate of $a$ is then defined for a number of nodes $n$ as the following expression, where $|p_u| = n$ indicates that $n$ nodes are on both the old and the new path.

\[
\min_{|p_u|=n} \frac{\min_r(p_u)}{n_r(p_u)}.
\]

**Exploitation of the Worst-Case Performance Ratio**  The worst-case performance rate is utilized in two manners.

1. First, plotting the evolution of the worst-case performance rate (WCPR) of an algorithm for an increasing number of nodes is an indication of the asymptotic performance of this algorithm with respect to WCPR. Note that computing WCPR becomes difficult for a single machine beyond a dozen nodes, hence such a plot is only an approximative indication of an asymptotic behavior, and not a proof.

2. Second, WCPR is a strong indication of the weaknesses of an algorithm. Indeed, if a policy update makes an algorithm’s performance rate equal to WCPR, then we know that this policy update presents some characteristics that an optimal algorithm would see but that the studied algorithm oversaw. Thus, analyzing this policy update enables us to discover how to improve further algorithms.

The module **Comparator** (see Section 6.2) is an implementation of a generator of a worst-case performance rate.
3.4.2 Further Objectives

Depending on the context, the time necessary for updating the whole network may not be the determining factor. The controller can favor some objectives over others, or try to strike a balance between them.

Minimizing Computational Cost  An important objective is the minimization of the computational cost. Clearly, if an unlimited computation power or an infinite amount of time is available, a minimal round sequence can be trivially obtained with Algorithm 1.

Algorithm 1 Greedy Optimal Algorithm

Input: old policy $p_1$, new policy $p_2$.
Output: round sequence $R$.
1: Generate all possible round sequences.
2: Filter out round sequences not fulfilling specified constraints.
3: Order round sequences depending on the degree to which they fulfill the specified objective.
4: return top-ranking round sequence.

Definition 24. An algorithm $a$ fulfills MINCOMP for $C$ a given set of constraints if for any policy update $p_u$ that can be realized in keeping with $C$, it computes a round sequence fulfilling $C$ with a minimal computational cost.

In case the round sequence can be computed offline, e.g., if the policy updated is scheduled in advance, such strategies can be used (see Algorithm 13). However, if the controller must compute a round sequence online, e.g., reacting to an unforeseen hazard, the duration of the computation of the round sequence is more critical.

Consequently, throughout this work, we will design algorithms that not only produce round sequences fulfilling specified constraints and objectives, but also generate these round sequences with a low computational cost. Possible trade-offs are explored in Section 4.3. Algorithms with different trade-offs can even be combined to combine advantages of both, as seen in Section 7.4.

Maximizing the Number of Updates per Round  Another classic objective is the maximization of the number of updates per round, MAXU. The motivation behind this objective is that, by maximizing the number of nodes updated in a single round, we ensure that as many nodes as possible follow the new policy. This is a natural objective used per example by [6]. A formal expression of MAXU is provided in Definition 25. Interestingly, fulfilling MAXU leads to round sequences longer than minimal (see Section 2.2.2), which is why this work mainly focuses on MINR.

Definition 25. For a round $r$, $u_r(r)$ denotes the ratio of the number of nodes updated during $r$ to the number of updatable nodes during $r$. An algorithm $a$ fulfills MAXU for a given set of constraints $C$ if for any policy update $p_u$ that can be realized in keeping with $C$, it computes a round sequence fulfilling $C$ such that $u_r(r)$ is maximal at each round $r$.

Updating a Single Node in a minimal Number of Rounds  Moreover, the policy update can be due to a load or a failure at a specific node, that is, a given node $v$ on the old path $p_1$ urgently needs to be removed from the currently used path, and the other node updates are only incidental. The round sequence then needs
to be minimal only up to the update of \( v \), and has no constraints to reach afterwards. This objective, called \( \text{MINRSINGU} \), is formalized in Definition \ref{def:minrsingu}. \( \text{MINRSINGU} \) is not the main focus of this work, but was only mentioned incidentally to illustrate the power of the line model (cf. Section \ref{sec:line_model}).

**Definition 26.** An algorithm \( a \) fulfills \( \text{MINRSINGU} \) with respect to a node \( v \) for a given set of constraints \( C \) if for any policy update \( p_u \) that can be realized in keeping with \( C \), it computes a round sequence fulfilling \( C \) such that the number of rounds before \( v \) is updated is minimal.

**Resiliency** Another desirable property is the resiliency to a delay in the update of a single node, simply called \( \text{RES} \). That is, the extent to which the network remains consistent with respect to given properties, such as LF, WE, as well as the distance between the current network and the new policy \( p_2 \), in case a node cannot be updated. This distance could for instance be expressed as the number of nodes updated, but many more metrics are conceivable.

A *tagging* solution does not tackle this issue, since the network either implements \( p_1 \) or \( p_2 \), but never any transient policy. This has the positive consequence that no undesired transient policy can emerge, but that comes at the price of a non-existent \( \text{RES} \): if any node needs time to be updated, all the network needs to wait for it, which leads to \( p_1 \) potentially waiting forever. One solution could then consist in letting the controller only wait up to a given duration, after which it would notify the authority responsible for policy design. It would then not only indicate it cannot enforce \( p_2 \) but also provide a list of problematic nodes and any relevant network information. Afterward, the policy design authority can generate an alternative new policy \( p_2' \) mindful of problematic nodes.

On the contrary, *update scheduling*, the method all the solutions in this work implement, does not rely on tagging, but schedules updates in such a way that given properties are ensured. Accordingly, in case a node is not responding, the controller may still be able to update a large part of the network. For example, assuming LF is to be ensured, if the node not responding is the forward node on which a long chain of backward nodes depend, such as the node \( v_2 \) on Figure \ref{fig:dependency_chain}, the controller cannot even update a single node apart from \( v_1 \). However, in a case such as that of Figure \ref{fig:relay_node}, a node not responding hardly poses any problem for the global network update, that is, the remaining policy would still be similar to \( p_2 \).

![Figure 30: Example of a tight dependency chain: \( v_1 \) and \( v_2 \) can be updated right away, any other node must wait for its predecessor.](image)

Accordingly, on the contrary to *tagging*, *update scheduling* offers \( \text{RES} \).

Nonetheless, having for a long time a policy enforced in the network that administrators did not envision could also be undesirable, since loads can appear at locations where they are not expected, and result in congestion. Such properties should also be taken into account in the definition of resiliency, that is, a
resilient round sequence should be a schedule that avoids as much as possible going through undesirable transient policies. The property RES is not studied in this work, but will probably be of interest in the future.
4 Algorithms

In this section, we present a range of algorithms minimizing the number of rounds to update the network policy.

4.1 Dependencies

The DF uses three types of node dependencies, the simple dependency, the backward dependency, and the extended dependency. If not specifically mentioned otherwise, the DF works with the backward dependency, but some algorithms using simple dependency- or extended dependency-DFs will be introduced (e.g., algorithms 10 and 14). These three dependencies generally become invalid after an update order is sent, and thus need constant update.

While this work limits its focus to these three dependency types, two more dependencies are presented to complete the picture and to get a better understanding of the problem, the weak update dependency and the strong update dependency. They differ strongly from the three former dependencies in that they do not require updating on-the-fly: they are computed at the beginning and can be relied upon for the remaining of the policy update.

Definition 27. A dependency for a specified set of constraints $\mathcal{C}$ at the beginning of a round $r$ is a function $D: \mathcal{D} : v \rightarrow D(v)$, where $D(v)$ is a subset of nodes, $v \notin D(v)$ fulfilling the following condition:

If a node $\tilde{v}$ is updatable with respect to $\mathcal{C}$ in round $r$ and the controller only updates $\tilde{v}$ during $r$, then any node $v$ such that $\tilde{v}$ is in $D(v)$ is updatable in round $r + 1$.

4.1.1 Update Dependencies

When speaking of dependencies, one wishes a dependency such that the controller can simply update one parent after a child, never needing to update the dependency during the network update. Such dependencies exist, but are not trivial to compute. We present them here for the sake of completeness.

Strong Update Dependency An ideal dependency forest would be such that as long as the controller always updates a node $v$ after having updated at least one of $v$’s children, the specified constraints are ensured, we call such a dependency a strong update dependency (Definition 28).

Definition 28. Let $D$ be a dependency for a specified set of constraints $\mathcal{C}$. $D$ is a strong update dependency always having updated a dependency path from a leaf to $v$ before updating $v$ is a condition strong enough to ensure $\mathcal{C}$.

Several strong update dependencies may exist for a given policy update. One trivial but valid strong update dependency is for instance a function that maps any node to the node preceding it on the new path. Relying on a strong update dependency, the controller could simply send an update order to $v$ as soon as it receives an update acknowledgment from one of $v$’s children, independently from the progress of other nodes’ updates (asynchronous network update). A strong update dependency has two main properties:

1. At any instant, the set of updatable nodes will simply equal the set of nodes having an updated child, without any need to update the dependency forest on the fly.

2. Dependency paths can be updated in parallel without any path needing to wait for another.
As an example, consider the configuration in Figure 32. Shown in Figure 33, its corresponding dependency forest is a valid strong update dependency because as long as the controller always updates a parent after at least one of its children has been updated, the network remains loop-free.

A strong update dependency could for instance be computed in a trial and error fashion, as is done in Algorithm 2. It should be noted that this algorithm is only a first step, it relies on randomization and does
not offer any guarantee to compute a valid dependency forest in a finite time.

**Algorithm 2** Algorithm for computing a **strong update dependency** forest

**Input:** old policy \( p_1 \), new policy \( p_2 \).

**Output:** **strong update dependency** forest \( F \).

1: **empty** the dependency forest
2: **while** nodes remain to be inserted in the dependency forest **do**
3: **select** at random a node \( v \) not in the dependency forest
4: **for each node** \( v' \) in the dependency forest **do**
5: **if** making \( v, v' \)'s child does not violate the consistency of the dependency forest with respect to the **strong update dependency** **then**
6: Go to Line 3
7: Go to Line 3
8: **return** the round set

**Weak Update Dependency**  A dependency weaker than a strong update dependency but also allowing a significant parallelization would be a dependency such that as long as the controller updates a specified set of nodes at a time, and waits for all their update acknowledgments to broadcast the next update batch, specified constraints are enforced. We call the time between one update broadcast and the next a round. To obtain this property, a **weak update dependency**, formalized in Definition 29, is such that as long as the network updates in each round parents of all the nodes it updated in the previous round, specified constraints are ensured. As an example, a weak update dependency can be computed using Algorithm 3.

**Definition 29.** A dependency \( D \) for a specified set of constraints \( C \) is a **weak update dependency** if \( C \) is ensured as long as

1. All leaves are updated in the first round.
2. If \( v \) is updated in a round \( r \), all \( r \)'s fathers are updated in the round \( r + 1 \).

As an example, consider the configuration in Figure 34. Shown in Figure 35, its corresponding dependency forest is a valid weak update dependency because as long as the controller always updates a parent after at least one of its children has been updated, the network remains loop-free.

![Figure 34: Policy update with a corresponding dependency forest in Figure 35](image-url)
Figure 35: Dependency forest according to the **weak dependency** corresponding to the policy update of Figure 34.

Note that a strong update dependency is in particular a weak update dependency. Indeed, if the controller is allowed to update \( v \)'s father as soon as \( v \) is updated for any \( v \) (no rounds), then it is in particular allowed to update the network in rounds and in each round update the fathers of the nodes updated in the previous round.

### 4.1.2 Topological Dependencies

While update dependencies, i.e., dependencies indicating directly what node should be updated after which other node, are convenient, compiling them requires the consideration of all possible outcomes, which is why we introduce dependencies less expressive, but computable at a lower cost.

They are only dependencies with respect to loop-freedom.

**Simple Dependency** The backward node \( v \) depends on the node \( v' \) according to the simple dependency if \( v' \) is the first node after \( v \) on the new path and its update would allow \( v \)'s update. As an example, on Figure 36, \( v_4 \) can depend on \( v_3 \), but not on \( v_2 \). A formal definition follows.

**Definition 30.** Let \( v \) and \( v' \) be two nodes, \( v \) backward. \( v \) exhibits a simple dependency for \( v' \) if all three conditions are fulfilled.
Algorithm 3 Algorithm for computing a **weak update dependency** forest

**Input:** old policy \( p_1 \), new policy \( p_2 \).

**Output:** weak update dependency forest \( \mathcal{F} \).

1. **empty** the dependency forest
2. **while** nodes remain to be inserted in the dependency forest **do**
3. **select** at random a node \( v \) not in the dependency forest
4. **for each node** \( v' \) in the dependency forest **do**
5. **if** making \( vv' \)’s child does not violate the consistency of the dependency forest with respect to the weak update dependency then
   6. Go to Line 3
   7. Go to Line 3
8. **return** the round set

Figure 36: Excerpt of a policy update illustrating the simple dependency. According to the simple dependency, \( v_4 \) depends on \( v_3 \) but not on \( v_2 \).

1. \( v' \) is the first node after \( v \) on the old path whose successor on the new path is not between \( d_n(v) \) and \( v \)
2. all nodes between \( v \) and \( v' \) on the new path are updated.
3. Updating singularly \( v \) and then \( v' \) does not violate specified constraints

A forward node does not depend on any other nodes and is a leaf in the dependency forest. A backward node not depending on a node is excluded from the dependency forest.

