Don't disturb my Flows: Algorithms for Consistent Network Updates in Software Defined Networks

Author(s):
Förster, Klaus-Tycho

Publication Date:
2016

Permanent Link:

https://doi.org/10.3929/ethz-a-010733901 →

DISS. ETH NO. 23703

Don't disturb my Flows: Algorithms for Consistent Network Updates in Software Defined Networks

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

Klaus-Tycho Förster

Diplom-Mathematiker, Braunschweig University of Technology, Germany
Diplom-Informatiker, Braunschweig University of Technology, Germany
Assessor des Lehramts, Studienseminar Goettingen, Germany

born on 19.05.1984

citizen of
Germany

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, ETH Zurich, examiner
Ratul Mahajan, Ph.D., Microsoft Research, co-examiner
Prof. Dr. Stefan Schmid, Aalborg University, co-examiner

2016

Abstract

In this dissertation, we consider the problem of finding efficient methods and complexity classifications for the consistent network update problem, with special focus on Software Defined Networks (SDNs). While the original and updated set of rules might both be consistent, disseminating the rule updates is an inherently asynchronous process, resulting in potentially inconsistent states. We study two fundamental consistency properties, first, loop freedom of forwarding rules, and second, consistent flow migration.

For the consistency property of loop freedom, we focus on hardness results. We start with longest prefix matching rules, and show that both maximizing the number of rules updated at once and finding the fastest update schedule is an NP-complete problem. Our results are then extended by proving that deciding if a sublinear schedule exists is also NP-complete already for two destinations. For single-destination routing, the number of rules updated at once can be approximated well in polynomial time, but finding an optimal update set is NP-complete too. We also consider related problems, specifically the hardness of fast blackhole-free updates under memory restrictions, the use of labeling schemes for local loop-free updates, and flipping the approach by generating efficient failure scenarios for reachability testing of a SDN controller.

For the consistent migration of flows, we show that for most scenarios involving splittable flows, the problem is in P, but NP-complete to decide for unsplittable flows. Specifically, we start by considering the standard flow migration model, where the bandwidth of every edge should be respected, no matter if each individual flow is in its old or its new state. We then extend this model in three ways: First, we develop non-mixing flow migration, where each packet respects waypointing and service chains. Second, we show the standard model to be susceptible to packet loss, and develop a lossless flow migration model. Third and last, we decouple the flow migration problem from a fixed new set of paths, and just consider the new set of demands: For the case of multi-commodity flows with one destination, this change allows to greatly improve the number of updates needed in the worst case. For all of these models, we present the first algorithms that can decide if a consistent flow migration for splittable flows exists, and also

provide an implicit schedule. On the hardness side, we show the decision problems for unsplittable flows to be NP-complete, whereas previous work only considered fastest update schedules.

Zusammenfassung

Thema der vorliegenden Dissertation ist die Problematik von konsistenten Netzwerkaktualisierungen in programmgesteuerten Netzwerken, mit besonderem Augenmerk auf effizienten Algorithmen und Komplexitätsklassifizierungen. Selbst wenn die aktuellen und gewünschten Netzwerkzustände in einem konsistenten Zustand sind, kann der Wechsel zwischen beiden Zuständen zu Inkonsistenzen führen, da die Dissemination der Änderungen ein inhärent asynchroner Prozess ist. Wir betrachten zwei fundamentale Konsistenzprobleme, zuerst die Kreisfreiheit von Netzwerkpaketweiterleitungen, dann die konsistente Migration von Netzwerkflüssen.

Für die Thematik der Kreisfreiheit legen wir unseren Fokus auf Komplexitätsresultate. Wir beginnen mit Präfixregeln und zeigen, dass sowohl die Maximierung der gleichzeitig aktualisierten Regeln als auch die Minimierung der Aktualisierungsplandauer ein NP-vollständiges Problem sind. Unsere Resultate lassen sich auch auf den Spezialfall mit zwei Senken erweitern, für den wir zeigen, dass das Problem der Aktualisierungsplandauer auch NP-vollständig für sublineare Werte ist. Sollte nur eine Senke im Netzwerk sein, dann kann die Maximierung der gleichzeitig aktualisierten Regeln gut approximiert werden, jedoch ist die optimale Lösung dieses Problems auch NP-vollständig. Wir betrachten in diesem Kontext auch verwandte Probleme, etwa Schwarze Löcher unter Speicherrestriktionen, die Verwendung von Knotenetikettierungen für rein lokale konsistente Aktualisierungen, und zuletzt eine in gewisser Sicht umgedrehte Problemstellung, bei der die Wiederherstellung der Erreichbarkeit im Netzwerk in effizient gewählten Fehlerfällen thematisiert wird.

Bezüglich der konsistenten Migration von Netzwerkflüssen können unsere Ergebnisse grob wie folgt zusammengefasst werden: Das jeweilige Entscheidungsproblem ist in P für teilbare Flüsse, aber NP-schwer für unteilbare Netzwerkflüsse. Wir betrachten zunächst das Standardmodell für konsistente Flussmigration, in dem die Kapazität jeder Kante nicht durch die aufliegenden Flüsse verletzt werden soll, egal ob die einzelnen Flüsse im alten oder neuen Zustand (nach der Aktualisierung) sind. Wir entwickeln dieses Modell dann auf drei Arten weiter: Zuerst betrachten wir nicht-vermischende Migration, bei der jedes Paket gewählte Wegpunkte und Dienstketten respektiert.

Danach zeigen wir, dass das Standardmodell für Paketverluste anfällig ist, und entwickeln ein verlustfreies Flussmigrationsmodell. Als drittes Modell entkoppeln wir das Flussmigrationsmodell von den gegebenen neuen Flusspfaden, und betrachten nur eine Migration zu den neuen Flussgrössen: Für Netzwerke mit nur einer Senke können wir dadurch die Dauer der konsistenten Flussmigration erheblich verbessern. Für alle diese Modelle ist unsere Arbeit die erste, die in polynomieller Zeit das Entscheidungsproblem für die konsistente Migration von teilbaren Netzwerkflüssen lösen kann. Für die entsprechenden Entscheidungsprobleme für unteilbare Flüsse können wir deren NP-Schwere beweisen, vorherige Arbeiten betrachteten nur die NP-Schwere der jeweiligen Optimierungsprobleme.

Contents

1 Introduction **1**
1.1 The Consistent Network Update Problem 2
1.2 Thesis Overview 3

I Loop Freedom **9**

2 On Loop-Free Network Updates **11**
2.1 A Model for Loop-Free Network Updates 12
2.1.1 Loop-Free Network Updates 12
2.1.2 Dynamic and Scheduling Loop-Free Network Updates 13
2.2 Related Work and Background 14
2.3 State of the Art for Loop-Free Updates & Results 16
2.4 Hardness of Multi-Destination Loop-Free Updates 18
2.4.1 Hardness of the Dynamic Case 19
2.4.2 Hardness of the Scheduling Case 25
2.5 Hardness of Scheduling Two-Destinations 28
2.6 Single-Destination Loop-Free Updates 31
2.6.1 Hardness of the Dynamic Case 32
2.6.2 Approximation of the Dynamic Case 35
2.7 Hardness of Blackhole Freedom 37

2.8 Loop-Free Updates and Local Checkability 44
2.8.1 A Local Migration Scheme 46
2.8.2 Beyond a Single Update 48
2.8.3 Further Applications in SDNs 48
2.9 Flipping the Approach: Reachability Testing 48
2.9.1 Problem Properties and Algorithms 49
2.9.2 Evaluation 52
2.9.3 Further Applications in SDNs 52

II Consistent Migration of Flows 55

3 On Consistency for Flow Updates 57
3.1 Models for Consistent Flow Migrations 58
3.2 Related Work and Background 60
3.3 State of the Art and Results 64

4 Consistent Flow Migration 67
4.1 Model 68
4.2 Hardness of Unsplittable Flow Migration 71
4.3 Consistent Migration for Splittable Flows 73
4.4 Insertion of Unsplittable Flows 84
4.5 Increasing Splittable Flows 85
4.6 Summary 90

5 Non-Mixing Flow Migration 91
5.1 Motivation 92
5.2 Model 92
5.3 Hardness of Unsplittable Flow Migration 94
5.4 An Algorithm for Two-splittable Flow Migration 96
5.5 Summary 101

6 Lossless Flow Migration 103
6.1 Motivation 104
6.2 Practical Evaluation 106
6.3 Model & Problem Setting 109
6.4 Checking Unsplittable Flow Network Updates 113

6.5 Checking for a $\forall$-Consistent Migration 117
6.6 Checking for a ℓ-Consistent Migration 122
6.7 Summary 124

7 Augmenting Flow Migration **125**
7.1 Motivation 126
7.2 Flow Augmentation Background 128
7.3 Model 128
7.4 Augmenting Flows for Multiple Commodities 131
7.5 Consistent Flow Migration with Augmentation 138
7.6 Strongly Consistent Flow Migration 141
7.7 Hardness of Flow Migration to new Demands 149
7.8 Augmenting Flows beyond a Single Destination 152
7.9 Summary 153

III Outlook **155**

8 Future Directions **157**

1

Introduction

The Internet as a whole is a wild place, full of autonomous participants. As such, it is naturally difficult to control centrally; instead, routing and congestion control is achieved through a selection of distributed protocols such as BGP and TCP. However, distributed protocols degrade performance, BGP cannot find the least congested path, and TCP will only crudely approximate the available bandwidth on the path selected by BGP. As a result, a loss of performance is to be expected and accepted. Many desirable properties such as drop freedom of packets, good utilization of links, or packet coherence are not as important as robustness. In contrast, individual networks that make up the Internet are controlled by single administrative entities. These include enterprise networks, ISP networks, data center networks, and wide area networks that connect the data centers of large organizations. The owners of these networks want to get the maximum out of their massive financial investment, which often runs into hundreds of millions of dollars per year (amortized). Towards this end, they have started replacing inefficient

distributed protocols.

The technological driver to this paradigm shift are so-called Software Defined Networks (SDNs): In an SDN, the data plane is separated from the control plane, allowing the decision of where and how much data is sent to be made independent of the system that forwards the traffic itself. A logically centralized controller monitors the current state of the network, then calculates a new set of forwarding rules, and distributes them to the routers and switches [10, 11, 37, 41].

Are centrally controlled SDNs the beginning of the end of distributed protocols? Not so fast! After all, the central SDN controller has to inform the switches about updates, and networks are inherently asynchronous, where nodes might even be temporarily not accessible to the controller [41].

1.1 The Consistent Network Update Problem

Network operators continuously strive for increased performance, especially to improve end-to-end latency or increase bandwidth utilization [41]. As such, the paths of network flows (as defined by the forwarding rules) are always in a state of flux, either by automated or manual *network updates*.

Network updates may temporarily destabilize a network, as mixing old and new forwarding rules may, e.g., form a loop. The chaos that can occur with the update is seen as an inevitable evil, as eventually the network will be in an improved state, outweighing the temporary losses by far.

Even though networks provide a best-effort service, e.g., congestion is highly problematic especially in contexts such as WAN and Data-Centers. As such, there has been (not only) recent interest in mechanisms to provide some sort of consistency for network updates [11, 58, 60, 63, 68], notably to prevent bandwidth violations [37, 42] and transient forwarding loops [25, 52, 58], but also for waypointing violations [51, 53], or ensuring that packets either take the old or the new path [68].

The central issue is that even if the network is under (logically) centralized control, the updates are performed in a distributed asynchronous environment: As the update happens in multiple parts of the network, a completely synchronized update is not possible, the individual routers could execute their updates in any ordering, cf. [36, 42, 49]. Even if perfect synchronization of updates were possible, a router could fail to update for some

reason, leaving the network in a disorganized state until either the failed update has been corrected, or, if this is not possible right away, all other routers roll back to their previous state. Time synchronization also ignores packets in flight, as it only deals with time at the routers' themselves, leading to transient congestion as we will show in this thesis. As such, we will study how to perform network updates in a consistent manner:

Problem 1 (The Consistent Network Update Problem, cf. [81])**.** *Let N be a network with an* old *and* new *state, each satisfying some consistency property. Find an ordered set of intermediate states (each being a network update), such that any asynchronous change between two consecutive states does not violate the consistency property, or output that no such set of states exist.*

In other words, we will investigate algorithms and complexity pertaining the consistent *migration* of networks, where a consistent migration solves the consistent network update problem by providing a sequence of appropriate network updates. The consistent network update problem can be extended by various constraints, especially regarding optimization problems (find the best individual update, find a shortest migration sequence) and model restrictions (memory limits, splitting possibilities).

We will focus on two fundamental network update problems in this thesis: First, the avoidance of transient forwarding loops for destination-based routing, and second, the minimization of congestion for multi-commodity network flows.

1.2 Thesis Overview

The presented thesis is logically organized in two parts, where the loop-free part focuses mostly on hardness results, whereas the flow migration part is more centered on an algorithmic point of view.

Loop Freedom First, in Chapter 2, we study loop-free network updates for destination based routing. After a short motivation, we formally define the model for loop-free network updates in Section 2.1: In particular, we give an introductory example and focus on the two cases of dynamic (greedy) and scheduled updates. We also provide an overview of the current state of the

art in tabular form in Section 2.3, providing the reader with a classification of our work in the area of loop-free updates.

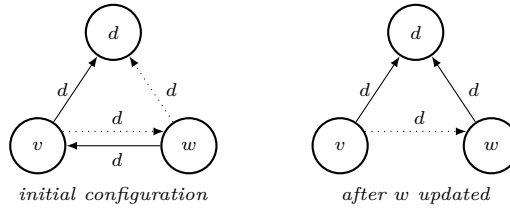


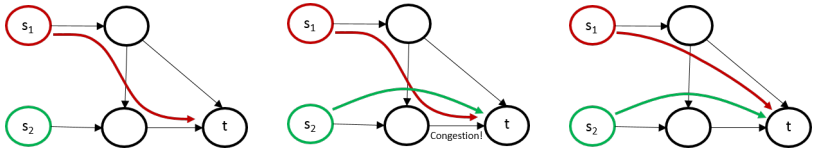
Figure 1.1: In this introductory example the initial network configuration is depicted on the left side. Both v and w have old forwarding rules for the destination d, drawn with solid lines. The desired new forwarding rules for d are drawn dotted. If v and w update in the same update, v could update before w due to asynchrony, leading to a loop between v and w. On the other hand, if just w updates to its new forwarding rule, no loop can appear, resulting in the network configuration depicted on the right side. Then, in another update, v can update loop free to its new forwarding rule by sending all packets for destination d to w.

We then begin in Section 2.4 with the study of multi-destination loop-free updates. In particular, we show both the dynamic and scheduling problem to be NP-hard. For the specific case of two-destination updates in Section 2.5, we prove the scheduling problem to be NP-hard for any sublinear schedule. For the dynamic single-destination case studied in Section 2.6, we show that it is closely related the feedback arc set problem by proving an NP-hardness reduction and corresponding approximation algorithm. These lines of work build upon the model of Mahajan and Wattenhofer [58], and were developed in parallel to the work by Schmid et al. [3, 16, 51–53]. Afterwards in Section 2.7, we show that the closely related problem of blackhole freedom under memory restrictions is NP-hard too, where packets arriving at a switch must always have a rule present.

We also investigate the use of labeling schemes for local loop-free updates in Section 2.8 and flip the loop-free approach by generating controlled failure scenarios for reachability testing of SDN controllers in Section 2.9.

Lastly, we give concluding remarks and discuss open research questions in the area of loop freedom for consistent network updates in Part III.

Consistent Migration of Flows In the second part of this thesis, we study the consistent migration of flows. We begin in Chapter 3 with motivating the problem, before giving a high-level overview and comparison of the four formal models considered in this thesis in Section 3.1. We then present related work and background on consistent flow migration in Section 3.2, and also give a tabular overview of the current state of the art in Section 3.3.



(a) Initial network with one flow from s_1 to t.

(b) If the lower flow is inserted first, there will be congestion.

(c) Desired flow placement with two flows of two commodities.

Figure 1.2: This figure depicts a small network to introduce the concept of consistency for flows. In the above examples, all flows have a size of one and all edges have a capacity of one as well. If the SDN controller desires to migrate the network from Subfigure 1.2a to Subfigure 1.2c in order to add a flow for the second commodity outgoing from s_2, then the commodity outgoing from s_1 has to be moved first. Else, due to asynchrony, s_2 could start a flow before the last edge is free, causing congestion (Subfigure 1.2b).

In the remainder of this part, we give the first complete study of consistent splittable flow migration for various models. Previous work could not decide the general case, and focused on restricted cases such as allowing every flow to be moved only once. Similarly, we prove NP-hardness for the decision problem for unplittable flows, whereas previous work only considered the fastest schedule. Our results builds upon the work by Mahajan and Wattenhofer et al. [37, 42, 58].

We start with the model proposed by [37] in Chapter 4, where the capacity of any edge should not be violated whether each flow is in the old or new state. We develop a polynomial augmentation technique that can decide if consistent migration for splitable flows is possible and also gives an

implicit schedule. Should no consistent migration be possible, we provide an LP-based approach to maximize the new demands in a consistent way. Beyond proving NP-hardness of the unsplittable variant, we also consider the case of unit size flows and give approximation thresholds.

Splitting flows can easily lead to waypointing and service chain violations, leading to the model of non-mixing flow migration in Chapter 5. We consider unsplittable flows along paths, where each packet should only be routed via the old or new path. As we prove the corresponding migration problem to be NP-hard to decide, we turn our attention to two-splittable flows: Again, we can show that the two-splittable flow migration problem is decidable in polynomial time, including implicit schedule generation. Compared to other splittable migration variants, the controller only needs to change the flow size distribution at the respective source, requiring no further communication with the remainder of the network beyond removing/adding the two old/new flow paths.

In the subsequent Chapter 6, we show how the notion of consistency from [37] can be extended to losslessness. Our practical evaluation proves that packets in flight can cause a single flow to congest itself, leading to packet loss and ongoing latency issues. To this end, we develop a new flow migration model, which can be considered as more restrictive as the one in Chapter 4, but allowing more updates than in Chapter 5. We distinguish the cases of fixed edge latencies and arbitrary edge latencies, with surprising results in the respective complexities. For one, fixed edge latencies make the flow migration problem NP-hard to decide, already for a single splittable flow. But on the other hand, if we take a worst-case approach with arbitrary edge latencies, we can once more decide the respective flow migration problem and implicit schedule generation in polynomial time.

Next, in Chapter 7, we tackle the flow migration problem regarding the number of required updates. The previously mentioned models can require an unbounded number of updates for consistency, as the final flow configuration is fixed. We take an orthogonal approach, and only take the new demands as an input. Extending the concept of flow migration to consistent migration, we show how to migrate in a polynomial number of updates if there is only one destination (or one source) in the network. We develop algorithms for both the consistency models of Chapter 4 and 6, the latter trading in additional updates for losslessness.

Lastly, concluding remarks and open research questions in the area of

consistent flow migration are presented in Part III.

Future Directions After treating our work on loop freedom in Part I and on consistent flow migration in Part II, we conclude our thesis in Part III. As noted earlier in this Thesis overview, we will summarize our work of both parts and the current state of the art, and pose open research questions with conjectures. We also propose to go beyond the consistent network update problem by integrating the desired network configuration into the problem setting.

Part I

Loop Freedom

2

On Loop-Free Network Updates

In loop-free updates, the SDN controller would like to switch from a set of old forwarding rules to a new set of forwarding rules, but without inducing (temporary or transient) forwarding loops in the process. Even if the forwarding loop just persists for miliseconds, a large amount of data will be lost in networks with a high throughput [25, 26, 52, 82, 83]. In this chapter, we will study loop-free updates from the viewpoint of algorithms and complexity.

We begin in Section 2.1 by formally defining the loop-free update problem, before discussing the state of the art and highlighting our results in the Sections 2.2 and 2.3. We then study the case of multiple destinations in Section 2.4, before investigating the special cases of two (Section 2.5) and single destinations (Section 2.6). Afterwards, we consider problems adjacent to the standard loop-free update problem: We study blackhole-free updates in Section 2.7 and local update schemes in Section 2.8. Lastly, we flip the loop-free update approach by generating failure test-cases for the centralized

controller in Section 2.9.

2.1 A Model for Loop-Free Network Updates

We model a network as a set of connected routers and switches (from now on, *nodes*). Packets must be forwarded to their destination without loops. Formally, a network is a directed multi-graph with a set of nodes V, a set of destinations $D \subseteq V$, and a set of destination-labeled edges s.t. all edges labeled with the same set of destinations will not contain a directed loop. The edges form a directed spanning tree with d being the root and all edges being oriented towards d.

Definition 1 (Single-Destination Network)**.** *Let $T_d = (V, E_d)$ be a directed graph with V being the set of nodes, $d \in D$ being the sole destination, and E_d being the set of edges each labeled with d. The edge from $u \in V$ to $v \in V$ for destination d is noted as $(u, v)_d$. The labeled directed graph T_d is a single-destination network, if T_d is a spanning tree with all directed edges being oriented towards d.*

Definition 2 (Multi-Destination Network)**.** *Let V be a set of nodes and $D \subseteq V$ be a set of destinations. For all $d \in D$, let $T_d = (V, E_d)$ be a single-destination network and let $E_D = \bigcup_{d \in D} E_d$. Then the labeled directed multi-graph $T_D = (V, E_D)$ is a multi-destination network.*

2.1.1 Loop-Free Network Updates

When a network needs to be updated, some (potentially all) nodes receive a new set of forwarding rules, leaving the network in a sort of limbo state. Due to the inherent asynchrony in networks, it is not possible to control the order in which the nodes contained in an update U change from old to new. At some point all nodes will be updated, but until then, the network might not be consistent, i.e., it might induce loops. Thus, we call an update loop-free if the union of the old and the updated forwarding rules is loop-free, cf. Figure 2.1. For subsequent updates, we define the current network forwarding state as old.

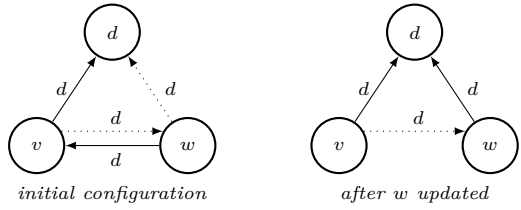


Figure 2.1: In this introductory example the initial network configuration is depicted on the left side. Both v and w have old forwarding rules for the destination d, drawn with solid lines. The desired new forwarding rules for d are drawn dotted. If v and w update in the same update, v could update before w due to asynchrony, leading to a loop between v and w. On the other hand, if just w updates to its new forwarding rule, no loop can appear, resulting in the network configuration depicted on the right side. Then, in another update, v can update loop free to its new forwarding rule by sending all packets for destination d to w.

Definition 3 (Loop-Free Single- and Multi-Destination Network Updates)**.** *Let $T_D^{old} = (V, E_D^{old})$ and $T_D^{new} = (V, E_D^{new})$ be multi-destination networks for the same set of nodes V and destinations D. Then the update $U_D = (V, E_D^{old}, E_D^{new})$ is called a multi-destination network update. If the labeled directed multi-graph $T_D = (V, E_D^{old} \cup E_D^{new})$ does not contain any loops of edges with the same label, then the update U_D is called consistent or loop-free. A single-destination network update U_d can be defined analogously.*

2.1.2 Dynamic and Scheduling Loop-Free Network Updates

Asynchronicity is not a technicality, as nodes in a production network can often react slowly (some switches might take up to 100× longer than average to update [42]), or may not be accessible for some time to the controller [41]. Thus, solutions in which the network can quickly start using as many of the new rules as possible, while maintaining the consistency properties, are preferable. This motivates the idea of dynamic loop-free network updates, where we update as many rules as we can at once:

Problem 2 (Dynamic Loop-Free Network Updates)**.** *Let $U_D = (V, E_D^{old}, E_D^{new})$ be a multi-destination network update. Find a set $E_D^{max} \subseteq E_D^{new}$, s.t.*

i) $U_D^{max} = (V, E_D^{old}, E_D^{max})$ *is a loop free multi-destination network update ii) for all loop free multi-destination network updates* $U_D^{other} = (V, E_D^{old}, E_D^{other})$ *with* $E_D^{other} \subseteq E_D^{new}$ *it holds that they do not contain more edges, i.e.,* $|E_D^{other}| \leq |E_D^{max}|$.

Even though asynchronicity is inherent in current hardware solutions (e.g., node failures [41] or highly deviating update times [42]), one could imagine these issues being tackled in future work. For example, the method of updating routing information could be decoupled from the remaining computational load of a node, resulting in roughly the same update time for all nodes in a network. Then one would want to find a shortest sequence of precomputed updates that migrate the network from the current old to the desired new routing rules. I.e., the controller will send out a first loop free multi-default update and wait until all affected edge changes are confirmed. This sending out of updates is iterated until all nodes switched their edges to the new desired routing rules.

Problem 3 (Scheduling Loop-Free Network Updates)**.** *Let* $U_D = (V, E_D^{old}, E_D^{new})$ *be a multi-destination network update. Find a sequence of* r *loop-free multi-destination network updates* $U_D^1 = (V, E_D^{old}, E_D^{new_1}), U_D^2, \ldots, U_D^r$ *with vertex sets* V *and corresponding pairwise disjoint new edge sets* $E_D^{new_1}, E_D^{new_2}, \ldots, E_D^{new_r}$ *s.t.* $E_D^{new_1} \cup E_D^{new_2} \cup \cdots \cup E_D^{new_r} = E_D^{new}$ *s.t.* $r \in \mathbb{N}$ *is minimal.*

We note that we might drop the D from the notation (e.g., U opposed to U_D) if the set of destinations is unambiguous in the context.

2.2 Related Work and Background

Loop-free routing is a classic problem in networking[1], but the study of transient loops during re-configurations is still relatively new [25, 58].

Ito et al. [40] investigated avoiding loops in shortest-path routing by increasing link costs to bypass single links. Shortly after, the study of loop free network updates was initiated by François et al. in their investigation of the convergence of link-state routing protocols [25, 26]. They studied the scheduling of single-destination updates, showing that one can always update in a number of rounds equivalent to the depth of a routing tree induced by the new forwarding rules.

[1] We refer to Section VIII of [25] for a short survey of previous work.

Their update model was later enhanced to a dependency forest setting in [58] by Mahajan and Wattenhofer, showing that any dynamic update scheduling strategy will lead to all nodes being updated when considering single-destination routing. They show that a greedy update strategy outperforms the tree-depth based scheduling of updates in a practical setting, cf. [23]. We extend their line of work in this chapter regarding multi-destination (Section 2.4) and Blackhole (Section 2.7) update NP-hardness, and an approximation of the dynamic case (Subsection 2.6.2).

Ludwig et al. [52] further[2] studied the scheduling of loop-free updates, and showed that this problem is NP-hard for 3 rounds, but that a schedule for 2 rounds can be found in polynomial time. Moreover, they show that a simple greedy update strategy is only $\Omega(n)$-competitive for scheduling updates, and also introduce the notion of relaxed loop-freedom: In this setting, there should be no loop on the source-destination path, always allowing for a $O(\log n)$-scheduling with their *Peacock* algorithm. Building upon their results, we explore the NP-hardness of scheduling sublinear two-destination updates in Section 2.5.

Amiri et al. [3] explore the problem of greedily updating single-destination forwarding rules. In parallel to our work in Subsection 2.6.1, they showed the problem to be NP-hard, but also proved the NP-hardness of the relaxed variant. Furthermore, they give polynomial optimal and approximation algorithms for special graph classes.

Dudycz et al. studied a setting similar to Section 2.5 of this work, but allow to break up forwarding rules [16]: They prove that minimizing the number of interactions (*touches*) with the switches for loop freedom is NP-hard in this case, and also study how to efficiently compose schedules for single destinations.

Vanbever et al. [82] consider a modified model for scheduling loop free updates for $O(n)$ destinations, in their variant all forwarding rules of a node have to be updated in the same round, making their decision variant of finding a per-router ordering NP-hard as well.

The packet stamping solution of Reitblatt et al. [68] can ensure loop freedom by adding version numbers to packets, but because it ensures the much stronger property of packet coherence, it is slow and has high memory

[2] We note that the setting in [3, 16, 52, 53] is slightly different from ours in general, as they assume the old and the new routes to be simple paths between one source and one destination.

overhead. The whole network needs to be updated first, before being able to use the system—a long delay in updating single node induces a long delay for the complete network. Further, despite the extensions of Katta et al. [45], which trade-off switch memory for speed, packet stamping has high memory overhead because it simultaneously stores both old and new rules. Switch memory is a scarce commodity, with even future generations of switches reaching their memory limit easily when optimizing the network [37].

Loop freedom can also be in conflict with waypoint enforcement, as shown in [53]. In fact, even deciding if a loop-free waypoint-enforcing schedule exists is NP-hard [51], leading Vissicchio and Cittadini [83] to apply the idea of packet stamping (or 2-phase commit) from the seminal work by Reitblatt et al. [68] in the loop-free setting.

Furthermore, Mizrahi et al. [60] study the problem with a time-based approach, aiming to synchronize concurrent network updates to reduce the inherent asynchrony; their work is also applicable to the migration of flows. An overview of their *TimedSDN* project is available at `http://tx.technion.ac.il/~dew/TimedSDN.html`.

Lastly, recent work in model checking [59, 87] has also considered the problem of loop-free network updates. In a sense, their work is orthogonal to ours, as they explore "*the space of possible solutions*" [59], without a specific focus on algorithmic worst-case guarantees.

2.3 State of the Art for Loop-Free Updates & Results

For ease of readability, we summarize the previously discussed related work and our results for loop-free updates in two tables, one for the dynamic case (Table 2.1), and one for the scheduling case (Table 2.2). Open problems are depicted by a ?-entry. Beyond the results listed in the Tables 2.1 and 2.2 on the next page, we also show the following in this chapter:

Section 2.7: The fastest scheduling of blackhole-free updates under memory restrictions, while maintaining routing without loops, is NP-hard.

Section 2.8: How to update in a loop-free way without acknowledging the individual nodes' updates to the controller, but instead relying on local verification.

Section 2.9: We flip the approach of updating in a loop-free manner by

Dynamic	Per-router ordering	Multi-Destination	Single-Destination	Relaxed
Optimal hardness	Polynomial [82]	NP-complete [2.4.1], [3]	NP-complete [2.6.1], [3]	NP-complete [3]
Optimal polynomial algorithms	Check all nodes individually [82]	If $P = NP$	For trees with 2 leaves & bounded tree-width [3]	For trees with 2 leaves & bounded tree-width [3]
Polynomial approx. algorithms	(see above)	?	2/3-approx with 3 leaves, 7/12-approx with 4 leaves, MAS [3] & FAS [2.6.2] in general	MAS [3]

Table 2.1: Overview of the state of the art in dynamic (greedy) loop-free updates. For the results proven in this thesis, please refer to the Subsections 2.4.1, 2.6.1, and 2.6.2. Furthermore, the abbreviations MAS and FAS stand for equivalent approximation results as for maximum acyclic subgraph and feedback arc set.

Scheduling	Per-router ordering	Multi-Destination	Double-Destination	Single-Destination	Relaxed	S.-D. touches
Decision hardness	NP-complete [82]	? (in NP)	? (in NP)	Always works [25, 58]	Always works [52]	Always works (see s.-d.)
# updates hardness	n-schedule or not solvable (model)	3-schedule is NP-complete [2.4.2]	sublinear schedule is NP-complete [2.5]	3-schedule is NP-complete [52]	?	Minimizing NP-hard [16]
Optimal polynomial algorithms	If $P = NP$	?	?	2-schedule [52]	?	Combining $O(1)$ optimal policies [16]
General polynomial algorithms, in # of updates	(see above)	If permitted, split into single-destination policies [58]	If permitted, split into single-destination policies [58]	$\Theta(n)$ [25, 52, 58]	$O(\log n)$ [52]	Different opt. goal ($O(n)$ updates, see s.-d.)

Table 2.2: Overview of the state of the art in scheduling loop-free updates. For the results proven in this thesis, please refer to Subsections 2.4.2 and 2.5.

focusing on how to test the centralized controller to maintain reachability without loops in the presence of failures.

2.4 Hardness of Multi-Destination Loop-Free Updates

Interval routing and longest-prefix matching are common routing techniques for large networks. In interval routing (introduced in [72], cf. [30]), destinations $\{d_1, \ldots, d_{|D|}\}$ are ordered cyclically, and forwarding rules for a node are defined as disjoint intervals over the destinations, cf. [19, 24, 80].

In contrast, longest-prefix routing defines forwarding rules via prefixes of the destination IDs, which may overlap: If two rules are in conflict, the one with the longer matching prefix is chosen, cf. [13, 76].

Both techniques have great practical advantages, since multi-destination routing does not scale well: Even when considering just IPv4 (and not IPv6), no router on the market could store an individual rule for every IP–address. Furthermore, this fine–grained information is not available, since the complete knowledge over a network is usually restrained to one's own Autonomous System.

A subset of both techniques is multi-destination routing with the possibility of default routes. Nodes can either have individual forwarding rules for each destination or a default rule, cf. [27], i.e., all packets go to a specific other node (except for those that reached their destination at the current node).

Definition 4 (Default Routes)**.** *Let $T_D = (V, E_D)$ be a multi-destination network and let $u, v \in V$. If all outgoing edges from u point at v in E_D, then those edges E_u may be merged into a* default edge*, labeled with all labels from D (but packets for a destination u do not get forwarded from u). We denote such an edge with $(u, v)_\forall$. I.e., we remove E_u from E_D and add $\{(u, v)_\forall\}$. Let the resulting set of edges of this iterated process be $E_{D,\forall}$. We call $T_{D,\forall} = (V, E_{D,\forall})$ a multi-destination network with default routes or* multi-default network.

Loop-free network updates can be defined analogously in the presence of just single-destination and default routes:

Definition 5 (Loop-Free Network Updates with Default Routes)**.** *Let $T^{old}_{D,\forall} = (V, E^{old}_{D,\forall})$ and $T^{new}_{D,\forall} = (V, E^{new}_{D,\forall})$ be multi-default networks for the same set*

of nodes V and destinations D. Then $U_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new}_{D,\forall})$ is called a multi-default network update. *If the labeled directed multigraph $T_{D,\forall} = (V, E^{old}_{D,\forall} \cup E^{new}_{D,\forall})$ does not contain any loops of edges with the same label, then the update $U_{D,\forall}$ is called* consistent *or* loop-free.

In this section, we show that both maximizing dynamic and fast scheduling of loop-free network updates are NP-hard problem – and therefore also NP-hard for both supersets of interval routing and longest-prefix matching, and especially the multi-destination case in general.