It should be noted that testing the first two conditions is done in a linear time. A node \( v \) always simply-depends on at most one node, called the simple dependency child of \( v (sdc(v)) \).

**Backward Dependency** If a node is **backward**, it obviously cannot be updated without creating a loop.

**Definition 31.** Let \( v \) and \( v' \) be two nodes, \( v \) backward.
$v$ backward-depends on $v'$ if $v'$ is between $d_n(v)$ and $v$ and $d_n(v')$ is not between $d_n(v)$ and $v$ (both properties defined according to the current policy).

It should be noted that if a node $v$ depends on a node $v'$ according to the simple dependency, then $v$ depends on $v'$ according to the backward dependency. Hence, simple dependency implies backward dependency.

It should be noted that the backward dependency is not an update dependency, as simply updating any dependency path from a leaf to $v$ does not necessarily make $v$ updatable. Consider as an example Figure 37; $v_4$ (backward-)depends on $v_2$ and $v_3$ and $v_2$ depends on $v_1$. However, having $v_1$ and $v_2$ new is not a sufficient condition for making $v_4$ updatable, and only $v_2$’s update makes $v_4$’s possible.

Figure 37: Policy update where a backward dependency-path cannot be used as a round sequence. According to the backward dependency based on LF, $v_4$ depends on $v_2$ and $v_3$, $v_2$ depends on $v_1$. However, updating $v_1$, $v_2$ and then $v_3$ violates LF.

$v$ always backward-depends on zero or several nodes joined by a or-relationship.

Extended Dependency Some dependencies are not captured by the backward dependency, such as an x-structure (see Definition 6), a finer dependency is thus introduced, the extended dependency.

Definition 32. Let $v$ and $v'$ be two nodes, $v$ backward.

$v$ depends on a node set $\mathcal{V}$ if updating only $\mathcal{V}$ and then $v$ would not create any loops.

It should be noted that if a node $v$ depends on a node $v'$ according to the backward dependency, then $v'$ belongs to a set on which $v$ depends according to the extended dependency. Hence backward dependency implies extended dependency.

Using the extended dependency, $v$ always depends on sets of nodes joined by an or-relationship, each of them comprising nodes combined by an and-relationship.

4.1.3 Dependency Map

4.2 Algorithmic Model

All the algorithms presented in this work rely on an algorithmic frame that the following subsection sketches out.

4.2.1 Algorithmic Definitions

An algorithm $a$ is described as an $o - c$ algorithm, where $o$ is the objective it attempts to fulfill, and $c$ the constraint with which it complies. The following definitions are introduced to describe the degree to which an algorithm fulfills its objective.
**Minimization of the number of rounds**

**Definition 33.** An algorithm is said to be minimum if it provides a round sequence of minimal size.

**Definition 34.** An algorithm is said to be approximate if it does not always provide a round sequence of minimal size.

If not otherwise specified, algorithms presented here are approximate.

**Maximization of the number of updates per round**

**Definition 35.** An algorithm is called maximal if at each round, it provides an update set such that no node can be added to the update set.

**Definition 36.** We call a maximum algorithm an algorithm that provides a round sequence so that, for each round, the update set is of maximum size –any other update set will have a size smaller or equal to that of this set–.

**Computational Cost** The terms lightweight, middleweight, and heavyweight are used to described qualitatively the computational cost of algorithms, in increasing order of computational cost.

### 4.2.2 Algorithmic Constraints

**No Packet Drop** In the remainder of this work, we make the assumption that no packet is dropped, written \(ND\) (Assumption 5), motivated by the idea that, even if in practice packet drops are standard practice, they are not desirable in the network, so that an update should avoid creating some.
Assumption 5. No packet should be dropped throughout the network update.

Without ND, namely if packet drops are allowed, the trivial Algorithm 4 can always be used. Indeed, if all old rules are removed before all else, if the controller waits subsequently long enough so that no packet is left in the network, then we have a packet consistency. Thus, provided the old and the new policies fulfill the constraints set on the update (which is assumed to be the case because otherwise no valid update is possible), Algorithm 4 is a valid solution. Techniques to remove all packets from the network in a valid manner are reviewed in Section 7.1.

Algorithm 4 Algorithm without ND
Input: policies $p_1, p_2$.
Output: round sequence $R$.
1: remove all (old) rules at $p_1$ concurrently
2: wait $\Theta$ so that no packet forwarded in accordance with $p_1$ remain in the network.
3: add (new) rules at nodes of $p_2$ concurrently
4: return the round set

Never Making a new Node Old Again: Constant Progress (CP) CP is obeyed when no new node is allowed to become old again. In the frame of our model, CP would be expressed formally as follows.

Assumption 6. The condition of constant progress, CP, is met by an algorithm when, for any network $N$, the round sequence it provides is progressing.

The constant progress constraint, CP, equivalently means that the controller is only allowed to update nodes, and cannot downgrade them: a new node is never allowed to become old again. CP will be observed in the remainder of this work. However, the possibility of relaxing CP will be discussed in Section 7.6.3.

4.2.3 Update Management on $p_2 \setminus p_1$: Touchability Enforcement

For clarity, some denominations are introduced. $p_2 \setminus p_1$ and $p_1 \setminus p_2$ represent $p_2 \setminus p_1$, respectively, $p_1 \setminus p_2$, and $wo_o(v)$ denote the path along $p_2$ between $d_n(v)$ and the last successor of $v$ on $p_2$ that does not belong to $p_1$ (both ends included), respectively. $wo_o(v)$ denote the path along $p_1$ between $d_o(v)$ and the last successor of $v$ on $p_1$ not in $p_2$.

Definition 37. Let $p_1 \cap p_2$ be the set of relevant nodes, and $p_2 \setminus p_1 \cup p_1 \setminus p_2$ the set of irrelevant nodes. A relevant node $v$ is said to be touchable if $wo_o(v)$ is fully new.

As will be discussed later on in this section, the update of irrelevant nodes can be easily scheduled for a given schedule of touchable nodes. Theorem 4 formulates $reqUpdRelNodes$, the necessary condition for packet loss avoidance:

Theorem 4. Let $C$ be a set of constraints to be enforced throughout the update, $C$ not supporting packet drops. For any touchable node $v$,

1. if $d_n(v) \not\in p_1, wo_o(v)$ should be updated before $v$ (v touchable)
2. if $d_o(v) \not\in p_2, wo_o(v)$ should be updated after $v$ (no removing a link along $wo_o(v)$)
Proof. In the case (1), if a single node \( v' \) is not updated on \( w_0(v) \) when \( v \) becomes new, then a packet \( p \) reaching \( v \) after it became new can go along \( w_0(v) \) up to \( v' \) and \( v' \) will have no rule installed for \( p \) when it receives \( p \). \( p \) will thus be dropped.

Reversely, in the case (2), if a single node \( v' \) is updated on \( w_0(v) \) before \( v \) becomes new, then a packet \( p \) leaving \( v \) before it became new can go along \( w_0(v) \) up to \( v' \) and \( v' \) will have no rule installed for \( p \) when it receives \( p \). \( p \) will thus be dropped.

Moreover, if \( d_o(v) = d_n(v) \), the update of \( v \) would have no effect, such forwarding rules should be removed from the update set in a preprocessing step, and the node \( v \) can then be removed from the set of nodes to update. This preprocessing step was implemented in Algorithm optCONFBFS described in Section 4.4.4. It should be noted that, provided \( d_n(v), d_o(v) \in p_1 \times p_2 \), no specific order can be given a priori to the update of \( v \).

Enforcing the condition reqUpdRelNodes can be realized by an automatized process independently from the scheduling of touchable nodes. The module enforcing reqUpdRelNodes is called the touchability enforcer, TOUCHENF. Two solutions to realize it are presented below: (1) SYNTOUCHENF, synchronous, allows for a simple but slow policy update, and (2) ASYNTOUCHENF, asynchronous, provides a complex but quick policy update.

### Synchronous Touchability Enforcer
SYNTOUCHENF (Algorithm 5) updates in a first step all nodes on the new path and not on the old path, and only then updates touchable nodes. In this way, the controller is sure that all touchable nodes are touchable whenever it schedules their updates.

This solution can be deemed slow because the touchability of all touchable nodes is ensured before the first round \( r_1 \), even though only the touchability of the nodes in \( r_1 \) is necessary. However it is simple and straightforward because the controller never needs to wait for a touchable node to become touchable and can send all update orders of \( r_{i+1} \) concurrently right after it has received the last update acknowledgment of \( r_i \).

#### Algorithm 5 SYNTOUCHENF: Synchronous Touchability Enforcer

**Input:** policies \( p_1, p_2 \).
**Output:** round sequence \( R \).

1: update concurrently all nodes in \( p_{2\setminus1} \)
2: update nodes in \( p_1 \cap p_{2\setminus1} \) in compliance with constraints \( C \), after the update of each node \( v \), also update \( w_0(v) \)
3: return the round set

### Asynchronous Touchability Enforcer
On the contrary, ASYNTOUCHENF (Algorithm 6) lets the controller send update orders to all nodes in \( p_{2\setminus1} \), and starts the round sequence without waiting for acknowledgments from any node in \( p_{2\setminus1} \). This requires the controller to wait when it schedules a touchable node for it to become touchable before sending an update order to it.

This solution can be deemed complicated because the controller sends at the beginning of each round \( r_i \) only update orders to the touchable nodes of \( r_i \), and then to the remaining nodes of \( r_i \) as they gradually become touchable. However, this is quick since the touchability of a single node is only awaited when the node is actually scheduled to be updated.

Furthermore, the controller could even adapt its strategy to the set of currently touchable nodes, so as to be, to some extent, quicker. Such methods are discussed in 7.5.
Algorithm 6 ASYNTOUCHENF: Synchronous Touchability Enforcer

**Input:** policies $p_1, p_2$.

**Output:** round sequence $\mathcal{R}$.

1: **update** concurrently send update orders to $p_2 \setminus p_1$
2: **update** touchable nodes according to a valid round sequence, each time a node $v$ is to be updated, wait first until $v$ is touchable and then send an update order to $v$
3: **return** the round set

### 4.3 Letting the Dependency Forest Evolve

The dependency forest ($DF$) applied to the specific loop-freedom (LF) constraint has been introduced and defined in Definition 4. Additionally, some simple applications of this tool were sketched out. In that section, the $DF$ considered was the initial $DF$, i.e., was computed based on the old and new policies before any update has been done.

However, while updates are executed, the policies change. Throughout the policy update, the network policy is neither $p_1$, nor $p_2$, but a mix of both. This evolution of the current policy should be reflected by the $DF$, which is why the dependency introduced in Definition 3 and defined exclusively by $p_1$ and $p_2$, is extended to dependencies based on the current policy, that are sketched below in Section 4.1.

#### 4.3.1 Embedding WE

While the $DF$ presented so far relies on a LF-dependency, it can also be computed with regard to LFWE. A node $v$ then LFWE-depends on a node $v'$ if it LF-depends on $v'$ and if updating any path from a leaf to $v'$ creates a configuration such that $v'$’s update would not violate WP.

#### 4.3.2 Updating the Dependency Forest

The dependency forest ($DF$) catches dependencies throughout the network as regards the comparison of the old policy $p_1$ and the new $p_2$. Unfortunately, these dependencies evolve with time, some become stale, and some are made simpler.

For instance, consider the setting presented in Figure 39. Initially, in the dependency forest, $v_5$ would backward-depend on $v_3$ and $v_4$. However, if in the first round, $v_2$ is updated, then $v_3$ is no longer between $d_0(v_5) = v_2$ and $v_5$, and updating $v_3$ can no longer render $v_5$ updatable. Thus, the $DF$ currently contains an inaccurate information that could mislead algorithms using it. If an algorithm, for some reason, wanted to update $v_5$ and tried updating $v_3$ to reach this result, this strategy would fail.

To avoid this issue, the inaccurate information on the $DF$ should be removed while updating. Since the information can only be rendered stale by a node update, we recompute the $DF$ after each update acknowledgment.

Updating the $DF$ can simply consist in building the dependency forest again from scratch. This needs to be done in the case of the simple dependency, because new links that did not exist previously get added. However, in the case of the backward dependency and the extended dependency, no new link appears, only obsolete links disappear. One thus simply needs to go over all links and filter out the obsolete ones, namely those that, based on the current policy, do not hold anymore. For example, in the case described above, after the first round, $v_5$ would still depend on $v_4$, so that the (dependency) link $v_5 \rightarrow v_4$ would be kept, while the link $v_5 \rightarrow v_3$ would be removed.
4.3.3 Weight System

We introduce *weights* to rank updatable nodes in keeping with our confidence that their updates would help reduce the minimal number of rounds to update the remaining of the network. By selecting the nodes of highest weights, we hope to build a round sequence of length close to minimal.

**Set of Closest Leaves** Let $\mathcal{L}_{\text{min}}$ be the function that maps any node $v$ in the dependency forest to its closest leaves. $\mathcal{L}_{\text{min}}(v)$ is the set of the leaves of the shortest dependency path of $v$.