2.4.1 Hardness of the Dynamic Case

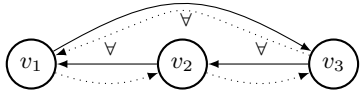


Figure 2.2: Illustrating circular dependencies with default routes, from [58]. Note that both in the old and new rules, no packet will loop: E.g., in the old rules, a packet sent out from v_1 will be forwarded to v_3, and possibly to v_2, but never to v_1 again - as all possible destinations were already reached on the path.

Let us start with an example of just three nodes for the dynamic case in Figure 2.2, as observed in [58]. We want to update the three old default edges (drawn solid) to the three new default edges (drawn dotted). However, due to circular dependencies, not even a single edge can be updated without causing a loop. This problem can be handled by relaxing the constraints of default routing: One can prevent loops by breaking a single (default) rule into one helper rule for each of the two other destinations, introducing these rules during the update process and then removing them later.

In general, this is not desirable, as memory constraints on routers can easily prevent introducing these additional helper rules, cf. [37]. Nonetheless, one can directly check if a non-empty update exists: Check each new edge individually, since adding more edges cannot remove existing cycles. However, even if a multi-default network can be updated with some edges, it is a hard optimization problem.

We thus define the problem of dynamically updating multi-default networks as finding the maximum number of edges that can be included in an update at once:

Problem 4 (Dynamic Loop-Free Default Updates)**.** *Let* $U_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new}_{D,\forall})$ *be a multi-default network update. Find a set* $E^{max}_{D,\forall} \subseteq E^{new}_{D,\forall}$*, s.t. i)* $U^{max}_{D,\forall} = (V, E^{old}_{D,\forall}, E^{max}_{D,\forall})$ *is a loop-free multi-default network update ii) for all loop-free multi-default network updates* $U^{other}_{D,\forall} = (V, E^{old}_{D,\forall}, E^{other}_{D,\forall})$ *with* $E^{other}_{D,\forall} \subseteq E^{new}_{D,\forall}$ *it holds that they do not contain more edges, i.e.,* $|E^{other}_{D,\forall}| \leq |E^{max}_{D,\forall}|$.

Theorem 1. *Problem 4 is NP-complete.*

Our proof is a reduction from the classic NP-complete problem 3-SAT, in the variant with exactly three pairwise different variables per clause [29]. We show that for every instance I of 3-SAT, we can construct in polynomial time an instance I' of the corresponding decision problem of Problem 4 s.t. I is satisfiable if and only if I' is a *yes*-instance. The general idea of the proof can be described via the gadgets listed in Figure 2.3 and 2.5:

1. Consider the routes for destination Y in the triangle-gadget from Figure 2.3. If node $\overline{X}_i$ updates, then node X_i cannot update without inducing a loop for Y, and vice versa. Choosing one of the two update rules corresponds to a variable assignment for a variable x_i in the instance I of 3-SAT: x_i is either *true* or *false*, but not both.

2. Let C be a clause in the instance I of 3-SAT. If there is a variable assignment S that satisfies I, then updating the triangle-gadgets for the variables according to S does not induce a loop for any destination C in the cycle-gadget for the corresponding clause in Figure 2.5. If no such variable assignment S exists, then at least one triangle-gadget cannot be updated at all without causing a loop for a destination representing a clause.

3. Let k be the number of variables in I. If k rules from the nodes $X_i, \overline{X}_i$ in the triangle-gadgets can be updated loop free, then there exists a variable assignment S that satisfies the instance I of 3-SAT. If less than k rules can be updated from the nodes $X_i, \overline{X}_i$ in the triangle-gadgets, then I cannot be satisfied.

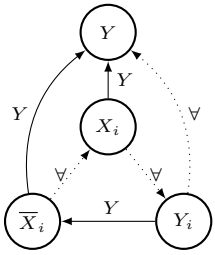


Figure 2.3: Triangle-gadget for a variable x_i. New edges are drawn dotted, old solid.

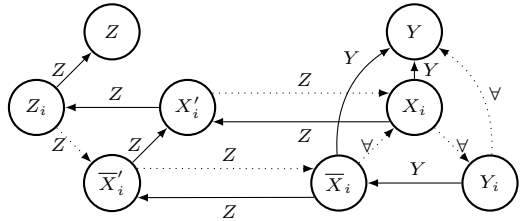


Figure 2.4: Extension of the triangle-gadget for a variable x_i from Figure 2.3. New edges are drawn dotted, old solid. Edges not shown point at their destination. The four possible cycles for destination Z are *i)* X_i, X'_i, *ii)* $\overline{X}_i, \overline{X}'_i$, *iii)* $\overline{X}'_i, X'_i, Z'_i$, *iv)* $\overline{X}'_i, \overline{X}_i, X_i, X'_i, Z_i$. No other new cycles are introduced.

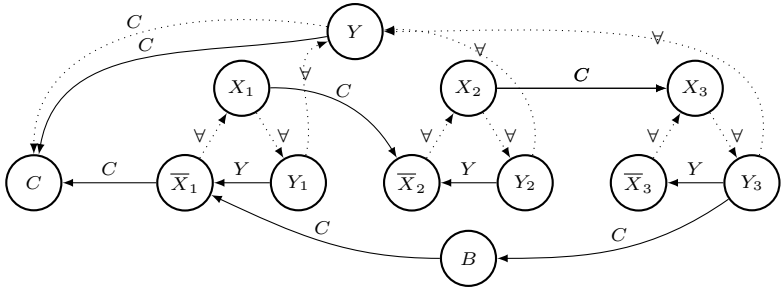


Figure 2.5: Cycle-gadget for the clause $C = (x_1 \vee x_2 \vee \overline{x}_3)$. All edges not shown point directly at their destination. Only if all three nodes $\overline{X}_1, \overline{X}_2, X_3$ update their forwarding rule for C, then there is a loop for the label C (via $B - \overline{X}_1 - X_1 - \overline{X}_2 - X_2 - X_3 - Y_3 - B$). E.g., $C = (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$ could only induce a cycle via $B - X_1 - Y_1 - X_2 - Y_2 - \overline{X}_3 - X_3 - B$.

# in sequence	conflicting clauses	variable false	variable true
1	Y, Z, Y_i	$Y, Z, Y_i, \overline{X}_i$	Y, Z, Y_i, X_i
2	$X_i, \overline{X}_i$	$\overline{X}'_i, X_i$	$X'_i, \overline{X}_i$
3	$X'_i, \overline{X}'_i$	X'_i, Z_i	$\overline{X}'_i, Z_i$
4	Z_i	$\emptyset$	$\emptyset$

Figure 2.6: Table depicting the fastest possible migration scenarios for the nodes in Figure 2.4. *i)* X_i cannot update before X'_i, *ii)* $\overline{X}_i$ not before $\overline{X}'_i$, *iii)* Z'_i not before $\overline{X}'_i$ or X'_i, and *iv)* X_i or X'_i must update before $\overline{X}'_i$ and $\overline{X}_i$ and Z_i can all three be updated. Note that Y, Z, Y_i can always update right away. However, if there are conflicting clauses (i.e., the corresponding instance is not satisfiable), then neither $\overline{X}_i$ nor X_i can update right away, but must wait for the next update to be sent out – after the conflicts with the clauses have been cleared, thus requiring a sequence of length four. Else, one could update with a sequence of length three, as shown in the two rightmost columns.

Proof. We will use similar notation for the elements of the instance I from 3-SAT and for elements from the instance I' for ease of readability.

The problem is in NP: Observe that the problem is indeed in NP: Given an update U of new forwarding rules, checking the union of the old and to be updated forwarding rules to be loop free can be performed in polynomial time, e.g., by Tarjan's algorithm [77] for each destination.

Construction of the new Instance I'
Let I be an instance of 3-SAT, with variables $x_1, x_2, \ldots x_k$, the literals X_i and $\overline{X}_i$ for each variable x_i with $1 \leq i \leq k$, and the clauses $C_1, C_2, \ldots, C_j$ with the corresponding literals $l_{h_1}, l_{h_2}, l_{h_3}$ for $1 \leq h \leq j$. We construct an instance I' of Problem 4 as follows: We create the three nodes X_i, $\overline{X}_i$, and Y_i for every variable x_i, two nodes C_h and B_h for every clause C_h, and one node Y. I.e., $V = \{X_1, X_2, \ldots, X_k, \overline{X}_1, \overline{X}_2, \ldots, \overline{X}_k, Y_1, Y_2, \ldots, Y_k, C_1, C_2, \ldots, C_j, B_1, B_2, \ldots, B_j, Y\}$.

The set of destinations is chosen as $D = \{C_1, C_2, \ldots, C_j, Y\}$. For each variable x_i, we define a gadget with the nodes X_i, $\overline{X}_i$, and Y_i, s.t. only either X_i or $\overline{X}_i$ may update its forwarding rule without creating a loop, see Figure 2.3. For each clause C_h we create a gadget as well, see Figure 2.5: If all three nodes (that correspond to the literals $\neg l_{h_1}, \neg l_{h_2}, \neg l_{h_3}$ not in the

clause C_h) update, then there is a loop for packets with the destination C_h.

More formally, the old set of edges $E^{old}_{D,\forall}$ is defined as follows:

- $\forall$ nodes C_h with $1 \leq h \leq j$ there is an edge $(Y, C_h)_Y$ from Y to C_h labeled with C_h.
- $\forall$ clauses C_h, $1 \leq h \leq j$, there are edges labeled with C_h as follows: $(B_h, \neg l_{h_1})_{C_h}$, $(\Upsilon_{h_1}, \neg l_{h_2})_{C_h}$, $(\Upsilon_{h_2}, \neg l_{h_3})_{C_h}$, $(\Upsilon_{h_3}, B_h)_{C_h}$, with
  - $\Upsilon_{h_r} = X_{h_r}$, if the corresponding literal l_{h_r} in the clause is a positive literal, for $1 \leq r \leq 3$
  - $\Upsilon_{h_r} = Y_{h_r}$, if the corresponding literal l_{h_r} in the clause is a negative literal, for $1 \leq r \leq 3$
- $\forall$ nodes Y_i with $1 \leq i \leq k$ there is an edge $(Y_i, \overline{X}_i)_Y$ from Y_i to $\overline{X}_i$ labeled with Y.
- If a destination d is still undefined for a node v, there is an edge $(v, d)_d$.

The desired set of edges $E^{new}_{D,\forall}$ is:

- $\forall i$ with $1 \leq i \leq k$ there are edges $(\overline{X}_i, X_i)_{all}$, $(X_i, Y_i)_{all}$, $(Y_i, Y)_{all}$.
- As before, if a destination d is still undefined for a node v, there is an edge $(v, d)_d$.

We now pose the following decision problem for the constructed instance $I' = (V, D, E^{old}_{D,\forall}, E^{new}_{D,\forall})$: *Is the maximum number of edges that can be consistently updated at least* $\left(|E^{new}_{D,\forall}| - k\right)$*?*

What loops can be induced by the update?
We first note that individually, both sets of edges $E^{old}_{D,\forall}$ and $E^{new}_{D,\forall}$ are loop free for every destination $d \in D$. What cycles can appear for any destination $d \in D$ when applying the full update, i.e., when looking at $E^{old}_{D,\forall} \cup E^{new}_{D,\forall}$? Except for the edges $(\overline{X}_i, X_i)_{all}$, $(X_i, Y_i)_{all}$, $(Y_i, Y)_{all}$ for $1 \leq i \leq k$, all updated edges labeled with a destination d point directly at d, meaning that they cannot induce a loop for the destination d. The edges $(Y_i, Y)_{all}$ cannot generate loops either: All outgoing edges from Y, which are labeled with a destination d, point directly at d. This leaves only the edges of the type

$(\overline{X}_i, X_i)_{all}$, $(X_i, Y_i)_{all}$ to induce loops when applying an update. Note that the total number of these edges is $2k$.

If for any i with $1 \leq i \leq k$ both $(\overline{X}_i, X_i)_{all}$ and $(X_i, Y_i)_{all}$ are in the update, then there is a loop for the destination Y (cf. Figure 2.3), but this is the only way to induce loops for the destination Y: All other edges labeled with Y in the update point at Y.

Is there a loop for a destination C_h? We do not have to consider edges pointing directly at the node C_h (they cannot induce a loop), but at the same time, we only need to consider edges labeled with C_h: The only way to create a loop for the destination C_h is to include the three edges labeled with $\forall$ outgoing from the nodes $\neg l_{h_1}$, $\neg l_{h_2}$, and $\neg l_{h_3}$, cf. Figure 2.5.

Thus, the only way to have a loop-free update of size at least $\left(|E^{new}_{D,\forall}| - k\right)$ is $\forall i$ with $1 \leq i \leq k$ to include either the outgoing edge labeled with $\forall$ from the node X_i or from the node $\overline{X}_i$.

Solving any instance of 3-SAT by maximizing updates
We now show that any instance I of 3-SAT can be reduced to such an instance I' s.t. I is satisfiable if and only if there exists an update of size at least $\left(|E^{new}_{D,\forall}| - k\right)$ for I'. We note that I' can be constructed from I in polynomial time.

If I is satisfiable, then there is a variable assignment S that satisfies I. S satisfies I in such a way, that there is no clause C_h s.t. all literals l_{h_1}, l_{h_2}, l_{h_3} are false. Therefore we can loop-free update the edges labeled $\forall$ in I' outgoing from the nodes X_i if x_i is true in S and those from $\overline{X}_i$ if x_i is false in S. This gives us already a set of k edges that can be included in the update and a set of k edges not to be included in the update. Since all other edges from the update can be included, this gives a solution for I' of size $\left(|E^{new}_{D,\forall}| - k\right)$.

If I is not satisfiable, then there is a no variable assignment S that satisfies I. For every variable assignment S' for I, there is at least one clause C_h s.t. all literals l_{h_1}, l_{h_2}, l_{h_3} are false. This means that we can only choose at most $k-1$ edges from the edges labeled with *all*, which are outgoing from the nodes X_i and $\overline{X}_i$, for the update without inducing a loop – at least $k+1$ of these edges cannot be included in a loop-free update. This means that the maximum number of edges to be included in the update for instance I' is at most $\left(|E^{new}_{D,\forall}| - k - 1\right)$. □

Corollary 1. *The problem of dynamic loop-free network updates for interval routing, longest-prefix matching, or multi-destination routing in general is NP-complete.*

2.4.2 Hardness of the Scheduling Case

Similar as in the last subsection, we cover the special case of single-destination and default routes for proving the hardness of the fastest scheduling, which in turn shows the hardness of the encompassing update problems as well.

Problem 5 (Scheduling Loop-Free Network Updates with Default Routes)**.** *Let $U_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new}_{D,\forall})$ be a multi-default network update. Find a sequence of r loop-free multi-default network updates $U^1_{D,\forall} = (V, E^{old}_{D,\forall}, E^{new_1}_{D,\forall})$, $U^2_{D,\forall}, \ldots, U^r_{D,\forall}$ with vertex sets V and corresponding pairwise disjoint new edge sets $E^{new_1}_{D,\forall}, E^{new_2}_{D,\forall}, \ldots, E^{new_r}_{D,\forall}$ s.t. $E^{new_1}_{D,\forall} \cup E^{new_2}_{D,\forall} \cup \cdots \cup E^{new_r}_{D,\forall} = E^{new}_{D,\forall}$ s.t. $r \in \mathbb{N}$ is minimal.*

Theorem 2. *Problem 5 is NP-complete.*

Note that the construction for the proof of Theorem 4 is not enough to show that Problem 5 is NP-hard: While it is NP-hard to decide if k rules from the nodes $X_i, \overline{X}_i$ in the triangle-gadgets can be updated, the whole network in the proof can always be updated in a sequence of just two updates. In the first step, one would update all nodes (except for the nodes $X_i, \overline{X}_i$ in the triangle-gadgets). Then, in the second step, all the nodes $X_i, \overline{X}_i$ in the triangle-gadgets can be updated, since the possibility of loops in the gadgets created from variables and clauses have vanished after the first update. However, we can extend our construction s.t. for a solution of sequence-length three, all k triangle-gadgets need to update either $X_i, \overline{X}_i$ in the first element of the sequence of updates. Else, a sequence of length four would be needed. The construction is described in the Figures 2.4 and 2.6.

Proof. **The problem is in NP:** Observe that the problem is indeed in NP: As noted before, given an update U of new forwarding rules, checking the union of the old and to be updated forwarding rules to be loop free can be performed in polynomial time, e.g., by Tarjan's algorithm [77] for each destination. As the maximal number of non-empty updates is polynomial, each update schedule can be checked in polynomial time for correctness as

well.

The problem is NP-hard Note that it follows from the proof of Theorem 1, that in the construction of new instance I' for each $1 \leq i \leq k$, not both $\overline{X}_i$ and X_i can migrate in the update, since this would induce a cycle with the node Y_i. Furthermore, only if the instance I of 3-SAT is satisfiable, then exactly one node of $\overline{X}_i$ or X_i can migrate in the corresponding triangle-gadget. Else, i.e., I is not satisfiable, neither $\overline{X}_i$ nor X_i can migrate in the current situation of instance I'. Thus, it is NP-hard to decide if it holds for every triangle-gadget that exactly one of the nodes of $\overline{X}_i, X_i$ can migrate in a first update.

We now extend the construction of instance I' as shown in Figure 2.4 for every triangle-gadget in the instance I':

- For $1 \leq i \leq k$ we add the nodes $\overline{X}'_i, X'_i, Z_i$, i.e., three nodes for every one of the triangle-gadgets.
- We add the node Z.
- For the set of old rules and for $1 \leq i \leq k$, we add the edges $(\overline{X}_i, \overline{X}'_i)_Z$, $(X_i, X'_i)_Z$, $(\overline{X}'_i, X'_i)_Z$, $(X'_i, Z_i)_Z$.
- For the set of new rules and for $1 \leq i \leq k$, we add the edges $(\overline{X}'_i, \overline{X}_i)_Z$, $(X'_i, X_i)_Z$, $(Z_i, \overline{X}'_i)$.
- As before, if a destination d is still undefined for a node v for the set of old or new rules, we add an edge $(v, d)_d$.

Edges pointing directly at their destination can never introduce loops, thus we only need to look at the remaining newly introduced edges – which are all inside the extended triangle-gadgets, and do not create new routes outside single extended triangle-gadgets (again, except for directly pointing at destinations). Note that all these remaining newly introduced edges are for the destination Z, hence only loops for this destination can be introduced by this extension of the construction of the instance I'. When considering the edge-set created by joining the set of old rules and new rules, only four new cycles for the destination Z are created for each extended triangle-gadget: *i)* X_i, X'_i, *ii)* $\overline{X}_i, \overline{X}'_i$, *iii)* $\overline{X}'_i, X'_i, Z'_i$, *iv)* $\overline{X}'_i, \overline{X}_i, X_i, X'_i, Z_i$. All these

four cycles induce dependencies for updating the rule including destination Z:

- *i*): X_i cannot update before X_i'.
- *ii*): $\overline{X}_i$ cannot update before $\overline{X}_i'$.
- *iii*): Z_i' cannot update before $\overline{X}_i'$ or X_i'.
- *iv*): X_i or X_i' must update before $\overline{X}_i'$ and $\overline{X}_i$ and Z_i can all three be updated.

Note that these four cycles cause the length of the sequence to update all rules in the triangle-gadget to be at least three: Due to *iii*), Z_i' must wait for $\overline{X}_i'$ or X_i', but due to *i*) and *ii*), these two nodes must wait for $\overline{X}_i$ or X_i respectively. Furthermore, all rules except for those in the extended triangle-gadget can be updated in the first element of the sequence of updates, see the proof of Theorem 1.

Hence, if the corresponding 3-SAT instance I is not satisfiable, then there exists at least one extended triangle-gadget where neither $\overline{X}_i$ nor X_i can update – causing the sequence length to be four, see the second column in the table in Figure 2.6 for a valid ordering for the nodes in the extended triangle-gadget.

Should the corresponding 3-SAT instance I be satisfiable, then either $\overline{X}_i$ or X_i can be updated in every extended triangle-gadget, allowing for a sequence of length three to update all rules in the network, see the two rightmost columns in the table in Figure 2.6 for a valid ordering for the nodes in the extended triangle-gadget.

Thus, it is NP-hard to decide if the whole network can migrate in a sequence of either length three or four. □

Corollary 2. *It is NP-complete to approximate the length of the sequence of updates needed for Problem 5 with an approximation ratio strictly better than* 4/3.

Corollary 3. *The problem of scheduling loop-free network updates for interval routing, longest-prefix matching, or multi-destination routing in general is NP-complete.*

2.5 Hardness of Scheduling Two-Destinations

In this section we consider the problem of scheduling loop free updates for two destinations d, d': Given a network N and old and new forwarding rules, find a sequence of r loop free updates $U_1, U_2, \ldots, U_r$ s.t. after U_r (i.e., r updates), the current network forwarding state is identical to f_{new}.

For the case of two destinations d, d', each node can either have two separate old forwarding rules for d, d' or a combined one, analogously for the new rules. We note that the separate forwarding rules of a node might still point towards the same node, and that a node can have separate old and combined new forwarding rules, or vice versa. If a node has a combined new forwarding rule, then updating to this rule replaces the old rules. For the case of separate forwarding rules however, they can be updated in different updates Recall that an update for two destinations loop-free, if the union of the old and the updated forwarding rules is loop free for each destination.

We first restate a result from Ludwig et al. [52], adjusted to our notation, which tells us that deciding if a 3–update sequence exists is NP-complete. In the remainder of this section, we will extend this proof to a sublinear number of updates for 2 destinations, i.e., less than $n^{1-\varepsilon}$.

Theorem 3 ([52]). *The problem of scheduling loop free updates for one destination d is NP-complete for a 3–update sequence.*

The proof of Theorem 3 is quite involved, yet elegant. As we will use it as a black box in our own construction, we omit its proof details here.

In fact, adding a second destination allows us to make use of the following idea: An update of a node v in the construction of Theorem 3 can be delayed by adding an additional destination, cf. Figure 2.7.

This concept can be performed on all forwarding rules, allowing every update to be delayed by one additional update by adding $\Omega(n)$ additional destinations. Iterating this idea beyond one update yields the NP-completeness of the problem for a logarithmic number of updates with a linear number of destinations.

Nonetheless, the deciding factor in this hardness augmentation comes from adding a second destination, as we will show in the following:

Theorem 4. *Let $0 < \epsilon < 1$. For any $3 \leq r \leq n^{1-\epsilon}$, there is a $r^* \geq r$ s.t. the problem of scheduling loop free updates for two destinations d, d' is NP-complete for a r^*–update sequence.*

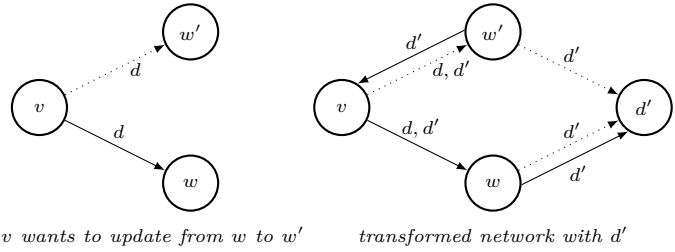


Figure 2.7: Old forwarding rules are solid, new ones dashed, the node for destination d is omitted. To delay the update of v from w to w', we transform the network on the left side to the network on the right side. Now, node v cannot update its combined new rule for the destinations d, d' to the node w' until w' switches its rule for d' from v to d'. Note that in general, w' could have arbitrarily many incoming new forwarding rules, but may only have one outgoing old forwarding rule for d'; Thus, a new destination is needed for every delay of an edge when using this construction.

Proof. **The problem is in NP:** Given a sequence of (supposedly loop free) updates, each single one can be checked sequentially in polynomial time to be loop free by using, e.g., Tarjan's algorithm [77].

NP-hardness of $r = 4$: We start by proving the theorem for $r = 4$, before extending it to larger $r \leq n^{1-\epsilon}$. The case of $r = 3$ was already covered above in Theorem 3. We will again use the idea of delaying the update of every forwarding rule by one update akin to the construction in Figure 2.7, but we will just use one additional destination d', cf. Figure 2.8.

First, we add a new destination d' to the network. Now let v, w' be two nodes in the network s.t. there is a new forwarding rule for d from v to w'. We add a new node v', and replace the forwarding rule for d from v to w' with one for d, d' from v to v'. Additionally, we add old and new forwarding rules for d from v' to w', a new forwarding rule for d' from v' to d', an old forwarding rule for d' from v' to v, and old and new forwarding rules for d' from v to d', see Figure 2.8.

After applying this construction for every new forwarding rule in the original network, we added $O(n)$ additional nodes and forwarding rules. We note that every node from the original network has at most one incom-

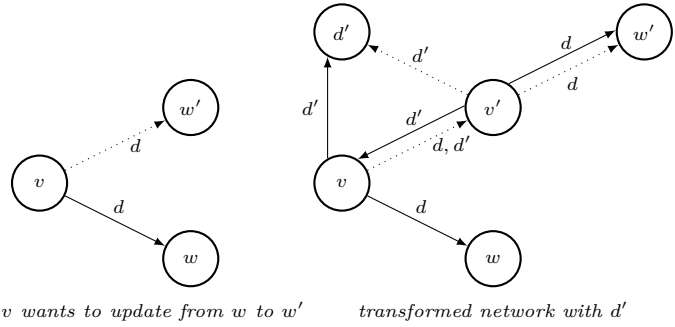


Figure 2.8: In this construction, every new forwarding rule for d from a node v to w' gets replaced by a forwarding rule for d, d' to a node v', the node for destination d is omitted among some further edges for simplicity of the illustration. For the node v', old and new forwarding rules for d to w' are added. Again, v cannot update to its new combined forwarding rule to v' before v' has updated its forwarding rule for the destination d'. Observe that in this construction, even when applied to every new forwarding rule of the original network, every node from the original network has only one incoming/outgoing forwarding rule for the new destination d'.

ing/outgoing forwarding rule for the new destination d'. Now, the update of every forwarding rule is delayed by one update, i.e., it is NP-hard to decide if the network can be updated loop free in $r = 4$ updats.

NP-hardness of $r \in \mathbb{N}, r \geq 3$: We can extend the construction used above for the NP-hardness of $r = 4$ to any larger natural number, cf. the construction in Figure 2.9: By adding the node v'' to the construction, the problem becomes NP-hard for $r = 5$, as v cannot update before v', which in turn cannot update before v''.

This can be iterated with v''' for the NP-hardness of $r = 6$, v'''' for $r = 7$, and so on. Note that if the original network has n nodes, the newly constructed network still has $O(n)$ nodes for any fixed $r \in \mathbb{N}$.

NP-hardness for $\leq n^{1-\epsilon}$: In the construction for $r \in \mathbb{N}$, we achieved NP-hardness for r in a network with $\Theta(n + rn) \in \Theta(n)$ nodes. The situation changes when r is dependent on n, e.g., by setting $r = n$, we get networks

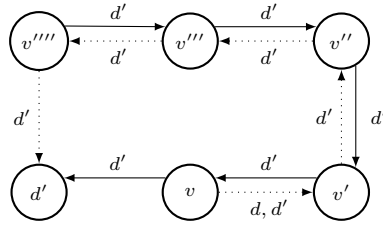


Figure 2.9: In Figure 2.8, v' had a new forwarding rule for d' pointing directly at d', inducing a delay of one update for the update of the new forwarding rule of v. This delay can be extended by adding additional nodes $v'', v''', \ldots$ as depicted in the construction shown in this Figure.

with $n' \in \Theta(n^2)$ nodes, meaning that the additional delay is just $\Theta(\sqrt{n'})$ opposed to $\Theta(n')$.

However, by setting the delay in the construction to n^x, $n \in \mathbb{N}$, it is NP-hard to decide if a schedule of length r^* exists for some $r^* > \frac{n^x}{n^{x+3}} = n^{1-\frac{3}{x+3}}$. Hence, the theorem holds by setting $x > \frac{3}{\epsilon} - 3$ for any ϵ, $0 < \epsilon < 1$. □

Thus, adding a second destination increases the complexity of scheduling loop free updates remarkably.

2.6 Single-Destination Loop-Free Updates

We already showed in Section 2.4.1 that for a number of $\Theta(n)$ destinations, the problem of dynamic loop-free updates is NP-hard via a reduction from 3-satisfiability.

However, is this problem really hard due to adding a linear amount of destinations – or is the complexity already hidden in the two choices every node has with one destination? Should a node 1) update, or 2) not update? As it turns out, already these two choices make the problem hard (Subsection 2.6.1), but one can approximate it well (Subsection 2.6.2).

2.6.1 Hardness of the Dynamic Case

Theorem 5. *The problem of dynamic loop free updates for one destination is NP-complete.*

Proof. **The problem is in NP:** Observe that the problem is indeed in NP: Given an update U of new forwarding rules, checking the union of the old and to be updated forwarding rules to be loop free can be performed in polynomial time, e.g., by Tarjan's algorithm [77].

NP-hardness: It is left to show that the problem is NP-hard. Our proof is a reduction from the classic NP-complete problem Feedback Arc Set (FAS) [29]: Given a directed graph, are there c edges s.t. their removal results in a loop free graph?

We show that for every instance I of FAS, we can construct in polynomial time an instance I' of the corresponding decision problem of the dynamic loop free update problem, s.t. I is satisfiable if and only if I' is a *yes*-instance.

For an illustration of the technique used in the remainder of the construction we refer to the Figures 2.10, 2.11, where the network in Figure 2.11 represents an instance I' created from the instance I of the network in Figure 2.10.

Construction of the new Instance I': Let I be an instance of FAS, i.e., a directed graph $G = (V, E)$ and $c \in \mathbb{N}$: Is there a set of at most c edges $E_c \subseteq E$, s.t. the removal of those edges yields a loop free graph $G^* = (V, E \setminus E_c)$? W.l.o.g., let $V = \{v_1, \ldots, v_n\}$, and let $\Delta(v_i)$ be the out-degree of the node v_i, with the edges $\{e_{i,j_1}, \ldots, e_{i,j_{\Delta(v_i)}}\}$. Again, w.l.o.g. let e_{i,j_k} be the edge from v_i to v_k.

We construct an instance $I' = (V', E_d^{old}, E_d^{new})$ by first defining the set of nodes, then the set of old forwarding rules, and then the set of new forwarding rules.

V' consists of a destination $d \in V$, the nodes V, and $\forall v_i \in V$ we add $\Delta(v_i)$ nodes $\{v_{i,j_1}, \ldots, v_{i,j_{\Delta(v_i)}}\}$. Thus, V' has in total $1 + |V| + |E|$ nodes.

The set of old forwarding rules is defined as follows: For each node $v_i \in V$, we construct a directed path starting at v_i and ending at d as $v_i, v_{i,j_1}, \ldots, v_{i,j_{\Delta(v_i)}}, d$. I.e., we basically have an in-tree for d that consists of $|V|$ paths. The total number of forwarding rules in E_d^{old} is $|V| + |E|$.

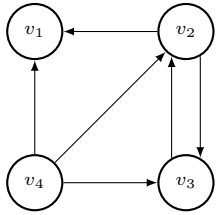


Figure 2.10: In this network instance I, there is exactly one loop between v_2 and v_3. Removing either of the two edges would solve the Feedback Arc Set problem in an optimal fashion.

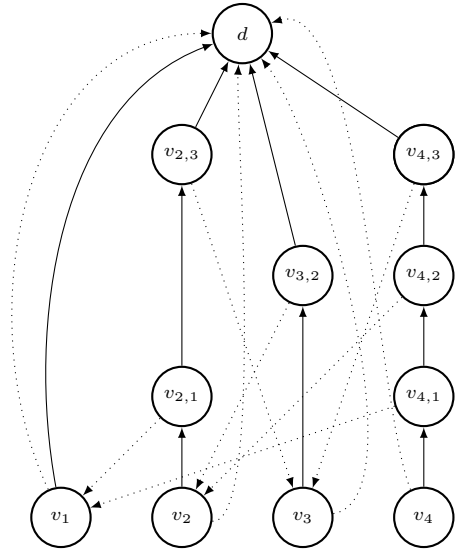


Figure 2.11: The corresponding network instance I' from I in Figure 2.10. The loop between v_2 and v_3 in I is represented by the loop between the nodes $v_2, v_{2,1}, v_{2,3}, v_3, v_{3,2}, v_2$. No further loops exist in I'.

The set of new forwarding rules mimics the edges of the graph G in the instance I. For each edge e_{i,j_k} in E, we construct a forwarding rule from v_{i,j_k} to v_{j_k} in E'. Furthermore, for each node $v_i \in V$, we add a forwarding rule $e_{i,d}$ from v_i to d in E'. Thus, the total number of forwarding rules in E_d^{new} is again $|V| + |E|$.

This set of new forwarding rules is loop free: *i*) The destination d has no outgoing rules, so all $|V|$ forwarding rules pointing at d cannot be part of a loop. Note that those $|V|$ forwarding rules origin from nodes v_i. *ii*) All other $|E|$ forwarding rules point at nodes v_i, and thus cannot be part of a loop either. Hence, both the old and the new forwarding rules are loop free.

Note that the instance I' can be constructed from the instance I in polynomial time. We now pose the following decision problem for the constructed instance $I' = (V', E_d^{old}, E_d^{new})$: *Is the maximum number of forwarding rules that can be updated loop free at least* $(|E_d^{new}| - c)$*?*

If I is a *yes*-instance, then I' is a *yes*-instance Let E_c be a set of c edges, s.t. the removal of those edges from E yields a loop free directed graph $G^* = (V, E \setminus E_c)$. Let I^*, be the instance I', with the forwarding rules constructed out of E_c removed, i.e. $I^* = (V', E_d^{old}, E_d^{new,*})$. Note that I^* still has $1+|V|+|E|$ nodes. By definition, G^* is loop free. For contradiction, let us assume that $(V, E_d^{old} \cup E_d^{new,*})$ contains a loop L. L must contain a mix of old and new forwarding rules, since each set is loop free individually. Since the outgoing new forwarding rules from v_i point directly at d, the loop L cannot contain two new forwarding rules consecutively.

We now contract the paths of old forwarding rules originating at v_i in I^*, i.e. $v_i, v_{i,j_1}, \ldots, v_{i,j_\Delta(v_i)}, d$ into just one new node v_i^{contr} that points at d with an old edge. The new contracted node v_i^{contr} has all ingoing new forwarding rules/edges from v_i and all outgoing new forwarding rules/edges from the nodes $v_{i,j_1}, \ldots, v_{i,j_\Delta(v_i)}$, plus one old forwarding rule/edge that points at d. We do this for all nodes $v_i \in V$ in I^*, leading to a contracted graph with $|V|+1$ nodes, given by $v_1^{contr}, \ldots, v_n^{contr}, d$. If we were to remove the node d from the contracted graph (and all ingoing edges to d), we would have a graph isomorphic to $G = (V, E)$: Each node v_i^{contr} corresponds to the node v_i in G^*. The same holds for the adjacency relations, as the contracted graph without d contains only new forwarding rules/edges, which in turn were created out of the edges from the edges of G^*.

Thus, if $(V, E_d^{old} \cup E_d^{new,*})$ were to contain a loop L, then the contracted

graph would contain a loop as well, and hence the graph G^* too. However, by definition, G^* was loop free, leading to a contradiction.

If I is a *no*-instance, then I' is a *no*-instance Let us assume that no set of c edges $E_c \subseteq E$ exist, s.t. the graph $G^* = (V, E \setminus E_c)$ is loop free. Thus, I is a *no*-instance. Now, let E_c be any set of at most c edges, but let E_c be fixed, and let $G^* = (V, E \setminus E_c)$. Note that G^* contains at least one loop L by definition. Again, as in the argumentation above, we look at the contracted graph without the destination d constructed from the instance I^*. Due to isomorphism, G^* contains a loop as well, which concludes the proof of Theorem 5. □

As seen in the above proof, the problem of dynamic loop free updates is strongly related to the Feedback Arc Set problem. The best known approximation ratio which can be achieved for FAS in polynomial time is $O(\log n \log\log n)$, as shown by Even et al. in their seminal paper [17]. Thus, we can show that finding a better guarantee for the dynamic loop free update problem cannot be achieved unless simultaneously improving the general case of the FAS problem:

Corollary 4. *A polynomial time algorithm for the problem of dynamic loop free updates for one destination with a better approximation ratio than $O(\log n \log\log n)$ would imply a better polynomial approximation ratio than $O(\log n \log\log n)$ for the Feedback Arc Set problem.*

Proof. Observe that in the proof of Theorem 5, the construction increases the number of nodes from n to $n+m+1$. As $O(\log n \log\log n)$ is equivalent to $O(\log n + m + 1 \log\log n + m + 1)$ due to logarithmic identities, a better approximation bound would immediately imply a better approximation bound for the Feedback Arc Set problem. □

In the next subsection, we will make use of this duality between dynamic loop-free updates and FAS.

2.6.2 Approximation of the Dynamic Case

While a greedy solution might seem like a good approach at first glance, it can be far from optimal regarding the number of updates sent out in one

flush, see Figure 2.12: Even for just one destination, an update can be of size up to $|E_d^{new}| - 1$, but a greedy one might just be 2 edges.

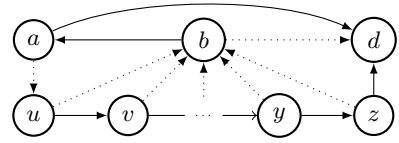


Figure 2.12: An update of the nodes a and b is a greedy update, but an update of the nodes $u, v, \ldots, y, z$ and b would be an update of maximum size.

Can we do better? Since we want to include as many edges as possible, we are essentially solving restricted instances of the NP-complete Feedback Arc Set Problem (FAS) [29]: Given a directed graph, what is the minimum number of edges that needs to be removed to break all cycles. FAS can also be considered in a variant with weighted edges: This allows us to exclude old edges from removal, by giving all old edges an arbitrarily high weight, and all new edges a weight of just 1. The best known approximation algorithm for weighted FAS has an approximation ratio of $O(\log n \log\log n)$ [17], allowing us to improve on a simple greedy algorithm:

Input: Nodes V and E_d^{old}, E_d^{new} as old/new rules, each loop-free.
Output: A loop-free update (V, E_d^{old}, E_d')

1. Set the weight of all edges contained in E_d^{old} to ∞, and the weight of all other edges to just 1.
2. Calculate a FAS F for the weighted graph $(V, E_d^{old} \cup E_d^{new})$ according to [17].
3. Set $E_d' = E_d^{new} \setminus F$.

Algorithm 1: Dynamic Loop-Free Updates for One Destination

Theorem 6. *The update calculated by Algorithm 1 is loop-free. The number of removed edges from E_d^{new} can at most be reduced by a factor of $O(\log n \log\log n)$.*

Proof. The removal of a FAS implies by definition loop freedom for the network. However, old edges are not allowed to be removed: But since all edges contained in the set of old edges $E_d^{old} = E_d^{old} \cup \left(E_d^{old} \cap E_d^{new}\right)$ have their weight set to infinity, there is always an infinitely better solution than removing any old edge. One would just set the edges being in $E_d^{'}$ to $\emptyset$, which results in a loop-free network by definition. □

Furthermore, by applying Algorithm 1 iteratively, the network will be migrated loop-free to the new forwarding rules in at most $O(n)$ updates, cf. [58]. Note that this scheduling of updates does not promise any approximation ratio regarding the fastest update schedule, even though there are instances where $\Omega(n)$ updates are optimal (e.g., reversing a chain of forwarding rules).

2.7 Hardness of Blackhole Freedom

Related to the study of loop freedom is the study of blackholes: A *blackhole* occurs when a packet arrives at a switch, but there is no matching rule to handle the packet. A straightforward way to avoid blackholes is to install some default rule which handles all packets not matched by some other rule. E.g., send packets to some neighboring node. However, this can easily introduce loops, or, if all packets are modified and sent back to the controller, a computational overload for the controller and congestion in the network.

Another issue occurs when forwarding rules are updated: Should one delete the old rule and then add the new rule, packets arriving in the meantime will be blackholed, unless some backup rule exists. Again, with enough memory, there is an easy fix: Introduce the new rule with a lower priority, and then delete the old rule. Should memory be limited however, we run into the problem of updating the switches in an efficient way:

Problem 6 (Blackhole-Free Routing under Memory Restrictions)**.** *Let $c_i \in \mathbb{N}$ be the total rule memory of a switch v_i, the combined number of rules in current use and the rules it can receive in one update. Let $G = (V, E)$ be the directed graph on which packets can be routed, with the destinations $D \subseteq V$ and the sources $S \subseteq V$ for the packets. In one round, a central controller can send out a set of any rules as an update to each node in the network. What is the minimum number of rounds, to migrate the network*

from a set of blackhole free old rules to a new set of blockhole free rules, if no blackholes should be introduced during migration and routing without loop should be possible at all times?

Theorem 7. *Problem 6 is NP-hard.*

The proof for Theorem 7 is based on a reduction from the NP-hard directed Hamiltonian Cycle problem (HC), cf. [29]: Given a directed graph $G = (V, E)$, is there a cycle that visits each node exactly once? The construction with further details is shown in Figure 2.13: It is possible to migrate blackhole free in two rounds if and only if there is a Hamiltonian Cycle in G, thus allowing to first use the cycle for intermediate routing via default rules, and then installing the new rules; Else it will take three rounds, one for each new rule. Thus, it is NP-hard to decide whether one can migrate in two or three rounds, even if the diameter is just two.

Proof. **Construction of the new Instance I'**
Let $G = (V, E)$ be an instance I of HC with the graph $G = (V, E)$. We will construct an instance I' of Problem 6 by first defining the vertex set V', then the set of packet destinations $D \subseteq V'$, followed by the set of packet sources $S \subseteq V'$, and the rule memory c_i for each switch. Afterward, we define the directed edge set E', the set of old rules, and the set of new rules.

The node set V' consists of six nodes $v_1^{old}, v_2^{old}, v_3^{old}, v_1^{new}, v_2^{new}, v_3^{new}$ and the node set V. W.l.o.g. let $V = \{v_1, v_2, \ldots, v_n\}$. The set of destinations D for packets is V, and the set of sources S for packets is V. The rule memory is 4 for each node in V, and $|V|/3$ for the remaining six nodes $v_1^{old}, v_2^{old}, v_3^{old}, v_1^{new}, v_2^{new}, v_3^{new}$.

The edge set E' consists of:

- The set of Edges E.
- $\forall v \in V$ and $\forall i$ with $1 \leq i \leq 3$ there are edges (v, d_i^{old}) from v to d_i^{old} ($3|V|$ edges in total).
- $\forall v \in V$ and $\forall i$ with $1 \leq i \leq 3$ there are edges (v, d_i^{new}) from v to d_i^{new} ($3|V|$ edges in total).

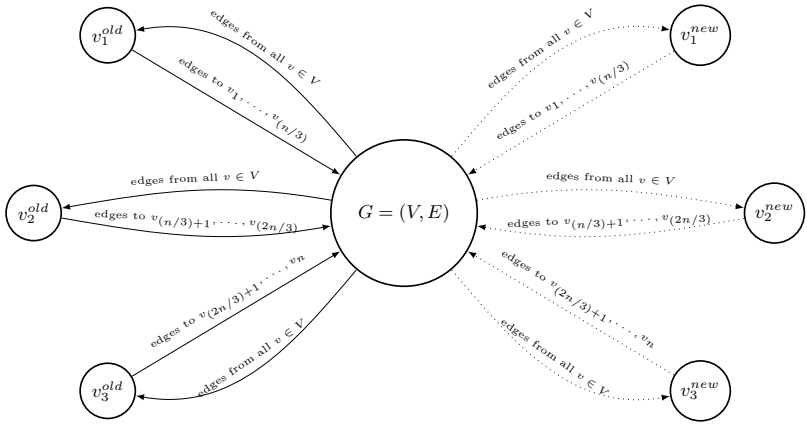


Figure 2.13: Overview of the construction of an instance I' for the proof of Theorem 7. The center node represents the graph $G = (V, E)$ from an instance I of the directed Hamiltonian Cycle problem. The sets of edges to and from the outer six nodes are bundled into single edges in this figure, their outgoing or ingoing nodes are labeled on top of each edge. Each node in $V = S = D$ has a memory limit c of four rules. The solid edges represent the edges used for the three old rules $\forall v \in V$, the dotted edges the edges used for the three new rules $\forall v \in V$. All nodes in V currently use the three nodes v_1^{old}, v_2^{old}, v_3^{old} on the left for 2-hop routing to the respective destinations in $D = V$, and want to migrate to use the three other nodes v_1^{new}, v_2^{new}, v_3^{new} on the right for 2-hop routing. If the network aims to be migrated blackhole free in two rounds, then each of the nodes in V needs to switch to a default rule pointed at some node in V in the first round. If and only if the set of these default edges form a Hamiltonian Cycle, then this first round of updates is blackhole free: Any packet from S will reach its destination in D by just using the default routing rules, which will guide the packet along a Hamiltonian cycle in V. Then, in a second round of updates, all nodes in V can switch to their three new rules. Should no Hamiltonian Cycle exist in $G = (V, E)$, then the update cannot be performed in two rounds, which can be proven by case distinction. Thus, it is NP-hard to decide whether one can migrate in two or three rounds, even if the diameter of the network is just two. We note that the construction for the memory limit of $c = 4$ for all nodes in V can be directly extended to any $c \in \mathbb{N}$ with $c \geq 4$. Furthermore, note that blackhole freedom is easy to guarantee for each node in the presence of default rules, if one does not care about routing: Just set a default rule to any neighboring node. While packets might not arrive at all (and in addition violate other consistency properties, e.g., congestion freedom), blackhole freedom is guaranteed.

- $\forall v \in \{v_1, \ldots, v_{(n/3)}\}$ there are edges (v_1^{old}, v) and (v_1^{new}, v) ($(2|V|/3)$ edges in total).
- $\forall v \in \{v_{(n/3)+1}, \ldots, v_{(2n/3)}\}$ there are edges (v_2^{old}, v) and (v_2^{new}, v) ($(2|V|/3)$ edges in total).
- $\forall v \in \{v_{(2n/3)+1}, \ldots, v_n\}$ there are edges (v_3^{old}, v) and (v_3^{new}, v) ($(2|V|/3)$ edges in total).

The set of old rules for the n destinations V is defined as follows:

- $\forall v \in \{v_1, \ldots, v_{(n/3)}\}$ and $\forall v' \in V$ there are rules $(v', v_1^{old})_{v_1, \ldots, v_{(n/3)}}$.
- $\forall v \in \{v_{(n/3)+1}, \ldots, v_{(2n/3)}\}$ and $\forall v' \in V$ there are rules $(v', v_2^{old})_{v_{(n/3)+1}, \ldots, v_{(2n/3)}}$.
- $\forall v \in \{v_{(2n/3)+1}, \ldots, v_n\}$ and $\forall v' \in V$ there are rules $(v', v_3^{old})_{v_{(2n/3)+1}, \ldots, v_n}$.
- $\forall v \in \{v_1, \ldots, v_{(n/3)}\}$ there are rules $(v_1^{old}, v)_v$.
- $\forall v \in \{v_{(n/3)+1}, \ldots, v_{(2n/3)}\}$ there are rules $(v_2^{old}, v)_v$.
- $\forall v \in \{v_{(2n/3)+1}, \ldots, v_n\}$ there are rules $(v_3^{old}, v)_v$.

Basically, the set of destinations from V is split into three partitions, and each of the nodes from $v_1^{old}, v_2^{old}, v_3^{old}$ is responsible for routing to that partition.

We define the new rules for the n destinations V analogously :

- $\forall v \in \{v_1, \ldots, v_{(n/3)}\}$ and $\forall v' \in V$ there are rules $(v', v_1^{new})_{v_1, \ldots, v_{(n/3)}}$.
- $\forall v \in \{v_{(n/3)+1}, \ldots, v_{(2n/3)}\}$ and $\forall v' \in V$ there are rules $(v', v_2^{new})_{v_{(n/3)+1}, \ldots, v_{(2n/3)}}$.
- $\forall v \in \{v_{(2n/3)+1}, \ldots, v_n\}$ and $\forall v' \in V$ there are rules $(v', v_3^{new})_{v_{(2n/3)+1}, \ldots, v_n}$.
- $\forall v \in \{v_1, \ldots, v_{(n/3)}\}$ there are rules $(v_1^{new}, v)_v$.

- $\forall v \in \{v_{(n/3)+1}, \ldots, v_{(2n/3)}\}$ there are rules $(v_2^{new}, v)_v$.
- $\forall v \in \{v_{(2n/3)+1}, \ldots, v_n\}$ there are rules $(v_3^{new}, v)_v$.

The instance I' can be constructed from the instance I in polynomial time. Each of the rules is valid, as they only use edges that were constructed for E' before. Note that no rule uses an edge from E. Also, both the old and the new rules lead to a correct routing table and blackhole freedom: Packets are only created at sources, and for each source $s \in V = S$ there is a unique 2-hop rule path to each destination $d \in V = D$ via the nodes $v_1^{old}, v_2^{old}, v_3^{old}$ (for the old rules) or $v_1^{new}, v_2^{new}, v_3^{new}$ (for the new rules).

Furthermore, neither the old or the new set violates the memory constraints for the nodes. Each node $v \in V$ is assigned three rules in both cases, leaving one slot open. The six nodes $v_1^{old}, v_2^{old}, v_3^{old}, v_1^{new}, v_2^{new}, v_3^{new}$ with a memory limit of $|V|/3$ are assigned $|V|/3$ rules in both cases.

Note that the six nodes $v_1^{old}, v_2^{old}, v_3^{old}, v_1^{new}, v_2^{new}, v_3^{new}$ cannot be issued updated rules without inducing blackholes: For a new rule to be received, an old rule would need to be dropped first.

We now pose the following decision problem for the constructed instance I':
Can the network in instance I' be migrated without blackholes in two rounds to the new rules?

Can the network migrate in one round without blackholes?
Each node $v \in V$ has only enough memory to receive one more rule in an update. Dropping any rule beforehand would induce blackholes. Since each node from V needs three new rules, it is not possible to migrate the whole set of new rules in one round.

Can the network migrate in two rounds?
Unlike in the case of just one round, we could now send out intermediate (helper) updates in the first round, and then more updates in the second round. We will now show that a blackhole free migration in exactly two rounds is only possible, if the graph $G = (V, E)$ has a Hamiltonian Cycle. Note that only the nodes in $V \subset V'$ need to be migrated, all other nodes from V' have the same set of old and new rules.

First, assume that the graph $G = (V, E)$ in the instance I has a Hamiltonian Cycle. Then it is possible to migrate in two rounds as follows:

- In the first round, issue each node in V a default routing rule, s.t. the set of all these rules form a Hamiltonian Cycle in V. Since each node has enough memory to receive one additional rule, no memory limit is violated. Each node in V can now add the new default rule to their current rule list, and then remove all old three rules. Thus, the process is blackhole free. Also, each source node $v \in V$ now routes the packets via the Hamiltonian Cycle to each destination node in $v' \in V$.

- In the second round, all nodes in V have three free slots for new rules. Thus, each node $v \in V$ can be issued an update consisting of all three new rules.

  Each node v can then add the three new rules $(v, v_1^{new})_{v_1,\ldots,v_{(n/3)}}$, $(v, v_2^{new})_{v_{(n/3)+1},\ldots,v_{(2n/3)}}$, and $(v, v_3^{new})_{v_{(2n/3)+1},\ldots,v_n}$ to the current list of rules, and then dispose of the default rule.

- No inconsistent limbo state is induced due to asynchronicity: Should some nodes have implemented/discarded the default rule already, but others have not, it could be that the packet destination cannot be reached via the yet incomplete Hamiltonian Cycle, because the cycle of default rules is broken at a node w. Then the packet will still reach its destination, because w will route the packet in two hops to its destination via one of the nodes $v_1^{old}, v_2^{old}, v_3^{old}, v_1^{new}, v_2^{new}, v_3^{new}$.

- Thus, if in the instance I there is a Hamiltonian Cycle in $G = (V, E)$, then it is possible to migrate the instance I' blackhole free in two rounds.

To complete the proof, let us now assume that the graph $G = (V, E)$ in the instance I has no Hamiltonian Cycle. Then it is not possible to migrate blackhole free in two rounds:

- After the second round, all nodes in V need to be issued the three new rules. Since at most one rule can be transmitted in the first round, at least two new rules need to be issued in the second round to each node in V. We now argue by case distinction:

- Let us assume that there is one node w in V that received one of the three new rules in the first round, w.l.o.g. $(w, v_1^{new})_{v_1,\ldots,v_{(n/3)}}$. If the old rule $(w, v_1^{old})_{v_1,\ldots,v_{(n/3)}}$ is not dropped before the second round, then no new rules can be received by w (unless the newly received rule is deleted, making the first update pointless), and it is not possible to migrate blackhole free in two rounds. Thus, the node w has to add the rule $(w, v_1^{new})_{v_1,\ldots,v_{(n/3)}}$ to the list of current rules and then drop the rule $(w, v_1^{old})_{v_1,\ldots,v_{(n/3)}}$ before the second round. Other rules cannot be dropped without inducing blackholes. In the second round however, the node w can now at most receive one of the two missing rules, making a blackhole free migration in two rounds impossible.

- Hence, no node in V can receive any of the three new rules in the first round of updates, if one wants to finish in two rounds. The only possible way to still be able to migrate in two rounds would be to issue an update of another sort. Let us assume that one node w in V receives a rule R that is neither one of the three new rules, nor one of the three old rules, nor a default rule to one of the nodes in V:

  - If the by w received rule R were a rule to one of the six nodes v_1^{old}, v_2^{old}, v_3^{old}, v_1^{new}, v_2^{new}, v_3^{new}, then that update would induce a blackhole or be an update that needs to be discarded: If the set of destinations covered by the rule R contains a proper superset of the destinations covered by one of the old rules, then we would have a blackhole, because at least one destination would not be covered after the next hop. If R does not contain a proper superset, then the rule is of no use, it needs to be discarded to open a slot in the second round.
  - If the by w received rule R were a rule that covers only a proper subset of V and is pointed at a node in V, then we cannot finish the update in the second round. Since R only covers a proper subset of V, w needs to keep at least one more old rule for the second round to avoid blackholes. However, this only leaves two slots for new rules in the second round, but the node w is still missing three new rules, making a blackhole free migration in two rounds not possible.

- Hence, the only option left is for all nodes $w \in V$ to be issued a default rule in the first round, each pointing at some node in V. However, the graph $G = (V, E)$ has no Hamiltonian Cycle, meaning that the set of default rules does not induce a strongly connected component: The only way to create a strongly connected component with n nodes and n edges is a Hamiltonian Cycle!

- Thus, if in the instance I there is no Hamiltonian Cycle in $G = (V, E)$, then it is not possible to migrate the instance I' blackhole free in two rounds.

This concludes the proof of Theorem 7. □

We note that the construction for the memory limit of $c = 4$ for all nodes in V can be directly extended to any $c \in \mathbb{N}$ with $c \geq 4$.

Corollary 5. *It is NP-hard to approximate the number of rounds needed for Problem 6 with an approximation ratio strictly better than* $3/2$.

2.8 Loop-Free Updates and Local Checkability

So far, we relied on the following paradigm when applying updates: First, gather the state at the centralized SDN controller, then, compute updates and distribute them in the network, sending out further updates after their implementation was acknowledged. It would be advantageous to perform these costly global operations only if needed – and otherwise rely on inexpensive local verification [73].

As it turns out, networks managed by a central controller, such as Software Defined Networks, show a strong similarity to prover-verifier pairs [20] (i.e., proof-labeling schemes). The controller can take on the role of the prover, and the switches in the network itself the role of the verifiers. We show how the concept of local checkability can be used for graceful network reconfigurations with the example of migrating in a loop-free manner between forwarding rules.

While the current line of research on local checkability focuses extensively on its theoretical properties, the number of practical applications beyond verification of a proof is sparse to the best of our knowledge. E.g.,

Schmid and Suomela [73] use proof-labeling schemes to verify spanning trees in networks.

The methods discussed in Subsection 2.6.2 and those from [52, 58] are inherently non-local however, as the updates can be anywhere in the network. Consider the example in Figure 2.14, where the whole network can be updated in a few rounds of switch-controller interaction, but the nodes with new forwarding rules cannot decide with local communication whether it is safe to update loop-free or not.

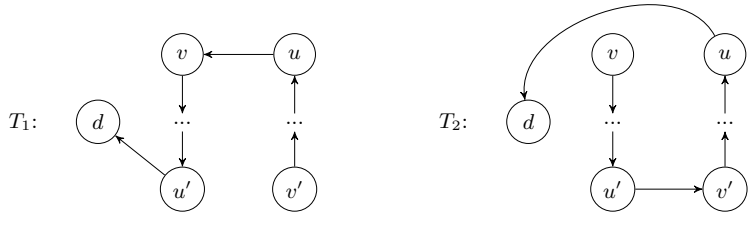


Figure 2.14: The loop-free migration from T_1 to T_2 can be handled in two updates by a centralized controller: First, u updates, and then after u confirmed the update to the controller, in a second update, the controller tells u' to update. As the distance between the nodes u, v and u', v' can be $\Omega(n)$, non-local communication via necessary: Else, u' cannot know when it is safe to update without inducing a loop.

A different approach is taken in [25, 26], which can be seen as analogous to a local checkability approach: For the new forwarding rules, defined as a tree T_2, label the root with (update) 0, then its children with (update) 1, and so on, labeling each node with an (update) number equivalent to its distance to the root of T_2. As T_2 could have a depth of $n - 1 \in \Omega(n)$, they need $\Omega(n)$ (or depth of T_2) subsequent updates in the worst case. Recall that no faster algorithm (dependent on n or the tree depth) can exist either for the general case: Consider a degenerated tree T_1, where every node has at most one child. If the parent-child relation is flipped in T_2 (with the only leaf becoming the only child of the root), then $\Omega(n)$ (or depth of T_2) updates are required, cf., e.g., [52].

2.8.1 A Local Migration Scheme

We will extend the scheme used by [25, 26] in this subsection in a way that every node will be able to decide locally if it can change its forwarding behavior to the new routing tree T_2.

We note that the nodes cannot verify if the new tree T' is actually T_2, as the very nature of T_2 is decided upon by the central controller. An analogous case can be about identifiers, which is why we will adopt the model of Schmid and Suomela [73] and assume every node has a unique identifier $\texttt{id}(v)$ (of size $O(\log n)$). As thus, we will only make sure that the nodes update to the tree specified by the sent out labels. We will still maintain the loop-free (i.e., tree) property at all times, no matter what labels are sent by the controller.

Input: Graph $G = (V, E)$ with the forwarding rules of the nodes forming a directed tree $T_1 = (V, E_1)$ with root $d \in V$. Every node $v \in V, v \neq d$ gets a label consisting of $\texttt{id}(p_2(v))$ of its parent $p_2(v)$ in T_2 and the depth $d_2(v)$ of v in T_2.

1: **while** v did not update yet **do**
2: **if** $p_2(v)$ sends depth $d_2(p(v)) = d_2(v) - 1$ or $p_2(v) = d$ **then**
3: Update forwarding rule to $p_2(v)$ and then send $d_2(v)$ to all neighbors
4: **end if**
5: **end while**

Algorithm 2: Update algorithm from the perspective of a node v.

We will now prove that Algorithm 2 works as intended, if the labels are correct:

Lemma 1. *Let the situation described in Algorithm 2 be correct: Then, executing Algorithm 2, the nodes $v \in V$ perform consistent updates, leading to a consistent migration of loop-free updates from the old forwarding rules T_1 to the new forwarding rules T_2.*

Proof. We will first prove that every update is loop-free, before showing that a loop-free migration to T_2 occurs.

Observe that initially, there is no loop in the network. Furthermore, note that during the whole execution of Algorithm 2, every node v has either a forwarding rule pointing at its parent $p_1(v)$ in T_1 or a forwarding rule pointing at its parent $p_2(v)$.

Assume for the sake of contradiction that the update of the node v' is the first occurrence of a loop in Algorithm 2. W.l.o.g., let this loop be $v' = v_0, v_1, v_2, \ldots, v_k, v' = v_{k+1}$. v_0 will only have updated to v_1 after v_1 has updated to v_2, with v_1 only updating to v_2 after v_2 has updated to v_3, and so on. As thus, for $1 \leq i \leq k+1$, v_i must be the parent of v_{i-1} in T_2. This leads to the desired contradiction: As T_2 is a tree, no loop $v' = v_0, v_1, v_2, \ldots, v_k, v' = v_{k+1}$ can exist in it. Hence, every update performed by Algorithm 2 is loop-free.

As every update is loop-free, Algorithm 2 performs a loop-free migration, but it is left to show that the migration will reach T_2. Now, assume, again for the sake of contradiction, that at some point no node can update any longer, but the forwarding rules do not form T_2. If every node (except for d) has updated, then the forwarding rules form exactly T_2. Thus, let $v' \neq d$ be a node which has not updated yet. We can use similar reasoning to the paragraph above: If v' has not updated yet, then $p_2(v')$ has not updated yet, which means in turn $p_2(p_2(v'))$ has not updated yet, and so on. As the graph is finite, this leads to the desired contradiction, as else a loop would need to exist. □

However, the network could still end up in a loop if the new forwarding rules do not form a tree T_2, but contain some loop, due to the error of the controller. As we incorporated the depth of each node into its label, this will be prevented:

Lemma 2. *Let the situation described in Algorithm 2 be correct, except that the new forwarding rules do not form a tree. Then, the updates performed by Algorithm 2 will still be loop-free.*

Proof. W.l.o.g., assume for the sake of contradiction that the update of a node v' is the first update of Algorithm 2 inducing a loop $v' = v_0, v_1, v_2, \ldots, v_k, v' = v_{k+1}$. As shown in the proof of 1, this loop must be exclusively induced by the new forwarding rules, which could be the case now as the new forwarding rules no longer have to form a tree. However, when updating, every node also checks if the label for the depth of the tree is smaller than its own by exactly one. Consider the smallest depth given as part of a labeling to a node v_i in the loop $v' = v_0, v_1, v_2, \ldots, v_k, v' = v_{k+1}$. When v_i updated, it checked the depth of its parent to be exactly one smaller than its own.

However, as v_i has the smallest depth in the loop, this is a contradiction: v_i would not have updated. □

2.8.2 Beyond a Single Update

We note that the local migration scheme can also be extended beyond updating from T_1 to T_2, to, e.g., T_3, without waiting for the network to converge to T_2 first. When a node $v \neq d$ receives its label for T_3, it waits until it receives $d_3(v) - 1$ or d from $p_3(v)$, and then updates its forwarding rule to $p_3(v)$ and sends $d_3(v)$ to all neighbors. This can be iterated, meaning that when the network is updating to T_i, it can be a mix of forwarding rules from T_1 to T_{i-1}, but will eventually converge loop-free to T_i.

2.8.3 Further Applications in SDNs

The loop freedom of forwarding rules is just one of many consistency properties to be considered when performing changes in the behavior of switches of Software Defined Networks via the controller. We envision that local updates for other consistency properties can be developed as well, e.g., for black hole freedom, per-packet consistency, waypoint enforcement, and bandwidth capacity constraints for network flows.

2.9 Flipping the Approach: Reachability Testing

So far, we covered the case of avoiding transient errors that occur during network updates, assuming the controller (and the network itself) behaves correctly.

Network failures are inevitable however. Interfaces go down, devices crash and resources become exhausted. It is the responsibility of the controller to provide reliable services on top of unreliable components and throughout unpredictable events. Guaranteeing the correctness of the controller under all types of failures is therefore essential for network operations. Yet, this is also an almost impossible task due to the complexity of the control software, the underlying network, and the lack of precision in simulation tools.

Instead, we argue that testing network control software should follow in the footsteps of large scale distributed systems, such as those of Netflix [7],

Amazon [79], Microsoft Azure [75], or Google [12], which deliberately induce live failures in their production environments during working hours, and analyze how their control software reacts.

2.9.1 Problem Properties and Algorithms

In this section, we look at how to compute failure scenarios that maintain the network-wide invariant of reachability, e.g., no loops should be induced. Reachability is indeed the most fundamental property that any controller must maintain. If a link (edge) fails, the controller should restore reachability provided the physical graph is still connected. We will compute failure scenarios that optimize two other objectives besides reachability: *coverage* and *speed*. In short, we aim at failing each link at least once, in as few iterations as possible.

Coverage. While distributed protocols are guaranteed to maintain reachability as long as the network is connected, SDN controllers do not. As such, we want to check if the controller can handle the failure of every single link.

A naïve algorithm would be to perform the following test for every link e. First, check if the network N is still connected without e. Second, if yes, fail the physical link e and see if routing is still possible between all nodes in the SDN; if not, inform the network operator that a single link of failure is in the network.

Coverage + Speed. While the runtime of this naïve algorithm is polynomial, its number of iterations is not acceptable in practice with large networks containing thousands of edges. In the best case, even huge networks should only need a small amount of iterations in average for all testable edges.

As such, we aim at solving the Connectivity Testing problem:

Problem 7 (Connectivity Testing)**.** *Find the minimum number of iterations k, where each k_i is a set of failed edges, that are needed to fail every edge at least once while still maintaining network connectivity.*

Finding a solution for $k = |E|$ is easy (cf. the naïve algorithm), but minimizing k turns out to be an algorithmically challenging problem. Theorem 8 proves that optimally solving Connectivity Testing requires at

least 2 iterations, in the best case; and as many iterations as nodes in the network, in the worst-case.

Theorem 8. *Let $N = (V, E)$ be a connected network where at least one edge can be failed while maintaining connectivity.* Connectivity Testing *needs at least* 2 *and at most $|V| = n$ iterations. These bounds are sharp.*

Proof. We start with the lower bound: If $e = (u, v)$ can be removed, then e is part of at least one 2-connected component in N. Then, $N' = (V, E \setminus E')$ is still connected, i.e., there is a path P from v to u in N' that joined with e yields a cycle, i.e., one iteration never suffices. For networks where 2 iterations suffice, consider a clique with at least 4 nodes: First, remove a spanning tree T, and second, remove all edges except for T.

We now prove the upper bound. Connectivity Testing for a ring of n nodes needs exactly n iterations. To show that no graph needs more than n iterations, observe that after failing a spanning tree, at most $n-1$ edges can be left to fail. As any of these $n-1$ edges will be part of a cycle (else they could not be failed), at least one edge can be failed per iteration, resulting in a sharp upper bound. □

Algorithms. Observe that networks for which all edges can be failed in 2 iterations are characterized by every iteration containing a spanning tree. With this in mind, we propose an algorithm, Greedy Killer (see Algorithm 3), for solving Connectivity Testing which first checks for two edge-disjoint spanning trees and, in the negative case, proceeds to fail all edges that can be failed in multiple iterations.

Lemma 3. Greedy Killer *will fail all edges that can be failed and mark all the others with weight 1.*

Proof. Greedy Killer will never fail an edge that cannot be failed, as it always leaves a spanning tree.

Assume that there is an edge $e = (u, v)$ that can be failed (and thus part of a cycle C), but Greedy Killer will not fail e. Now, consider the network N after Greedy Killer has finished. If there are edges (e.g., e) in C with weight 1, then there is a spanning tree with weight W that will fail at least one of these edges e' that includes all edges from C except e'. Any spanning tree with weight W or less will fail at least one edge, as the weight

```
1: compute_link_failures_scenarios(N = (V, E))
2: if N has spanning trees T_1 = (V, E_1), T_2 = (V, E_2), E_1 ∩ E_2 = ∅ then
3:     fail E \ E_1 and fail E \ E_2
4: else
5:     ∀e ∈ E set link weights of c(e) = 1
6:     repeat
7:         compute minimum weight spanning tree (MST) T = (V, E')
8:         fail all links E'' = E \ E' not in the MST T
9:         set sum of new edge failures λ = Σ_{∀e∈E''} c(e)
10:        ∀e ∈ E'' set c(e) = 0
11:    until λ = 0 or ∀e ∈ E : c(e) = 0
12: end if
```

Algorithm 3: GREEDY KILLER algorithm. It fails all edges in 2 iterations if N contains at least 2 disjoint spanning trees. Else, at most $|V|$ iterations are needed.

W is less than the sum of all edge weights in the network. Thus, GREEDY KILLER would not have been finished, leading to a contradiction. □

Identifying all bridges (*i.e.*, edges whose removal disconnect the network) can also be performed by other algorithms, e.g., [78], but GREEDY KILLER will mark them as well, allowing us to inform the network operator about all single points of failure in the topology.

Lemma 4. *In each iteration,* GREEDY KILLER *fails the maximum amount of edges that have not failed yet.*

Proof. Consider any fixed iteration. Let E' be the set of edges e' which either cannot be failed or have been failed in a previous iteration. Let $N' = (V, E')$ and let $C_1, \ldots, C_k$ be the connected components of N' that are maximal. Any spanning tree (which maintains connectivity) for N will thus need to contain $k - 1$ edges from of individual weight 1 to connect $C_1, \ldots, C_k$ (which are individually connected by edges in E'), i.e., no MST can have a weight of less than $k - 1$. As the weight of any MST will be exactly $k - 1$, Lemma 4 holds. □

We can now prove that GREEDY KILLER performs according to specification:

Theorem 9. GREEDY KILLER *is correct.*

Proof. Combining the observation on edge-disjoint spanning trees and from Lemma 3 leaves only one open question: if GREEDY KILLER always finishes after at most n iterations. Note that after one iteration, there are at most $(n-1)$ edges still left to fail, as the first iteration failed all edges not contained in a spanning tree. As the algorithm will fail all failable edges eventually (cf. Lemma 3), it cannot take more than n iterations: Then, there would be an iteration where no edge is failed, yielding a contradiction to Lemma 4. □

Running time. Checking if a network has two edge-disjunct spanning trees can be done in $O(n^2)$ time [70]. Kruskal's MST algorithm takes $O(m \log n)$ time [14]. As such, GREEDY KILLER runtime is $O(nm \log n)$.