As an example, consider the dependency forest in Figure 40. The resulting $\mathcal{L}_{\text{min}}$ is given in the Table 4.3.3 below.

**Distance to the closest Leaf: Height**
Table 1: Closest Leaves by Node for DF in Figure 40

<table>
<thead>
<tr>
<th>v</th>
<th>$\mathcal{L}_{\text{min}}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>${v_1}$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>${v_1}$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>${v_4, v_5}$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>${v_4}$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>${v_5}$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>${v_1, v_3, v_4}$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>${v_4}$</td>
</tr>
<tr>
<td>$v_8$</td>
<td>${v_4}$</td>
</tr>
<tr>
<td>$v_9$</td>
<td>${v_4}$</td>
</tr>
</tbody>
</table>

Definition 38. The height of a node $v$, $h(v)$, is the length of the shortest dependency path of $v$.

As an example, consider the dependency tree in Figure 40. The resulting $\mathcal{L}_{\text{min}}$ is given in the Table 4.3.3 below.

Table 2: Height for DF in Figure 40

<table>
<thead>
<tr>
<th>v</th>
<th>$\mathcal{L}_{\text{min}}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>1</td>
</tr>
<tr>
<td>$v_2$</td>
<td>2</td>
</tr>
<tr>
<td>$v_3$</td>
<td>2</td>
</tr>
<tr>
<td>$v_4$</td>
<td>1</td>
</tr>
<tr>
<td>$v_5$</td>
<td>1</td>
</tr>
<tr>
<td>$v_6$</td>
<td>3</td>
</tr>
<tr>
<td>$v_7$</td>
<td>2</td>
</tr>
<tr>
<td>$v_8$</td>
<td>3</td>
</tr>
<tr>
<td>$v_9$</td>
<td>4</td>
</tr>
</tbody>
</table>

Leaf Weights

Definition 39. The weight of a node $v$, $w(v)$, is defined as follows.

$$w(v) = \max_{\tilde{v} \mid v \in \mathcal{L}_{\text{min}}(\tilde{v})} (h(\tilde{v}))$$

As an example, consider the dependency tree in Figure 40. The resulting $\mathcal{L}_{\text{min}}$ is given in the Table 4.3.3 below. $v_1$ has the highest weight, namely 4, and should thus be the first to be added to the current update set.

The higher the weight of $v$, the more urgent $v$’s update is, as many nodes depend on it. This is the principle followed in most of the MINR-heuristics below.
4.4 Loop-Freedom (LF)

4.4.1 MAXU-LF Algorithms

**LFMAXALUPD: Maximal MAXU-LF Algorithm**  Algorithm LFMAXALUPD (Algorithm 7) computes at each round a maximal—but not maximum—update set. It should be noted that LFMAXALUPD has a polynomial complexity.

**Algorithm 7** Maximal Algorithm for LF policy update  
**Input:** old policy $p_1$, new policy $p_2$.  
**Output:** round sequence $R$.  
1: **until** impossible, select an updatable old node and update it (i.e., add it to the current round and make it limbo)  
2: **if** old nodes remain, create a new round and go back to Line 1  
3: **return** the round set

**LFMAXUMUPD: Maximum MINR-LF Algorithm**  Algorithm LFMAXUMUPD (Algorithm 8) computes at each round a round sequence of maximum size. LFMAXUMUPD’s complexity is exponential.

**Algorithm 8** Maximum Algorithm for LF policy update  
**Input:** old policy $p_1$, new policy $p_2$.  
**Output:** round sequence $R$.  
1: **generate** all LF rounds given the current network  
2: **select** a maximal round  
3: **if** the network is not fully updated, go back to Line 1  
4: **return** the round set

4.4.2 Lightweight MINR-LF Algorithms

**LFONEBYONE** (Algorithm 9) is the simplest algorithm possible for updating a policy in a loop-free manner. In each round, it updates a single node, from the end to the beginning, so that a packet always goes first along the old path, and then along the new path. As both paths are loop-free, LFONEBYONE is also loop-free. LFONEBYONE requires only a linear computation, but performs in general poorly, as it always needs as many rounds as the new path contain nodes independently of the update difficulty.

**LFONEBYONE: Algorithm updating LF one node at a time from end to start**

<table>
<thead>
<tr>
<th>$v$</th>
<th>$L_{min}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>4</td>
</tr>
<tr>
<td>$v_4$</td>
<td>2</td>
</tr>
<tr>
<td>$v_5$</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Height for $DF$ in Figure 40
Algorithm 9 Trivial Solution, LF\textsc{OneByOne}

\textbf{Input}: old policy $p_1$, new policy $p_2$.
\textbf{Output}: round sequence $R$.
1: \textbf{update} one node at a time from end to start
2: \textbf{return} the round set

\textsc{LFSIMPDEP: Very Lightweight Algorithm} Algorithm \textsc{LFSIMPDEP} (Algorithm 10) relies on the simple dependency (Definition 30).

The shortcuts for $v$ are all \textit{ways out} (5) for $v$ apart from the one the closest from $s_n(v)$.

\textsc{LFSIMPDEP} is a very lightweight algorithm. The computation necessary for generating dependencies is linear, as a single traversal of the new policy suffices. Moreover, it does not do any update that would prevent other updates, as it cannot prevent shortcuts from being reachable. Additionally, \textsc{LFSIMPDEP} differs from other algorithms in that the $DF$ is not a forest but only comprises parallel lines. Nodes each have either no child or a single one.

However, this simplicity comes at the price of a low performance with respect to \textsc{MINR}. Indeed, \textsc{LFSIMPDEP} fails to make good use of bridges, in the same way as in Section 2.2.2, and can accordingly be $O(n)$ far from optimal.

Since \textsc{LFSIMPDEP} is very light-weight, it is a good algorithm to start updating the network while slower algorithms are computing a better solution, as described in the section on approximations (Section 7.4).

\textbf{Definition 40}. Let $v$ be a node.
A shortcut for $v$ is a way out for $v$ that is not the simple dependency child of $v$ ($\textsc{sdc}(v)$).

Algorithm 10 \textsc{LFSIMPDEP}

\textbf{Input}: old policy $p_1$, new policy $p_2$.
\textbf{Output}: round sequence $R$.
1: \textbf{while} the network is not fully updated \textbf{do}
2: \textbf{for} each node $v$ \textbf{do}
3: \textbf{if} $v$ is a leaf \textbf{then}
4: Update $v$.
5: \textbf{return} $R$

4.4.3 Middleweight \textsc{MINR-LF} Algorithms

\textsc{LFbckDepHeur: MINR-LF Algorithm} Algorithm \textsc{LFbckDepHeur} is an heuristic that provides a round sequence $R$ with the objective of maximizing the update speed, i.e., minimizing the number of rounds. To do so, it builds at the beginning of each round a dependency forest and selects then leaves in descending order of weight (cf. Definition 39), so as to reduce the lower bound on the number of remaining nodes for updating the network.

\textsc{LFfreqUpd: MINR-LF Algorithm} As mentioned in Section 4.3.2, updating the $DF$ only after each round may lead to choices not as good as updating it after each update. \textsc{LFfreqUpd} (Algorithm 12) was designed to avoid working with a stale dependency forest. The leaf of largest weight is added to the round,
Algorithm 11 Heuristic for minimizing the lower bound of the dependency forest

**Input:** old policy $p_1$, new policy $p_2$.

**Output:** round sequence $R$.

1. compute DF, determine weights
2. compute a round, select in decreasing order of weight as many nodes as possible.
3. if the network is not fully updated, go back to Line 1
4. return $R$

then the dependency forest, and the new heaviest leaf is selected, and so on. Naturally, the question then arises of where to stop updating nodes, as updating too many nodes can cost $n/10$ nodes (cf. Lemma 5).

LFREQUPD solves this problem by considering a future dependency forest, the one that would exist at the beginning of the next round if no further node were added to the round. After each node update, both the current $DF$ and the future one are updated, and LFREQUPD tests whether the maximal height of the current $DF$ is strictly higher than that of the future one. If so, then updating a further node can still help reduce the maximal height in the next round, and accordingly help reduce the remaining number of rounds before the end of the update. Otherwise, updating another node cannot help reduce the future maximal height and can only prevent potentially useful updates, and should thus be avoided. Waiting is then a better choice.

Algorithm 12 Advanced Heuristic for MINR

**Input:** old policy $p_1$, new policy $p_2$.

**Output:** round sequence $R$.

1. compute DF, determine weights
2. if current maximal height higher than future one then
   3. update highest leaf
   4. else
   5. start a new round and go back to Line 1
3. else
   6. if network not fully updated then
      7. go to Line 1
4. return $R$

4.4.4 Heavyweight MINR-LF Algorithms

**LFOPTMINR:** Minimal MINR-LF Algorithm

LFOPTMINR (Algorithm 13) is a simple algorithm computing a round sequence of minimal size. LFOPTMINR goes over all possible successions of loop-free rounds and selects one of minimal size. LFOPTMINR has an exponential cost and thus an only limited applicability, but is used to test other algorithms’ performance.

Algorithm 13 Optimal Algorithm for MINR

**Input:** old policy $p_1$, new policy $p_2$.

**Output:** round sequence $R$.

1. compute all round sequences
2. select shortest
3. return $R$
LFExtDepHeur2: MINR-LF Algorithm relying on extended dependency

As soon as nodes start updating, some nodes can be backward but have no children (see Section 2.2.3). Fortunately, this information that eludes a dependency forest based on the backward dependency is caught with the LFExtDepHeur2 (Definition 32).

The LFExtDepHeur2 (Algorithm 14) realizes such a dependency forest. A node does not have single nodes as children anymore, but sets of nodes such that the parent would be updatable if they were updated.

It should be noted that LFExtDepHeur2 (Algorithm 14) has the same structure as LFBckDepHeur (Algorithm 11), but implements the dependency extended dependency instead of LFBckDepHeur.

Algorithm 14 LFExtDepHeur2: MINR-LF algorithm relying on extended dependency

Input: old policy $p_1$, new policy $p_2$.
Output: round sequence $R$.

1: compute LFExtDep-DF using only old sets, determine weights
2: compute a round, select in decreasing order of weight as many nodes as possible.
3: for each node in the highest level do
4:   if it is updatable then
5:     add it to the round
6:   else
7:     if the network is not fully updated, go back to Line 1
8: return $R$

LFExtDepHeur3: MINR-LF Algorithm relying on extended dependency

LFExtDepHeur3 (Algorithm 15) relies on a dependency forest using the extended dependency, but updates several levels in a single round.

As a reminder, LFExtDepHeur2 computes a dependency forest, then updates as many nodes of the highest level as possible, after which it starts a new round, computes a new dependency forest, etc. However, a fraction of the nodes LFExtDepHeur2 selects in the next round may already be updatable in the current round, in addition to the one LFExtDepHeur2 selects in this round.

LFExtDepHeur3 seizes this opportunity by updating levels from the highest downwards until it reaches a level it cannot completely update. It updates as many nodes of this level as it can, and starts a new round. It should be noted that if it kept going down, LFExtDepHeur3 would needlessly risk violating other nodes’ updatability.

Algorithm 15 LFExtDepHeur3: MINR-LF Algorithm relying on extended dependency

Input: old policy $p_1$, new policy $p_2$.
Output: round sequence $R$.

1: compute LFExtDep-DF using only old sets, determine weights
2: for each weight level of the DF in decreasing order do
3:   if all nodes of the level can be updated concurrently then
4:     add them to the round
5:   else
6:     add them to the round, start a new round, and go to Line 1
7:   if the network is not fully updated, go back to Line 1
8: return $R$
**LFEXTDEPHEUR4: MINR-LF Algorithm relying on extended dependency**  
Finally, LFEXTDEPHEUR4 (Algorithm 16) updates frequently a dependency forest relying on the extended dependency. After each node update, the dependency forest is recomputed. This stops when updating cannot decrease the maximal height anymore, as in LFFREQUPD (Algorithm 12).

**Algorithm 16 LFEXTDEPHEUR4: MINR-LF Algorithm relying on extended dependency**

**Input:** old policy \( p_1 \), new policy \( p_2 \).

**Output:** round sequence \( R \).

1. **compute** LFEXTDEP-DF using old and new sets, **determine** weights
2. **if** current maximal height higher than future one **then**
   3. update highest leaf
   4. **else**
   5. start a new round and go back to Line 1
3. **if** network not fully updated **then**
   6. go to Line 1
4. **return** \( R \)

**4.5 Waypoint Enforcement (WE)**

**4.5.1 WAYUP: MINR-WE with pausing**

Algorithm 17, WAYUP, ensure that packets go through the waypoint in *four* rounds, a minimal number of rounds.

1. In Line[1] WAYUP only updates backward rules, which cannot violate waypoint enforcement.
2. In Line[2] WAYUP finishes updating \( p_2^{<wp} \). Any packet not taking \( p_1 \) will stay in loops for some time and finally take \( p_2^{<wp} \).
3. After the pause in Line[3] all packets have disappeared from links unused. Thus, from this point on, all packets in the network will go along \( p_2^{<wp} \) before all else, and WE cannot be endangered anymore.
4. The update is finished.