2.9.2 Evaluation

The performance of GREEDY KILLER was evaluated in [74], using 261 topologies extracted from the Internet Topology Zoo [48] and RocketFuel [57] topologies. The topologies were pre-processed, removing links whose removal would break network connectivity, i.e., all remaining networks were 2-connected. Most topologies could be failed in 5 or less iterations, 78% in 6 or less iterations, and 91% in 8 or less iterations.

Greedy Killer is optimal most of the time. We now compare the number of iterations computed by GREEDY KILLER with the optimal solution, *i.e.* the smallest number of iterations to fail all possible links. In a network $N' = (V', E')$ with $|V'| = n'$ and $|E'| = m'$, at least $n'-1$ edges always need to stay in N' to maintain reachability. Therefore, at most $m'-(n'-1)$ edges can be failed in each iteration, yielding a lower bound of $\lceil m'/(m'-n'+1) \rceil$ iterations.

It was shown in [74] that GREEDY KILLER produces the optimal number of iterations for at least 138 of the 265 networks (52%).

An open question we are working on is to find an optimal algorithm to succeed in the minimum amount of iterations possible.

2.9.3 Further Applications in SDNs

Although connectivity is preserved, the failures may not preserve more advanced properties. For instance, failing entire spanning trees can decrease

network capacity so much that congestion starts to appear. Similarly, security policies can also legitimately prevent reachability in connected graph.

Fortunately, it is easy to combine Greedy Killer with other network-wide invariants. For example, to test the congestion-free properties of a SWAN-like SDN controller [37], we can first monitor the current traffic to estimate the demands. Then, using Multi-Commodity Flow formulations [1], we can maximize the set of not yet failed edges in each iteration while maintaining sufficient network capacity.

Part II

Consistent Migration of Flows

3

On Consistency for Flow Updates

Network operators continuously strive for increased performance, especially to increase bandwidth utilization [41], an aspect we did not consider in the prior Part I: In Chapter 2, we targeted the topological consistency property of loop freedom, but not congestion directly. Even though networks provide a best-effort service, congestion is highly problematic especially in contexts such as WAN and Data-Centers. As such, there has been (not only) recent interest in mechanisms to provide some sort of consistency for network updates in the context of bandwidth violations [37, 42].

Motivated by applications in data centers and "elephant"-size data flows in general, congestion during network updates is mostly not considered for the topological case of updating individual forwarding rules, but rather from the viewpoint of multi-commodity flow theory: Following this approach, we will study consistent flow migration in this part of the presented thesis, where consistency refers to respecting bandwidth constraints on edges.

We start in Section 3.1, where we give a general overview over the models

for consistent flow migration in this part. We then give an overview of the background and related work in this area in Section 3.2, before summarizing the current state of the art and our results in tabular form in Section 3.3.

The following four chapters in this part deal with consistent flow migration under different aspects, but mostly with the following take-away message: Migrating flows in a consistent way is NP-hard when considering unsplittable flows, but allowing the flows the be split leads the problem into the tractable polynomial realm: More specifically, in Chapter 4 we study the seminal model of *SWAN* [37], before including waypoint constraints in Chapter 5, the impact of packets in flight in Chapter 6, and lastly, extending the idea of augmenting flows to flow migration to new demands in Chapter 7.

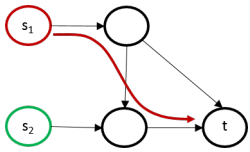
3.1 Models for Consistent Flow Migrations

Similar to the previous part, we will model a network as a directed connected graph with edge capacities, where the nodes represent the switches/routers. Each flow in the network has a source and a destination, with the packets being tagged for the respective flow. As such, unlike in the previous part, packets heading for the same destination arriving at some node can then take different paths.

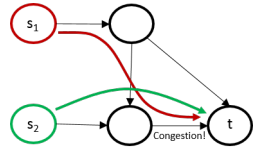
Furthermore, all update mechanisms in the following chapters will implicitly use the 2-phase commit by Reitblatt et al. [69] as follows: Before the update, all nodes will have forwarding rules for the flows in their old state. Then, all affected nodes will install rules for the flows in their new state. As soon as all these nodes acknowledged the installation of the new rules, the sources can be informed to tag their flow packets with the new rules.

However, even if the old and new rules are without congestion, there can still be bandwidth violations during the update as shown in Figure 3.1: Due to asynchrony, source s_2 starts sending packets even though the flow from s_1 has not updated to its new path yet. In this case, an easy fix is to update s_1 first, and then s_2. We will later see examples where consistency can only be maintained by splitting flows, or even not at all.

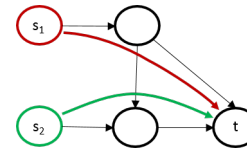
In the following chapters, we will essentially consider three different models of consistency for flow migration. For ease of accessibility, and due to small subtleties in the problem formulation, we give a formal model in each



(a) Initial network with one flow from s_1 to t.



(b) If the lower flow is inserted first, there will be congestion.



(c) Desired flow placement with two flows of two commodities.

Figure 3.1: This figure depicts a small network to introduce the concept of consistency. In the above examples, all flows have a size of one and all edges have a capacity of one as well. If the SDN controller desires to migrate the network from Subfigure 3.1a to Subfigure 3.1c in order to add a flow for the second commodity outgoing from s_2, then the commodity outgoing from s_1 has to be moved first. Else, due to asynchrony, s_2 could start a flow before the last edge is free, causing congestion (Subfigure 3.1b).

of the following four chapters. Nonetheless, we give a comparing overview here:

In Chapter 4, we use the seminal model introduced by *SWAN* [37], who implemented and evaluated the consistent migration of flows in SDNs with multiple production data center networks across three continents with tens of thousands of servers. In their model, the sum of the flows on each edge should not violate the edge's capacity, whether or not each flow is in its old or new state. As thus, they capture the effects of flows congesting other flows.

In Chapter 5, we let flows only use their old or new paths, but no mix of the two and no intermediate paths. As such, waypoints and service chains can easily be respected, but the solution space is strongly restricted opposed to the model in Chapter 4. Still, it may be the most pleasant model from an operator's point of view, as no additional paths need to be created and most updates only revolve around changing the split-ratio of flows at their sources.

The model of Chapter 6 is situated in the solution space between the models of Chapter 4 and 5. We lay our new focus on packets in flight by considering edge latencies, extending consistency to losslessness. Specifically, we also cover the impact of flows congesting themselves, opposed to

only inter-flow congestion.

To give a rough overview, updates consistent in the model of Chapter 5 are consistent in both other models, and consistency in Chapter 6 implies consistency in Chapter 4, but the reverse does not have to be true, respectively.

In the last Chapter 7 we relax the update problem, as we will not migrate to a specific new multi-commodity flow, but just to new demands. For the case of a single destination (or, source), this flexibility allows us to greatly reduce the number of updates needed in the worst case, from an unbounded to a polynomial amount. Interestingly, update problems in this model are always possible in a consistent way, but an extension to multiple destinations is not viable.

3.2 Related Work and Background

The two fields most related to our work are network flow optimization problems in general and consistent flow migration in software defined networks, which we cover in the following two subsections.

Network Flow Optimization Problems

Since the seminal work by Ford and Fulkerson [21], there has been a vast array of different methods to deal with (maximum) flow problems: Augmenting paths, linear/integer programming, approximation algorithms [32], the preflow-push method, Dinic's algorithm [15], or the recent algorithm of Orlin [65], to name just the more popular ones. These methods mainly deal with maximizing the flows according to some objective function, and as thus, are aimed at calculating the desired flows, but not describing how to migrate to them in a consistent fashion. We refer to the books of Ahuja et al. [1] and Cormen et al. [14] for a comprehensive overview.

Splitting flows Splitting flows is usually done in the context of generating a set of unsplittable flows whose union is the original splittable flow, to deploy them in *MPLS* or *OpenFlow* networks: Approximating the minimum number of unsplittable flows needed is an *NP*-hard problem however [33]. In a similar fashion, maximizing a k-splittable flow is *NP*-hard to approximate as well [6]. Splitting flows can lead to packet reordering problems in

computer networks, which needs to be addressed in practice, cf., e.g., [43]. Beyond hash based flow splitting, major techniques are flow(let) caches and round-robin splitting, cf. [35].

Software Defined Networks & Flow Migration Consistency

Dynamically changing the flow of traffic has also been considered in traditional networks [4, 28], but it was not before SDNs that consistency during the network update was studied extensively in the context of flow migration.

Consistent Network Updates for Flows The work by Reitblatt et al. [68] introduced a seminal form of consistency via 2-phase commit, called *per-packet / per-flow consistency*: By stamping each packet or flow with a version number, denoting old or new forwarding rules, one can ensure that a packet/flow is always routed according to just one set of rules. While their work prevents many effects that induce congestion, it is not lossless, cf. Subfigures 3.2a and 3.2b.

Dionysus [42] tries to find a consistent migration ordering by greedily searching through a dependency graph of possible migration steps. In their model, flows are not allowed to take intermediate paths and may only migrate as a whole to the desired path, see the Subfigures 3.2b,3.2c. They show the corresponding decision problem to be NP-hard under switch memory constraints. If no solution is found, some flows are rate-limited for congestion-free migration of the remaining flows. The corresponding MIP-formulation was considered in [55], and further local dependency resolving in [84].

The approach of *SWAN* [37] is twofold: First, if one guarantees that a fraction s of capacity (*slack*) is free on each link for the old and new flow, then one can migrate congestion-free in $\lceil 1/s \rceil - 1$ steps. E.g., if each edge never exceeds 95% capacity in both the current and the desired network state, it takes 19 steps to migrate at most. Should background traffic exist in the network, it can be rate-limited during migration, and restored to its normal state after. Second, they use binary search over the number of steps to find the optimal solution, i.e., they check with linear programming if a solution with x steps exists. However, they note that this is costly in practice for a long sequence of LPs. This search also works if there is no slack on some edges, but then the computation is not guaranteed to halt

when no consistent migration exists. We refer to the Subfigures 3.2c,3.2d for examples. The ideas of *SWAN* [37] were also used in a data center setting in [50].

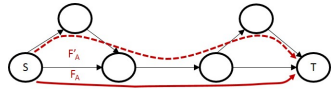
Zheng et al. [86] take an orthogonal approach, and fix the number of x intermediate update states. For given x, they give an MIP minimizing the maximum transient congestion, and also consider an LP-based approximation: However, if intermediate paths are allowed for the flows, the LP is of exponential size.

Also in the line of consistent flow updates, [66] handles dynamic flow arrivals, and [56] allows for user-specified deadlines of flow movements via MIP or an LP-based heuristic.

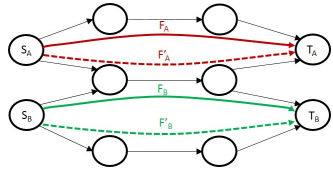
The main difference of the above works to ours is the following: They cannot decide if a consistent flow migration exist in general, as only MIP/LP-based approaches are considered, which only check if a solution of some x steps exists. As thus, deciding if a solution exists can only be done in the restricted case where the number of flow movements is arbitrarily limited. An overview of the above work and ours is also shown in Table 3.2 in the next Section, where we will also discuss the flow migration complexity in general.

As discussed shortly before in the loop-free part, Mizrahi et al. [60–63] tackle the problem of asynchrony of network updates: As network updates are installed in a distributed fashion in the network's routers, the atomicity of a network update cannot be guaranteed. The problem goes beyond standard clock synchronization, as timing the exact injection of rules is nearly impossible in the hardware deployed nowadays. Thus, they propose mechanisms called *TIMEFLIPs / timed-based consistent updates* to ensure that rules are installed when they should be installed: In the first, time is used as an alternative to version numbering, while in the second, various methods are employed to ensure accurate time synchronization. They also show with game-theoretic arguments, that for optimal traffic utilization, it can be needed on some edges to swap two flows.

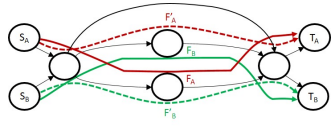
In specialized environments or topologies, such as Data-Centers, one can also take an orthogonal approach to network updates, by scheduling all the traffic, eliminating the need for the migration of flows [31, 44, 67]. Model checking is also used for consistent updates, but cannot handle bandwidth guarantees yet [59, 64].



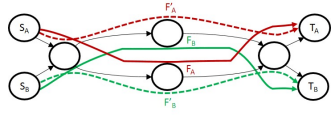
(a) When migrating from F_A to F'_A, the 2-phase commit method [68] will migrate the network in this example consistently in the model of [37]. This also holds for multiple commodities, as long as only one commodity is considered for migration. In both the non-mixing and lossless model introduced in Chapters 5 and 6, this update would be inconsistent, as packets can be lost due to congestion on the lower middle edge.



(b) Example where Dionysus [42] will migrate congestion-free by first migrating flow F_B to F'_B and then F_A to F'_A. However, the method from [68] might not migrate congestion-free, if F_A migrates to F'_A before F_B migrates to F'_B due to asynchrony.



(c) Example where SWAN [37] can migrate congestion-free by temporarily storing a flow on the topmost link. Dionysus [42] will not find a solution without rate-limiting one flow to zero, as intermediate paths are not considered.



(d) Example where the LP-method from SWAN [37] will keep computing forever until halted somehow manually or by a threshold of steps, since it is not possible to migrate congestion-free without rate-limiting.

Figure 3.2: Example networks where the methods of [37, 42, 68] do not succeed, due to violating congestion-freedom, rate-limiting flows, or by not halting their computation. All edges in the four examples have a capacity of one and all flows have a size of one. Initial flows are drawn solid, the new ones dashed.

3.3 State of the Art and Results

We already discussed related flow migration algorithms in the previous section, an overview of their and our work can be found in Table 3.2. The complexity of the consistent flow migration problem was not yet considered though in our coverage. Besides *Dionysus* [42] showing that it is NP-hard to decide if a consistent migration exists for splittable flows under memory restrictions, related work has not considered the decision problem in general: Using the Partition problem, Jin et al. [42], and later [56, 86] showed that finding the fastest schedule without memory restrictions is NP-hard.

We prove even stronger results, namely that already the general decision problem is NP-hard, though our Partition proofs are similar in principle. For the case of intermediate paths, our proof reduction via (MAX) 3-SAT also extends to flows of unit size, and is also the only result in this area giving inapproximability thresholds, cf. Section 4.4. A further overview is provided in Table 3.1.

Flow migration problem	Interm. Paths	Consistency Model	Decision problem hardness
Unsplittable	Yes	*SWAN* [37]	NP-hard [4.2]
		Lossless (fixed & arbitrary latencies)	Proof construction from [4.2] can be used
	No	Non-Mixing	NP-hard [5.3]
		SWAN [37]	
		Lossless (fixed & arbitrary latencies)	Proof construction from [5.3] can be used
Splittable	Yes	*SWAN* [37]	P [4.3]
		Lossless (fixed l.)	NP-hard [6.6]
		Lossless (arbitr. l.)	P [6.5]
	No	Non-Mixing	P [5.4]

Table 3.1: Table summarizing our decision problem results for flow migration. We note that the non-mixing model does not allow intermediate paths.

Ref.	Approach	(Un-)splittable model	Int. Paths	Computation	# Updates	Complete
Consistency model of *SWAN* [37], but restrict each flow to move only once						
[68]	Install old and new rules, then switch from old to new	Both	No	Polynomial	1	No bandwidth guarantees
[42]	Greedy traversal of dependency graph	Both	No	Polynomial	Linear	No (rate-limit flows to guarantee completion)
[55]	MIP of [42]	Both	No	*Exponential*	Linear	Yes
Further practical extensions						
[84]	Extends approach of *Dionysus* [42] with local dependency resolving					
Consistency model of *SWAN* [37], general setting						
[37]	Partial moves according to free slack capacity s	Splittable	No	Polynomial	$\lceil 1/s \rceil - 1$	Requires slack on flow edges
[86]	Mininmize transient congestion for fixed x updates via LP	Both	No	Polynomial	Any $x \in \mathbb{N}$	Approx. min. trans. cong. > 0 by $\log n$ factor
			Yes	*Exponential*		
[86]	... via MIP	Both	Both	*Exponential*	Any $x \in \mathbb{N}$	For any given x yes, but cannot decide in general
[37]	Binary search of intermediate states via LP	Splittable	Yes	Polynomial in # of updates	Unbounded	Cannot decide if migration not possible
C.4	Create slack with intermediate states, then use partial moves of [37]	Splittable	Yes	Polynomial	Unbounded	Yes
C.7	Use augmenting flows to find updates	Splittable, 1 dest., paths not fixed	Yes	Polynomial	Linear	Yes
Further practical extensions						
[50]	Extends approach of *SWAN* [37] in a data center setting					
[66]	Considers reconfiguration for dynamic flow arrivals					
[56]	User-specified deadlines and requirements via MIP (or LP heuristic)					
Non-mixing consistency model, each flow-packet only on old or new path						
C.5	Split unsplittable flows along old and new paths	2-Splittable	No	Polynomial	Unbounded	Yes
Lossless migration						
C.6	Find intermediate paths, use partial moves of [37]	Temporarily splittable	Yes	Polynomial	Unbounded	Yes (arbitrary latencies)
C.7	Use augmenting flows to find updates	Splittable, 1 dest., paths not fixed	Yes	Polynomial	Polynomial	Yes

Table 3.2: Overview of our algorithms and the related work from Section 3.2

4

Consistent Flow Migration

In this chapter, we begin our study of the consistent migration of flows: Given a current and a desired network flow configuration, we give the first polynomial-time algorithm to decide if a consistent migration in the seminal model of *SWAN* [37] is possible (Section 4.3). All known approaches will resort to breaking consistency by, e.g., dropping flows even if this is not necessary. The splittable flow consistency model of *SWAN* [37] is presented in Section 4.1, along with some notation and preliminaries for this chapter. Furthermore, we show that if flows have to be unsplittable, or integer or unit size, the corresponding decision problem is NP-hard (Section 4.2).

In a similar line of thought, we investigate the problem of consistently increasing the flow of traffic between a source and a terminal node, while keeping all other traffic flows intact. Current methods such as RSVP-TE consider unsplittable flows and assign weights according to the importance of the flow. Then, less important flows are removed until there is enough capacity, and finally the desired flow is added. We can show that deciding

what flows to remove is an NP-hard optimization problem with no *PTAS* possible unless P = NP (Section 4.4). If flows are splittable, we show how the maximum traffic increase can be approximated arbitrarily well: We can calculate in polynomial time (independent of the chosen approximation ratio) a viable new flow placement, s.t. *i*) the demand increase for the desired commodity is within $(1-\epsilon)$ of the maximum, *ii*) the demand of all other commodities is unchanged, and *iii*) consistent migration is possible (Section 4.5).

4.1 Model

We start by modeling a network as a directed graph, with a flow of a commodity respecting flow conservation, demand satisfaction, and capacity constraints:

Definition 6. *Let $G = (V, E)$, with $|V| = n$ and $|E| = m$, be a simple directed graph. For every node $v \in V$ denote the set of outgoing edges by $\texttt{out}(v)$ and the set of incoming edges by $\texttt{in}(v)$. A* network *N is a pair (G, c), where $c : E \to \mathbb{R}^+$ is a function that assigns a* capacity *$c(e)$ to each edge $e \in E$. Also, a* commodity *K in N consists of a pair (s, t) where s and t are nodes in G. We call a map $F : E \to \mathbb{R}_{\geq 0}$ a* (single-commodity) flow *for K if the following conditions are satisfied:*

- $\forall v \in V \setminus \{s, t\} : \sum_{e \in \texttt{out}(v)} F(e) = \sum_{e \in \texttt{in}(v)} F(e)$,
- $\sum_{e \in \texttt{out}(s)} F(e) = d_F = \sum_{e \in \texttt{in}(t)} F(e)$,
- $\forall e \in E : F(e) \leq c(e)$,

where d_F is called the demand *of K (w.r.t. F). We also call d_F the* size *of F.*

Note that this standard formulation allows cycles to exist where the flow of a commodity is greater than zero on each edge, which can however be easily removed by flow decomposition.

Definition 7. *We call a single-commodity flow F for commodity $K = (s, t)$ with demand d_F* cycle-free *if there is no cycle C in G, s.t. $\forall e \in C : F(e) > 0$.*

For ease of notation, all flows are considered to be cycle-free from now on unless noted otherwise. Since we study the migration of flows between different nodes, we extend the flow definition to multiple commodities.

Definition 8. *We call a tuple $\mathcal{K} := (K_1, \ldots, K_k)$ of commodities a* multi-commodity*. Let $F_1, \ldots, F_k$ be single-commodity flows for $K_1, \ldots, K_k$, respectively. Then we call the tuple $\mathcal{F} := (F_1, \ldots, F_k)$ a* multi-commodity flow *(for $\mathcal{K}$) if the following condition holds: $\forall e \in E : \sum_{i=1}^{k} F_i(e) \leq c(e)$.*

In later sections, we will also study flows that cannot be split up among different paths or are of integer value, i.e.:

Definition 9. *We call a single-commodity flow F for commodity K* unsplittable *if there is a simple path from s to t such that F assigns a value greater than zero only to edges along the path. Similarly, we call a multi-commodity flow unsplittable if all its single-commodity flows are unsplittable.*

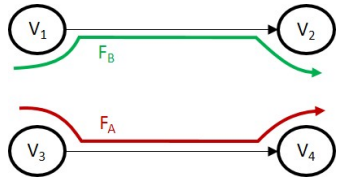
Definition 10. *We call a single-commodity flow F for commodity K* integer *if F assigns only integer values. Similarly, we call a multi-commodity flow integer if all its single-commodity flows are integer.*

Lastly, we define the term consistent migration formally. We follow the definition as used in *SWAN* [37]. Due to asynchrony, when (parts of) multiple flows are being simultaneously migrated to other parts of the network, one cannot control in which order the flows migrate: Even if the flows in the network before and after the migration are congestion-free, congestion can occur during migration. Moreover, rate-limiting/dropping is only allowed if it is required by the flow. We refer to Figure 4.1 for further illustration of Definition 11:

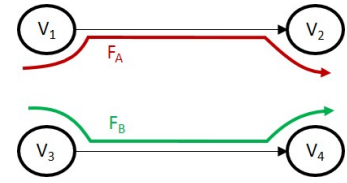
Definition 11. *Let N be a network and let $\mathcal{K}_{all} = (K_1, \ldots, K_k)$ be a multi-commodity in N. Let $\mathcal{F}, \mathcal{F}'$ be multi-commodity flows for $\mathcal{K}, \mathcal{K}' \subseteq \mathcal{K}_{all}$, respectively. We call the tuple $U = (N, \mathcal{F}, \mathcal{F}')$ a* consistent migration update *if the following condition holds:*

$$\forall e \in E : \sum\nolimits_{1 \leq i \leq k} \max\left(F_i(e), F_i'(e)\right) \leq c(e) \tag{4.1}$$

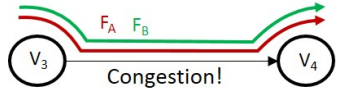
where we assume $F_i(e) = 0$ (respectively, $F_i'(e) = 0$) if $K_i \notin \mathcal{K}$ (respectively, $K_i \notin \mathcal{K}'$). We call a sequence $((N, \mathcal{F}, \mathcal{F}_1), (N, \mathcal{F}_1, \mathcal{F}_2), \ldots, (N, \mathcal{F}_j, \mathcal{F}'))$ of consistent migration updates a consistent migration if for each commodity the demand is monotonically increasing or decreasing over all elements of the sequence $\mathcal{F}, \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_j, \mathcal{F}'$.



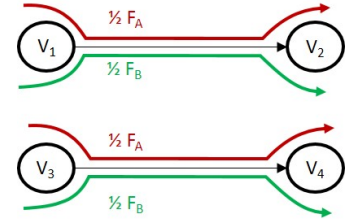
(a) Old/start flow placement



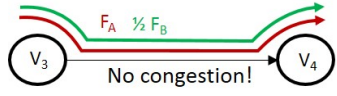
(b) New/desired flow placement



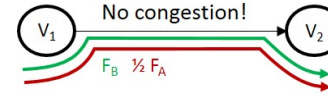
(c) When migrating in one step, the capacity constraints for each edge might be violated, e.g., for (V_3, V_4).



(d) However, with this intermediate placement, one can migrate congestion free in two steps.



(e) If half of F_B migrates first due to asynchrony, then the edge capacity of 3 of (V_3, V_4) is not violated.



(f) The same holds if half of F_A migrates first due to asynchrony, the edge capacity of 3 is not violated.

Figure 4.1: In this example, we want to migrate consistently from Subfigure 4.1a to Subfigure 4.1b. Each edge has a capacity of 3 and the flows F_A, F_B have a size of 2 each. If one migrates in one step, then congestion could occur. E.g., as shown in Subfigure 4.1c, F_B might migrate before F_A, inducing congestion on (V_3, V_4). However, if we add an intermediate step as shown in Subfigure 4.1d, congestion cannot occur, see Subfigure 4.1e and 4.1f.

4.2 Hardness of Unsplittable Flow Migration

We begin studying consistent migration of flows by considering unsplittable flows. How hard is it to decide if it is possible to migrate consistently at all with unsplittable flows?

Problem 8. *Let N be a network and $\mathcal{F}, \mathcal{F}'$ be unsplittable multi-commodity flows in N for multi-commodities $\mathcal{K}, \mathcal{K}' \subseteq \mathcal{K}_{all}$. Is there a consistent migration $(N, \mathcal{F}, \mathcal{F}_1), \ldots, (N, \mathcal{F}_j, \mathcal{F}')$ s.t. all flows from $\mathcal{F}_1$ to $\mathcal{F}_j$ are unsplittable?*

Theorem 10. *Problem 8 is NP-hard.*

We first describe the intuition of our proof via reduction from *3-SAT* [29]. Consider the following setting: Parts of two unsplittable flows F_A, F_B need to be swapped, but the only way to do so is by temporarily storing one on a helper path P. However, that helper path P is blocked by other unsplittable flows, which need to be temporarily stored in other parts of the network as well. Storing these unsplittable flows however requires solving the *3-SAT* problem, as each of them can be stored in up to three variable gadgets (one can think of them as clauses). These variable gadgets can be set to true or false – and if and only if one finds a variable assignment that satisfies each clause, then the path P can be freed up temporarily to allow swapping parts of F_A, F_B. An illustration with a small unsatisfiable instance can be found in Figure 4.2.

Proof. Our proof is a reduction from instances I of the NP-hard problem *3-SAT*. For ease of notation, we assume exactly three pairwise distinct variables per clause [29]. Let I consist of the variables $x_1, \ldots, x_k$, the corresponding literals $X_1, \overline{X}_1, \ldots, X_k, \overline{X}_k$, $k \geq 3$, and clauses $C_1, \ldots, C_h$, $h \geq 2$.

Construction of the new instance I'
We construct an instance I' of Problem 8 as follows: We add the nodes $S_A, S_B, S, A, B, T, T_A, T_B$ and edges $(S_A, S), (S_B, S), (S, A), (S, B), (A, T)$, $(B, T), (T, T_A)$, and (T, T_B). For each clause C_j with the contained variables $x_{j_1}, x_{j_2}, x_{j_3}$ we add the nodes $C_{j,S}, C_{j,T}, C_{j,P,0}, C_{j,P,1}, C_{j,x_{j_1},0}, C_{j,x_{j_1},1}$, $C_{j,x_{j_2},0}, C_{j,x_{j_2},1}, C_{j,x_{j_3},0}, C_{j,x_{j_3},1}$.

Also, the three edges $(C_{j,S}, C_{j,P,0}), (C_{j,P,0}, C_{j,P,1}), (C_{j,P,1}, C_{j,T})$, and the nine edges $(C_{j,S}, C_{j,x_{j_1},0}), (C_{j,x_{j_1},0}, C_{j,x_{j_1},1}), (C_{j,x_{j_1},1}, C_{j,T}), (C_{j,S}, C_{j,x_{j_2},0})$, ... For each variable x_r, appearing in the u clauses $C_{r_{t_1}}, \ldots, C_{r_{t_u}}$ as

a positive literal X_r and in the w clauses $C_{r_{f_1}}, \ldots, C_{r_{f_w}}$ as a negative literal $\overline{X}_r$, we add the nodes $x_{r,S}, x_{r,T}$ and furthermore also the edges $(x_{r,S}, C_{r_{t_1}}, x_r, C_{r_{t_1},x_r,1}, C_{r_{t_2},x_r,0}), \ldots, (C_{r_{t_u},x_r,1}, x_{r,T})$ as part of the *false* path for x_r and, as part of the *true* path for x_r, the edges $(x_{r,S}, C_{r_{f_1},x_r,0})$, $(C_{r_{f_1},x_r,1}, C_{r_{f_2},x_r,0}), \ldots, (C_{r_{f_w},x_r,1}, x_{r,T})$. We also add a path P from S to T by adding the edges $(S, C_{1,P,0}), (C_{1,P,1}, C_{2,P,0}), \ldots, (C_{h,P,1}, T)$. All edges in instance I' have a capacity of exactly one.

We now construct the desired multi-commodity flows $\mathcal{F}, \mathcal{F}'$, both with the commodities (all of them having a demand of one) $(S_A, T_A), (S_B, T_B)$, $(C_{1,S}, C_{1,T}), \ldots, (C_{h,S}, C_{h,T}), (x_{1,S}, x_{1,T}), \ldots, (x_{k,S}, x_{k,T})$.
All flows in $\mathcal{F}, \mathcal{F}'$ are unsplittable and have a size of exactly one. We begin with two flows in $\mathcal{F}$: We start by adding flows F_A from S_A to T_A via A and F_B from S_B to T_B via B. We add nearly the same two flows to $\mathcal{F}'$: Flows F'_A from S_A to T_A via B and F'_B from S_B to T_B via A. We note that the only other option for these flows would be to route them via the nodes $C_{1,P,0}, \ldots, C_{h,P,1}$. The following flows are added both to $\mathcal{F}$ and $\mathcal{F}'$: For each variable x_r, we add a flow F_{x_r} from $x_{r,S}$ to $x_{r,T}$ via their true path. By construction, F_{x_r} can only be routed along the true or false path of x_r. For each clause C_z, we add a flow F_{C_z} from $C_{z,S}$ to $C_{z,T}$ via $C_{z,P,0}$ and $C_{z,P,1}$. By construction, the only other routing options for F_{C_z} would be along a part of a true/false path for one of its contained variables. Consider a clause C_z that contains the variable x_r as a positive literal. Then, the flow F_{C_z} can (partly) use the false path of variable x_r if and only if F_{x_r} is routed along the respective true path. An analogous statement is true for variables that are contained as a negative literal. Thus, the flow of a clause can be (temporarily) routed along a path different from its initial path if and only if there is at least one variable flow which is contained in the clause as a positive literal and routed along the true path, or contained as a negative literal and routed along the false path. In other words, if and only if the variable assignment given by the variable flow routing satisfies the above clause.

Solving any instance of 3-SAT by unsplittable flow migration
What is left to show is that I is satisfiable if and only if we can migrate consistently from $\mathcal{F}$ to $\mathcal{F}'$ in I' with unsplittable flows. Note that the construction of I' can be done in polynomial time. We start by assuming that I is not satisfiable. Then, there is no variable assignment s.t. all clauses

can be satisfied. I.e., for every variable assignment, at least one clause is not satisfied, meaning that no matter if each variable flow F_{x_r} chooses a true or false path, at least one clause flow F_{C_z} has to choose its initial routing path at each moment of a supposed consistent migration. Thus, at each moment the path P is blocked for flows originating from S_A and S_B, meaning that the in/outgoing edges from A and B are always at full capacity. Hence, the flows F_A and F_B cannot migrate consistently to their new placements F'_A and F'_B. Let us now assume that I is satisfiable. Then, there is a variable assignment s.t. every clause is satisfiable. If we migrate the variable flows according to this assignment along their true/false paths, every clause flow can migrate away from the path P. Thus, we can migrate, e.g., F_A to the path P, then migrate F_B to its desired placement F'_B, and then migrate F_A to its desired placement F'_A. Afterwards, the clause flows can migrate to their initial placement, followed by the variable flows. □

We note that the construction only uses capacities and flows of size one. Thus, we can extend Theorem 10 to integer flows:

Problem 9. *Let N be a network and let $\mathcal{F}, \mathcal{F}'$ be integer multi-commodity flows in N. Is there a consistent migration $(N, \mathcal{F}, \mathcal{F}_1), (N, \mathcal{F}_1, \mathcal{F}_2), \ldots, (N, \mathcal{F}_j, \mathcal{F}')$ s.t. all flows from $\mathcal{F}_1$ to $\mathcal{F}_j$ are integer (of unit size)?*

Corollary 6. *Problem 9 is NP-hard.*

4.3 Consistent Migration for Splittable Flows

As we saw in the last section, it is NP-hard to decide if consistent migration is possible if flows have to be integer or unsplittable. Thus, we turn our attention to splittable flows in this section. As we show, this relaxed problem is actually solvable in polynomial time:

Problem 10. *Let N be a network and let $\mathcal{F}, \mathcal{F}'$ be multi-commodity flows for multi-commodities $\mathcal{K}, \mathcal{K}' \subseteq \mathcal{K}_{all}$. Is there a consistent migration from $\mathcal{F}$ to $\mathcal{F}'$?*

Theorem 11. *Problem 10 is in P.*

For better readability, we first present some preliminaries before actually proving Theorem 11. First, consider a special case of the problem where

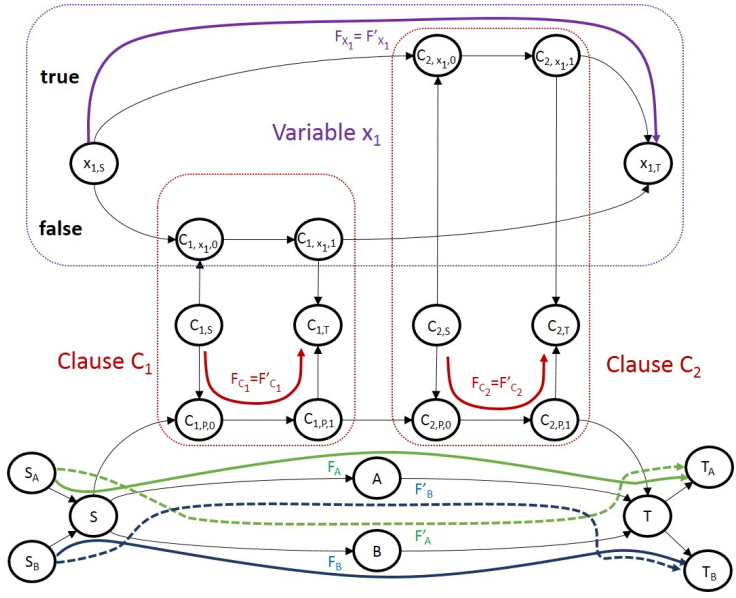


Figure 4.2: An example for $(x_1) \wedge (\neg x_1)$. All edges have a capacity of one and all flows have a size of one. Note that the formula is not satisfiable. For the green flow F_A and the blue flow F_B to swap (parts of) their paths, one of them needs to migrate to the path $S, C_{1,P,0}, C_{1,P,1}, C_{2,P,0}, C_{2,P,1}, T$ above them. This requires temporary migration of the two red flows F_{C_1}, F_{C_2} along the variable gadget, since else the path will not be free. However, the violet flow F_{X_1} will always block one of the temporary migration options of one of the red clause flows, no matter where the violet flow migrates to. Thus, it is not possible to swap the green and the blue flow in this instance.

$\mathcal{K} = \mathcal{K}'$ and the demand of each commodity does not differ between $\mathcal{F}$ and $\mathcal{F}'$. We note that if a commodity only exists in either $\mathcal{K}$ or $\mathcal{K}'$ or has a higher demand in one of the two multi-commodity flows, then one could drop the corresponding excess before migration or insert it afterwards without violating monotonicity.

Problem 11. *Let N be a network and let $\mathcal{F}, \mathcal{F}'$ be multi-commodity flows for the same multi-commodity $\mathcal{K}$ s.t., for each $K \in \mathcal{K}$, the demand of K is the same w.r.t. $\mathcal{F}$ as w.r.t. $\mathcal{F}'$. Is there a consistent migration from $\mathcal{F}$ to $\mathcal{F}'$?*

We now introduce the concept of slack, i.e., an edge is not used at full capacity by a (multi-commodity) flow:

Definition 12. *Let $\mathcal{F} = (F_1, \ldots, F_k)$ be a multi-commo-dity flow in N. We say that an edge e in N has* slack *w.r.t. $\mathcal{F}$ if the following condition holds: $\sum_{1 \leq i \leq k} F_i(e) < c(e)$. If for a consistent migration update $U = (N, \mathcal{F}, \mathcal{F}')$ an edge e has slack w.r.t. $\mathcal{F}'$, but not w.r.t. $\mathcal{F}$, then U* induces *slack on e.*

Also, note that if every edge is used at full capacity (i.e., without slack), then consistent migration is not possible for the above problem if $\mathcal{F} \neq \mathcal{F}'$, as any consistent migration update to a multi-commodity flow $\mathcal{F}^* \neq \mathcal{F}$ would violate the capacity constraint of some full edge:

Observation 1. *Let N be a network and let $\mathcal{F} \neq \mathcal{F}'$ be multi-commodity flows in N for the same multi-commodity $\mathcal{K}$ s.t. for each commodity $K \in \mathcal{K}$ the corresponding flows in $\mathcal{F}, \mathcal{F}'$ have the same size. If every edge is used at full capacity in $\mathcal{F}$ and $\mathcal{F}'$, then consistent migration is not possible.*

On the other hand, if every edge that needs to change its flows has slack, then one can migrate consistently, cf. [37]:

Observation 2. *Let N be a network and let $\mathcal{F}, \mathcal{F}'$ be multi-commodity flows for $\mathcal{K}$ s.t. for each commodity $K \in \mathcal{K}$ the corresponding flows in $\mathcal{F}, \mathcal{F}'$ have the same size. If every edge where $\mathcal{F}$ and $\mathcal{F}'$ differ has slack w.r.t. both $\mathcal{F}$ and $\mathcal{F}'$, then consistent migration is possible.*

This gives rise to the following question: What happens when an edge does not have slack w.r.t. $\mathcal{F}$ or $\mathcal{F}'$, but not all edges are used at full capacity – is consistent migration possible?

The first step to answering this question is to identify the edges which will never admit slack (after any consistent migration) by Algorithm 4. Note that edges which never admit slack do not change their flow assignment in any consistent migration update, this would violate Condition (4.1).

Input: A network N and a multi-commodity flow $\mathcal{F}$ for a multi-commodity $\mathcal{K}$.
Output: A multi-commodity flow $\mathcal{F}^*$ for $\mathcal{K}$ s.t. *a*) for each commodity $K \in \mathcal{K}$ the corresponding flows in $\mathcal{F}, \mathcal{F}^*$ have the same size and *b*) only the edges, which never admit slack (after any consistent migration from $\mathcal{F}$), have slack w.r.t. $\mathcal{F}^*$.

1. Pick an edge $(u, v) = e_1 \in E$ without slack.
2. Pick a commodity $K \in \mathcal{K}$ used on edge e_1. We denote the corresponding flow as F. Let s^* be some positive real number s.t. 1) for all edges which have slack, s^* is smaller than the minimal slack of these edges and 2) for all edges e with $F(e) > 0$, $s^* < F(e)$.
3. Outgoing from the endpoint v of e_1, perform a BFS, where a node v'' is a child-node of a node v', if *a*) there is an edge $e = (v', v'')$ with $F(e) > 0$ or *b*) there is an edge (v'', v') with slack.
4. If the BFS from step 3 visits the node u, then divide the set of edges traversed in the corresponding node sequence $(v, \ldots, u)$ into 1) the set E_K which contains the edges selected by condition *a*) in step 3 and 2) the set E_s which contains the edges selected by condition *b*) from step 3. For all edges in E_K and for e_1, reduce the flow of commodity K by s^*, and for all edges in E_s, increase the flow of commodity K by s^*. Remove any cycling subflows, should they exist.
5. If the BFS from step 3 does not visit the node u, then repeat steps 2 to 4 for the remaining commodities used on the edge e_1, until a BFS from step 3 visits node u or all commodities have been chosen.
6. Repeat steps 1 to 5 for all other edges without slack.
7. Repeat steps 1 to 6 until either all edges have slack or until steps 1 to 6 were performed without inducing any new slack on an edge that had no slack before.

Algorithm 4: Creation of slack

Lemma 5. *Algorithm 4 produces a correct output, i.e., all edges with potential slack after a consistent migration are identified. The runtime is in* $O(|\mathcal{K}||E|^3)$*, i.e., polynomial.*

To prove Lemma 5, we need a lemma which establishes a relation between

the existence of a consistent migration update which induces slack and step 4 of Algorithm 4.

Lemma 6. *Let N be a network and let $\mathcal{F}$ be a multi-commodity flow for $\mathcal{K}$. Let $e = (u, v)$ be an edge without slack w.r.t. $\mathcal{F}$. Then the following are equivalent:*

(*i*) *There exists a consistent migration update $(N, \mathcal{F}, \mathcal{F}')$, where $\mathcal{F}'$ is a multi-commodity flow for the multi-commodity $\mathcal{K}$ s.t. for each commodity $K \in \mathcal{K}$ the corresponding flows in $\mathcal{F}, \mathcal{F}'$ have the same size and e has slack w.r.t. $\mathcal{F}'$.*

(*ii*) *There is a commodity $K \in \mathcal{K}$ with a positive flow of K on e w.r.t. $\mathcal{F}$ and a sequence of nodes $(v = v_1, v_2, \ldots, v_j = u)$ s.t. for each $1 \leq i \leq j - 1$ there is a) an edge (v_i, v_{i+1}) with a positive flow of commodity K w.r.t. $\mathcal{F}$ or b) an edge (v_{i+1}, v_i) with slack w.r.t. $\mathcal{F}$.*

Before proving Lemma 6, we first illustrate it (and the steps 3 and 4 of Algorithm 4) in Figure 4.3.

Proof. We begin by showing that (*i*) implies (*ii*). Let $K \in \mathcal{K}$ be a commodity where for the corresponding flows $F \in \mathcal{F}$ and $F' \in \mathcal{F}'$ holds: $F(e) > F'(e)$. Such a $K = (s, t)$ exists as the consistent migration update $(N, \mathcal{F}, \mathcal{F}')$ induces slack on e. We denote the set of all edges e_K with $F(e_K) > 0$ by E_K (with $e \in E_K$) and the set of all edges e_s with slack w.r.t. $\mathcal{F}$ by E_s. Furthermore, we denote the set of all edges e^- with $F(e^-) > F'(e^-)$ by E^- and the set of all edges e^+ with $F(e^+) < F'(e^+)$ by E^+. Note that all edges in E^- are contained in E_K and all edges in E^+ are contained in E_s.

Let V^e be the set of all nodes v' for which there exists a sequence of nodes $(v, \ldots, v')$ with the properties specified in (*ii*). If $u \in V^e$, then (*ii*) follows. Thus, assume $u \notin V^e$.

Let $E_{\mathtt{in}}$ be the set of all edges (x, y) with $x \notin V^e$ and $y \in V^e$. Let $E_{\mathtt{out}}$ be the set of all edges (x', y') with $x' \in V^e$ and $y' \notin V^e$. For any $(x, y) \in E_{\mathtt{in}}$, it holds that $e_{\mathtt{in}} \notin E_s$, as otherwise $x \in V^e$ by the definition of V^e. This implies $(x, y) \notin E^+$. Analogously, for all edges $(x', y') \in E_{\mathtt{out}}$, it holds that $(x', y') \notin E_k$. This implies $(x', y') \notin E^-$. Set $\Phi_F := \sum_{e_{\mathtt{in}} \in E_{\mathtt{in}}} F(e_{\mathtt{in}}) - \sum_{e_{\mathtt{out}} \in E_{\mathtt{out}}} F(e_{\mathtt{out}})$.

Note that $\Phi_F = \sum_{v^e \in V^e} \left(\sum_{e' \in \mathtt{in}(v^e)} F(e') - \sum_{e' \in \mathtt{out}(v^e)} F(e') \right)$, which implies the following by the defining conditions of a flow: If $s \in V^e, t \notin V^e$,

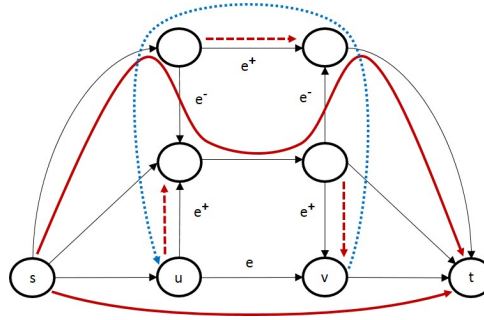


Figure 4.3: In this example, all edges have a capacity of one and the solid red flow of the single commodity has a size of one along each of the two paths. For an unobstructed view, other commodities are left out. To create slack on e, parts of the red flow migrate to the edges denoted with e^+, as shown in the dashed red arrows (see (i) of Lemma 6 and increasing flow in step 4 of Algorithm 4). The dotted blue path from v to u is found via step 3 of Algorithm 4 and corresponds to (ii) of Lemma 6. We note that the edges in e^- and e^+ do not have to alternate in the path from v to u. E.g., there could be also multiple e^+ or e^- edges in a row.

then $\Phi_F = -d_F$. If $s \notin V^e, t \in V^e$, then $\Phi_F = d_F$. Otherwise, $\Phi_F = 0$. We have analogous statements for F' and since $d_F = d_{F'}$, we obtain $\Phi_F = \Phi_{F'}$. On the other hand, $\Phi_{F'} - \Phi_F = \sum_{e_{\text{in}} \in E_{\text{in}}} (F'(e_{\text{in}}) - F(e_{\text{in}}))$ $+ \sum_{e_{\text{out}} \in E_{\text{out}}} (F(e_{\text{out}}) - F'(e_{\text{out}}))$. As shown above, all edges in E_{in} are not in E^+.

Thus, every summand in the first sum is ≤ 0. Analogously, since all edges in E_{out} are not in E^-, every summand in the second sum is ≤ 0. Furthermore, since $e \in E_{\text{in}}$ and $F(e) > F'(e)$, there is a negative summand in the first sum. Hence, $\Phi_{F'} < \Phi_F$, contradicting $\Phi_{F'} = \Phi_F$. Therefore, $u \in V^e$ which shows that (i) implies (ii).

It is left to show that (ii) implies (i). Let s^* be some positive real number s.t. 1) for all edges which have slack w.r.t. $\mathcal{F}$, s^* is smaller than the minimal slack of these edges and 2) for all edges e' with $F(e') > 0$, $s^* < F(e')$. Let E_1 be the union of the set $\{e\}$ and the set of edges described by a) in (ii) and E_2 the set of edges described by b) in (ii).

Now define $\mathcal{F}'$ as follows: For all commodities in $\mathcal{K}$ except K, take the

corresponding flow from $\mathcal{F}$. For K, set

$$F'(\bar{e}) := \begin{cases} F(\bar{e}) - s^* & \text{if } \bar{e} \in E_1, \\ F(\bar{e}) + s^* & \text{if } \bar{e} \in E_2, \\ F(\bar{e}) & \text{otherwise.} \end{cases} \tag{4.2}$$

By the definition of s^*, no capacity constraints will be violated by $\mathcal{F}'$. For all nodes which are not in the sequence given in (ii), the flow conservation condition holds w.r.t. $\mathcal{F}'$, as no ingoing or outgoing flows have been changed. For each node which is in the sequence given in (ii), the (two) incident edges with changed flow F' (compared to F) cancel each other out regarding flow conservation. If the nodes s and t are not in the sequence given in (ii), then $d_{F'} = d_F$ holds. W.l.o.g., let s be in the sequence given in (ii): Then there could be a cycle of commodity K but increasing the outgoing flow of commodity K for s. In that case, we transform F into a cycle-free flow by subtracting the cycling "subflows" from the affected edges. This does not violate flow conservation or capacity constraints and ensures $d_{F'} = d_F$.

Since we only change the flow of one commodity from $\mathcal{F}$ to $\mathcal{F}'$, Condition 4.1 is satisfied because no capacity constraints are violated by $\mathcal{F}'$ as shown before. Thus, $(N, \mathcal{F}, \mathcal{F}')$ is a consistent migration update. As $e \in E_1$, $(N, \mathcal{F}, \mathcal{F}')$ induces slack on e, which shows that (ii) implies (i). □

Now we are able to prove Lemma 5:

Proof. We begin with the observation that once we have slack on an edge, we never need to remove the slack on this edge completely in order to induce slack on other edges. Assume that there is a consistent migration update that induces slack on edge e_1 and removes slack on edge e_2. If we change the migration update to just migrate half the size of the migrated flows, we will still have slack on both edge e_1 and edge e_2. This is essential to the proof, as it cannot be done with unsplittable or integer flows. Furthermore, it shows that the size of the slack is not relevant for any further steps of an (appropriately designed) algorithm that tries to induce slack on as many edges as possible. As we consider splittable flows, for any consistent migration update, one can just reduce the size of the moved flow s.t. it still leaves slack on the newly used edges, but still induces slack on the previously more heavily used edges. Thus, we only need to differentiate for each edge whether i) it has slack, or ii) it has no slack.

Recall from Definition 11 that a consistent migration update is only possible if there is some slack on the edges where the flow is being moved to. Else Condition (4.1) will be violated. Consider any step 4 of Algorithm 4, where the BFS from step 3 visits the node u. The flow transformations being performed subsequently correspond to the construction of F' in the proof of Lemma 6 (given by (4.2)). Thus, by Lemma 6, all flow transformations performed by Algorithm 4 are consistent migration updates.

What is left to show (apart from the polynomial runtime) is that Algorithm 4 induces slack on each edge on which slack can be induced by any consistent migration. Thus, assume, for the sake of contradiction, that there is a consistent migration M after which there is slack on some edge e_x which had no slack initially, but Algorithm 4 does not induce slack on e_x.

Let E_i be the set of all edges which had no slack initially, on which M induces slack in update i. Let j be the smallest index s.t. E_j contains an edge e_j on which Algorithm 4 does not induce slack. Then, Algorithm 4 induces slack on all edges from $E_1 \cup E_2 \cup \cdots \cup E_{j-1}$. Let $K \in \mathcal{K}$ be a commodity for which the corresponding flow size on edge e_j gets reduced in the j-th update of M.

Let $\mathcal{F}$ be the initially given flow and $\mathcal{F}'$ the flow after the j-th update of M. Let $\mathcal{F}^*$ be the flow obtained by running Algorithm 4 on $\mathcal{F}$. Let $F \in \mathcal{F}$, $F' \in \mathcal{F}'$ and $F^* \in \mathcal{F}^*$ be the flows corresponding to commodity $K = (s, t)$. Let E^- denote the set of edges e with $F'(e) < F(e)$ and E^+ the set of edges e' with $F'(e') > F(e')$. Every edge $e \in E^+$ must have had slack at some point before the j-th update of M as some flow for commodity K has been added during the migration from $\mathcal{F}$ to $\mathcal{F}'$. By the definition of j, this implies that e has slack also w.r.t. $\mathcal{F}^*$. Furthermore, for each edge $e' \in E^-$, we have $F(e') > 0$, and since the design of Algorithm 4 ensures that any edge containing some flow for some commodity always keeps some positive flow for this commodity, we obtain $F^*(e') > 0$. Now define a flow $\overline{\mathcal{F}}$ for $\mathcal{K}$ by setting

$$\overline{F}(e) := \begin{cases} F^*(e) - r(F(e) - F'(e)) & \text{if } e \in E^-, \\ F^*(e) + r(F'(e) - F(e)) & \text{if } e \in E^+, \\ F^*(e) & \text{otherwise,} \end{cases} \tag{4.3}$$

for the flow $\overline{F}$ for commodity K and taking the flows from $\mathcal{F}^*$ for all other commodities in $\mathcal{K}$. The parameter $r \in \mathbb{R}_+$ in the definition of $\overline{F}$ will

be determined in the following. Note that the first two terms in the above definition are equal.

We have to show that $\overline{F}$ is indeed a flow (and $\overline{\mathcal{F}}$ indeed a multi-commodity flow). As $\overline{F}$ is a linear combination of the flows F^*, $F(e)$ and $F'(e)$ (which all satisfy the flow conservation condition on the nodes different from s and t), it also satisfies the flow conversation condition. Furthermore, as F and F' cancel each other out regarding the flow (size) outgoing from s and incoming in t, $\overline{F}$ satisfies also the second flow condition and the demand of K w.r.t. F is also the same as w.r.t. F^*, F and F'. By choosing r small enough, we can also ensure that the third flow condition is satisfied (in the general form required for multi-commodity flows, also given in the model section), i.e., that $\overline{\mathcal{F}}$ does not violate the capacity constraints. (This last claim follows from the fact that the only edges e with $\overline{F}(e) > F^*(e)$ are those in E^+ and we already showed that all of these edges have slack w.r.t. F^*.)

In order to avoid having "negative" flows on some edges, we must take care that for the edges in E^- (for which $F'(e) < F(e)$ holds), $F^*(e) - r(F(e) - F'(e))$ is positive. As we showed above, we have $F^*(e) > 0$ for all edges $e \in E^-$. Thus, we can ensure the required positivity by again choosing r small enough. Now fix r s.t. it is small enough for the arguments in the above discussion. Then $\overline{F}$ is a flow for commodity K and $\overline{\mathcal{F}}$ is a multi-commodity flow for $\mathcal{K}$. Furthermore, for all commodities in $\mathcal{K}$ different from K, the corresponding flows are identical in $\overline{\mathcal{F}}$ and $\mathcal{F}^*$. Thus, $(N, \mathcal{F}^*, \overline{\mathcal{F}})$ is a consistent migration update as Condition 4.1 must be satisfied (since both $\overline{\mathcal{F}}$ and $\mathcal{F}^*$ do not violate the capacity constraints of the edges). Moreover, for each commodity in $\mathcal{K}$, the corresponding flows in $\overline{\mathcal{F}}$ and $\mathcal{F}^*$ have the same size.

Consider the edge e_j. By its definition, we have $F'(e_j) < F(e_j)$ which implies $e_j \in E^-$. Thus, $\overline{F}(e_j) < F^*(e_j)$ and e_j has slack w.r.t. $\overline{F}$. So we have shown that $(N, \mathcal{F}^*, \overline{\mathcal{F}})$ is a consistent migration update as described in statement (i) of Lemma 6 whereas e_j is an edge without slack w.r.t. $\mathcal{F}^*$. By applying Lemma 6, we obtain the corresponding statement (ii) from Lemma 6 (where "e" $= e_j$ and "$\mathcal{F}$" $= \mathcal{F}^*$). It follows that after reaching flow $\mathcal{F}^*$, Algorithm 4 will pick the edge e_j in some step 1 and the commodity K in step 2. In the subsequent step 4, Algorithm 4 will find a node sequence which ends in the starting node of e_j (the existence of such a sequence is ensured by the above statement (ii) from Lemma 6). Thus, Algorithm 4 will perform a consistent migration update which induces slack on the edge

e_j. This is a contradiction to the assumption and Algorithm 4 produces a correct output.

It is left to show that Algorithm 4 runs in polynomial time: For the analysis, we first ignore the runtime contributed by step 4. Steps 1 to 5, excluding 4, can be performed in $O(|\mathcal{K}||E|)$ time – with step 6 iterating this process $O(|E|)$ times. Step 7 will repeat steps 1 to 6, excluding 4, again $O(|E|)$ times, resulting in a total runtime of $O(|\mathcal{K}||E|^3)$. Observe that step 4 will be executed at most $O(|E|)$ times, as it is only run when slack is generated for the first time on an edge. Increasing and decreasing the flows in step 4 can be performed in $O(|E|)$ time, leaving the cycle removal: By selecting an edge e with smallest flow size $F_K(e)$ of commodity K, we can determine if there is a cycle for the commodity K containing e, and if this is the case, remove such a cycle, setting $F_K(e) = 0$ in the process. Such a cycle removal with a runtime of $O(|E|)$ needs to be performed at most $|E|$ times, yielding a total runtime of $O(|E|^3)$ for all executions of step 4. Hence, the total runtime of Algorithm 4 is $O(|\mathcal{K}||E|^3)$. □

Furthermore, Problem 10 can be reduced to Problem 11 by essentially *i*) first dropping (parts of) commodities that do not need to migrate, and *ii*) adding new (parts of) commodities at the end. Then, Algorithm 4 can be applied to both $\mathcal{F}$ and $\mathcal{F}'$. Should there be edges in N, on which *i*) no slack can be induced starting from $\mathcal{F}$ or $\mathcal{F}'$ by means of consistent migration, and that *ii*) differ in their flow assignment in $\mathcal{F}$ and $\mathcal{F}'$, then consistent migration is not possible. Else, one can use the approach of *a*) applying Algorithm 4 to the old and new placement to ensure slack s on each edge and then migrate in at most $\lceil 1/s \rceil - 1$ consistent migration updates, or *b*) use the method of binary search via LPs from [37] to find a consistent migration. The formal proof of Theorem 11 is given in the following.

Proof of Theorem 11. We start to address this problem by first recalling Observation 1: I.e., if all edges are at full capacity, the commodities and demands stay the same, and some flows have to change their placement in the network, any consistent migration step would violate the capacity constraint of some edge. Furthermore, assume that a commodity only exists in the initial starting state, i.e., in $\mathcal{K}$, but not in $\mathcal{K}'$. Then, one can just remove the corresponding flow/commodity and consider this reduced problem. The same holds if the commodity just exists in the desired state, i.e., in $\mathcal{K}'$, but

not in $\mathcal{K}$. One can ignore this commodity when migrating – and just add its flow at the end, as there will be enough space on all of its used edges. Similarly, if there is some slack on each edge for both $\mathcal{F}$ and $\mathcal{F}'$, then it is possible to migrate consistently, see Observation 2.

Thus, we want to identify the edges whose usage by flows cannot be changed by consistent migration. We refer to Subfigure 3.2d for an example: As all of the edges are used to full capacity in both the old starting state with $\mathcal{F}$ and the new desired state with $\mathcal{F}'$, the flow assignment of any edge cannot be changed, unless one violates congestion-freedom or rate-limits some flow. Note that a necessary requirement for such an edge is that it is used at full capacity in both the old starting state $\mathcal{F}$ and the new desired state $\mathcal{F}'$.

Hence, when a commodity with the same source and sink has different demands $d_{F_{old}}, d_{F_{new}}$ in the old and new state with flows F_{old}, F_{new}, we only need to look at the minimum of $d_{F_{old}}, d_{F_{new}}$: Assume $d_{F_{old}} < d_{F_{new}}$ with $d_{F_{old}} = y * d_{F_{new}}$, $y > 1$. Then, we can reduce the flow of F_{new} by a factor of y on all of its used edges, and after a consistent migration, add the missing demand over all edges. The same holds if $d_{F_{old}} > d_{F_{new}}$ with $y * d_{F_{old}} = d_{F_{new}}$, $y > 1$: We reduce the flow of F_{old} by a factor of y on all of its used edges, and then consider consistent migration. In both cases, we will only lower the used capacity of edges by consistent migration steps. Therefore, we can simplify our original problem to one where the demands and commodities are equal for $\mathcal{F}$ and $\mathcal{F}'$, i.e., $\mathcal{K} = \mathcal{K}' = \mathcal{K}_{all}$. Note that the problem is symmetric in the sense that consistent migration from $\mathcal{F}$ to $\mathcal{F}'$ is possible if and only if consistent migration from $\mathcal{F}'$ to $\mathcal{F}$ is possible.

This means that if there is an edge whose usage by flow $\mathcal{F}$ or $\mathcal{F}'$ cannot be changed by any sequence of consistent migration steps, and it is not used in the same fashion by $\mathcal{F}$ and $\mathcal{F}'$, then consistent migration from $\mathcal{F}$ to $\mathcal{F}'$ is not possible! Conversely, if no such edge exists, we can migrate consistently: If each edge that needs to be changed in its usage of flows has some slack, we can migrate consistently by always changing a small part of the network in accordance with the slack. I.e., if the minimum slack is 10%, we can migrate in 9 steps.

Therefore, we consider all edges that are *i*) at full capacity without slack for $\mathcal{F}$ or $\mathcal{F}'$, and *ii*) not used the same by $\mathcal{F}$ or $\mathcal{F}'$, and try to figure out if we can find some sequence of consistent migration steps that induces slack on them. Thus, we only need to look at an even more restricted problem:

Let N be a network and $\mathcal{F}^*$ be a flow in N with commodities $\mathcal{K}^*$. On what edges can we induce slack by a sequence of consistent migration steps?

However, this is exactly the problem solved by Algorithm 4 in polynomial time. As all steps mentioned before can be performed in polynomial time as well, Problem 10 is in the complexity class P. □

4.4 Insertion of Unsplittable Flows

A natural method to insert/increase a flow in a network is to check first if there is enough space. In that case, the solution is straightforward – one just increases/adds the flow in question. However, things get more difficult if there is currently not enough capacity. Is it necessary to remove some flows, or is it enough to consistently migrate the existing flows?

A common method (cf. *RSVP-TE* [5]) is to assign levels of importance to all flows in the network, and then remove those of lesser importance if they block the new flow. We show that from a theoretical standpoint, this approach is problematic for unsplittable flows, as we prove the following corresponding decision problem to be NP-hard:

Problem 12. *Let N be a network and let $\mathcal{F}$ be an unsplittable multi-commodity flow in N for a multi-commodity $\mathcal{K}$. Let K_{new} be a commodity not contained in $\mathcal{K}$ and let M be a map of each commodity in $\mathcal{K}$ to a level of importance, i.e., $M : \mathcal{K} \to \mathbb{N}$. Let r be some integer. Is there a set $\mathcal{K}_r \subseteq \mathcal{K}$ of commodities with summed up importance at most r, an unsplittable multi-commodity flow $\mathcal{F}'$ in N for the multi-commodity $\{\mathcal{K} \setminus \mathcal{K}_r\} \cup K_{new}$, and a consistent migration $(N, \mathcal{F}, \mathcal{F}_1), (N, \mathcal{F}_1, \mathcal{F}_2), \ldots, (N, \mathcal{F}_j, \mathcal{F}')$ s.t. all flows $\mathcal{F}_i$ are unsplittable?*

Theorem 12. *Problem 12 is NP-hard.*

Proof. The proof follows directly from the proof of Theorem 10 if one considers the following case: Importance $r = 0$, $\mathcal{F}$ and $\mathcal{K}$ as given in the proof of Theorem 10, and K_{new} with source S and sink T, cf. Figure 4.2. □

This implies that the corresponding optimization version of Problem 12 can also not be approximated well: Any (constant) approximation ratio for r would mean that one could decide if the problem is satisfiable or not.

Corollary 7. *The optimization version of Problem 12, i.e, minimizing r, does not admit a PTAS unless $P = NP$.*

We can take this problem even one step further and ask about the hardness of approximation regarding the additive error. Maybe one could always migrate in the construction from our proof if one just removes just a constant amount of the flows of the lowest importance? If we assign the clause flows the importance 1 and the remaining flows the importance # *of clauses*, then this problem reduces to finding a variable assignment that satisfies as many clauses as possible – i.e., solving *MAX 3-SAT*. However, as shown by Håstad [34], this is NP-hard to approximate better than $7/8 + \epsilon$.

4.5 Increasing Splittable Flows

As shown in Section 4.4, the consistent increase/insertion of flows is an NP-hard problem if flows are unsplittable. This leads to the natural question of *how much* the demand of a commodity can be increased under the condition of consistency and splittable flows. Then, one can make an informed decision if it is worth it to violate consistency or not: Maybe the new demand is for a critical application that absolutely needs the bandwidth – or maybe it is just some background data that is not time critical.

In Section 4.3 we showed that it is decidable in polynomial time if consistent migration is possible. However, this does not answer the question of to what unknown new desired flow placement one should migrate. One could just solve an LP maximizing the demand of one commodity while keeping the demand of the other commodities fixed (cf. the LP in Figure 4.4 without restriction 4)). But it can be the case that consistent migration is not possible for the resulting multi-commodity flow of the LP, cf. Figure 4.5. We thus formulate the following problem:

Problem 13. *Let N be a network and let $\mathcal{F}$ be a multi-commodity flow in N for the multi-commodity $\mathcal{K}$. Let $K \in \mathcal{K}$. Find a multi-commodity flow $\mathcal{F}'$ for the multi-commodity $\mathcal{K}$ that i) maximizes the demand of commodity K, ii) leaves the demand of all other commodities unchanged, and iii) can be migrated to consistently from $\mathcal{F}$.*

For ease of notation, we assume that if one wants to maximize a new commodity, it is denoted as $K \in \mathcal{K}$ with a current demand of 0. As it turns

Maximize $\sum_{i:e_i \in \texttt{out}(s_1)} x_{i1}$
subject to

1. $\forall 1 \leq j \leq k \forall v \in V \setminus \{s_j, t_j\}$:
   $\sum_{i:e_i \in \texttt{out}(v)} x_{ij} = \sum_{i:e_i \in \texttt{in}(v)} x_{ij}$,
2. $\forall 2 \leq j \leq k : \sum_{i:e_i \in \texttt{out}(s_j)} x_{ij} = d_j = \sum_{i:e_i \in \texttt{in}(t_j)} x_{ij}$,
3. $\forall 1 \leq j \leq k : \sum_{i=1}^{k} x_{ij} \leq c(e_j)$,
4. $\forall 1 \leq i \leq m$ s.t. $e_i \in E_{\texttt{fix}} \forall 1 \leq j \leq k : x_{ij} = F_j(e_i)$,
5. $\sum_{i:e_i \in \texttt{in}(s_1)} x_{i1} = 0$.

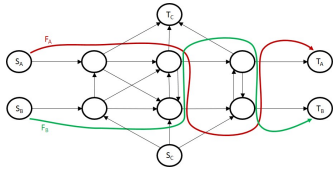
Figure 4.4: The LP doesn't alter the flow on the edges from $E_{\texttt{fix}}$ due to 4).

out, we can approximate the maximum consistent increase of the demand of commodity K in Problem 13 arbitrarily well:

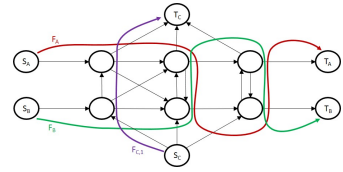
Theorem 13. *Let $\epsilon > 0$. Finding a multi-commodity flow for Problem 13 that satisfies the conditions ii) and iii) and approximates the maximum demand of commodity K in condition i) with an approximation ratio of $(1 - \epsilon)$ can be done in polynomial time (independent of the chosen ϵ).*

Proof. Let $\mathcal{F} = (F_1, ..., F_k)$ be the given (initial) multi-commodity flow for the multi-commodity $(K_1, ..., K_k)$ where commodity $K_j = (s_j, t_j)$ with demand d_j w.r.t. $\mathcal{F}$. Let $K = K_1$. Using Algorithm 4, we can determine all the edges which do not admit slack after any consistent migration starting from $\mathcal{F}$. Let $E_{\texttt{fix}}$ denote the set of these edges. Due to Condition (4.1), consistent migration updates cannot change the flow assignments on $E_{\texttt{fix}}$, so we would like to fix them for the (approximate) maximum flow we are looking for.

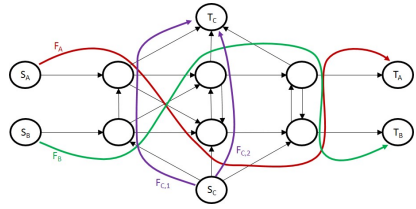
By solving the LP given in Figure 4.4, we can find a multi-commodity flow $\mathcal{F}^*$ that maximizes the demand of commodity K. Any found optimal solution of the LP represents a multi-commodity flow $\mathcal{F}^*$ with the above-mentioned properties by setting $F_j(e_i) := x_{ij}$, $1 \leq i \leq m$, $1 \leq j \leq k$. The first three constraints represent the usual flow prerequisites. The fourth constraint guarantees that nothing changes (from $\mathcal{F}$ to $\mathcal{F}^*$) on the edges in $E_{\texttt{fix}}$. As the LP does not check if the solution(s) represent cycle-free flows,



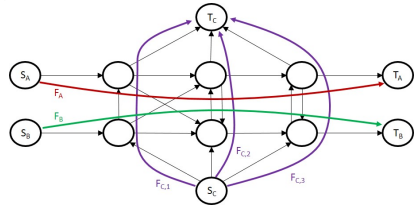
(a) Old/start flow placement



(b) The flow $F_{C,1}$ can be directly inserted. All other paths are currently blocked by F_A or F_B.



(c) We can migrate F_A and F_B consistently to their shown positions, allowing the insertion of flow $F_{C,2}$.