**Algorithm 17 LFWE algorithm (without Pausing)**

**Input:** old policy \( p_1 \), new policy \( p_2 \).

**Output:** round sequence \( R \).

1. **update** all backward nodes of \( p_1^{>wp} \cap p_2^{<wp} \)
2. **update** \( p_2^{<wp} \cap p_1^{<wp} \) and forward nodes of \( p_2^{<wp} \cap p_1^{>wp} \)
3. **wait** (Pausing)
4. **update** \( p_2^{>wp} \)
5. **return** \( R \)
4.5.2 WPWithoutPauses: MINR-WE without pausing

WPWithoutPauses (Algorithm 18) updates the network in two rounds without Pausing. It relies on the prerequisite discussed in Theorem 3, namely $p_2^{>wp} \cap p_1^{<wp} = \emptyset$.

1. In Line 1, only nodes after the waypoint are updated. They can only forward checked packets so that no security breach is possible.

2. In Line 2, nodes before the waypoint are updated. However, since $p_1^{>wp}$ is fully updated and $p_1^{<wp}$ is included in $p_2^{<wp}$, any packet forwarded from $p_1^{<wp}$ to $p_1^{>wp}$ eventually comes back to $p_1^{<wp}$, and ultimately reach the waypoint. The waypoint traversal is enforced.

Algorithm 18 LFWE algorithm (without Pausing)

| Input: | old policy $p_1$, new policy $p_2$. |
| Output: | round sequence $R$. |
| 1: | update $p_1^{>wp}$ concurrently |
| 2: | update $p_1^{<wp}$ |
| 3: | return $R$ |

4.6 Combined Loop-Freedom and Waypoint Enforcement (LFWE)

4.6.1 LFWWithPauses: MINR-LFWE with Pausing

LFWWithPauses (Algorithm 19) is an algorithm that realizes an update both LF and WE. It relies on an unspecified algorithm able to update a policy in a loop-free manner. It should be noted that LFWWithPauses can reach a dead-end and fails to provide a round sequence in cases such as depicted in Figure 25.

- LFWWithPauses allows for fewer transient configurations than WayUp (Algorithm 17). As WayUp enforces WE, so does LFWWithPauses too.
- Additionally, LFWWithPauses realizes only LF rounds, and is thus overall LF.

Algorithm 19 LFWWithPauses algorithm (without Pausing)

| Input: | old policy $p_1$, new policy $p_2$. |
| Output: | round sequence $R$. |
| 1: | update LF all backward nodes of $p_1^{>wp} \cap p_2^{<wp}$ |
| 2: | update LF $p_2^{>wp} \cap p_1^{<wp}$ and forward nodes of $p_2^{<wp} \cap p_1^{>wp}$ |
| 3: | wait (Pausing) |
| 4: | update LF $p_2^{>wp}$ |
| 5: | return $R$ |
4.6.2 LFWPWithoutPauses: minR-LFWE without Pausing

LFWPWithoutPauses (Algorithm 18) is an algorithm that realizes an update both LF and WE. It relies on an unspecified algorithm able to update a policy in a loop-free manner.

- LFWPWithoutPauses (Algorithm 19) allows for fewer transient configurations than WPWithoutPauses. Accordingly, since WPWithoutPauses enforces WE, LFWPWithoutPauses does it too.

- Additionally, LFWPWithPauses realizes only LF rounds, and is thus overall LF.

\[\text{Algorithm 20 LFWE algorithm (without Pausing)}\]
\textbf{Input:} old policy }p_1\text{, new policy }p_2\text{.
\textbf{Output:} round sequence }R\text{.
1: update }p_1^{\geq \text{up}}\text{ loop-free
2: update }p_1^{\leq \text{up}}\text{ loop-free
3: return }R\]
5 Experimental Results

In this section, we present the first results of experimentations with some of the algorithms presented above minimizing the number of rounds under the loop-free constraint.

The main goal of the experiments is the computation of the performance metric worst-case performance rate (Definition 23). The performance ratio is the ratio of the number of rounds needed by a given algorithm \((a)\) to implement a given policy update over the minimal number of rounds. For a given number of nodes \((n)\), the worst-case performance rate is the lowest performance ratio computed over all policy updates with \(n\) nodes. However, this project’s task goes beyond this simple function, and is actually triple:

1. It considers all possible policy updates for a given number of nodes \(n\) and computes the worst-case performance rate of the given algorithm \(a\).
2. It renders a policy update corresponding to the worst-case performance rate, as well as the round sequence computed by \(a\) and a minimal round sequence (of minimal size) for this policy update.
3. It saves all relevant data related to round sequence computations for each set of policies with a given number of nodes, both by \(a\) and by an optimal algorithm.

The first task, computing the worst-case performance rate, consists in considering all policies and computing for each one a candidate worst-case performance rate with the number of nodes the tested algorithm needs and the minimal number of rounds, respectively. If it is lower than the currently saved worst-case performance rate, then the latter is replaced by the former.

The second task, determining a worst policy update, is executed in parallel with the first. Each time the worst-case performance rate is updated, the corresponding policy update, augmented with some high-level information regarding the round sequence computation, is saved and replaces the previous one.

The third task, logging, is realized on the fly. This has first a forensic purpose, namely debugging in case an algorithm fails. Additionally, one can identify the policies for which the tested algorithm needed more rounds than the minimum, and compare its round sequence with a minimal one. Then a better algorithm can be designed, accounting for the effect that its predecessor oversaw.

5.1 Optimal Algorithm

The minimal algorithm, OPTCONFBS, produces as expected a round sequence of minimal length. Computing minimal round sequences for around \(12!\) configurations of 12 nodes each takes approximately 72 hours. In average, OPTCONFBS needs thus around 1 millisecond for computing a solution with 12 nodes.

Unfortunately, OPTCONFBS has an exponential computational cost. Hence, it would not be usable in an online scenario for policy updates so large or so complex that independent clusters contain a high number of nodes, e.g., 20. However, OPTCONFBS remains applicable if policy updates are not too complex. Moreover, in the offline case, OPTCONFBS’s high computational time is not of major importance, so that it is then usable for a larger number of nodes. Additionally, OPTCONFBS could be rendered faster, e.g., by saving dynamically partial solutions and reusing them.

5.2 Suboptimal Algorithms

So far, only LFbckDepHeur, LFextDepHeur2, and LFextDepHeur3, have been completely implemented and tested. The resulting worst-case performance rates (metric presented in Definition 23) are presented below in Figure 41.
Figure 41: Performance Comparison of LFbckDepHeur, LFextDepHeur2, and LFextDepHeur3 regarding worst-case ratio

LFbckDepHeur, LFextDepHeur2, and LFextDepHeur3 have a worst-case performance rate decreasing with the number of nodes. This can be interpreted by the increase of network structures both in variety and complexity. More configurations appear, and, while optCONFbfs keeps returning sequences of minimal length, more corner-cases escape the three other algorithms, and they drift further from minimal lengths.

Additionally, for any number of nodes, LFextDepHeur3 performs better than LFextDepHeur2, which in turn outperforms LFbckDepHeur. Indeed, as algorithms increase in complexity, they are able
to capture more information, and, better-informed, select in each round nodes more likely to yield a short round sequence.
6 Simulation Environment

In order to test the algorithms presented in Section 4, we have implemented a project in Java, using the Eclipse environment. The role of this implementation is dual, it firstly tests algorithms’ performance, and secondly gives much valuable information for improvement. The results of these tests are presented in Section 5.

6.1 Design Overview

The project is divided into five main components. In Figure 42 is provided the architecture overview of the whole project.

- The Main component defines a number of nodes for which the worst-case performance rate should be computed.
- The Policy Generator component gives in turn all policies with a given number of nodes.
- The Optimal Algorithm component computes a round sequence of minimal length.
- The Heuristic computes a round sequence according to an heuristic.
- The Comparator component interacts with all other components. It receives a number of nodes and computes the corresponding worst-case performance rate. It considers all policies for this number of nodes, and in turn uses the number of rounds the Optimal Algorithm, respectively the Heuristic, need to update this policy, to compute a candidate worst-case performance rate. At the end, the lowest candidate worst-case performance rate is kept as the worst-case performance rate.

Each component corresponds to a package in the Eclipse project. For simplicity, each package contains a single public class, called manager class, that communicates with the other components. All other classes of a package are private.

When saving information, two different files are used, with specific purposes.
- The log file is used by any ongoing process to store temporary information. For example, the Heuristic component stores the DFs in it while they are constructed. This file has a size of 1GB for \( n \geq 12 \).
• The output file is used exclusively for storing final results. The Worst Case Manager class of the Comparator component saves the worst-case performance rates in it.

Additionally, the network is simulated by a Network class implementing core functions required by all other algorithms. It is responsible for low-level tasks such as update execution, as well as high-level tasks, such as compiling distances. It also implements a range of test functions, e.g., indicating whether a node is certainly updated (new) or has not responded yet to an update order (limbo).

The project we designed has a modular architecture allowing easy updates and modifications.

6.2 Comparator

The Comparator component interacts with all other components. Its task is to iterate over all policies for a given number of nodes, to compute each time a candidate worst-case performance rate, and finally to extract from this list the worst-case performance rate.

The Comparator component comprises two classes, Worst Case Manager and Worst Case Finder. In Figure [43] is provided the architecture overview of the Comparator.

![Figure 43: Design of the Comparator package](image)

• The Worst Case Manager class is responsible for handing a number of nodes over to the Worst Case Finder and saving information relative to the worst-case performance rate queried from the Worst Case Finder. It is the class communicating with other modules.

• The Worst Case Finder class considers policies one by one and computes in turn a candidate worst-case performance rate. Each time a new lowest worst-case performance rate is found, it updates the information relative to it.

It should be noted that the Comparator uses OPTCONFBS to compute a minimal round sequence, so that, as OPTCONFBS only supports LF, only the performance of an algorithm enforcing LF is currently tested.

The Worst Case Manager class leverages a simple test to avoid unnecessarily running the Worst Case Finder. When handed a given number of nodes, it starts by checking in the output file whether the corresponding worst-case performance rate has already been computed, and ignores the query if it is so.
Additionally, for consistency, the *Worst Case Finder* does not only computes the worst-case performance rate for the number of nodes for which it has been queried. It also computes the worst-case performance rates for all lower numbers of nodes not considered so far.

### 6.3 Policy Generator

The *Policy Generator* component is responsible for creating, managing, and retrieving policies for the other components. Depending on the other components’ wishes, *Policy Generator* can create a predefined policy, a random policy, and iterate over all policies constructible with a given number of nodes.

For simplicity, other components only communicate with the class *Policy Generator Manager*, that will then work with a specialized class of the *Policy Generator* responsible for the task required by the component having queried *Policy Generator Manager*. In Figure 44 is provided the architecture overview of the *Policy Generator*.

![Figure 44: Design of the Policy Generator package](image)

- The *Predefined Policy Generator* class creates a policy specified by an index. For example, the policy with the index 1 contains seven nodes. $v_1$ and $v_2$ point at $v_6$ and $v_7$, respectively, and all other nodes point at the node preceding them.

- The *Random Policy Generator* class generates a policy at random. It receives parameters describing the Gaussian distribution that the neighbor assignment should follow, and assign neighbors accordingly.

- The *Policy Iterator* class receives a number of nodes and iterates over all valid policies with this number of nodes. Each time it is queried, it gives a policy not provided so far, until exhaustion.

Three versions of the *Policy Iterator* have been implemented.

1. The *first* version stores all policies in memory, regardless of whether they are invalid or whether a node has points at the same node on the old and the new paths. When queried for a policy, the *Policy Iterator* class takes the policy following the one it gave last, tests it, and if it is found invalid, consider the following one, and so on, until a valid one is found.

2. The *second* version stores all valid policies. In a preprocessing step, policies are filtered, so that the *Policy Iterator* simply gives the following policy upon query.
3. The third version computes the next policy iteratively solely on the basis of the previous one, using a predefined ordering scheme.

The first version could not scale beyond 8 nodes (8! policies) on a standard host. Likewise, the second version ran out of memory beyond 10 nodes (10! policies). Thus, all experiments are run with the third version, storing only one policy at a time and creating thus no memory constraint.

6.4 Heuristic

The Heuristic is the component that implements the suboptimal algorithms implementing the dependency backward dependency (Definition 31) namely LFBCDHEUR (Algorithm 11), LFCDEUR2 (Algorithm 14), LFCDUR3 (Algorithm 15) and LFCDEUR4 (Algorithm ??).

The first architecture of the implementation LFBCDHEUR is provided in Figure 45. It comprises three classes.

![Figure 45: Design of the Heuristic package with the triple architecture](image)

1. The Forest Manager class creates a DF based on the LFBCDHEUR. When given a node, it can provide its list of children.

2. The Weight Manager class computes the weights and retrieves lists of nodes in decreasing order of weight.

3. The Heuristic Manager class coordinates the other components.

The second architecture of the implementation LFBCDHEUR is provided in Figure 46. It differs from the previous architecture in that the Weight Manager is split into two classes, a Weight Manager in charge of the weight computation, and a Level Manager responsible for sorting nodes and managing lists of nodes.

The Level Manager sorts nodes first by weights and then using a distance metric. It can support more metrics and their order is customizable. The Level Manager gives iteratively one list at a time.
6.5 Optimal Algorithm

The Optimal Algorithm (Algorithm 13) is responsible for computing a round sequence of minimal length. It does so by computing all possible round sequences for a given policy update and selecting one of shortest length.

The architecture’s implementation is provided in Figure 47. Each class relies on the output of the previous one.

1. The Cluster Manager class receives a policy update and computes independent clusters (cf. Definition 1) of nodes that can be updated independently from one another. It manages them and gives them in turn to the Optimal Algorithm Manager that computes for each a minimal round sequence.
2. The *Optimal Algorithm Manager* class implements an interface for other components and coordinates the other classes inside the *Optimal Algorithm* package. On each independent cluster, the *Optimal Algorithm Manager* computes, using the five other components, a minimal round sequence, and ultimately combines them.

3. The *Loop-Free Configuration Manager* class generates and list of all possible transient update configurations. It hands over configurations iteratively to *Loop-Free Configuration Manager*.

4. The *Loop-Free Configuration Manager* (loop-free configuration manager) class filters and manages the loop-free configurations. It iterates over the list managed by *Loop-Free Configuration Manager* and keeps only the loop-free configurations. It hands over loop-free configurations iteratively to *Loop-Free Configuration Manager*.

5. The *Single Loop-Free Update Distant Configuration Manager* (single loop-free update distant configuration manager) class creates a forest of (loop-free) configurations such that a parent configuration can become one of its children in a single update. It iterates over the list of loop-free configurations managed by *Loop-Free Configuration Manager*, and for each configuration, *Single Loop-Free Update Distant Configuration Manager* computes the list of configurations into which it results after a single update, filters out the ones with loops, and finally mark the resulting list as its children.

6. The *Single Loop-Free Round Distant Configuration Manager* (single loop-free round distant configuration manager) class compiles a forest of (loop-free) configurations such that a parent configuration only requires one loop-free round to become one of its children. It uses the *Single Loop-Free Update Distant Configuration Manager* to verify that any round sequence is loop-free.

7. The *Update Path Finder* class determines a shortest loop-free round sequence. To do so, it assigns distances to the new policy to all transient policies with a breadth-first search (BFS) on the forest managed by *Single Loop-Free Update Distant Configuration Manager* from the new configuration to the old one. Note that each time a configuration receives a distance that makes it update its distance to the new policy, *Update Path Finder* records the configuration whose advertisement triggered this update. Thus after the BFS finishes, *Update Path Finder* can recompute a shortest path from the old policy to the new.

### 6.6 Future Implementations

As a side note, it should be mentioned that *Worst Case Finder* currently only provides a single policy update corresponding to the worst-case performance rate, as well as a single minimal round sequence, since this information is already enough to spot weaknesses. However, worst-case performance rate could be easily tuned so as to hand over all policy updates resulting in *a* having a *performance ratio* equal to the worst-case performance rate, along with all minimal round sequences for each of these policy updates.

Moreover, the *Worst Case Finder* only supports the constraint LF so far, but could be modified to alternatively support WE and LFWE. To do so, three different optimal algorithms should be eligible as $opt$, namely not only $opt$CONF$BFS$, optimal with respect to LF, but also modified versions of $opt$CONF$BFS$, optimal regarding WE respectively LFWE. Implementations of these modifications are described in Section 4.4.4. With these three versions of $opt$ in hand, *Worst Case Finder* could then measure the performance of algorithms enforcing not only LF but also WE or LFWE.

These extensions will probably be of some use for future work.
7 Discussion

This work is only a first attempt to uncover the problems and solutions pertaining to the synchronization of updates in a network. Many more aspects would require studying.

7.1 Minimizing Pauses: Cleaning

Pausing (Section 2.3.3), namely letting the controller wait so that all packets having entered the network before it started waiting have reached their destination, is only one of many solutions to ensure that a specific set of packets have been delivered at a given location, a process we dub cleaning. Two subprocesses belonging to cleaning are studied in this work: (1) making sure that all packets having left a node before a given instant have been received by the node on the other end of the link, which is called local cleaning, and (2) making sure that all packets having entered the network before a defined point in time have left it, called global cleaning (pausing is a global cleaning technique). Methods and considerations with respect to both aspects are reviewed in the remaining of this section.

7.1.1 Local Cleaning

The first way to clean the network, local cleaning, consists in the controller waiting long enough to make sure that a given set of links is free of packets, namely clean:

**Definition 41.** A link \((v, v')\) is said to be clean when no packet remains on it. A path \(\{v_1, ..., v_p\}\) is said to be clean when all its links \((v_i, v_{i+1})\) are clean.

**Local Cleaning** is relevant for specific applications, such as pausing or enforcement of ND (Section 4.2.2). Indeed, partial pausing makes sure that no packets are on a specific path, and relies on this assumption to realize consistent updates. Likewise, ND requires that \(v\)'s ingress link (the link connecting \(s_o(v)\) to \(v\)) is clean before updating \(v\): when \(v\) removes its forwarding rule without adding a new one, a packet on its way to \(v\) would automatically be dropped.

Two exemplary techniques to realize local cleaning are sketched out below, one requiring an interaction between each node \(v\) and its successor \(d_o(v)\), the other an interaction between each node and the controller.

The first solution for local cleaning is distributed. It consists in the use of circular labels assigned to packets in a round-robin fashion: The packet \(p'\) sent right after \(p\) should have the label \(l' = l + 1 \mod m\) with \(m\), the modulo, large enough to avoid collision (two packets with the same label). \(d_o(v)\) could then ask \(v\) the label of the last packet it sent to \(d_o(v)\), so as to make sure that it has received all packets from \(v\) and can now consider the link as obsolete. A special packet would then indicate that the link has been closed. This solution may however imply a high overhead, and the loss of any packet would lead the network to a dead end.

The second solution for local cleaning is centralized, i.e., only relies on the controller’s doing: the controller always waits at least \(\theta\) between the update of \(v\) and that of \(d_o(v)\), for any \(v\) in the network, where \(\theta\) is an upper bound for the time required for a packet to traverse any given link. That is, provided \(v\)'s update should be scheduled, the controller only sends an update to \(v\) after \(\max(0, \theta - \tau)\), where \(\tau\) is the time elapsed since \(s_o(v)\) was updated.

In practice, both solutions are only completely safe with the use of infinite resources: the first solution would require an infinite modulo \(m\), while the second would require an infinite \(\theta\). The higher they are, the stronger the confidence that local cleaning was effective, i.e., that no packet remain on the link, and the more probable it is that few packets remain on the link. Accordingly, these parameters should be tuned so as to ensure a desired quality of local cleaning.
7.1.2 Global Cleaning

The second way to clean the network, *global cleaning*, consists in the controller waiting long enough to make sure no packets having entered the network before it started waiting are still underway.

The *clean start* (Assumption 1), the assumption that, at the beginning of a policy update, no packets remains partially forwarded according to a previous policy, can be ensured by *global cleaning*. Moreover, pausing, waiting a given time $\Theta$ selected large enough to ensure *global cleaning*.

The first solution to realize *global cleaning* consists in pausing: the controller waits a duration $\Theta$ that is an upper bound for the time required for a packet to traverse the entire network (*transfer time*), e.g., $n\theta$.

Another solution would be to introduce a global ordering of packets: each packet incoming at the source is assigned a label in a round-robin fashion (modulo $M$), so as to uniquely characterize the ordering of incoming packets. When a *global cleaning* is started, the controller notifies the source of it, which in turn indicates the controller the index $i_{last}$ of the last packet it sent before receiving the *global cleaning* notification. The controller then hands out $i_{last}$ to the sink that ultimately notifies the controller when all packets with indexes in $i_{last} - M/2 ... i_{last}$ have passed through it. For this to work, the modulo $M$ of the round-robin system should be selected high enough so that the source cannot send a packet with a given label when another packet with the same label remains in the network.

In practice, the same problem arises for the *global cleaning* as for *local cleaning*: No finite $\Theta$ or $M$ can fulfill the required conditions, namely ensure that all packets existing before have traversed the network. The higher these thresholds, the higher the confidence that no packets previously present remains in the network, so that they should be chosen accordingly to the controller’s needs.

7.1.3 Cleaning and Loop-Freedom

While the concept of pausing and more generally that of *cleaning* were introduced to solve WE-problems otherwise unsolvable, it can also be applied to LF-problems.

An example of such a situation was introduced in Section 2.4.4. Two algorithms were presented, one updating the network without any *cleaning* technique, and one with pausing. The first algorithm needs $n - 2$ rounds, and the second one only 2. The second algorithm appears to be advantageous, but can actually be slower or quicker than the first one. Indeed, Pausing requires to wait a given time $\Theta$, while $n - 3$ take a non-deterministic time, say upper and lower bounded by $b_{low}$ and $b_{upp}$. If $b_{upp} < \Theta$, then the first algorithm should be favored, if $b_{low} > \Theta$, then the second algorithm is preferable, but otherwise none of them is clearly faster than the other. For future work, experiments should characterize how often each of the three previous cases occurs.

7.1.4 Limitations of Cleaning

As explained before, *cleaning* parameters, introduced in *local cleaning* and *global cleaning* are chosen to satisfy specific security requirements under specific conditions. These requirements consist in administrative wishes, e.g., the acceptable likelihood that a packet bypasses a waypoint. Conditions, on the contrary, evolve with the policy update, they can be data-plane conditions, namely expectations on the time a packet needs to go through some links or the whole network, or control-plane conditions, i.e., forward paths currently existing or likely to exist. Clearly, the more information taken into account in the computation of *cleaning* parameters, the more complex it becomes, but at the same time, lower parameters can then be used to reach the same level of security.
To understand the gain of precise information, let us consider two situations, in both of which the controller is required to clean a link $l$, and the requirements are that, with a probability of 90 percent, at least 99 percent of the packets currently on $l$ should leave $l$ during the *local cleaning*. Let us now assume that in the first situation, the controller lacks information (throughput, congestion risk...), e.g., because the information it has on $l$ is stale, while in the second, it has a very precise idea of $l$’s state. While in the first situation, the controller needs to use a high amount of resources (to be on the safe side), for example, wait a long time, to fulfill these requirements, in the second one, the controller does it with a much lower expense, e.g., wait a short time.

Similarly, while in all algorithms presented in this work, the only cleaning technique used is the global cleaning method pausing, where the controller waits a duration $\Theta$ long enough to renew entirely the set of packets currently in the network, such a high (temporal) expense can be spared if more information is included. For example, if the current control plane is analyzed, the controller may determine a small set of links needing to be locally cleaned to fulfill given requirements, and only focus on these.

Going even further, since any form of cleaning takes precious resources such as computation, tagging space, or time, future algorithms should resort to cleaning as rarely and as locally as possible. In this regard, WAYUP (Algorithm [17]) may not be optimal, since it uses pausing, taking a high amount of time. Another algorithm, relying on successive local cleaning techniques, may be preferable, but that would come at the expense of more computation on the controller’s side, as links to be waited for should be identified or the time to wait should be quantified.

In conclusion, cleaning methods are not a panacea, and should be used with care, as (1) they are never completely reliable, and (2) they always take a scarce resource, time. The lightest cleaning methods should be preferred, and cleaning should only be done when it cannot be avoided. For simplification, only pausing has been considered in this work, and, faithful to the time considerations, it was only used when it could not be avoided. Future work should also make use of local cleaning methods, and a balance between a high number of rounds and a high time spent waiting remains to be struck.

### 7.2 Imprecision of the dependency forest’s Lower Bound

While the maximal height of the initial dependency forest relying on backward dependency is a lower bound for the minimal number of rounds necessary for updating the network without creating a loop, this lower bound may be far below the actual minimal number of rounds.

**Lemma 6.** In the worst case, the maximal height of the DF running on backward dependency can be $(n - 1)/4$ times higher than the minimal number of rounds.

*Proof.* The configuration corresponding to this worst-case performance is displayed in Figure 49. On the initial DF— for the backward dependency—, presented in Figure 48, all $r$ backward nodes have the long-ranging forward node as a child. The DF thus has a maximal height of one and implies a lower bound on the number of rounds necessary for updating the whole network equal to two.

Nevertheless, an optimal algorithm *minimal number of rounds* would have the following round sequence. In the first round, it would update all forward nodes, and in the following ones, it would update backward nodes one by one from left to right (one in each round). This would take $r + 1$ rounds in total, where $r$ is the number of backward nodes.

The only remaining step to take is expressing $r$ in function of $n$. Each backward node is followed on $p_2$ by a forward node, accounting for $2r$ nodes, to which the source, the sink, and the node pointing at the sink should be added, i.e., three nodes. Thus, $n$ equals $2r + 3$, and $r = (n - 3)/2$. 

59
Finally, the ratio of the minimal number of rounds necessary for updating this network divided by the lower bound computed using the backward dependency is $(r + 1)/2 = (n - 1)/4$.

This lemma makes it impossible to estimate the duration of a policy update using the lower bound of the DF. The problem lies here in the lemma that the information that all backward nodes compete for the same way out (Definition 5), the long-ranging forward node, is not captured by the backward dependency. They all want to use the same way out to become updatable but as soon as one of them takes it, it stops being usable for the others.

To understand the phenomena at hand and why it is not detected by the backward dependency, some definitions need to be introduced. Let $m$ be the node preceding the sink on $p_2$, the set of long-ranging forward nodes be $F$, respectively that of long-ranging backward nodes be $B (m \notin F)$.