(d) If one does not care about consistent migration, then one could re-arrange the flows as shown in this subfigure and also insert the flow $F_{C,3}$.

Figure 4.5: In this example, we want to maximize the flow from S_C to T_C, with the flows F_A and F_B currently in the network as shown in Subfigure 4.5a. All edges have a capacity of one and the flows F_A and F_B have a size of one as well. Under the restriction of consistent migration, it is possible to increase the size of the flows from S_C to T_C to two, but not more – as the flows F_A and F_B block each other's consistent migration in the right part of the network.

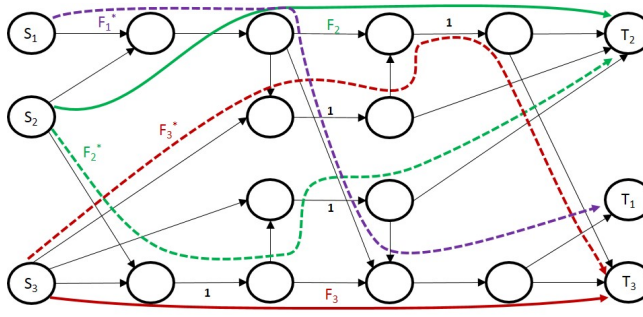


Figure 4.6: In this network, all edges have a capacity of two – except for the ones denoted with a capacity of 1. The commodities K_2 and K_3 have a demand of one. In the initial flow $\mathcal{F}$, consisting of F_2 and F_3, one can achieve slack on all edges by rerouting some parts of F_2 and F_3 along alternate paths. In the optimal solution $\mathcal{F}^* = (F_1^*, F_2^*, F_3^*)$ of the LP from Figure 4.4 that maximizes the demand for K_1 (which is then two), the flows F_2 and F_3 of size one have to be rerouted along the paths denoted by F_2^* and F_3^*. However, by means of consistent migration it is not possible to induce slack on the 1-edges starting from (F_2^*, F_3^*), even if F_1^* is not inserted yet: F_2^* would need to use at least one of the 1-edges fully occupied by F_3^* and vice versa. Thus, it is not possible to consistently migrate between $\mathcal{F}^*$ and $\mathcal{F}$.

we need the last constraint: It ensures that no part of the flow leaving s_j returns to s_j (such a part would not contribute to the flow F_j from s_j to t_j, but to the sum we are trying to maximize). If we obtain a solution to the LP with a cycle, then we can transform it into a cycle-free flow by subtracting the cycling "subflows" from the affected edges. Thus, we may still assume cycle-freedom.

While it is not ensured that it is possible to migrate consistently from $\mathcal{F}$ to $\mathcal{F}^*$, the obtained maximized demand $d' := \sum_{i:e_i \in \mathtt{out}(s_1)} x_{i1}$ gives us an upper bound for the demand of commodity K w.r.t. flow $\mathcal{F}^*$, subject to the condition that all other demands remain as they are. If we demand consistent migration, then the demand d' cannot necessarily be achieved, but we come arbitrarily close. We refer to Figure 4.6 for an example where d' cannot be achieved.

However, the initial flow $\mathcal{F}$ and an arbitrary optimal solution $\mathcal{F}^*$ of the LP can be combined to a multi-commodity flow $\mathcal{F}'$ which can be migrated

to consistently from $\mathcal{F}$. The combination is parametrized by some $0 < r < 1$ which determines the demand of the commodity whose demand we are trying to maximize, and thus also determines the approximation ratio. We define $\mathcal{F}'$ by $F'_j(e) := rF^*_j(e) + (1-r)F_j(e)$ for all $1 \leq j \leq k$ and $e \in E$. It follows directly from the definition that $\mathcal{F}'$ is a multi-commodity flow for which the demands $K_2, ..., K_k$ did not change from $\mathcal{F}$. What is left to show is that we can migrate consistently from $\mathcal{F}$ to $\mathcal{F}'$. Recall that starting from $\mathcal{F}$, slack can be achieved on exactly the edges which are not in $E_{\texttt{fix}}$. We show that the same is true for $\mathcal{F}'$:

We can "divide" the given network N into two networks N_1 and N_2 which have the same underlying graph as N but have less capacity – for N_1 each edge capacity in N is multiplied by r, for N_2 by $(1-r)$. We can imagine the flow $r\mathcal{F}^*$ as living only on the N_1 part of N and $(1-r)\mathcal{F}$ as living only on the N_2 part. Now any consistent migration of $\mathcal{F}$ in N represents a consistent migration of $(1-r)\mathcal{F}$ in N_2 and thus a consistent migration of $\mathcal{F}'$ in N. It follows that slack (starting from $\mathcal{F}'$) can be achieved on all edges except possibly on those in $E_{\texttt{fix}}$. We will show now that slack cannot be induced on any $e \in E_{\texttt{fix}}$ starting from $\mathcal{F}'$. By the above discussion, there is a consistent migration from $\mathcal{F}'$ to a flow $\overline{\mathcal{F}}$ (for the same multi-commodity) for which exactly the edges not in $E_{\texttt{fix}}$ have slack (just apply only updates which do not affect N_1).

As discussed in the proof of Lemma 5, the result of Algorithm 4 in terms of which edges have slack, does not depend on any slack size, but just on whether an edge has slack or not. The set of edges which have slack is the same w.r.t. $\overline{\mathcal{F}}$ as w.r.t. the flow Algorithm 4 returns after running on $\mathcal{F}$ (namely, $E \backslash E_{\texttt{fix}}$). Thus, running Algorithm 4 (again) for those two flows will yield the same set of edges with slack. As the latter has already achieved slack on all edges where this is possible, slack cannot be induced on any $e \in E_{\texttt{fix}}$, starting from $\overline{\mathcal{F}}$. Thus, the same is true for $\mathcal{F}'$.

So slack can be achieved on the edges not in $E_{\texttt{fix}}$, both starting from $\mathcal{F}$ and $\mathcal{F}'$. Furthermore, as $\mathcal{F}^*$ is a solution to the above LP, we have $F^*_j(e) = F_j(e)$ for all $1 \leq j \leq k$, $e \in E_{\texttt{fix}}$. Thus, the flow assignment on the edges in $E_{\texttt{fix}}$ is the same for $\mathcal{F}$ and $\mathcal{F}'$. Ignoring these edges (whose flow assignments do not need to be changed), we can use the technique provided by [37] to obtain a consistent migration between $\mathcal{F}$ and $\mathcal{F}'$ as all edges that need to change their flow assignments have slack w.r.t both $\mathcal{F}$ and $\mathcal{F}'$ (after applying Algorithm 4). Hence, we obtain that there is a consistent

migration between the multi-commodity flows $\mathcal{F}$ and $\mathcal{F}'$.

It is left to show that the runtime is polynomial (independent of the chosen ϵ): Solving a linear program as the LP in Figure 4.4 can be done in polynomial runtime (e.g., with the *interior point method*). Furthermore, Algorithm 4 has a polynomial runtime as well, see Lemma 5. ϵ comes only into play when "dividing" the network into two networks – but there it is only used for multiplication to achieve the desired approximation ratio for the demand of commodity $K = K_1$. □

4.6 Summary

We studied the problem of migrating flows consistently, i.e., without violating the consistency model of *SWAN* [37] or rate-limiting. We were able to show that for splittable flows, it is possible to decide in polynomial time if consistent migration is possible. All previous approaches could not decide if consistent migration is possible or applied only to specific subsets of the consistent migration problem. We proved that splittable flows are essential for this result, as the decision problem is NP-hard for unsplittable or integer (even unit size) flows. Furthermore, we also studied the problem of consistently increasing or inserting new flows into the network. A current practice is to drop obstructing flows: We showed that optimizing this technique is NP-hard to as well. However, we proved that one can approximate the maximal consistent increase for the size of a (new) flow arbitrarily well in polynomial time.

5

Non-Mixing Flow Migration

In this chapter, we extend our study on consistent flow migration to deal with waypointing and service chain requirements for unsplittable flows, a property we call non-mixing consistency. We first show that this problem is NP-hard as well for unsplittable flows in Section 5.3: Due to our proof construction, the NP-hardness also follows for consistent flow migration without intermediate paths. Subsequently, we relax the unsplittable requirement, and consider two-splittable flows in Section 5.4: Akin to the previous chapter, we can give a completely polynomial classification and schedule generation as well. Our technique is designed with easy deployment in mind: Beyond establishing flows rules for the old and new paths, all network updates just have to change the flow splitting ratios at the sources.

5.1 Motivation

Network flow routes often have to adhere to waypointing, or even service chaining (e.g., firewalls, caches, etc.) [54]. As thus, it is sometimes a requirement that an unsplittable flow should only be routed along its old path or its new path, but not a mix of these two, and especially not along totally different paths. This is easy to guarantee if the old and the new flow path are node-disjoint, but not so much when the old and the new path mix: When old and new packets arrive at a switch, marked as being from the same flow, forwarding them according to either or the old rule will lead to the violation of waypointing or the traversal of network functions in the wrong order.

Therefore, we extend the packet stamping method from [68] in the context of congestion-avoidance[1]: We introduce flow rules F_i^{old} for the old path and F_i^{new} for the new path, allowing each packet to respect the service chains.

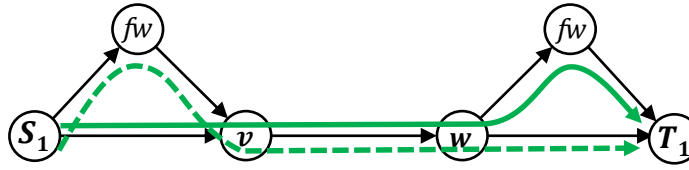


Figure 5.1: In this network, the old flow path is drawn solid, the new dotted. Assume that the flow would be split along the old and the new path, and that there would be firewalls on the nodes marked with *fw*. If there is just one flow rule at w, namely to split the flow along the two outgoing edges, then there are flow packets not traversing any firewall. As thus, we introduce separate *old* and *new* flow rules for the non-mixing property.

5.2 Model

We model a network N as a directed graph $G = (V, E)$ with non-negative edge-capacities c. An (unsplittable) flow F_j of size d_j starts at a source

[1] Reitblatt et al. [68] use them to guarantee the non-mixing property (which they call per-flow/per-packet consistency), but do not consider congestion.

$S_j \in V$ and is routed along a cycle-free path P_j of enough capacity to its destination $T_j \in V$, i.e., $\forall e \in P_j : d_j \leq c(e)$. We call a set of k flows $F_1, \ldots, F_k$ a multi-commodity flow $\mathcal{F}$ if $\forall e \in E : \sum_{i=1}^{k} F_i(e) \leq c(e)$, i.e., their combined sizes do not violate any capacity constraints. For the sake of simplicity, we assume that the old flow size d_j on P_j is identical to its new size d'_j on $P'_j \neq P_j$, else one could, e.g., reduce the flow size to d'_j on P_j before updating, or migrate and then increase on P'_j.

Inspired from a combination of [37] and [68], cf. Figure 5.1, we now define the concept of a non-mixing consistent migration update:

Definition 13 (Non-Mixing Consistent Network Update)**.** *A network update of unsplittable flows is called* non-mixing consistent*, if the following condition holds:*

$$\forall e \in E : \sum_{i=1}^{k} \max\left(F_i^{old}(e) + F_i'^{old}(e)\right) + \max\left(F_i^{new}(e) + F_i'^{new}(e)\right) \leq c(e) \ . \tag{5.1}$$

Observe that the SWAN [37] consistency model, cf. Definition 11, and non-mixing consistency are identical if the old and new flow paths are disjoint, the only difference is in the case when the old and the new path are joint at some point.

Consistent Non-Mixing Flow Migration So far we described flow updates of one round in Condition 5.1, but due to dependencies it can easily be the case that one needs to apply various updates before the desired outcome is reached, cf. [42]. E.g., as already described in earlier chapters, F_1 wants to move to the path of F_2, but before that can happen, F_2 first needs to be moved to its new path.

Definition 14 (Non-Mixing Migration)**.** *Let $\mathcal{F}$ be the old flow assignment and $\mathcal{F}'$ be the new flow assignment, with identical demands. We call a series of non-mixing consistent flow updates* a non-mixing migration *from $\mathcal{F}$ to $\mathcal{F}'$, if the following two conditions are met:* 1) *each flow packet may only use the old or the new flow path,* 2) *the final flow assignment is $\mathcal{F}'$.*

5.3 Hardness of Unsplittable Flow Migration

While it is well studied if a set of demands can be met by unsplittable flows, what about the case of migrating between two known flow assignments? We know that if the unsplittable flows are allowed to take any path, this problem is NP-hard, see Section 4.2. We now consider the case where the unsplittable flows are just allowed to take either the old or the new path:

Theorem 14. *Let $\mathcal{F}$, $\mathcal{F}'$ be unsplittable multi-commodity flows. Deciding if there is a non-mixing consistent flow migration from $\mathcal{F}$ to $\mathcal{F}'$ is NP-hard.*

Our proof will be a reduction from the NP-hard problem Partition [29]:

Definition 15 (*Partition* [29])**.** *Let $\mathcal{A}$ be a multiset of k positive real-valued elements $a_1, \ldots, a_k$, and set $A := \sum_{i=1}^{k} a_i$. Is it possible to partition $\mathcal{A}$ into two sets $\mathcal{A}_1, \mathcal{A}_2$ s.t. the sums $A_1 := \sum_{a_i \in \mathcal{A}_1} a_i$, $A_2 := \sum_{a_i \in \mathcal{A}_2} a_i$ of their respective elements are identical, i.e., $A_1 = A_2 = \frac{A}{2}$?*

Theorem 15 ([29])**.** *The* Partition *problem from Definition 15 is NP-hard.*

Proof. For each instance I of Partition, we will create an instance I' of the migration problem s.t. I is a *yes*-instance if and only if I' is a *yes*-instance.

Construction of the new Instance I'
The network N in the instance I' consists of the nodes $S_1, \ldots, S_k$, S_b, $T_1, \ldots, T_k$, T_b, v_{11}, v_{12}, v_{21}, and v_{22}. There is a directed edge with capacity A from each $S_1, \ldots, S_k$ to v_{11} and v_{21}, and from each v_{12}, v_{22} to $T_1, \ldots, T_k$. Furthermore, there is a directed edge from v_{11} to v_{12} and from v_{21} to v_{22}, each with capacity A. Lastly, we have edges with capacity A from S_b to v_{11} and v_{21}, and from v_{12} and v_{22} to T_b. We refer to Figure 5.2 for an illustration.

The old flow configuration $\mathcal{F}$ is given as follows. For $1 \leq i \leq k$, there is a flow F_i of size a_i from S_i to T_i along the path S_i, v_{11}, v_{12}, T_i. Also, there is a flow F_b of size $A/2$ from S_b to T_b via S_b, v_{21}, v_{22}, T_b In the new flow configuration $\mathcal{F}'$, the flows F_i instead take the other path S_i, v_{21}, v_{22}, T_i, while the flow F_b takes the path S_b, v_{11}, v_{12}, T_b. Observe that both $\mathcal{F}, \mathcal{F}'$ are valid multi-commodity flows.

If I is a *no*-instance, then I' is a *no*-instance
Assume that it is not possible to partition $\mathcal{A}$ into two sets of summed up

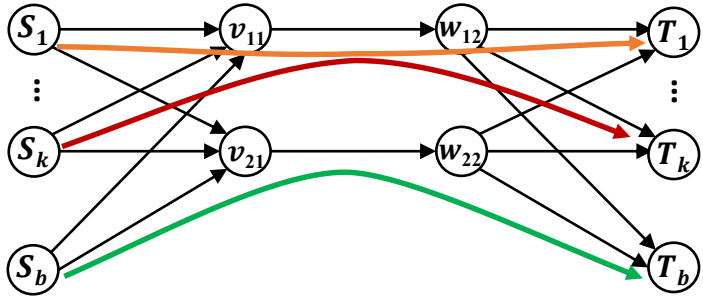


Figure 5.2: In this network, there are k flows from $S_1, \ldots, S_k$ to $T_1, \ldots, T_k$ via the upper path v_{11}, v_{12} of combined size A, and one flow from S_b to T_b via v_{21}, v_{22} of size $A/2$. All edges have a capacity of A. The task is to swap the assignments, i.e., move the k flows to the lower path, and the one bottom flow to the top path. Observe that the only consistent way to do so is to move flows of combined size $A/2$ to the bottom path, then the bottom flow up, then the remaining flows down. As the k unsplittable flows correspond to a Partition instance, it is NP-hard to decide if this is possible.

size $A/2$ each. Note that for F_b to migrate to its new path S_b, v_{11}, v_{12}, T_b, there has to be a free capacity of $A/2$ on the edge from v_{11} to v_{12}. The only possibility for that to happen is to select a subset of the paths from $F_1, \ldots, F_k$ of summed up size $A/2$ to be routed along the edge from v_{21} to v_{22}. But, as the partition instance I is not solvable, no such subset exists.

If I is a *yes*-instance, then I' is a *yes*-instance
Assume that it is possible to partition $\mathcal{A}$ into two sets $\mathcal{A}_1, \mathcal{A}_2$ of summed up size $\mathcal{S}/2$ each. We can then migrate as follows:

We select the flows corresponding to $\mathcal{A}_1$ and move them to their new path, their combined size is $A/2$ and the edge from v_{21} to v_{22} has a free capacity of $A/2$. Then, we move the flow F_b to its new path, which now also has a free capacity of $A/2$. Lastly, we move the flows corresponding to $\mathcal{A}_2$ to their new path, rejoining the flows corresponding to $\mathcal{A}_1$ on the edge from v_{21} to v_{22}. □

Note that the old and new path for each flow was disjoint in this proof construction. As thus, the hardness result also holds for the standard con-

sistency model for the special case where no intermediate paths are allowed:

Corollary 8. *Let $\mathcal{F}$, $\mathcal{F}'$ be unsplittable multi-commodity flows. Deciding if there is a consistent flow migration from $\mathcal{F}$ to $\mathcal{F}'$, s.t. each unsplittable flow may only be on its old or new path, is NP-hard.*

5.4 An Algorithm for Two-splittable Flow Migration

We saw in the last Section 5.3 that unsplittable flow migration is NP-hard, even if the old and new flow paths are known. However, we will now use the power of two to turn the problem of non-mixing consistent flow migration tractable.

As it turns out, by allowing the flows to be two-splittable, we can decide in polynomial time if a non-mixing consistent flow migration is possible. E.g., in Figure 5.2, one could move half of each flow from the top path, move the bottom flow up, and lastly, move the remaining half of the original top flows down. We will now first give an overview of the problem before describing our algorithm, then prove its correctness and completeness, before lastly stating some additional methods for the case that no consistent migration exists.

Creation of slack. We are given two unsplittable multi-commodity flows, with the task to migrate from the old to the new one in a non-mixing consistent way. As we allow the flows to be two-splittable, each unsplittable flow can be separated into two distinct parts, each assigned to the old and new path, respectively.

The fundamental inherent problem is posed here by edges e where all capacity is used. In the non-mixing consistent model, no (part of a) flow can be moved to such an edge e until some free slack capacity has been created on e, else the capacity constraints in the consistency property would be violated.

Observation 3. *For a non-mixing consistent network update to increase the combined sizes of any subset of flows on an edge e by x, the combined total sizes of the flows on e must be at most $c(e) - x$.*

The slack s on an edge e is defined as the ratio of the non-used capacity and the capacity on e. E.g., an edge with capacity 10 and flows of combined size 9 on it has a slack of 1/10. As thus, the question is: Can slack be

created on all such edges e, with each flow being only split along its old and new path, i.e., with two-splittable flows?

The following algorithm will try to create slack on all relevant edges, by iteratively attempting to move parts of flows in a non-mixing consistent way until either slack has been created on all edges of $\mathcal{F}, \mathcal{F}'$, or a situation is reached when such movement is not possible. The idea is that we will only create slack, but never remove it completely from any edge.

Input: (Old) multi-commodity flow $\mathcal{F}$ and a desired mcf $\mathcal{F}'$.
Output: A sequence of non-mixing consistent network updates, starting from $\mathcal{F}$, to create slack on all edges used by $\mathcal{F}, \mathcal{F}'$, or output that this is not possible.

1. Let x be the smallest free capacity on any edge in N used by $\mathcal{F}^*, \mathcal{F}'^*$.
2. Is there a flow $F_i^* \in \mathcal{F}_i^*$ that has no slack on some edge of their path, but there is slack on all edges of the corresponding path of $\mathcal{F}'^*$?
   (a) If yes, perform a network update where the flow size of F'^* is increased by $x/2$ and the flow size of F^* is decreased by $x/2$. Then, go to step 1.
   (b) Else
      i. If all edges of $\mathcal{F}^*$ have slack, output **yes** & all performed network updates so far.
      ii. If there is still an edge of $\mathcal{F}^*$ without any slack/free capacity, output **no**.

Algorithm 5: Creation of slack

Lemma 7. *The updates performed by Algorithm 5 are non-mixing consistent and just use the old and new flow paths.*

Proof. The lemma holds as parts of a flow are just moved to their new path if the new path has enough free capacity, no other paths are used. □

Lemma 8. *The runtime of Algorithm 5 is $O(k^2 n)$, with k being the number of flows/commodities. The number of network updates performed is at most k.*

Proof. Steps 1 & 2 of Algorithm 5 need to be repeated at most k times, with k being the number of commodities/flows, resulting in at most k network updates: If a flow path already has slack on every edge, this flow does not need to be updated again. Furthermore, each iteration of Step 1 & 2 needs to check $O(k)$ flows of length $O(n)$ in the worst case, resulting in a total runtime of $O(k^2 n)$. □

We note that if a sequence of non-mixing consistent network updates leads to a non-mixing consistent migration from $\mathcal{F}$ to $\mathcal{F}'$, then this sequence can also be applied "backwards" to $\mathcal{F}'$, leading to a non-mixing consistent migration from $\mathcal{F}'$ to $\mathcal{F}$. We cast this observation into the following statement:

Observation 4. *A non-mixing consistent migration from $\mathcal{F}$ to $\mathcal{F}'$ exists if and only if a non-mixing consistent migration from $\mathcal{F}'$ to $\mathcal{F}$ exists.*

Non-mixing consistent migration. Let $\mathcal{F}_{slack}$ be a multi-commodity flow in N where every edge used by the flow has a slack of at least s, and similarly, let $\mathcal{F}'_{slack}$ be a multi-commodity flow in N where every edge used by the flow has a slack of at least s. We can then use a method from *SWAN* [37] to migrate in a non-mixing consistent fashion with $\lceil 1/s \rceil - 1$ updates: Every network update moves a share of s (possibly less in the last update) of each flow to its new path, resulting in the given number of updates. E.g., if the slack is 1/10, then 10% of the original flow size will be moved in every update, resulting in 9 updates in total. As the invariant of a slack of at least s will be maintained after every update, each performed update is non-mixing consistent.

Input: (Old) multi-commodity flow $\mathcal{F}$ and a desired multi-commodity flow $\mathcal{F}'$.
Output: **yes**, if a non-mixing consistent migration from $\mathcal{F}$ to $\mathcal{F}'$ is possible, **no** otherwise.

1. Run Algorithm 5 on $\mathcal{F}, \mathcal{F}'$ and $\mathcal{F}', \mathcal{F}$. If either output is no, then output **no**, else output **yes**.

Algorithm 6: Deciding non-mixing consistent migration

Theorem 16. *Algorithm 6 decides in a runtime of $O(k^2 n)$ if a non-mixing consistent migration is possible.*

We defer the complete proof of Theorem 16 and first give the following algorithm for non-mixing consistent migration:

Input: (Old) multi-commodity flow $\mathcal{F}$ and a desired multi-commodity flow $\mathcal{F}'$.
Output: Either a sequence of non-mixing consistent network updates, starting from $\mathcal{F}$, which form a non-mixing consistent migration to $\mathcal{F}'$, or an output that this is not possible.

1. Run Algorithm 5 on $\mathcal{F}, \mathcal{F}'$ and $\mathcal{F}', \mathcal{F}$. If either output is no, then output **no**. Else, denote the resulting networks gotten by applying updates $\mathcal{U}$,$\mathcal{U}'$ by $\mathcal{F}_{slack}$ and $\mathcal{F}'_{slack}$.
2. Let s be the smallest slack on any edge in N used by $\mathcal{F}_{slack}, \mathcal{F}'_{slack}$.
3. Apply the calculated non-mixing consistent updates $\mathcal{U}$ to $\mathcal{F}$, resulting in $\mathcal{F}_{slack}$.
4. Perform $\lceil 1/s \rceil - 1$ non-mixing consistent updates, each moving an original share of s to its new path, resulting in $\mathcal{F}'_{slack}$.
5. Apply the non-mixing consistent updates $\mathcal{U}'$ in reverse, resulting in $\mathcal{F}'$, and output **yes** .

Algorithm 7: Performing non-mixing consistent migration

Combining the above argumentation and Lemma 7, 8 yields:

Corollary 9. *The migration performed by Algorithm 7 is non-mixing consistent, with at most* $2k + \lceil 1/s \rceil - 1$ *updates.*

As thus, we know that a migration performed by Algorithm 7 is non-mixing consistent, but we still need to show that an output of **no** is correct as well:

Lemma 9. *If Algorithm 7 outputs* ***no****, then no consistent non-mixing migration from* $\mathcal{F}$ *to* $\mathcal{F}'$ *is possible.*

Proof. Assume for the sake of contradiction that Algorithm 7 outputs **no**, but that a consistent non-mixing migration from $\mathcal{F}$ to $\mathcal{F}'$ exists with the updates $U_1, U_2, \ldots$. As Algorithm 7 outputs **no**, Algorithm 5 ouputs **no** as well for the case of $\mathcal{F}$ to $\mathcal{F}'$ or the case of $\mathcal{F}'$ to $\mathcal{F}$. Due to Observation 4, we can assume w.l.o.g it was at least for the case of $\mathcal{F}$ to $\mathcal{F}'$. Note that if Algorithm 5 would have output **yes** for both cases, then Algorithm 7 would have output **yes** as well.

Let U_j be the first update where a flow F_i was (partially) moved that Algorithm 5 failed to create slack for on its new path. By assumption, Algorithm 5 was able to create slack (or there was already slack) for all flows of $\mathcal{F}$ moved in $U_1, U_2, \ldots, U_{j-1}$. Note that if U_j was a non-consistent mixing network update, then we can create a non-consistent mixing network update U_j^* from U_j that just contains moving the respective parts of the flow F_i. Recall that Algorithm 5 never removed slack completely from any edge. As thus, Algorithm 5 would have been able to create slack for F_i, as it could have moved a part of flow F_i to its new path after making sure that there is slack for all new flow paths contained in the updates $U_1, U_2, \ldots, U_{j-1}$. Thus, no such non-mixing consistent update U_j could have existed, leading to the desired contradiction, which concludes the proof of Lemma 9. □

We now have all the methods necessary to prove Theorem 16:

Proof of Theorem 16. It directly follows that the runtime is $O(k^2 n)$, as we essentially just run Algorithm 5 twice, cf. Lemma 8. It is left to show that Algorithm 6 is correct, which we will infer from its usage in Algorithm 7: With Corollary 9 we know that an output of **yes** is correct. Similarly, with Lemma 9 we know that an output of **no** is correct, concluding the proof of Theorem 16. □

What if no non-mixing consistent migration exists? It can be the case that Algorithm 7 outputs that no non-mixing consistent migration of flows exists for two-splittable flows, but that the benefits of the desired new flow $\mathcal{F}'$ outweigh the downsides of congestion during the migration.

In this case, we can apply Algorithm 5 to $\mathcal{F}, \mathcal{F}'$ to pre-compute updates for as many edges with slack as possible in $\mathcal{F}_{slack}, \mathcal{F}'_{slack}$, and migrate non-mixing consistent to $\mathcal{F}_{slack}$ using these updates from Algorithm 5. Then, in the next step, we use the approach from *Dionysus* [42], which breaks consistency during some updates, but still migrates the flows from $\mathcal{F}_{slack}$ to $\mathcal{F}'_{slack}$. Lastly, we can migrate in a non-mixing consistent fashion from $\mathcal{F}'_{slack}$ to the desired multi-commodity flow $\mathcal{F}'$, using the pre-computed updates from Algorithm 5.

5.5 Summary

We introduced and motivated the concepts of non-mixing consistent flow updates and migration, where un-/two-splittable flows respect waypoint traversals and service chaining. For unsplittable flow migration, we showed that non-mixing consistency respecting old and new flow paths is NP-hard, with a proof that also applies for consistent migration of unsplittable flows without intermediate paths. However, when the flows are two-splittable, we give a fast polynomial algorithm for a non-mixing consistent migration, and outlined alternatives when no non-mixing consistent migration is possible.

6

Lossless Flow Migration

In this chapter, we are going to discuss that all current network update techniques overlook the effect of flows congesting their own path during a network update, with the reason being latency on the links (Section 6.1). Furthermore, while congestion will be resolved eventually after the network update, the buffers of the affected routers can be filled for a long time period, leading to the following paradox: A flow is moved to a path with less latency, but the latency stays the same! As flows are often migrated because of latency concerns, this is highly undesirable.

We show in Section 6.2 that these effects will occur in networks in practice, formally model the problem in Section 6.3, and give algorithms in Section 6.4 to check if a network update is consistent in this context. Furthermore, in Section 6.5 we also give a polynomial time algorithm to check if a sequence of intermediate lossless updates exists that will result in the desired network update, leading to the first polynomial time schedule generation algorithm for (temporarily) splittable multi-commodity flows. Nonetheless,

the migration problem turns out to be NP-hard for known latencies, already for a single splittable flow, see Section 6.6.

6.1 Motivation

Understanding network updates has issues besides the previously described consistency for flow migration: Namely, flows congesting themselves. We would like to motivate these issues with a simple example. In your small gigabit ethernet, you transport a single flow at a rate of one gigabit as well. The end-to-end latency of the flow is undesirably high, so you move the flow to a path with low latency. Naturally, you expect the latency of your flow to be reduced, however, paradoxically, the end-to-end latency of the flow does not change at all! Later, you decide to move your flow to yet another path with low latency. This time it works, but a lot of packets are dropped in the process! Both phenomena can be explained with a simple example network, see Figure 6.1.

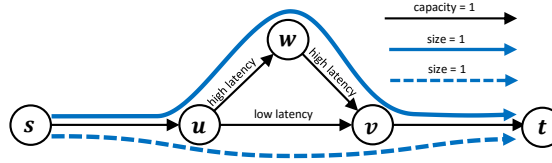


Figure 6.1: In this network, the flow along the solid path is moved to the dashed path, causing congestion at v. Current methods overlook this issue of flows congesting "themselves".

When the flow is moved from the solid path to the dashed path by, e.g., switching the forwarding rule at u, congestion will occur at node v. For some time, packets from both w and u will arrive at v, at a rate of 2 gigabits, twice the rate the outgoing link of v can manage. Thus, v will need to buffer the extra packets, or if the buffer is not large enough, drop them.

In the case of a *UDP* flow and large enough buffer size, all the extra delay from the old path is "moved" into the buffer at v. If the rate of the flow is not limited, this buffer will never decrease, causing the delay to be there permanently. In the case of a *TCP* flow, not only will packets be dropped if the buffer is not large enough, but the re-ordering of the packets will lead to

further problems: In addition to the lost packets, the source will decrease its sending rate, taking additional time until the original throughput is reached again.

Overview We start this chapter by confirming that latency-induced congestion upon updates does happen in practice. Motivated by our findings, we then develop the first theoretical network update framework which can deal with link latency during network updates for flows.

We describe provably correct algorithms, checking for link capacity violations for any set of given (or even unknown) delays, considering unsplittable flows.

Should a single update violate link capacities, we explore the option of migration via intermediate updates, and also consider the case of migrating multi-commodity flows.

A heterogeneous picture unfolds: On the one hand, checking for a lossless migration with known latency is NP-hard already for a single flow. On the other hand, if we allow to split the flows into a linear number of paths, we can check in polynomial time if a lossless migration exists with unknown latencies – even for multi-commodity flows. We then turn this decision procedure into a polynomial time algorithm to give a schedule for lossless migration, using only a linear number of additional flow splittings.

Contribution The contribution of this chapter is a combination of a practical evaluation and an in-depth theoretical analysis, with each part of independent interest, but together forming a framework on how to deal with edge latency for network flow updates.

Practical Evaluation In Section 6.2, we show that the effects described in the introduction for the network in Figure 6.1 appear in practice, by setting up a network and collecting data during and following network updates. We study both TCP and UDP, and give measurements for packet loss and latency.

Theoretical Analysis and Algorithms We perform an extensive theoretical analysis regarding the impact of link latency on lossless flow migration during network updates. After defining the model in Section 6.3, we completely explore the problem in two dimensions, distinguishing between the cases where the network latencies are known (called ℓ-consistency), respectively unknown (called $\forall$-consistency).

We start in Section 6.4, where we show how to check in linear time if a given network update is consistent for a single unsplittable flow. We extend this line of thought by exploring if a set of intermediate consistent updates exists s.t. one can migrate in a lossless fashion.

For unknown latencies and n-splittable flows, we give a polynomial time algorithm in Section 6.5 for lossless migration, which also works for multi-commodity flows.

Surprisingly, if the latencies are fixed, the problem of lossless migration turns out to be NP-hard, even for a single splittable flow, cf. Section 6.6.

6.2 Practical Evaluation

In this section, we show that the congestion effects highlighted in Figure 6.1 do occur in practice, inducing congestion during potentially seconds upon updates. We start by describing a brief model, that formalizes the effects described in the introduction, followed by a description of our testbed. We then evaluate the effects considering both UDP and TCP flows, confirming our model assumptions.

Model We start with some notations and assumptions. Consider the network in Figure 6.1 and let F be the old flow arriving via w and F' be the new flow arriving via the lower path from u. Denote the size of F by $d_F = F((w,v)) \geq 0$ and the size of F' by $d_{F'} = F'((u,v)) \geq 0$. Let the latency $\Delta_{old} > 0$ of the path u, w, v be larger than the latency $\Delta_{new} > 0$ of the path u, w and denote the time difference by Δ. Lastly, let the buffer size of v be $B(v) \geq 0$ and let the outgoing link of v have a capacity of $c > 0$ with $d_F + d_{F'} > c$.

We conjecture that the following effects will appear after the node u has switched its forwarding from w to v:

- The buffer at v will be filled up at most to:

$$\min\left((d_F + d_{F'} - c) \cdot \Delta, B(v)\right), \tag{6.1}$$

  i.e., the buffer will at most be filled with surplus data, not being able to be drained, arriving during the time when both flows arrive at v.

- The amount of data dropped at v will be:

$$(d_F + d_{F'} - c) \cdot \Delta - B(v), \tag{6.2}$$

i.e., the amount of surplus data not fitting into the buffer.

- The buffer at v will be drained after the following time:

$$\frac{(d_F + d_{F'} - c) \cdot \Delta}{d_{F'} - c}, \tag{6.3}$$

i.e., after the incoming flow F from w has stopped, the drain of the buffer is equivalent to the difference between the capacity of the outgoing link and the throughput of the remaining new flow F'.

We note that in theory, if the new flow F' has the same size as the capacity c of the outgoing link, the term 6.3 implies that the buffer at v will never be drained.

Methodology & Testbed To validate our theoretical model in practice, we replicated Figure 6.1 in a testbed composed of five servers connected through Gigabit links. Each of the machines has at least the following specifications: Ubuntu 64bit server 14.04 with kernel 3.16, 8GB of RAM, 2x Intel Xeon quadcore 2.4GHz. We configure the one-way latency along the links (u, w) and (w, v) as well as the buffer size at v using `tc`.

We create traffic in the network by establishing UDP and TCP connections from s to t with `iperf`. Initially, we configure the network so that traffic is forwarded via the (slow) path $s - u - w - v - t$. After about 15 seconds, we switch u to the fast-path $s - u - v - t$. In parallel, we monitor the buffer at v and the round-trip time from s to t using `iperf`. After the switch, we let the flows run for 60s. We repeated all our measurements at least 30 times and report the median values in the following.

Upon updates, congestion can appear for several seconds when link utilization is high. We start by considering the effect of updating a network whose link utilization is high such as a Wide-Area Network (WAN) where link utilization is often above 90% [37]. For this, we run a single UDP connection at a rate close to the link capacity, yet without creating any congestion. We set the buffer size $B(v)$ such that no packet would be dropped at v.

Figure 6.2 depicts the evolution of the delay prior and after the update. The dotted vertical line indicates the moment traffic is switched to the low delay path. We observe that the end-to-end delay stays stable up to 5 seconds after the switch. As described in our model, this is due to the

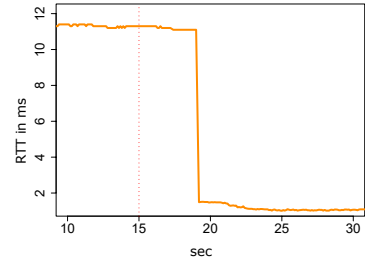


Figure 6.2: Median end-to-end latency over 30 UDP experiments. The dotted vertical line denotes the update time. It takes seconds for the congestion induced by latency to disappear.

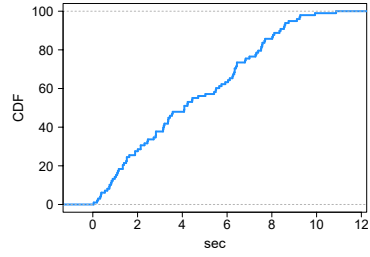


Figure 6.3: CDF of the duration length. In 50% of the experiments, congestion lasted for more than 4 seconds, in 10%, for more than 8 seconds.

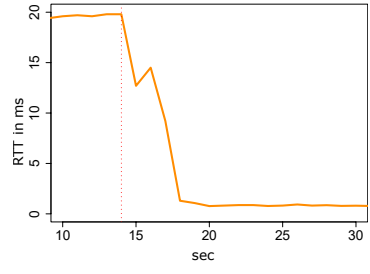


Figure 6.4: Median end-to-end latency over 30 TCP experiments. The dotted vertical line denotes the update time. Latency-induced congestion is also detrimental for TCP flows, albeit for shorter time.

latency from the packets in flight along the slow path $u - w - v$ which was moved into the buffer at v after the switch. Figure 6.3 depicts the CDF of the number of seconds it takes for congestion to disappear across all experiments. In the majority of the cases, congestion appeared for *more than 4 seconds*. In 10% of the case, the congestion lasted for 10 seconds or more. This clearly shows that not accounting for latency (as SWAN [37] or zUpdate [50]) can be highly detrimental for network traffic. Observe also that with additional cross traffic at v, the congestion measured above will last even longer.

Congestion did not appear when the rate of the UDP flow was below 50%. In these situations, the buffer drained nearly immediately after the switch, together with the dropping of the additional delay.

Congestion also appears for TCP flows, but last shorter due to congestion avoidance mechanisms. We now show that latency-induced congestion also impacts TCP flows, which accounts for the vast majority of the Internet traffic.

Figure 6.4 reports the evolution of the delay prior and after the update when running 10 concurrent TCP connections between s and t instead of one UDP connection. The effect of congestion (higher end-to-end delay) is still clearly visible. It takes about 3 seconds for the flows to stabilize around the minimum delay. This is explained as the throughput of TCP flows fluctuates due to congestion avoidance mechanisms. As such, the link is not perfectly filled all the time as some flows back off when they experience a packet loss or receive three duplicate ACKs. Such backing off enables the buffer to drain.

Our model precisely captures the practical effects measured. In all our experiments, the amount of packets dropped and the maximum buffer size at v was consistent with the terms 6.1, 6.2 with delays ranging from 2ms to 40ms.

6.3 Model & Problem Setting

We first define some common notation, such as network, latency, or flow, before describing network updates in Subsection 6.3. Afterwards, in Subsection 6.3, we investigate why current systems overlook the effect of latency

on network flow updates, leading to our own approaches in the subsequent sections.

Network, graph, capacity, latency We define a network as a simple (i.e., no self-loops) directed graph with edge capacities.

Definition 16. *Let $G = (V, E)$ be a simple connected directed graph with $n = |V|$ nodes, representing the routers, and $m = |E|$ edges, representing the links. We denote the set of outgoing edges (v, u) of a node $v \in V$ by $\texttt{out}(v)$ and the set of incoming edges (u, v) by $\texttt{in}(v)$. A* network *N is a pair (G, c), with $c : E \to \mathbb{R}^+$ being a function assigning each edge $e \in E$ a* capacity *of $c(e)$.*

For ease of readability, we model delays in the network as latency on the edges, i.e., the time it takes from arriving at some router to the next hop along the path.

Definition 17. *Let $N = (G, c)$ be a network. The latency of N is a function $\ell : E \to \mathbb{R}^+$ assigning each edge $e = (u, v) \in E$ a* latency *of $\ell(e)$, with $\ell(e)$ being the time T it takes data to arrive at v from u.*

Buffers We note that that besides not dropping packets due to edges being over capacity, we also want to avoid congestion in the form of buffer build-ups. As thus, we effectively model the available buffer sizes as zero, meaning in turn that our methods will work for any current buffer utilization and sizes.

Flows Next, we define an unsplittable flow according to the standard flow constraints, i.e., demand satisfaction, flow conservation, and capacity constraints. As common in this context, we only consider cycle-free flows in this chapter.

Definition 18. *Let $N = (G, c)$ be a network. A map $F : E \to R_{\geq 0}$ is called an (unsplittable)* flow *(from s to t) if it is cycle-free and fulfills the standard flow constraints, i.e.,*

$$\forall v \in V \setminus \{s, t\} : \sum\nolimits_{e \in \texttt{out}(v)} F(e) = \sum\nolimits_{e \in \texttt{in}(v)} F(e), \tag{6.4}$$

$$\sum\nolimits_{e \in \texttt{out}(s)} F(e) = d_F = \sum\nolimits_{e \in \texttt{in}(t)} F(e), \tag{6.5}$$

$$\forall e \in E : F(e) \leq c(e), \tag{6.6}$$

with d_F being the size of F and the edges with $F(e) > 0$ forming a simple path from s to t.

Definition 19. *Let $F_1, F_2, \ldots, F_k$ be a set of unsplittable flows. $\mathcal{F} = (F_1, \ldots, F_k)$ is called a* multi-commodity flow*, if for all edges e in E holds: $\sum_{i=1}^{k} F_i(e) \leq c(e)$.*

Network Updates

In this chapter we only consider network updates for flows, i.e., given a set of *old* forwarding rules for some flow F, we want to change to a set of *new* forwarding rules for another flow F'.

Definition 20. *Let N be a network and let F, F' be flows in N, both from node s to t. A* network update *is a triple (N, F, F').*

Atomicity of network updates We assume that the change from a flow F to a flow F', both from s to t, is performed as an atomic operation on the ingress router s. In practice, this can be achieved by a two-phase protocol as described in, e.g., [68]: All forwarding rules in the network for F' are installed by the SDN controller first. When these installations are confirmed, the ingress router s will start tagging all packets, that previously were marked with F, with F'. Note that with this method, if the latency from s to some node in the network is the same for F and F', the flow F' will arrive after F has departed already.

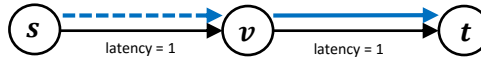


Figure 6.5: This picture depicts the network one time unit after the ingress router s switches from F (solid) to F' (dashed).

Updates under known latency We can now define when a network update is consistent under the effects of latency:

Definition 21. *Let (N, F, F') be a network update where the ingress router s switches from F to F' at time $T = 0$. Let ℓ be the latency of N. The network update (N, F, F') is* ℓ-consistent*, if there is no time $T \geq 0$ s.t. the capacity limit of some edge is violated at time T.*

For an example of Definition 21 being applied, consider the network update in Figure 6.1: When the ingress router switches from F (solid path) to F' (dashed path), the flow of F' is always behind F, until F' takes a shortcut to v: Then, for a time equal to the latency differences between both paths, the router v will have an incoming flow of 2, even though the only outgoing edge has a size of 1. Thus, there will be congestion, and the network update is not ℓ-consistent.

Updates under unknown latency We will also cover the special case that the latency of the network is unknown and say that a network update is $\forall$-consistent if it is ℓ-consistent for all ℓ:

Definition 22. *Let (N, F, F') be a network update where the ingress router s switches from F to F' at time $T = 0$. The network update (N, F, F') is* $\forall$-consistent*, if for all possible latencies ℓ of N holds: (N, F, F') is ℓ-consistent.*

For a small example, consider the network in Figure 6.1: The network update from F to F' is not $\forall$-consistent, and the reverse situation from F' to F would not be $\forall$-consistent either, as the edges depicted with low latency could have high latency and vice versa.

Effects on Current Systems

To the best of our knowledge, current network update mechanisms that aim at lossless migration overlook at large the effects of latency in the network.[1] Systems like *SWAN* keep their focus on the asynchrony caused by changing multiple flows at once, which is not an atomic operation such as changing just one flow. E.g., *zUpdate* defines a network update of multiple flows $F_1, \ldots, F_k$ to be *lossless* if the following condition holds:

$$\forall e \in E : \sum\nolimits_{1 \leq i \leq k} \max\left(F_i\left(e\right), F_i'\left(e\right)\right) \leq c\left(e\right). \tag{6.7}$$

Even though this is an elegant way to capture the asynchrony of different flows F_i, F_j with each other, it does not account for flows F_i, F_i' congesting the same edge due to latency. When considering Figure 6.1, all edges satisfy

[1] We note that flow scheduling systems, which by design do not move flows while they are running, do not suffer from this problem.

Condition (6.7), yet there is congestion: For some time, the old flow F and the new flow F' use the same edge outgoing from v, violating the edge's capacity.

Enforcing enough space for old and new flow A straightforward way to "fix" Condition (6.7) would be to replace the maximum operation with the addition operation similar to Definition 13 in the last chapter, i.e.,

$$\forall e \in E : \sum\nolimits_{1 \leq i \leq k} \left(F_i(e) + F'_i(e) \right) \leq c(e) . \tag{6.8}$$

Now, the network update in Figure 6.1 is no longer detected as lossless. However, Condition (6.8) is "too strong", cf. Figure 6.6: In this example, there will be no congestion, as the flows do not meet again once they divert from each other's path. Yet, the network update in Figure 6.6 violates Condition (6.8)!

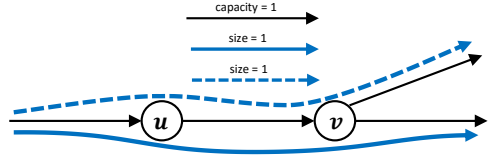


Figure 6.6: When changing from the solid (F) to the dashed path (F'), Condition (6.7) is satisfied on all edges, but Condition (6.8) is violated on the incoming edges of u and v.

Thus, we need different methods to deal with congestion during network updates, which will be covered in the next sections.

6.4 Checking Unsplittable Flow Network Updates

In this section, we will start by checking if a network update for unsplittable flows is consistent for known latency values. Afterwards, we show how to check the consistency of a network update if the latency in the network is unknown.

Known Network Latencies

The main idea for ℓ-consistent network updates for unsplittable flows can

be described as follows: If the new flow F' utilizes some edge e while the old flow F is still using edge e, then the capacity of e needs to be large enough to accommodate both, i.e., $F(e) + F'(e) \leq c(e)$. We refer to Figure 6.7 for a first illustration, before giving an algorithm to solve the problem.

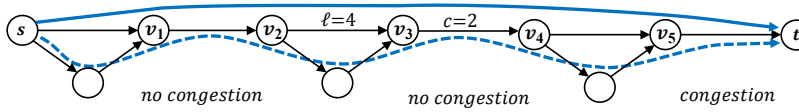


Figure 6.7: In this network, the solid line represents the old flow F, while the dashed line represents the new flow F', both of size one. All edges have a delay and capacity of one, unless denoted otherwise. On the edge (v_1, v_2) there is no congestion, as F' arrives when the edge is not used anymore by F. For one time unit F, F' are on the edge(v_3, v_4), but there is enough capacity for both flows. On the last edge (v_5, t) however, the capacity is violated for one time unit, as its capacity is just one.

Theorem 17. *A network update (N, F, F') is ℓ-consistent if and only if the $\Delta_{\max}$ computed by Algorithm 8 is 0.*

Proof. We start by assuming that the network update (N, F, F') is not ℓ-consistent and that Algorithm 8 will output that (N, F, F') is ℓ-consistent. Let T^* be the first time when the capacity of some edge $e_i = (v_{i-1}, v_i)$ is violated, i.e., the capacity of all other edges will not be violated until time T^*.

The capacity of e_i is violated because there is flow arriving at v_{i-1} from F and F' at time T^*, even though $F(e_i) + F'(e_i) > c(e)$ holds. However, for that to happen, the latency of the path of F' from s to v_{i-1} needs to be lower than the latency of the path of F from s to v_{i-1}. This leads to a contradiction, as Algorithm 8 checks exactly for this Condition.

We now consider the reverse case, i.e., that the network update (N, F, F') is ℓ-consistent and Algorithm 8 will output that (N, F, F') is not ℓ-consistent. The idea is essentially analogous, if Algorithm 8 detects a $\Delta_{\max} > 0$ for some edge, then the update cannot be ℓ-consistent. □

Runtime By choosing data structures with an $O(1)$-access time and all input values being pre-sorted, Algorithm 8 can be implemented to run in

Input: A network $N = (G, c)$, a latency ℓ, and flows F, F' along the paths $s = v_0, \ldots, v_p = t$ and $s = v'_0, \ldots, v'_{p'} = t$

1: Set latency at s to zero for F and F' with $v_0^F := 0$ and $v_0'^{F'} := 0$
2: **for** $i \leftarrow 1$ to p **do**
3: $\quad v_i^F := v_{i-1}^F + \ell((v_{i-1}, v_i))$ ▷ calculate latency for F
4: **end for**
5: **for** $i \leftarrow 1$ to p' **do**
6: $\quad v_i'^{F'} := v_{i-1}'^{F'} + \ell((v'_{i-1}, v'_i))$ ▷ calculate latency for F'
7: **end for**
8: $\Delta_{\max} := 0$ ▷ Initialize max latency error to 0
9: **for** $\forall\ (v_{i-1}, v_i) = (v'_{j-1}, v'_j) =: e_i$ **do** ▷ all edges in both F, F'
10: $\quad$ **if** $v_{i-1}^F - v_{j-1}'^{F'} > \Delta_{\max}$ and $F(e_i) + F'(e_i) > c(e_i)$ **then**
11: $\quad\quad \Delta_{\max} := v_{i-1}^F - v_{j-1}'^{F'}$
12: $\quad$ **end if**
13: **end for**
Output: $\Delta_{\max}$

Algorithm 8: The algorithm computes for unsplittable flows F, F' how much ($\Delta_{\max}$) the new flow F' has to be delayed s.t. F' will not utilize an edge that F still uses with not enough capacity to support both F and F'. Should $\Delta_{\max}$ be > 0, then the network update (N, F, F') is not ℓ-consistent, else (N, F, F') is ℓ-consistent.

linear time. Else, at most a logarithmic overhead needs to be added for sorting the values.

Corollary 10. *Algorithm 8 has a runtime of* $O(n \log n)$.

Unknown Network Latencies

In this subsection, we will deal with unknown latency in the network, i.e., checking if a network update is $\forall$-consistent. A first thought could be to re-use Algorithm 8, and just set the latency of all edges with F' to ϵ and the latency of all edges with F to some arbitrarily large value ∞. However, how do we deal with edges e where $F(e) > 0$ and $F'(e) > 0$ holds?

We propose the following Algorithm 9, which checks for a first edge e along the path of F after F' rejoins the path of F where $F(e) + F'(e) > c(e)$ holds.

Input: A network $N = (G = (V, E), c)$, and flows F, F' along the paths $s = v_0, \ldots, v_p = t$ and $s = v'_0, \ldots, v'_{p'} = t$

```
1: AWAY:=false                                   ▷ F' has not left the path of F yet
2: CONSISTENCY:=true                             ▷ ∀-consistency not violated yet
3: for i ← 1 to p do
4:     if F'((v_{i-1}, v_i)) = 0 then
5:         AWAY:= true                           ▷ F' has left the path of F
6:     end if
7:     if F'((v_{i-1}, v_i)) + F((v_{i-1}, v_i)) > c((v_{i-1}, v_i)) and AWAY= true then
8:         CONSISTENCY:=false                    ▷ ∀-consistency violated
9:     end if
10: end for
```

Output: CONSISTENCY

Algorithm 9: The algorithm computes if the network update is ∀-consistent by checking if F' rejoins the path of F on an edge without enough capacity

Theorem 18. *The network update (N, F, F') is ∀-consistent if and only if Algorithm 9 outputs true.*

Proof. Recall that the flows F, F' are cycle-free. Assume that AWAY stays as false forever. Then the path of F and F' is identical and the network update (N, F, F') is ∀-consistent, i.e., the output is correct.

Now consider that AWAY gets set to false for the first time due to some edge e_j having the property $F'(e_j) = 0$. As the path of F and F' has been identical before e_j, ∀-consistency will not be violated if F and F' do not share any further edges behind e_j along the path of F or only edges e with $F(e) + F'(e) \leq c(e)$.

Assume that Algorithm 9 now outputs false. Then, there was a first edge $e_{j'}$ behind e_j along along the path of F with $F(e_{j'}) + F'(e_{j'}) > c(e_{j'})$. We define a latency ℓ as follows: All edges except e_j have a latency of some arbitrarily small ϵ and the edge e_j has a latency of some arbitrarily high value ∞. Now, after at least a time of $n \cdot \epsilon$, the utilization of F' on $e_{j'}$ is $F'(e_{j'})$ – but at the same time, the utilization of F on $e_{j'}$ is $F(e_j)$. As it holds that $F(e_{j'}) + F'(e_{j'}) > c(e_{j'})$, the considered network update (N, F, F') is not ∀-consistent, which concludes the proof. □

Runtime As only a path of length $p \leq n$ needs to be checked, it follows

that the runtime is linear.

Corollary 11. *Algorithm 9 has a runtime of $O(n)$.*

6.5 Checking for a ∀-Consistent Migration

So far, we just checked the losslessness for a single given network update, for just a single unsplittable flow. However, if the update is not consistent, maybe there is still a way to reach the desired flow by a migration with intermediate update steps, also handling multi-commodity flows? One central problem appearing in this context is the asynchrony of updating multiple flows at once: As the update happens in multiple parts of the network, a completely synchronized update is not possible, the individual routers could execute their updates in any ordering, cf. [36, 42, 49].

Unsplittable ∀-consistent migration is NP-hard For both the cases of the flows only being allowed to be either on the old or new path, or also allowing arbitrary flow reroutings, the constructions in the previous Sections 4.2 and 5.3 can be easily adapted to show that unsplittable lossless migration is NP-hard to decide for unknown latencies. Hence, we will allow the splitting of flows to turn the migration problem tractable.

Splitting flows vs. n-splittable flows A natural approach, inspired from mathematical flow theory, is to allow a single flow to be splittable: I.e., the flow packets arriving at routers are to be split according to some pre-set distribution.

However, the standard hash based flow splitting is not exact from a theory point of view: It is based on probabilistic assumptions, meaning that in practice, there can be a sequence of packets which will not be split according to the pre-set distribution. As we do not want to drop packets, we are going for another approach, as done by, e.g., [41]: We will split each flow into at separate (unsplittable) flow paths, each having its own flow rules, totally eliminating any probabilistic splitting.

Linear programming vs combinatorial algorithms Linear programming is currently the method of choice when considering flow migration in partial stages, cf., e.g., [37, 50, 86]. We see two main problems with this approach:

1. The number of possible unsplittable flow paths in a graph can be exponential. As thus, when checking for all path options (e.g., when adopting the LP formulation in [86] for our case), the LP grows to be of exponential size.

2. The number of intermediate updates for a lossless migration can be superpolynomial. As thus, an LP calculating the intermediate steps will take superpolynomial runtime too, and cannot decide if a migration is possible at all. E.g., *SWAN* [37] performs a binary search over the number of intermediate steps, with the corresponding halting problem addressed in Chapter 4, while the work in [86] takes the number of intermediate steps as an input.

Therefore, inspired by our two previous chapters, we will take a combinatorial approach, checking for intermediate flow placements in a lossless manner.

∀-consistent migration In the following, we will speak about splittable flows, but each splittable flow F_i is in reality a collection of unsplittable flows $F_{i,1}, F_{i,2} \ldots$ from s_i to t_i. A ∀-consistent migration will begin and end with a set of unsplittable multi-commodity flows $\mathcal{F}, F'$, but in intermediate updates, each flow F_i can consist of multiple paths:

Definition 23. *A sequence of r ∀-consistent network updates $(N, (F_1, \ldots, F_k), (F_1^1, \ldots, F_k^1)), (N, (F_1^1, \ldots, F_k^1), (F_1^2, \ldots, F_k^2)), \ldots, (N, (F_1^{r-1}, \ldots, F_k^{r-1}), (F_1', \ldots, F_k'))$, with $d_{F_i} = d_{F_i'} = d_{F_i^j}$ for all $1 \leq i \leq k$ and $1 \leq j \leq r-1$, is called a ∀-consistent migration (from $\mathcal{F} = (F_1, \ldots, F_k)$ to $\mathcal{F}' = (F_1', \ldots, F_k')$).*

Should $d_{F_i} > d_{F_i'}$, then one could first reduce the size of d_F to the one of $d_{F'}$ before migrating, or vice versa for $d_F < d_{F'}$.

Let's generate slack A key concept used by [37] is the concept of *slack*, i.e., free capacity on the edges. When we speak about slack s, we mean free capacity of size s, while relative slack s_R denotes a free fraction of capacity. One of the central ideas by *SWAN* [37] is that (lossless) migration is easy if there is relative slack s_R on every edge: E.g., if s_R is 1/10, one can migrate in 9 ∀-consistent updates by moving 10% of the flow utilization each time to the new flow rules, never breaking any capacity constraints.

We note that for ∀-consistent updates, we can weaken this requirement of all edges having slack: Given an old flow $F_i \in \mathcal{F}$ and a new flow $F_i' \in$

$\mathcal{F}'$, we introduce the concept of a *divergence node* $v_{F,F'}$. For the old flow path P_{F_i} and the new flow path $P_{F'_i}$, the divergence node $v_{F,F'}$ is the first node, starting from s_i along both paths, beyond which both flows take a different edge to continue. E.g., in Figure 6.6, the divergence node is v. For completeness reasons, we define $v_{F,F'}$ to be t_i if $P_{F_i} = P_{F'_i}$. As both flows are the "same" until $v_{F,F'}$, we do not need any slack until $v_{F,F'}$ when migrating between F_i and F'_i. Note that other flows on these edges might enforce the need for slack though when using the method of *SWAN*.

In the following Algorithm 10 we will give a method to check how slack can be generated on as many edges as possible under the restriction of ∀-consistent network updates:

Input: Network N and m.-c. flow $\mathcal{F} = (F_1, \ldots, F_k)$.

1. For every edge $e = (u, v)$ without slack, check via BFS if there is a flow $F_{i,j}$ on e s.t. there is a path P with slack $s > 0$ from u to t_i. If yes, let w be the node in $F_{i,j}$ closest to s_i if you travel along $F_{i,j}$, that is also contained in P. Let P' be the subpath of P from w to t_i. Perform a network update where $F_{i,j}$ is split into two by reducing the size of $F_{i,j}$ by $\min\left\{d_{F_{i,j}}/2, s/2\right\}$ and adding a new unsplittable flow of size $\min\left\{d_{F_{i,j}}/2, s/2\right\}$ along the path of $F_{i,j}$ to u and then via P' to t_i.
2. Repeat Step 1 until either all edges have slack or there has been one iteration of Step 1 without any network updates (i.e., all edges without slack were checked for all flows).

Output: The obtained m.-c. flow $\mathcal{F}^* = (F_1^*, \ldots, F_k^*)$.

Algorithm 10: The algorithm creates slack on every edge where slack can be created by a sequence of ∀-consistent network updates. Note that the slack generation is non-destructive in the sense that if an edge has slack at some point, it will always keep some slack greater than zero.

Lemma 10. *Using ∀-consistent updates, Algorithm 10 creates slack on every edge where slack can be created via a ∀-consistent migration.*

Proof. Observe that the performed network updates by Algorithm 10 are ∀-consistent, as flows are only split and rerouted via paths of free capacity. Assume that there exists some ∀-consistent migration $\mathcal{M}$ that creates slack on further edges: Consider the first update U of $\mathcal{M}$ where free capacity on

an edge e' was created where Algorithm 10 was not able to create slack. In U, a flow F using e' is rerouted (partially) along some path P not containing e'. Denote the flow induced in M before U was applied by $\mathcal{F}^U$. $\mathcal{F}^*$ has slack on every edge where $\mathcal{F}^U$ has slack as well. As thus, we can also reroute F (partially) along the same path P, inducing slack on e'. □

Complexity of slack generation Observe that Algorithm 10 creates slack on at least one edge when a new flow (path) is introduced. As thus, at most $|E| = m$ new flows are created with at most m network updates. Note that each flow is split at most n times. Furthermore, a BFS can be performed in $O(m)$ time. With $O(n)$ destinations and m edges, the total computational runtime is therefore $O(nm^3)$.

Corollary 12. *Algorithm 10 has a runtime of $O(nm^3)$, performing at most m network updates and introducing at most m new flows.*

We can now use Algorithm 10 to check (and in the positive case, perform) for a ∀-consistent migration. The idea is as follows: We see if we can create slack on all edges beyong the divergence points of the old and new flows. If yes, we can migrate between these two states step by step using the slack method of *SWAN*. Else, no ∀-consistent migration is possible.

Lemma 11. *If Algorithm 11 performs a migration it is ∀-consistent, else no ∀-consistent migration is possible.*

Proof. We begin by showing that the migration is ∀-consistent. The migration via Algorithm 10 is ∀-consistent due to Lemma 10. Note that every ∀-consistent network update is also ∀-consistent if applied in reverse. Hence, using Algorithm 10 in reverse is a ∀-consistent migration as well. Lastly, when using the SWAN method, we only insert an amount of flow to edges that is at most the available slack, which is ∀-consistent.

We now show that else no ∀-consistent migration is possible. Assume that for some i there is an edge e^* behind $v_{F_i,F_i'}$ in F_i s.t. Algorithm 10 cannot induce slack on e^* for $\mathcal{F}$ (the symmetric case for F_i' and $\mathcal{F}'$ can be handled analogously due to the reversibility of ∀-consistent updates), but that there is a ∀-consistent migration $\mathcal{M}$ from $\mathcal{F}$ to $\mathcal{F}'$. As e^* is behind $v_{F_i,F_i'}$, there must have been some update U in $\mathcal{M}$ that reroutes some part of F_i going along e^*. The rerouted flow part cannot use e^* directly after

Input: Network N and m.-c. flows $\mathcal{F}, \mathcal{F}'$.

1. Execute algorithm 10 on N and $\mathcal{F}$.
2. For every corresponding pair of flows $F_i \in \mathcal{F}, F_i' \in \mathcal{F}'$, check if all edges behind $v_{F_i,F_i'}$ in F_i have slack in the output of Step 1.
3. Repeat Steps 1, 2 for $\mathcal{F}'$ instead of $\mathcal{F}$.
4. If the answer is yes in both executions of Step 2, then migrate from $\mathcal{F}$ to $\mathcal{F}'$ as follows. Else, no ∀-consistent migration is possible.
   (a) Migrate from $\mathcal{F}$ to $\mathcal{F}^*$ as described in Algorithm 10.
   (b) Migrate from $\mathcal{F}^*$ to $\mathcal{F}'^*$ using the technique from *SWAN*.
   (c) Migrate from $\mathcal{F}'^*$ to $\mathcal{F}'$ using Algorithm 10 in reverse.

Output: A ∀-consistent migration from $\mathcal{F}$ to $\mathcal{F}'$, if it exists.

Algorithm 11: The algorithm first utilizes two executions of Algorithm 10 to check if a ∀-consistent migration is possible. If yes, then the algorithm proceeds in three steps by first creating slack on $\mathcal{F}$, migrating to a modified version of $\mathcal{F}'$ with slack, and lastly migrating to $\mathcal{F}'$, all with ∀-consistent network updates.

U, as ∀-consistency requires some slack on e^*. Thereby, U could be used to create slack on e^*, which is a contradiction to the above assumption. □

Complexity of ∀-consistent migration Note that the number of network updates and new flow rules created by the executions of Algorithm 10 are at most $2m$, respectively. When migrating between $\mathcal{F}^*$ and $\mathcal{F}'^*$, no further flow rules are needed for the updates of the *SWAN* method. The number of network updates for this step are $\lceil 1/s_R \rceil$, with s_R being the smallest relative slack of $\mathcal{F}^*$ and $\mathcal{F}'^*$. For the computational runtime, note that the implicit schedule generation of the *SWAN* method (perform $\lceil 1/s_R \rceil$ network updates of the same type) is dominated by the executions of Algorithm 10 with runtimes of $O(nm^3)$.

Corollary 13. *Let s_R be the smallest relative slack created by Algorithm 10 on $\mathcal{F}, \mathcal{F}'$. Algorithm 11 performs at most $\lceil 1/s_R \rceil + 2m$ network updates with at most $2m$ additional flows. The runtime for the (implicit) schedule generation is $O(nm^3)$.*

Waypoint enforcement Lastly, we note that in the current form, our

migration algorithm does not respect waypointing (e.g., firewalls) or service chains in general, cf. [54]. However, observe that each intermediate (splitted) flow has its own forwarding rules, as we use the method of [68]. As thus, we can extend our migration algorithm to enforce such rules. E.g., intermediate paths are only allowed if they pass a firewall.

6.6 Checking for a ℓ-Consistent Migration

In the previous Section 6.5, we checked for $\forall$-consistent migrations using splittable flows. While every $\forall$-consistent network update is ℓ-consistent by definition, the reverse is not true, cf. Figure 6.1. However, the problem of a lossless migration becomes intractable for ℓ-consistency, even for a single splittable flow.

ℓ-consistent migration is NP-hard The problem of checking for a ℓ-consistent migration is already hard for a single flow, due to a reduction from the Hamiltonian path problem. Essentially, in order to find an alternative path on where to temporarily store the current flow, one needs to find a longer path, which is intractable.

Theorem 19 ([29])**.** *Let $G = (V, E)$ be a graph. The problem of finding a cycle-free path of $|V| - 1$ edges in G, denoted as* Hamiltonian path*, is NP-hard.*

Theorem 20. *Deciding if a ℓ-consistent migration from F to F' exists is NP-hard.*

Proof. Let $I = (G = (V, E))$ be an instance of the Hamiltonian path problem, with $|V| = x+2$ nodes. We construct an instance I' for ℓ-consistent migration, which is solvable if and only if I contains a Hamiltonian path (of length $x + 1$). The idea of the construction is depicted in Figure 6.8.

I' contains $G = (V, E)$ and seven additional nodes s, t, z, w, w', u, u'. s has outgoing edges to all nodes of V with a latency of 1, while w has incoming edges from all nodes of V with a latency of 1 as well. Furthermore, there are edges (s, u'), (s, u), (u', u), (u, z), (z, w') and (w, t), with a latency of 1 too, but the edges (z, w) and (w', w) have a latency of x. All edges in I' have a capacity of 1. The flow F of size 1 is routed via $s - u' - u - z - w - t$, while the flow F' of size 1 is to be routed via $s - u - z - w' - w - t$.

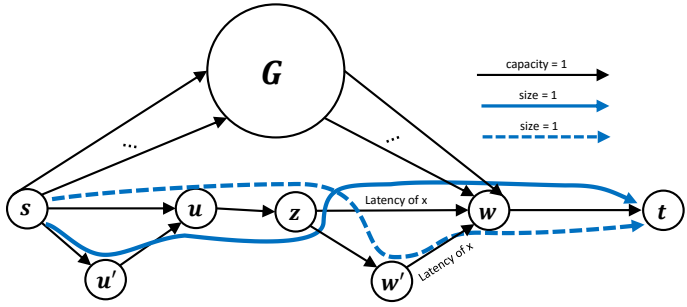


Figure 6.8: In this network, the task is to migrate in a ℓ-consistent way from the flow F, depicted in solid blue, to the flow F', depicted as a blue dashed path. All edges have a capacity of 1 and a latency of 1, except for (z, w) and (w', w) with a latency of x. The only way to migrate in a ℓ-consistent fashion is if there is a path P via G from s to w that has a length of exactly $x + 3$. Then, one can migrate F to $P - w - t$, and then to F', both of which are ℓ-consistent updates.

Observe that a network update from F to F' is not ℓ-consistent, as the capacity of (u, z) will be violated for some time. It would be possible to update F to be routed via w', but then the flow between s and w has a latency of $x + 4$ instead of $x + 3$.

As thus, the only option is to find an alternate path of length $\geq x + 3$ from s to u via G. However, such a path only exists if and only if G contains a Hamiltonian path $\mathcal{H}$ of length $x+1$ Then, one can update in a ℓ-consistent way to $s-\mathcal{H}-w-t$, and then to F', as the latency for both to w is $x+3$. □

Note that the above proof of Theorem 20 still works even if the flows are splittable (during the migration), as the problem of finding a longer path remains. Additionally, both edges of latency x could be replaced with a path of length x, where each edge has a latency of one.