Initially, $B$ are backward but they can all be updated as soon as $m$ is new. No other node or combination of nodes can make nodes in $B$ updatable. However, after $m$ has become new, each time a node $v$ in $B$ is updated, if $s_o(n)$ is still old, $d_n(n)$ is backward.

To avoid this problem, the backward dependency should be augmented, so that the DF exhibits a more complete information. This way, more complex dependencies could be already spotted from the beginning, and a more precise estimation of the minimal number of rounds could be given. An attempt to do so is the extended dependency (Definition 32).
7.3 Cost of Cautiousness

While being too liberal with updates, updating too many nodes, can lead to a performance \( n/10 \) times worse than optimal (Section 2.2.2), not updating enough nodes is also dangerous. Indeed, algorithm LFSIMPDEP (Algorithm 10), updates a backward node \( v \) only if all nodes between \( v \) and \( sdc(v) \) along \( p_2 \) are new—a straightforward solution to ensure that \( v \)’s update cannot deprive any other node from its updatability, and is thus a very cautious algorithm. Unfortunately, LFSIMPDEP can need up to \( n/6 \) times more rounds than an optimal algorithm. This section presents the policy update corresponding to this worst-case performance.

Lemma 7. LFSIMPDEP can require \( n/6 \) times more rounds than minimal number of rounds to update a network’s policy.

Proof. Such a configuration is presented in Figure 50. One can immediately see that this structure is the same as that of the worst-case policy update in Section 2.2.2 i.e., Figure 19 but for the bridges that existed in Figure 19 and not in this one.

Confronted with that configuration, LFSIMPDEP would generate the following round sequence. In the first round, LFSIMPDEP would update all forward nodes. In the second round, all backward nodes after the long chain of short backward edges \( C \) will be updated, as well as the first backward node of \( C \). Then, in turn, backward nodes of \( C \) will be updated in turn, one in each round. This would thus take overall \( r + 1 \) rounds, where \( r = |R| \).

On the contrary, an optimal Algorithm minimal number of rounds would need only the constant number
of two rounds to update the same configuration. In the first round, minimal number of rounds would update all forward nodes, and in the second one, all backward nodes.

Now, \( r \) remains to be expressed in function of \( n \). Except for the last, each backward node on \( C \) is followed (on \( p_1 \)) to a long-ranging forward edge followed (on \( p_2 \)) by a long-ranging backward edge, so that \( r_1 \) nodes correspond each to 2 nodes. The last backward node is only followed by a single long-ranging forward edge, and then by the sink, adding 3 nodes to the sum. Finally, the first two nodes \( v_1 \) and \( v_2 \) on \( p_1 \) have to be added to this structure, as well as the (backward) node following \( v_2 \) on \( p_2 \), accounting for 3 additional nodes. This results in \( n \) being equal to \((r - 1) + 3 + 3 + 3\), that is, \( 3r + 3 \). Equivalently, \( r = n/3 - 1 \).

Finally, the corresponding worst-case performance rate of LFSIMPDEP (see Definition 23), ratio of the number of nodes LFSIMPDEP would need and that minimal number of rounds would need, equals \((r + 1)/2\), i.e., \( n/6 \).

LFSIMPDEP is only an exhibit of a broader problem, namely that cautious algorithms, delaying updates as much as possible, are not a solution for MINR. However, as seen in Section 2.2.2, updating too early can also dramatically fail to reach MINR. Accordingly, both extremes come with a worst-case performance rate
(Definition 23) in the order of $n$. Good algorithms need to carefully select nodes to update at each round, and take neither too many, nor too few.

### 7.4 Optimized Hybrid Approximation Scheme

As mentioned in Section 3.4.2, as computational resources are always limited, controller, so that computing a round sequence takes time. We are thus facing a trade-off: while a short round sequence implies a low update time –time between the first update order and the last update acknowledgment–, it comes with a high computational time –time the controller needs to compute it–. Conversely, simple algorithms take a short computational time but a high update time. A trade-off should thus be found between too light algorithms and too heavyweight algorithms.

A solution to this problem would be for instance to start two algorithms in parallel as soon as the controller is asked to implement a new policy. The first algorithm would compute quickly a very approximate solution, while the second would compute within a longer time a better solution. As soon as the first algorithm’s computation ends, corresponding update orders are sent, and the second algorithm is notified of all update orders so that it integrates the change of configuration in its computation. Then, when the second algorithm’s round sequence is computed, the controller follows it and finishes the network’s update fast.

This way, an approximate solution is found quickly, so that not too much time is wasted computing, but an better solution is produced soon after, so that round sequences do not become too long due to the initial haste. The ability of a slower algorithm to detect and correct mistakes of a quicker will be dependent on whether the controller is allowed to **downgrade** nodes –make them old again–.

As an example, LFSIMPDEP (cf. Section 4.4.2) is very lightweight, but would automatically discard any **shortcut** (cf. Definition 40). LFSIMPDEP would discard **shortcuts** but as soon as a better algorithm takes over, they would be put to use, so that this mistake would not have too negative consequences –only a loss of a few rounds. There is no need to **downgrade** nodes to compensate such errors.

However, it can happen that an approximate algorithm would make a mistake that could not be repaired by any algorithm, no matter how optimal, as long as CP holds, such as described in 2.2.2 –number of rounds increased from $O(1)$ to $O(n)$ because of a wrong start–. The better algorithm would need to waive CP to correct this initial misstep, that is, make some nodes old again, and start again from, which would be a small waste of time compared to starting directly with a good round sequence. Thus, errors made by the approximate algorithm would not be without consequences but at least, these would be lessened in comparison with the approximate algorithm running alone.

It should be noted that in such a case, not using the approximate algorithm, but simply waiting for the better one to complete its computation, would have been advantageous. Indeed, situations in which approximating may be harmful can occur. To avoid the risk of having to **downgrade**, cautious –avoiding updates that would need to be undone– approximative algorithms should be used, such as LFSIMPDEP.

To conclude, combining an approximate algorithm and a slower but better performing one would offer advantages of both. The updates would start shortly after the controller is asked to enforce a new policy and the update itself would take a short time. However, the better algorithms may need to undo what the quicker algorithms have done to compensate their imprecision efficiently, which, as seen in Section 7.6.3 implies a very high computational cost.

### 7.4.1 Formal Approximation Process

Going even further, instead of having two algorithms running in parallel, a continuous range of algorithms of increasing MINR-performance and computational time could be executed in parallel. The controller would
start with the round sequence of the simplest algorithm and then in turn switch to a better round sequence each time a heavier algorithm’s computation comes to its end.

Formally, the combination of approximate algorithms works in the round model (7.4.3): An algorithm computes, based on an old policy \( p \) and a new one \( p_2 \), a round sequence that the controller simply follows without adapting on the fly to variations in update time (i.e., in a static way). The event-based model will be studied in the next section (7.4.3) but for simplification, we remain in the round model in this section. The combination of approximate algorithms then works as follows.

Initially, a new policy \( p_2 \) is given to the controller to implement. It is handed to the algorithms \( \{a_1...a_b\} \) (in decreasing order of speed) are handed \( p_1 \) as an origin policy and \( p_2 \) as a goal. During the network update, the origin policies of some algorithms will be modified, and we assume that these modifications do not delay them (techniques to ensure this assumption will be reviewed in 7.4.2). Under this assumption, \( a_j \) will clearly finish before \( a_j' \) for any couple \( j, j', j < j' \).

\( a_1 \) being the quickest, it is the first to produce a round sequence, say \( \{R_{1,1}, R_{1,1,t}\} \). The controller starts \( R_{1,1} \), and notifies \( \{a_2...a_b\} \) about it, so that they redefine their source policy as \( p_1 \). \( R_{1,1} \) where nodes of \( R_1 \) are new (equivalently, add the constraint that \( R_1 \) is the first round). If \( a_2 \) does not finish before the end of \( R_{1,1} \) i.e., before the controller receives all update acknowledgments of \( R_{1,1} \), then the controller starts \( R_{1,2} \), and notifies \( \{a_2...a_b\} \) again. However, if \( a_2 \)'s computation ends during \( R_{1,1} \), then \( R_{2,2} \) is used as a second round. If \( a_2 \) and \( a_3 \) finish during \( R_{1,1} \), then \( R_{3,2} \) is used as a second round, since \( a_3 \)'s result is better. Likewise, the algorithm of highest index is always favored.

More generally, let \( A_{||} \subset \{a_j|1 \leq j' \leq j\} \) be the set of algorithms having finished their computation during a round in \( \{R_{k',k'}\}, k' \leq k \}, \) with \( j_k \) largest index in \( A_{||} \). The next round is then computed on the fly according to the following pattern: \( R_{k+1} = R_{j_k,k+1} \).

### 7.4.2 Avoiding Delays

Initially, the algorithms \( a_1...a_b \) are started in parallel, provided with the old and new policies, and their input is modified on the fly as nodes get updated. As mentioned in the previous section (7.4.1), we assume that this process does not delay the algorithms. For some algorithms, this assumption is true, and for others it only holds approximately.

For instance, OPTCONF(BFS (Algorithm 13) builds a forest –and a directed acyclic graph without CP– upon which it computes the distances and the shortest path from any intermediate policy to the new policy. Accordingly, it can provide a round sequence for a policy different from the initial one without any additional computational cost.

Along the same lines, to make algorithms robust to changes of the initial policy, instead of building round sequence from the old policy \( p_1 \) to the new policy \( p_2 \), they should construct them reversely, i.e., from the last round to the first round. Algorithms should compute a round sequence from \( p_2 \) to \( p_1 \), which is equivalent to a round sequence from \( p_1 \) to \( p_2 \), but with this technique, there is a hope that only a few of the last rounds need to be modified when \( p_1 \) is updated, and dependencies are not affected.

### 7.4.3 Approximations with Event-Based Updates

In the frame of event-based updates (Section 7.5), algorithms’ initial policy is not recomputed at the beginning of each round anymore, but upon each update acknowledgment or update order. This is a major difference with the round model, in which algorithms were simply told to consider the nodes to which orders are sent during the set, as new instead of old at the beginning of each round. Indeed, in the event-based model, when an update order is issued for a given node, its status is updated as limbo, and only when its
update acknowledgment is received does it become new. Thus, the frequency of updates that algorithms are prompted to take into consideration on the fly is much higher, and induces a significant computational stress.

Another difference, more fundamental, is that no algorithm can foresee what update acknowledgment will be received next, as it is an indeterministic process. However, as discussed in Section 7.5.3, algorithms pre-compute the update orders they would make the controller send in case the one or the other update acknowledgment is received, but quicker algorithms will have pre-computed more of them, and be therefore less likely to be surprised—meaning not to have any updates to recommend to the controller when an update acknowledgment arrives—.

7.4.4 Algorithms with various Precision Levels

Future work should investigate the (asymptotic) guarantees offered by only computing dependencies up to a given number of hops or up to a given distance in trees.

7.5 Event-Based Updates

So far, in this work, the controller sent at the beginning of a round $r_i$ all update orders of $r_i$ in a single batch. Then, the controller waited until it had received the update acknowledgments of all nodes in $r_i$, and only then sent the batch of $r_{i+1}$. Besides, nodes are assumed to be all touchable (i.e., following nodes on $p_2$ are updated, formalized in Definition [37]) at the beginning of a round, so that the controller can send update orders to them in a single batch (methods to ensure that condition are reviewed in Section 4.2.3). Updating nodes in rounds is a straightforward approach, and has the advantage of giving a simple metric to compare the performance of different algorithms: an algorithm $a$ will be deemed better as an algorithm $a'$ regarding the objective MINR (minimizing the number of rounds) if $a$ requires less rounds than $a'$, namely if the controller sends less update batches. $a$ is then said to be quicker than $a'$.

However, in practice, the very way the controller deals with rounds is far from optimal. Two algorithms are introduced below to improve on the classic round update mechanism described above, namely partEBU, remaining in the spirit of rounds, and fullEBU, abolishing it.

7.5.1 partEBU: Partially Dynamic Algorithm

In partEBU (Algorithm [21]), instead of waiting for $r^u_i$ (set of nodes to be updated in $r_i$) to be entirely new to start updating $r^u_{i+1}$, the controller does update $r^u_i$ with top priority, but also updates $r^u_{i+1}$ as soon as it becomes possible (according to the considered set of constraints). Thus, the updating process is drastically sped up.

**Algorithm 21** partEBU: Improved Round Update

| Input: rounds $r_i$, $r_{i+1}$. |
| Output: $r^u_i$ updated |
| 1: send update orders to $r^u_i$ |
| 2: until $r^u_i$ entirely new: |
| 3: each time an update acknowledgment is received from a node in $r^u_i$, send update orders to the the nodes of $r^u_{i+1}$ having just become updatable |
7.5.2 fullEBU: Completely Dynamic Algorithm

Going even further, in fullEBU (Algorithm [22]), instead of having offline-defined rounds that the controller is supposed to follow whatever happens, it adapts its strategy continuously to current policy. In an initial round, the controller sends update orders to all nodes. Then, as soon as the controller receives an update acknowledgment from a touchable node or if a touchable node becomes touchable (\(wo_t(v)\) fully new, cf. [37]), it hands in to \(a\) as input \(p_1\) the current policy, defined by the current sets of touchable old, limbo and new nodes, as an input to \(a\), and as \(p_2\) always the same \(p_2\), namely the new policy to enforce ultimately. \(a\) will then, based on the knowledge of the current policy, compute a set of nodes to immediately update (that may be empty) so that the controller immediately sends update orders to them. In this manner, the controller will hence always adopt an optimal strategy to reach \(p_2\), and make the best use of its current knowledge of the network policy.

Nevertheless, this technique has limitations. First, the line model is lost, since \(p_1\) is not a path policy and even has two forwarding rules attached to limbo nodes. This makes the round sequence computation harder. Second, instead of computing a new round sequence each time the controller is requested to enforce a new policy, \(a\) now has to compute a round sequence after each update acknowledgment, that is, \(n\) times more often. Clearly, if \(a\) is an optimal algorithm and if the update time is low, \(a\) would not be able to provide a new round sequence between two update acknowledgments, or at a high cost on resources, and the question may then arise whether it is worth it. Future work should shed light on a trade-off between a high computation frequency and a high computation cost.

Algorithm 22 SYNTOUCHENF: Synchronous Touchability Enforcer

| Input: policies \(p_1, p_2\). |
| Output: network in \(p_2\). |
| 1: send concurrently update orders to all nodes |
| 2: upon update acknowledgment reception, recompute set of nodes to update and send |
| 3: return the round set |

7.5.3 Minimization of the number of updates per round

Two possibilities exist, the sets of update orders can be computed on the fly, namely right after an update acknowledgment has been received, or can be pre-computed, i.e., computed in advance for all update acknowledgments that can arrive.

The first approach, online computation, has the drawback that some time is lost between the update acknowledgment and the emission of the next update orders, but avoids the unnecessary computation of computing round sequences that will never be used. If all nodes need a priori the same amount of time to update themselves, an optimal algorithm along these lines would be dynEvBa (Algorithm [23]). Whenever the network is in an undetermined state because some nodes are limbo, the transient policies that can be reached without risking to violate specified constraints are called accessible transient policies. Note that, unless dropping packets is allowed, nodes are always touchable when the controller sends them update orders.

On the contrary, the second approach, pre-computation, has the disadvantage of being computationally expensive, but avoids any latency between the reception of an update acknowledgment and the controller sending the new update orders. If all nodes need a priori the same amount of time to update themselves, an optimal algorithm would then be optEvBa (Algorithm [24]).

As an example of a run of optEvBa, let us present the possible schedules of updates using optEvBa to
Algorithm 23 dynEvBa: algorithm computing on the fly update sets minimizing the expected time until the new policy is enforced

**Input:** old and new policies.

**Output:** network in new policy

1: while network not in new policy do
2: for each new update acknowledgment received do
3: compute $S_1$, set of (transient) policies in which the network can be
4: compute $S_2$, set of (transient) policies accessible with a single batch of updates from all policies in $S_1$, and $p$ policy where all limbo nodes are new
5: select in $S_2 \cup \{p\}$ a policy $p_g$ a minimal number of rounds away from the new policy
6: send simultaneously update needed to take network from $p$ to $p_g$ orders in $S_2$ a policy a minimal number of rounds away from the new policy

Algorithm 24 optEvBa

**Input:** old and new policies.

**Output:** network in new policy

1: compute and store all possible sequences of batches of update orders using dynEvBa
2: for each new update acknowledgment received do
3: Look up batch of update orders to send

implement the policy update shown in Figure 52. Since the closest policy of the old policy is the policy $p_{1,2}$ (policy where $v_1$ and $v_2$ are new, and $v_3$ is old), the controller first sends the update orders $v_1$ and $v_2$. If the controller receives $v_1$’s acknowledgment first, it knows that the network is either in $p_1$ or in $p_{1,2}$. The closest policy available from the set $\{p_1, p_{1,2}\}$ is $p_{1,2,3}$, which is $v_3$’s update away from $p_{1,2}$, thus the controller issue $v_3$’s update order. On the contrary, if the controller receives $v_2$’s acknowledgment first, it knows that the network is either in $p_2$ or in $p_{1,2}$. The closest policy available from the set $\{p_2, p_{1,2}\}$ is $p_{1,2}$, which is $v_3$’s update away from $p_{1,2}$, thus the controller keeps waiting, and only sends an update order to $v_3$ after it receives $v_1$’s acknowledgment.

![Figure 52: Policy optimally updated using optEvBa](image)

If the computational power allows it, *pre-computation* will thus be used, as it saves time. However, in practice, a trade-off may be required, that is, pre-computing outputs for only a fraction of the update acknowledgments that can arrive—for example, the ones expected the soonest—.

### 7.6 Allowing for Time Flexibility

In this subsection, we investigate the consequences of removing assumptions we made throughout this work about the update time. First, we let the update time be unbounded and identify some mechanisms that could
define the controller’s behavior in such a case. Second we consider the possibility of allowing new nodes to become old again, e.g., for error correction.

7.6.1 Unfortunate Future: Statistically Optimized Decision Process

Making in the first place the unrealistic assumption that the update time is deterministic, update time is likely to vary from nodes to nodes. Accordingly, at the beginning of a round $r_i$, when the controller sends concurrently update orders to $r_i^n$, it may receive acknowledgments from all but a single within a short time, and needs wait for the last acknowledgment for a long time, before finally being able start the round $r_{i+1}$.
The update time information can also be embedded in the computation of the rounds, so as to bring together nodes of similar update times, for example. However, this solution has two flaws: (1) time would still be wasted waiting for acknowledgment, and (2) more rounds may be necessary (a trade-off between the a minimization of the number of rounds and a minimization of update time imbalances within rounds would have to be found).

Getting rid of the round model with the methods introduced in the previous section (Section 7.5), the controller never waits needlessly: the controller always sends an update order when it is relevant to do it. However, these methods also discard the update time information, and would thus still not be optimal. Indeed, if, for some reason, a node $v$ needs to be updated, and if $v$ depends on $v_1$ or on $v_1'$, while $v_1'$ is currently not updatable and depends on $v_2'$, then $a$ will ask the controller to update $v_1$, since it seems to make $v$ updatable quicker. However, if $v_1$ has an update time larger than the sum of the update times of $v_1'$ and $v_2'$, updating $v_1'$ and then $v_2'$ would be a better strategy. Accordingly, it appears that a good algorithm should take into account the update time.

Unfortunately, the update time is nondeterministic... yet perhaps not completely unknown: Let $r_{ut_v}(\tau)$ be the remaining time the controller needs to wait before receiving an update acknowledgment from $v$, knowing that the time $\tau$ has elapsed since it sent an update order to $v$, if $r_{ut_v}(\tau)$ can be approximated or modeled as a (Poisson) random variable, then a statistically optimal decision process can be set up. That is, the controller can then determine at each instant $t$ whether updating a further node or waiting for a further update acknowledgment is statistically the best choice, and then act according to what is statistically best (to its current knowledge). This assumes, first, that such a statistical knowledge is available, and, second, that a sort of scoring function would be able to bind all remaining update times to a metric expressing the distance from the current policy to $p_2$. Both are left for future work.

7.6.2 Unfinished Policy Updates

Due to the constraints laid on the policy update, the controller may very well be left with no update it is allowed to do, and that could last for an arbitrary amount of time. Indeed, so long as the acknowledgment of a given node $s$ has not been received by the controller, it has to consider $s$ as limbo and if several nodes are limbo, the controller may not be allowed to update any further node at all, for fear of an unsafe transient configuration. However, in an asynchronous environment, acknowledgments may never come, resulting in this dead-end configuration lasting forever. To avoid this, the only solution available consists in releasing some constraints, i.e., allowing for transient unsafe configurations. A simple approach in this case would be, for example, to define a threshold after which the controller stops waiting for update acknowledgment, and allows for transient updates: The controller would consider that the node with the largest remaining update time is old, or that the one with the lowest remaining update time is new, and see if nodes become updatable under this assumption. If not, it would then make further assumptions, until progress is possible. Future work should cast light on where to place the trade-off between giving up too early and too late.
Moreover, the hypothesis that policy updates never overlap, part of the line model, defined in Assumption ??, and the assumption that no packet forwarded according to a previous policy remain (Assumption [1]) can be invalidated if the frequency of policy changes is too high. That is, the controller may be required to set up the policy \( i \) while some forwarding rules of the policies \( i - 2 \) and \( i - 1 \) are still running, the line model would then not be applicable anymore, that is, the policy update would start from a policy that would not be a path policy. This would simply be equivalent to the recomputation of the set of nodes to send update orders upon reception of an update acknowledgment, and should be treated using event-based techniques, such as SYNTOUCHENF(Section 7.5). However, instead of only allowing for changes in node states (untouchable, touchable, old, limbo, or new), changes in new policy should also be accounted for.

Unfortunately, by updating from \( p_1 \) to \( p_2 \), the network may have reached a state in which no node is LFWE-updatable to \( p_3 \), assuming the CP constraint (i.e., no node can be downgraded to a policy other than the last one). As an illustration, consider Figure 53; both updating from \( p_1 \) to \( p_3 \) and from \( p_2 \) to \( p_3 \) is trivial and can be done by updating all nodes concurrently (as soon as they become touchable). However, if the controller is asked to update the network from \( p_1 \) to \( p_2 \), and subsequently, while not done yet with the update to \( p_2 \), to \( p_3 \), then the path from source to sink could at this instant consist of \( p_1^{\text{wp}} \) and \( p_2^{\text{wp}} \) (assuming no node is limbo). If that is the case, then the controller is faced with a policy update such as presented on Figure 25 in Section 2.2.2, that is, a policy update impossible to update, even with Pausing. However, without CP, i.e., if nodes can be downgraded, then the controller has an easy way out: the controller can reverse back the network from its current state to \( p_1 \) (cf. Section ??), and from there on update trivially the network from \( p_1 \) to \( p_3 \) by updating all nodes concurrently.

![Figure 53: Policy update requiring a pause. solid line: initial path \( p_1 = \{v_1, v_2, v_3, v_5\} \), dashed line: second path \( p_2 = \{v_1, v_3, v_4, v_5\} \), fine dashed line: third path \( p_3 = \{v_1, v_4, v_3, v_2, v_5\} \).](image)

Additionally, if policy updates keep coming without the controller having enough time to execute them completely, stale rules of previous policies will accumulate, making the controller’s task of enforcing a new policy update harder, since more dependencies need to be taken into account. However, the assumption that policies are defined both by the initial source of the packet and by its final destination (Assumption ??) creates small policies, and makes it unlikely that the controller cannot catch up with policy updates. For future work, experiments should be made to assert how often the controller is overwhelmed, namely is asked to execute a new policy update while still enforcing a previous one.

In any event, if effectively the rate of policy updates is too high for the controller to implement one before being asked to implement the next, this rate will eventually decrease, and the controller will be able to implement the last policy update, albeit it might take a long time to clean all resilient stale rules. On the contrary, to provide for such events, a tagging scheme would need to have as many labels as simultaneous policy updates can happen, and will be overwhelmed if the number of simultaneous policy updates increase the foreseen boundary.
7.6.3 Relaxing CP

**Error Corrections** Discussed in Section [4.2.2] CP was always observed throughout this work.

However, allowing downgrading (i.e., making a new node old) may allow for better round sequences. For instance, consider an algorithm $a$ computing rounds iteratively: $a$ computes the $i$th round and then considers $\{r_j, j \leq i\}$ as immutable and computes the $i + 1$th round based on $\{r_j, j \leq i\}$, so as to bring the current policy closer to $p_2$ (a metric between policies would have to be informally or formally defined). In case $a$ were provided the network introduced in section [2.2.2], it may make the mistake of updating bridges in a given round $r_i$ and, by doing so, rendering shortcuts (nodes with long-range forward outgoing edges) unusable for the future. If $a$ could somehow notice in a latter round, say $r_i'$, that it did a bad choice, $a$ would have the two following possibilities to correct its mistake.

The first approach would be for $a$ to go back on its decision and to modify the set of nodes in $r$, however this going back and forth would probably amount to a computational load, possibly exponential.

Another solution would then consist in $a$ downgrading the bridges, and coming in $r_{i'+1}$ to a better policy (once more, from the considered metric viewpoint) where shortcut nodes are once more reachable. In other words, instead of going back on its computation and preventing the error, $a$ corrects it on the fly. Clearly, this second approach does not provide a solution as good as the first one, but has the advantage of avoiding expensive iterative computations.

Interestingly, if a round sequence is loop-free, then its reverse is also loop-free.

**Allowing Optimal Solvers to downgrade** A consideration orthogonal to the previous one is that even an optimal round sequence may require downgrading: Perhaps, an optimal algorithm would need to make a node $v$ new because it helps updating a subset $S_\infty$ of the network, then make $v$ old again because it helps updating a subset $S_\in$, and finally render $v$ new again to help update $S_\infty$. For future work, OPTCONFBFS, described in [4.4.4] currently forbidding downgrading, could be tuned to allow it by making links between configurations bidirectional, and optimal not progressing round sequence may thus be detected.
8 Related Work

The Software-Defined Networking (SDN) paradigm separates the network management and operation (control plane) from the actual packet forwarding (data plane). Converting a traditional network into a software-defined network is not a trivial task, but once this done, it is highly beneficial to network management [10]. With SDN, many complex network guarantees and functionalities become straightforward to implement, while they were unconceivable in traditional networks. This is why SDN has already been widely implemented both in enterprise networks [1, 15, 16, 17, 18] and for research application [14], network simulations and virtualization engines [19, 20, 21, 22], and state test tools [23, 24, 25, 26]. Additionally, recent work has enabled hybrid networks (comprising both SDN switches and legacy switches) to obtain SDN capabilities, in particular Panopticon [10].

However, during a policy update, although both the initial and the final policies are in compliance with specified guarantees, the network may go through transient policies violating them. This had a significant impact both on performance considerations —congestion— and on security concerns —middlebox temporarily bypassed. To ensure transient consistency, a range of tools have emerged, developed as a framework [2, 13, 27] or a language, such as Frenetic [28], NetCore [29], or Pyretic [30], and presenting a set of functionalities with specific consistency guarantees. The user then designs SDN applications calling these primitives which automatically deal with required constraints without the SDN application needing to consider consistency issues. This set of primitive functions is often dubbed Network Operating System [31, 32] —in analogy to operating systems of computers.

SDN primitives constantly improve both in the level of functionality they provide and in the strength of consistency they ensure. To do so, they rely on tools in development. Regarding security, early work allowed the user to specify a middlebox [9], and later on, methods were developed to specify allowed end-to-end paths [33, 34], and middlebox management was simplified [35]. Fault-tolerance is also a key factor, for which both provisioning resources [36, 37] and pre-installing rules [38] are widely explored solutions. Network utilization guarantees (e.g., bandwidth, load) are also of utmost relevance since they are intrinsically bound to performance guarantees; they are widely studied [8, 16, 39, 40], in particular in relation to application needs [41, 42, 43, 44]. We expect that the algorithms presented in this work can be integrated into these primitives to enrich further the panel of functionalities they offer.

Additionally, there is a recent trend to partially migrate computing from the control plane to the data plane, both for scalability and for performance issues [38, 45, 46]. In this context, the control is handed to an intermediary agent in charge of enforcing the policy in the local environment. This agent must deal with specific local requirements (e.g., time, security, or performance constraints) and available resources (e.g., current network state, computational power). At the same time, local policies must be combined together concurrently, which represents new challenges [45, 47]. Since this work gives an overview of available algorithms with variable costs and guarantees, we hope that it will enrich the agent's function panel and help it make the most of real-time conditions in its administrative local network as well as give insight for policy coordination.

In parallel to the improvement and extension of automatic solutions to ensure consistency guarantees, much work has been done in understanding phenomena at hand. A number of network problems have been analyzed from the SDN viewpoint, including security [48], fault-tolerant and performing networks [49, 50], and a range of caveats have been uncovered [51]. Regarding updates, low-latency policy updates have been considered [26], and consistency has lately received much attention [5, 6, 7, 8, 52]. A line of work has been considering strong consistency properties, such as per-packet consistency (a packet will either be forwarded according to the old policy or the new one but never a mix of both) [7] or combining policies by either
installing them throughout the network or discarding them \[52\]. In parallel, \[6\] and \[5\] analyzed the impact of consistency on update latency, and maximized the number of nodes updated per round, which, as this work shows, does not necessarily go along with a low latency. Accordingly, we designed in this work algorithms minimizing latency under loop-freedom constraint, giving a lower latency. Additionally, by displaying and illustrating a wide variety of use cases and sketching out solutions for loop-freedom and waypoint enforcement, this work contributes to the understanding of the impact of consistency on performance and design choices.
9 Conclusion

This work is only a first step in the optimization of round sequences. Several directions need to be more developed.

1. A more flexible working model. The model used in this work could be rendered more complex, and thus perform better in reality. As an example, the controller waits until it has received all update acknowledgments of the round to start the next round, while it can already emit further update orders as soon as it receives an acknowledgment (an optimization would then be based on update time prediction).

2. Better algorithms. Using the framework we devised, more experiments can be done, and algorithms with a lower computational cost or a better performance (measured with the worst-case performance rate) should be designed.

3. More precise definitions. While most cases met throughout this work could be understood and defined, several interesting cases would be worth studying further, e.g., the impact of allowing a node to become old again.

4. Experimental measurements. The algorithms presented in this work could be implemented, e.g., using OpenFlow [53]. Of special interest would then be

   on the one hand measurements of the (online) computation time on-the-fly compared to the time the controller takes to realize the policy update, in the objective of minimizing the sum of both times, and

   on the other hand assessments of the possibility to precompute and cache (offline) round sequences and a study of the trade-off between the precomputation cost and the benefit in the minimization of the round sequence length.

The main objective of this work is to design methods for minimizing the number of rounds a controller needs to execute a specified policy update under the loop-freedom, waypoint enforcement constraints, and their combination.

Our key contributions are the following.

1. We highlight and characterize problems and impossibility results regarding LF- and WE-compliant policy updates, e.g., that under specific circumstances, the controller has no choice but to wait for packets to traverse the whole network before issuing a new update order if WE is to be observed. We also present a trade-off between, on the one hand, waiting a long time between broadcasts of update orders and sending many update orders at a time and, on the other hand, waiting little between broadcasts of update orders but sending only few update orders at a time. Our hope is that future work can lean on these results and make informed decisions.

2. We construct a theoretic framework for making updates quick and consistent. We introduce several forms of dependencies and discuss their potential and computational cost. Moreover, we introduce a wide range of algorithms minimizing the time to update the network, from very simple heuristics with little performance guarantees to algorithms updating the network in a minimal number of rounds. These tools can be reused and expanded in future work.
3. We present an experimental framework that does not only allow for an easy implementation of algorithms but also embed a performance evaluation mechanism. We have already implemented several algorithms, whose tests we present, but the modular architecture of the experimental framework allows for an easy development and testing of further solutions.

4. We sketch out some possibilities to make the model more flexible. We abstract from the notion of rounds and introduce algorithms updating the network at an optimal speed by sending updates as soon as they can, instead of waiting for all acknowledgments. At times, relaxing some of the specified constraints can be a good solution, e.g., if switches cannot be updated in a timely manner and it has acceptable consequences.
References


Zusammenfassung


Während einzelne Policies festgelegte Vorgaben beachten, kann das Netz für eine beliebig lange Zeit in einem Übergangszustand bleiben, der diese verletzt. Um dem vorzubeugen, sollte der Controller sicherstellen, dass das Netzwerk nur durch bestimmte zugelassene Übergangszustände geht (d.h. in Übereinstimmung mit gegebenen Einschränkungen). Das braucht eine sorgfältige Planung der Updates und bedeutet möglicherweise, dass viele Switchs auf einige warten müssen, bis sie die neue Policy umsetzen dürfen. Eine Lösung besteht darin, ein Optimierungsproblem daraus zu definieren und einen kürzestmögliche Updateplan zu berechnen. Das kann jedoch einen hohen Rechenaufwand bedingen, den dem Controller möglicherweise nicht zur Verfügung steht.


List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Reduction of the node set to SDN-enabled switches.</td>
</tr>
<tr>
<td>4</td>
<td>Reduction of the node set to a specific destination (here, $v_1$), with a single old policy and a single new one.</td>
</tr>
<tr>
<td>5</td>
<td>Reduction of the node set to a specific source (here, $v_1$), with a single old path and a single new one.</td>
</tr>
<tr>
<td>6</td>
<td>Reduction of the node set to the intersection of the old and the new paths.</td>
</tr>
<tr>
<td>7</td>
<td>Line representation.</td>
</tr>
<tr>
<td>8</td>
<td>Example of a policy update for which a loop can appear with destination differentiation but not with path differentiation. <em>solid</em> line: old policy, <em>dashed</em> line: new policy. Old path for source $v_3$: $[v_4, v_3, v_2, v_1]$, old path for source $v_1$: $[v_1, v_5]$, new path for source $v_1$: $[v_1, v_2, v_4, v_5]$.</td>
</tr>
<tr>
<td>10</td>
<td>Excerpt of a policy update in which following a dependency path $(v_8, v_{10}, v_{12})$ violates LF. Corresponding (initial) dependency forest in Figure 11.</td>
</tr>
<tr>
<td>11</td>
<td>Excerpt of dependency forest in which a dependency path $(v_8, v_{10}, v_{12})$ is not reliable (corresponds to the policy update of Figure 10), a node points at another node if it depends on it (according to one of the dependencies defined in the section 4.1).</td>
</tr>
<tr>
<td>12</td>
<td>Policy update displaying a dependency path that cannot be followed from bottom to top: updating in turn $v_8,v_{10}$ and $v_{12}$ or $v_5,v_6,v_7,v_9$ and $v_{12}$ violates LF, although they are parents of one another. Corresponding initial dependency forest in Figure 13.</td>
</tr>
<tr>
<td>13</td>
<td>Dependency forest corresponding to the policy update of Figure 12.</td>
</tr>
<tr>
<td>14</td>
<td>Policy updates differentiating on sources and destinations.</td>
</tr>
<tr>
<td>15</td>
<td>Policy updates differentiating on destinations only.</td>
</tr>
<tr>
<td>16</td>
<td>Policy update in which the maximal height of the dependency forest equals the minimal number of rounds (both equal to three).</td>
</tr>
<tr>
<td>17</td>
<td>Excerpt of a policy update where a node (here, $v_4$) is visited $n/4$ times.</td>
</tr>
<tr>
<td>18</td>
<td>Unit pattern used in the Lemma 5.</td>
</tr>
<tr>
<td>19</td>
<td>Policy Update of Lemma 5. <em>solid</em> line: old policy $p_1$, <em>dashed</em> line: new policy $p_2$, <em>full</em> circles: SDN switches (nodes), <em>square</em>: unit pattern of Figure 18.</td>
</tr>
<tr>
<td>21</td>
<td>Policy update showing a <em>x-structure</em>, at the beginning of round 2, after the controller updated $v_3$ in round 1. $v_2$ and $v_5$ are both independent but cannot be updated together because they form an x-structure.</td>
</tr>
<tr>
<td>22</td>
<td>Example of a policy update violating WE. <em>solid</em> line: old policy $p_1$, <em>dashed</em> line: new policy $p_2$, <em>full</em> circles: SDN switches (nodes), <em>triangle</em>: waypoint.</td>
</tr>
<tr>
<td>23</td>
<td>Policy update where waypoint enforcement is impossible.</td>
</tr>
<tr>
<td>24</td>
<td>Policy update requiring a pause.</td>
</tr>
</tbody>
</table>
Policy update impossible to realize without a packet either bypassing a waypoint or taking a loop.

Policy update where at least one node can be updated in each round.

Ambiguous Policy. If $v_1$ is updated in the first round, the new policy cannot be enforced without risking that packets take a loop or bypass the waypoint.

Policy update illustrating the potential benefit of making a pause. With a pause, policy update done in three rounds, without a pause, in $n - 1$ rounds.

Example of a tight dependency chain: $v_1$ and $v_2$ can be updated right away, any other node must wait for its predecessor.

Example of a loose dependency chain: $v_i$ updatable right away for $i = 1[3]$ or $i = 3[3], v_i$ becomes updatable for $i = 2[3]$ as soon as $v_{i-1}$ and $v_{i+1}$ are new.

Policy update with a corresponding dependency forest in Figure 32.

Policy update with a corresponding dependency forest in Figure 35.

Excerpt of a policy update illustrating the simple dependency. According to the simple dependency, $v_4$ depends on $v_3$ but not on $v_2$.

Policy update where a backward dependency-path cannot be used as a round sequence. According to the backward dependency based on LF, $v_4$ depends on $v_2$ and $v_3$, $v_2$ depends on $v_1$. However, updating $v_1$, $v_2$ and then $v_3$ violates LF.

Dependency Map illustrating relationships between dependencies, where an arrow indicates that the source is implied by the destination.

Policy illustrating the need for a dependency forest update after each node update acknowledgment.

Example of a dependency forest, a node points at another node if it depends on it (according to one of the dependencies defined in the section 4.1).

Performance Comparison of $\text{LF}_{BCKDEPHER}$, $\text{LF}_{DEPHER2}$, and $\text{LF}_{DEPHER3}$ regarding worst-case ratio.

Design of the project at the package level.

Design of the Comparator package.

Design of the Policy Generator package.

Design of the Heuristic package with the triple architecture.

Design of the Heuristic package with the quadruple architecture.

Design of the Optimal Algorithm package.

Initial DF in the worst case for the lower bound estimation.

Topology displaying the worst-case pattern for the lower bound estimation, maximal height of the dependency forest of two (constant), minimal number of rounds of $(n - 3)/2$ (linear in $n$).

Policy Update of Lemma 7, square unit pattern of Figure 51.

Unit pattern used in the Lemma 7.

Policy optimally updated using optEvBa.
Policy update requiring a pause. *solid* line: initial path $p_1 = \{v_1, v_2, v_3, v_5\}$, *dashed* line: second path $p_2 = \{v_1, v_3, v_4, v_5\}$, *fine dashed* line: third path $p_3 = \{v_1, v_4, v_3, v_2, v_5\}$. . . 69