Corollary 14. *Deciding if a ℓ-consistent migration from F to F' exists is NP-hard, even for splittable flows and all edges having unit latency.*

Furthermore, if the nodes u', w' are removed, then it would be NP-hard to decide if there is any alternative path at all to which one can update:

Corollary 15. *Deciding if any flow F' exists s.t. the network update (N, F, F')*

is ℓ-consistent is NP-hard, even if the flows are splittable and all edges have unit latency.

Mitigating the longest path problem If we examine the polynomial computation time migration Algorithm 11 from Section 6.5, then it leaves the tractable realm when looking for alternative intermediate paths: All the $\forall$-consistent updates still work, but with ℓ-consistency, we can also perform updates such as from the dashed to the solid flow in Figure 6.1. If we could solve the problem of finding a longest path (which is NP-hard as well, [29]), we could decide if intermediate ℓ-consistent updates exist. In general though, no better approximation ratio for the longest path than $n^{1-\epsilon}$ exists for directed graphs unless P=NP [8], even if all weights are uniform. However, there are special cases where a polynomial time algorithm for the longest path problem is possible: The easiest one is if the number of paths in total is polynomial, as then we could check all possibilities. If the path length is bounded by a constant, then one could resort to dynamic programming, as the problem is fixed-parameter tractable, e.g., [9]. Lastly, should the longest path be of at most logarithmic edge length, one can use color-coding to find it in polynomial time [2].

6.7 Summary

We give the first study of network updates for flows under the effect of edge latencies. Our measurements show that packet loss and extended end-to-end latency appear in practice, for both UDP and TCP flows – to which existing frameworks are oblivious. We perform an extensive theoretical analysis and give algorithms to check if a network update is lossless for splittable flows for both known and unknown latencies.

Should a single network update not be consistent, we explore if a sequence of lossless intermediate update exists that lead to the desired migration. By allowing the multi-commodity flows to be temporarily split along a linear number of extra paths, we can give the first complete polynomial time algorithm for lossless updates under unknown latencies. Previous approaches were not lossless, not complete, or used exponential runtimes. For the case of known latencies, we proved that already the migration of a single splittable flow is NP-hard, and give methods how to find alternative intermediate paths in practice.

7

Augmenting Flow Migration

In this chapter, we study flow migration using the concept of flow augmentation algorithms. While augmenting s-t flows of a single commodity is a well-understood concept, we study updating flows in a multi-commodity setting.

After discussing the motivation and background in Sections 7.1 and 7.2, we continue with the model Section 7.3, where we formally define the flow migration problem discussed in this chapter: Given a directed network with flows of different commodities, how can the capacity of some commodities be increased, without reducing capacities of other commodities, when moving flows in the network in an orchestrated order? To this extent, we show in Sections 7.4 how the notion of augmenting flows can be efficiently extended to multiple commodities for applications with a single logical destination.

We develop algorithms for two settings, first for consistent migration in Section 7.5, and then for lossless migration in Section 7.6, the latter trading additional updates for dealing with packets in-flight.

Maybe the most fundamental take-away message is that both consistent and lossless migration to new demands is always possible in a polynomial number of updates in this context: For the related problem of migration to a new specific flow, no consistent/lossless solution has to exist, and even if, the number of updates needed can be unbounded.

Lastly, we prove the consistent/lossless migration to new demands to be NP-hard for unsplittable flows (Section 7.7), and discuss extensions for the case of multiple source-destination pairs (Section 7.8).

7.1 Motivation

The rise of *Software Defined Networks (SDNs)* has sparked an increasing interest in applying network flow algorithms. In contrast to networks that use standardized distributed protocols, SDNs allow for utilizing the available bandwidth almost completely. The algorithmic tool to manage network traffic in an efficient way is provided by flow algorithms.

Since network traffic is highly dynamic, existing SDN solutions [11, 37, 41, 42, 50] frequently re-compute the optimal way to route traffic demands, usually using an approach based on linear programming (LP), often accepting the overhead that a new solution will re-route many existing flows that did not change their demands.

However, these approaches (and our own in the previous chapters) are oblivious to how the new demands should be managed: A new network configuration is output by some LP, and the task of the network updates is to migrate to this new configuration, no matter the cost.

In this chapter we propose to abandon these oblivious solutions in favor of path augmentation, a technique developed and studied primarily in the era of The Beatles. Even though path augmentation is still taught in introductory lectures, little research has been devoted to it since, probably because more versatile (and also polynomial-time) LP-based techniques were introduced.

We believe that SDNs should be managed in an incremental way. If a commodity (a source-destination node pair) wants to reduce its bandwidth, we can simply do that without harm. If a commodity wants to increase its bandwidth (or a new commodity is introduced to the network), we try to increase its flow by using path augmentation. Maybe we are lucky, and the

increased demand fits without changing any of the other flows. Maybe we are less lucky, and re-routing (also called migration) of some other flows is necessary, conceivably even recursively.

Understanding flow migration is still in its infancy: It is not clear in general (1) to what solution one should actually attempt to migrate, and (2) how to reasonably bound the migration time.

Moreover, there is another problem: Apart from a few exceptions that we discuss in the background section, path augmentation was only developed for *s-t*-flow problems with a *single* commodity. In real networks we have *multiple* commodities, so we first need to generalize path augmentation to *flow augmentation*, a path augmentation technique supporting multiple commodities.

It turns out that generalizing path augmentation is not as easy as one may hope. As Hu notes in his influential article [38], "*it is unlikely that similar techniques can be developed for constructing multicommodity flows*".

This is why this article focuses on an important case of multi-commodity flow, the so-called *anycast* problem, cf. [76]. In the anycast problem, we have different commodities, one for each source node. All these commodities must be routed to an arbitrary set T of destination nodes. In contrast to general multi-commodity flow problems, it does not matter which commodity ends up at which destination, as long as the destination is in the set T. Commodities may route to *any* destination of the set T, hence anycast.

Using popular terminology, think of T as the set of servers of a cloud provider; customers do not care which server gets the (potentially enormous [85]) data, as long as "the data gets to the cloud".

Applying our method of flow augmentation, we develop an efficient algorithm for consistently migrating to any desired feasible set of traffic demands. We require only one augmenting flow per commodity, minimizing network overhead. Thus, for the anycast setting, we solve both issues (1), (2) mentioned above.

Furthermore, we show that our method can also be extended to stronger notions of consistency, by adding a polynomial number of intermediate updates to the flow migration. Lastly, we also prove that the consistent migration problem to new demands turns out to be NP-hard for unsplittable flows.

7.2 Flow Augmentation Background

The notion of augmenting paths for single-commodity flows has been introduced in the seminal works of Ford and Fulkerson [21, 22], with their concepts influencing thousands of publications to this day.

Hu [38] studied augmenting paths for a two-commodity setting and generalized the results of Ford and Fulkerson to maximize the simultaneous flow of two commodities. By limiting the problem to just two commodities, he introduced so-called backward and forward paths, which together allow for an augmentation of the network.

Furthermore, in 1978, shortly before the celebrated publication of the Ellipsoid method [46], Itai [39] published an improved version of Hu's two-commodity flow algorithm and showed that maximizing a two-commodity flow is as difficult as linear programming in the sense that they are polynomially equivalent.

However, while many further results were published for multi-commodity flow problems in general and augmenting path algorithms for the case of single-commodity flow problems in particular, the application of augmenting paths to multi-commodity flow problems has been sparse.

Rothfarb et al. applied augmenting paths in the following way [71]: To maximize multi-commodity flows with just one destination t, they added a logical super-source s, considered all commodities as the same commodity with new source s, and then solved the obtained single-commodity flow problem using the standard augmenting path method. Afterwards, the single-commodity flow is split into a multi-commodity flow again, using arc-chain decomposition. Since the arc-chain decomposition is independent of the initial flow, possibly all single-commodity flows are re-routed completely, even though already a small modification might have been sufficient. Furthermore, their algorithm does not deal with the problem of asynchrony in SDNs (which is not surprising, considering the concept of SDNs was still decades away), and as thus, will induce congestion if used for migration.

7.3 Model

We consider a network as a directed graph with edge capacities. For the definition of a multi-commodity flow, we first need to define a single-commodity flow via the usual flow constraints:

Definition 24. *Let $G = (V, E)$ be a simple connected directed graph with $|V| = n$ nodes and $|E| = m$ edges. Denote the set of edges outgoing from a node $v \in V$ by $\mathtt{out}(v)$ and the set of incoming edges by $\mathtt{in}(v)$. A* network *is a pair $N = (G, c)$ where $c : E \to \mathbb{R}_+$ is a map assigning each edge a positive capacity. We call a pair of distinct nodes $s, t \in V$ a* commodity *K. We define a* single-commodity flow *for K as a map $F : E \to \mathbb{R}_{\geq 0}$ s.t.*

$$F(e) \leq c(e) \qquad \text{for all } e \in E, \tag{7.1}$$

$$\sum_{e \in \mathtt{out}(v)} F(e) = \sum_{e \in \mathtt{in}(v)} F(e) \qquad \text{for all } v \in V \setminus \{s, t\}, \tag{7.2}$$

$$\sum_{e \in \mathtt{out}(s)} F(e) = d_F = \sum_{e \in \mathtt{in}(t)} F(e), \tag{7.3}$$

where d_F is called the demand *of K (w.r.t. F). We also call d_F the* size *of F.*

We now extend the definition of a single flow to multi-commodity anycast, for which we encompass all nodes in T in a single node t. Our results can be applied analogously to the "edge reversed" model variant, where an arbitrary set of source nodes S routes multiple commodities to their assigned distinct destinations.

Definition 25. *Let N be a network and let $K_i = (s_i, t)$ be commodities where $s_1, s_2, \ldots, s_k, t \in V$ are pairwise distinct nodes. Then we call a tuple $\mathcal{K} = (K_1, K_2, \ldots, K_k)$ a* multi-commodity*. Let F_i be a flow for the commodity K_i for all $1 \leq i \leq k$. A tuple $\mathcal{F} = (F_1, \ldots, F_k)$ is called a* multi-commodity flow *for $\mathcal{K}$ if*

$$\sum_{i=1}^{k} F_i(e) \leq c(e) \qquad \text{for all } e \in E. \tag{7.4}$$

We will assume in the following that all considered flows are cycle-free. In the presented algorithms, cycles may appear temporarily, but will always be explicitly removed. In particular, this implies that $\sum_{e \in \mathtt{in}(s)} F(e) = 0 = \sum_{e \in \mathtt{out}(t)} F(e)$ if $d_F > 0$. For the sake of simplicity, we assume that this equation also holds for the case of $d_F = 0$ (which is a natural assumption from a practical point of view as we want to study the traffic from a source s to a destination t).

Definition 26. *We call a flow F* cycle-free*, if there is no directed cycle C in N s.t. $F(e) > 0$ for all $e \in C$.*

Lastly, we will need the concept of a *partial flow* in the following sections:

Definition 27. *Let F be a single-commodity flow for the commodity $K = (s,t)$ and let $v \in V$. We call a flow F' for the commodity $K' = (v,t)$ a* partial flow of F starting in v *if the following conditions hold:*

$$F'(e) \leq F(e) \qquad \text{for all } e \in E$$

$$F'(e) = F(e) \qquad \text{for all } e \in \texttt{out}(v)$$

Furthermore, we call a flow F'' for the commodity K a subflow *of F if $F''(e) \leq F(e)$ for all $e \in E$.*

Note that, since we assume all considered flows to be cycle-free, all the traffic leaving v (in the flow F) must finally end up in t which implies that (for each $v \in V$) there exists a partial flow of F starting in v.

Consistent Migration

From a conceptual point of view, our following definition of the term *consistent migration* in this chapter is the same as before in Definition 11. However, due to the nature of network model in this chapter, we slightly change the notation s.t. our upcoming proofs can be more concise and easy to follow:

Definition 28. *Let N be a network and let $\mathcal{F} = (F_1, \ldots, F_k)$, $\mathcal{F}' = (F'_1, \ldots, F'_k)$ be multi-commodity flows for the multi-commodity $\mathcal{K}$ s.t. $d_{F_i} \leq d_{F'_i}$, $1 \leq i \leq k$. The tuple $(N, \mathcal{F}, \mathcal{F}')$ is a* consistent migration update *from $\mathcal{F}$ to $\mathcal{F}'$ if*

$$\sum_{i=1}^{k} \max\left(F_i(e), F'_i(e)\right) \leq c(e) \qquad \text{for all } e \in E. \tag{7.5}$$

A consistent migration *from $\mathcal{F}$ to $\mathcal{F}^*$ is a sequence of consistent migration updates $(N, \mathcal{F}, \mathcal{F}_1), (N, \mathcal{F}_1, \mathcal{F}_2), \ldots, (N, \mathcal{F}_j, \mathcal{F}^*)$.*

Note that for each $K \in \mathcal{K}$ the demand of K w.r.t $\mathcal{F}, \mathcal{F}_1, \ldots, \mathcal{F}_j, \mathcal{F}^*$ is non-decreasing. If the demand of flows was smaller in $\mathcal{F}'$, then one would drop corresponding parts of the flows before migration.

7.4 Augmenting Flows for Multiple Commodities

In the case of one source and one destination, it is well-known [22] how to use an obtained augmenting path P in order to transform a given flow into a new enhanced flow whose size is increased by the "capacity" of P. When we have multiple sources, the "standard" augmenting path does not account for moving multiple commodities at once, since it is only defined to modify the flow of a single commodity.

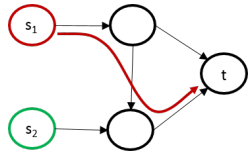
In the following Definition 29, we define an augmenting flow for the case of a multi-commodity flow where we have multiple sources (but only one destination). The augmenting flow may use edges from the residual network, which is created by adding a back-edge in the reverse direction for every edge with some flow on it, cf. the dashed edges in Subfigure 7.1b. Note that while the augmenting flow may use these back-edges, there will be never any "real" flow routed over these edges, as they are not part of the physical network and just used for our algorithms.

We further introduce the notion of a *farthest back-edge* which is a back-edge used by the augmenting flow "after which" the augmenting flow only uses forward edges.

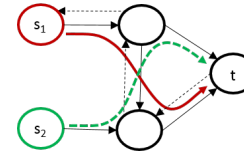
Definition 29. *Let N be a network and let $\mathcal{F}$ be a multi-commodity flow for the multi-commodity $\mathcal{K}$. We denote by $\overline{G}$ the graph obtained from G by adding an edge $e^* = (v, u)$ to G for any edge $e = (u, v) \in E$. Let E^* be the set of all newly added edges. If an edge $e^* \in E^*$ starts and ends in the same vertices as some edge in E, we still consider them as distinct edges. Set $\overline{N} := \left(\overline{G}, \overline{c}\right)$ where $\overline{c}(e^*) := \overline{c}(e) := c(e)$ for all $e \in E$. Let $K \in \mathcal{K}$. We call a cycle-free (single-commodity) flow F_A for K in $\overline{N}$ an* augmenting flow *w.r.t. $\mathcal{F}$ if $F_A(e) \leq c(e) - \sum_{F \in \mathcal{F}} F(e)$ and $F_A(e^*) \leq \sum_{F \in \mathcal{F}} F(e)$ for all $e \in E$. Set $E^*_{F_A} := \{e^* \in E^* | F_A(e^*) > 0\}$. We call an edge $(u, v) \in E^*_{F_A}$ a* farthest back-edge *if there is no path P from v to t s.t. for all edges $e \in P$ we have $F_A(e) > 0$ and there is an edge $e^* \in P$ with $e^* \in E^*_{F_A}$. Since F_A is cycle-free, such a farthest back-edge exists if $E^*_{F_A} \neq \emptyset$.*

Note that in this article, such an augmenting flow always "belongs" to a specific commodity K contained in the respective multi-commodity.

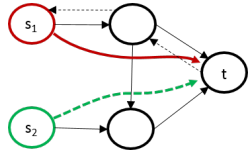
We develop a technique in the following Algorithm 12 to transform the given multi-commodity flow step by step into a multi-com-modity flow where the flow size for K is increased by the size of the augmenting flow, see



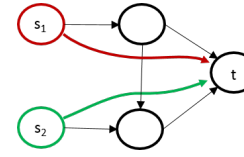
(a) Initial network with just one flow from s_1 to t. Currently, there is no space for a flow from s_2 to t, the red flow needs to be moved first.



(b) An augmenting flow for s_2 is found from s_2 to t, using a dashed edge in $\overline{G}$ that pushes the red flow back to the top.



(c) After the red flow has been pushed to the top, the resulting green augmenting flow uses only "real" edges from the network. Thus, in a next step, it can be replaced with a proper "real" flow.



(d) The resulting flows are feasible and use only edges in the "real" network, none of the (hidden) dashed ones from $\overline{G}$.

Figure 7.1: In this small introductory example network to illustrate augmenting flows, all edges have a capacity of one and all flows have a size of one. The solid edges are the "real" edges in the network N, while the dashed edges in Subfigure 7.1b exist just in $\overline{G}$: A reverse edge for every edge with some flow on it. Dashed edges from $\overline{G}$ are never used for routing, they are just used to find augmenting flows. If the task is to add a flow from s_2 to t, then one searches for an (augmenting) flow from s_2 to t – but not just in N, the dashed edges from $\overline{G}$ are allowed as well.

Theorem 21. A very small introductory example is given in Figure 7.1. We show that the transformation steps correspond to consistent migration updates, thus proving that the new (multi-commodity) flow can be obtained from the old one by a consistent migration.

The general idea is as follows: Given a multi-commodity flow and an augmenting flow, Algorithm 12 will perform a consistent migration update (Lemma 17) in the network.[1] Essentially, one execution of Algorithm 12 will process one edge of the augmenting flow. As the augmenting flow can have at most $m = |E|$ edges, the augmenting flow will be inserted consistently after a linear number of iterations of the algorithm (Theorem 21). We refer to Figure 7.2 for an advanced illustration of Algorithm 12.

In the following, we will state and prove various lemmas to lastly prove Theorem 21 in this section. We begin with Lemma 12 and Lemma 13, which state that the new augmenting flow F'_A will not violate the capacity constraints set in Definition 29:

Lemma 12. *$F'_A(e) + \sum_{F' \in \mathcal{F}'} F'(e) \leq c(e)$ for all $e \in E$.*

Proof. Since F_A is an augmenting flow w.r.t. $\mathcal{F}$, it holds that $F_A(e) + \sum_{F \in \mathcal{F}} F(e)$
$\leq c(e)$ for all $e \in E$. Thus, after step 1 we have $F'_A(e) + \sum_{F' \in \mathcal{F}'} F'(e) \leq c(e)$ for all $e \in E$. In step 2, $F'_A(e) + \sum_{F' \in \mathcal{F}'} F'(e)$ remains unchanged for all $e \in E$. In steps 3, 4, and 5 this is also the case since for each $e \in E$, $F'_A(e)$ is diminished by the same amount by which $\sum_{F' \in \mathcal{F}'} F'(e)$ grows larger, resp. vice versa. As the cycle removals in step 6 can only diminish the above sum, we obtain Lemma 12. □

Lemma 13. *$F'_A(e^*) \leq \sum_{F' \in \mathcal{F}'} F'(e)$ for all $e \in E$.*

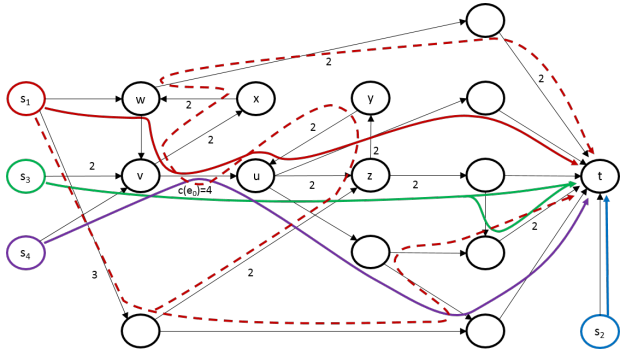
Proof. Since F_A is an augmenting flow w.r.t. the multi-commodity flow $\mathcal{F}$, it holds for $F'_A(e^*)$ that $F'_A(e^*) \leq \sum_{F' \in \mathcal{F}'} F'(e)$ for all $e \in E$ after step 1. In step 2, both $\sum_{F' \in \mathcal{F}'} F'(e_0)$ and $F'_A(e_0^*)$ are diminished by $F'_A(e_0^*)$, while nothing changes for the edges $e \neq e_0$. In step 3, $\sum_{F' \in \mathcal{F}'} F'(e)$ cannot decrease, while $F'_A(e^*)$ cannot be increased. Thus, at this point, $F'_A(e^*) \leq \sum_{F' \in \mathcal{F}'} F'(e)$ still holds for all $e \in E$. In step 4, the left hand side of the

[1] The calculation of the corresponding augmenting flow for Algorithm 12 is discussed in Section 7.5. Essentially, we will calculate one augmenting flow per commodity that needs to be augmented, and apply Algorithm 12 sequentially.

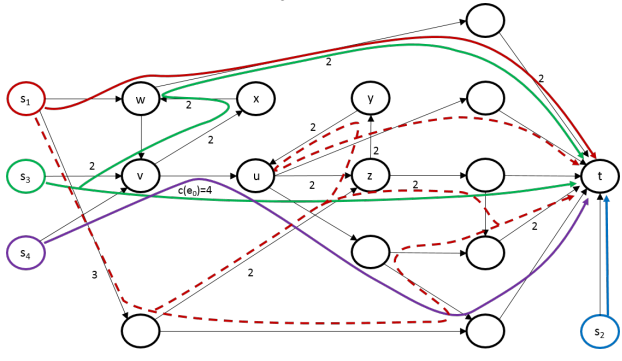
Let N be a network and $\mathcal{F} = (F_1, \ldots, F_k)$ be a multi-commodity flow for the multi-commodity $\mathcal{K} = (K_1, \ldots, K_k)$. Let F_A be an augmenting flow w.r.t. $\mathcal{F}$ for some commodity $(s,t) = K \in \mathcal{K}$. Let $E^*_{F_A}$ be non-empty and let $(u_0, v_0) = e_0^* \in E^*_{F_A}$ be a farthest back-edge. Let $F_{k_1}, ..., F_{k_q} \in \mathcal{F}, k_1 < \cdots < k_q$ be the flows[2] which assign the edge e_0 (i.e., the edge which induced the adding of e_0^* to G) a non-zero value, i.e., the flows which are present on this edge. Let r be the smallest index such that $\sum_{z=1}^{r} F_{k_z}(e_0) \geq F_A(e_0^*)$. Set $U := F_A(e_0^*) - \sum_{z=1}^{r-1} F_{k_z}(e_0)$. We migrate to a new multi-commodity flow $\mathcal{F}' = (F'_1, \ldots, F'_k)$ for $\mathcal{K}$ and a new augmenting flow F'_A w.r.t. $\mathcal{F}'$ as follows:

1. Begin by setting $F'_y(e) := F_y(e)$ for all $e \in E$ and all $1 \leq y \leq k$, and $F'_A(e) := F_A(e)$ for all $e \in E \cup E^*$.
2. Redefine $\mathcal{F}'$ on e_0 and F'_A on e_0^*: Set $F'_{k_z}(e_0) := 0$ for all $1 \leq z \leq r-1$, $F'_{k_r}(e_0) := F'_{k_r}(e_0) - U$, and $F'_A(e_0^*) := 0$.
3. Choose a partial flow of F_A starting in v_0 and choose a subflow F_a (of this partial flow) of size $F_A(e_0^*)$. (Note that $F_a(e^*) = 0$ for all $e^* \in E^*$ because e_0^* is a farthest back-edge.) Decompose F_a in r subflows $F_a^{(1)}, \ldots, F_a^{(r)}$ of sizes $F_{k_1}(e_0), \ldots, F_{k_{r-1}}(e_0), U$ such that, for each edge $e \in E$, we have $\sum_{z=1}^{r} F_a^{(z)}(e) = F_a(e)$. Now set $F'_{k_z}(e) := F'_{k_z}(e) + F_a^{(z)}(e)$ for all $1 \leq z \leq r$ and all $e \in E$, and set $F'_A(e) := F'_A(e) - F_a(e)$ for all $e \in E$.
4. For all $1 \leq z \leq r$, choose a partial flow of F_{k_z} starting in u_0 and choose a subflow $F^{(z)}$ (of this partial flow) of size $F_{k_z}(e_0)$ if $z \neq r$ and of size U if $z = r$. Then replace these subflows by the augmenting flow, i.e., set $F'_{k_z}(e) := F'_{k_z}(e) - F^{(z)}(e)$ for all $1 \leq z \leq r$ and all $e \in E$, and set $F'_A(e) := F'_A(e) + \sum_{z=1}^{r} F^{(z)}(e)$ for all $e \in E$.
5. Replace possible cycles for flows in $\mathcal{F}'$ by cycles for F'_A: If there is some flow $F' \in \mathcal{F}'$ which is not cycle-free, then find a (directed) cycle C s.t. $F'(e) > 0$ for all $e \in C$. Set $F'(e) := F'(e) - \min_{e' \in C} F'(e')$ for all $e \in C$, thus "removing" the cycle, and set $F'_A(e) := F'_A(e) + \min_{e' \in C} F'(e')$ for all $e \in C$. Continue removing (and replacing) cycles in this way (for all flows in $\mathcal{F}'$) until there are no cycles left in $\mathcal{F}'$. (Note that the removal of a cycle implies that there is some edge e which changes in the process from $F'(e) > 0$ to $F'(e) = 0$. Thus, all flows contained in $\mathcal{F}'$ are cycle-free after removing at most $O(mk)$ cycles.)
6. Remove possible cycles for F'_A: First remove cycles for the flow F'_A which consist only of an edge $e \in E$ and its corresponding edge $e^* \in E^*$, until no such cycles remain. Subsequently, remove arbitrarily chosen cycles for F'_A iteratively until F'_A is cycle-free. Analogously to the above, at most $O(m)$ cycles need to be removed.

Algorithm 12: Edge augmentation algorithm



(a) Multi-commodity flow $\mathcal{F} = (F_1, F_2, F_3, F_4)$ and augmenting flow F_A, *before* applying Algorithm 12 w.r.t. e_0^*.



(b) Multi-commodity flow $\mathcal{F}' = (F_1', F_2', F_3', F_4')$ and augmenting flow F_A, *after* applying Algorithm 12.

Figure 7.2: In this example network, all unmarked edges have a capacity of one. The green flow F_3 starting in s_3 has a size of two, all other solid flows have a size of one. The dashed augmenting flow F_A for the commodity K_1 starting in s_1 has a size of three. In Subfigure 7.2a, the (not drawn) edge $(u, v) = e_0^*$ in $\overline{N}$ is a farthest back-edge. When executing Algorithm 12, we obtain $q = 3$, $k_1 = 1, k_2 = 3, k_3 = 4$, $F_A(e_0^*) = 2$, $r = 2$, and $U = 1$. A part of the augmenting flow is re-routed in the node u via former paths of F_1 and F_3, whereas F_1 and half of F_3 are re-routed in the node v via a former path of the augmenting flow. The occurring cycles wvx and zyu are removed afterwards by Algorithm 12.

inequality is not increased, since $e^* \notin E$. As in this step $F'_A(e)$ is increased by the same amount by which $\sum_{F' \in \mathcal{F}'} F'(e)$ is diminished, we obtain $F'_A(e^*) \leq \sum_{F' \in \mathcal{F}'} F'(e) + F'_A(e)$, for all $e \in E$ after step 4. By an analogous argument, this new inequality holds also after step 5. After removing the "small" cycles in the first part of step 6 we have $F'_A(e^*) = 0$ or $F'_A(e) = 0$ for all $e \in E$, while the new inequality still holds. As the subsequent cycle removals in the second part of step 6 cannot decrease $\sum_{F' \in \mathcal{F}'} F'(e)$ to less than 0, the inequality given in Lemma 13 holds for all $e \in E$ with $F'_A(e^*) = 0$. Thus, consider the edges $e \in E$ with $F'_A(e) = 0$. For these edges, $F'_A(e^*) \leq \sum_{F' \in \mathcal{F}'} F'(e) + F'_A(e)$ implies $F'_A(e^*) \leq \sum_{F' \in \mathcal{F}'} F'(e)$ which yields the desired statement of Lemma 13 since the cycle removals in the second part of step 6 cannot decrease $\sum_{F' \in \mathcal{F}'} F'(e)$. □

Next, in Lemma 14 and 15, we show that the flows adhere to the flow conditions and keep their demand unchanged.

Lemma 14. *F'_A is a (single-commodity) flow for the commodity K and $d_{F'_A} = d_{F_A}$.*

Proof. We first show that F'_A is non-negative on all edges in $E \cup E^*$ and then that F'_A satisfies Conditions (7.1)–(7.3) from Definition 24.

The only step where F'_A can switch to a negative value on some edge $e \in E \cup E^*$ is step 3 and this can only be the case if $e \in E$. But since F_a is a subflow of a partial flow of F_A, we have $F_a(e) \leq F'_A(e)$ for all $e \in E$ and $F'_A(e)$ remains non-negative in step 3. Note that at the beginning of step 3, it holds that $F'_A(e) = F_A(e)$ for all $e \in E$.

By an analogous argument, $F'_y(e)$ is non-negative for all $1 \leq y \leq k$ and all $e \in E$. Since, in addition, $F'_A(e^*)$ is never increased in steps 2 to 6 for all $e^* \in E^*$ (which implies $F'_A(e^*) \leq F_A(e^*)$), Condition (7.1) holds due to Lemma 12 and Definition 29.

We will now show that Conditions (7.2) and (7.3), i.e., flow conservation and demand satisfaction, are maintained. Consider $D_A(v) := \sum_{e \in \mathtt{in}(v)} F'_A(e) - \sum_{e \in \mathtt{out}(v)} F'_A(e)$ for all $v \in V$. After step 1, $D_A(v) = 0$ for all $v \in V \setminus \{s, t\}$, and $-D_A(s) = D_A(t) = d_{F_A}$. However, in step 2, $D_A(u_0)$ is increased by $F'_A(e_0^*)$ and $D_A(v_0)$ is diminished by $F'_A(e_0^*)$. In step 3, $D_A(v_0)$ is increased by $F'_A(e_0^*)$, $D_A(t)$ gets diminished by $F'_A(e_0^*)$, and $D_A(v)$ remains unchanged for all other nodes v. In step 4, $D_A(u_0)$ is diminished by $F'_A(e_0^*)$,

$D_A(t)$ gets increased by $F'_A(e^*_0)$, and $D_A(v)$ remains unchanged for all other nodes v. Lastly, the replacement of cycles in step 5 and the removal of cycles in step 6 do not change any $D_A(v)$. Thus, $D_A(v) = 0$ for all $v \in V \setminus \{s,t\}$, and $-D_A(s) = D_A(t) = d_{F_A}$. Since F'_A is cycle-free, this implies $\sum_{e\in\mathtt{out}(s)} F'_A(e) = d_{F_A} = \sum_{e\in\mathtt{in}(t)} F'_A(e)$, i.e., $d_{F'_A} = d_{F_A}$. □

Lemma 15. *$\mathcal{F}'$ is a multi-commodity flow for $\mathcal{K}$ and $d_{F'_y} = d_{F_y}$ for all $1 \le y \le k$.*

Proof. Lemma 15 follows by a proof analogous to the proof of Lemma 14. Note that Condition (7.4) holds due to Lemma 12. □

Combining Lemmas 12 to 15, we obtain the following corollary:

Corollary 16. *F'_A is an augmenting flow w.r.t. $\mathcal{F}'$.*

Furthermore, we need to prove that Algorithm 12 actually makes progress, i.e., at least one of the edges e^* has an augmenting flow of zero afterwards.

Lemma 16. *The number of edges $e^* \in E^*$ with $F'_A(e^*) > 0$ is strictly smaller than the number of edges $e^* \in E^*$ with $F_A(e^*) > 0$.*

Proof. As observed in the proof of Lemma 14, we have $F'_A(e^*) \le F_A(e^*)$ for all $e^* \in E^*$. Thus, $F'_A(e^*) > 0$ implies $F_A(e^*) > 0$. Moreover, $F_A(e^*_0) > 0$, but $F'_A(e^*_0) = 0$ (due to step 2). The result follows. □

Lastly, we show that the update performed by Algorithm 12 is actually consistent:

Lemma 17. *$(N, \mathcal{F}, \mathcal{F}')$ is a consistent migration update.*

Proof. By Lemma 15, we only have to show that Condition 7.5 holds, i.e., that for all $e \in E$: $\sum_{y=1}^{k} \max\left(F_y(e), F'_y(e)\right) \le c(e)$. Let e be an arbitrary edge in E. We observe that, for all $1 \le y \le k$, the only step (after setting $F'_y(e) := F_y(e)$) where $F'_y(e)$ gets possibly increased is step 3. Moreover, a positive increase in step 3 is only possible if $y = k_z$ for some $1 \le z \le r$. More specifically, we have $F'_{k_z}(e) \le F_{k_z}(e) + F_a^{(z)}(e)$ after step 6 for all $1 \le z \le r$. Since $F_a^{(z)}(e) \ge 0$ for all $1 \le z \le r$, we obtain

$$\sum_{y=1}^{k} \max\left(F_y(e), F'_y(e)\right) \le \sum_{y=1}^{k} F_y(e) + \sum_{z=1}^{r} F_a^{(z)}(e) = F_a(e) + \sum_{y=1}^{k} F_y(e) \ .$$

Since F_a is a subflow of a partial flow of F_A (compare step 3), we have $F_a(e) \leq F_A(e)$. Thus,

$$\sum_{y=1}^{k} \max\left(F_y(e), F'_y(e)\right) \leq F_A(e) + \sum_{y=1}^{k} F_y(e) \leq c(e) \ .$$

The last inequality follows since F_A is an augmenting flow w.r.t. $\mathcal{F}$. □

We can now prove Theorem 21, which states that we can update consistently to the new augmented flow in just a linear number of updates[3], resulting from applying the augmenting flow:

Theorem 21. *Let N be a network and let $\mathcal{F} = (F_1, \ldots, F_k)$ be a multi-commodity flow for the multi-commodity $\mathcal{K} = (K_1, \ldots, K_k)$. Let F_A be an augmenting flow w.r.t. $\mathcal{F}$ for the commodity $(s,t) = K_x \in \mathcal{K}$ where $1 \leq x \leq k$. Then there is a multi-commodity flow $\mathcal{F}^*$ for $\mathcal{K}$ s.t. $d_{F^*_x} = d_{F_x} + d_{F_A}$ and $d_{F^*_y} = d_{F_y}$ for all $1 \leq y \leq k$ with $y \neq x$. Moreover, there is a consistent migration from $\mathcal{F}$ to $\mathcal{F}^*$, consisting of at most $m+1$ consistent migration updates.*

Proof. W.l.o.g., let $h \in \mathbb{N}$ be the number of times that the steps 1 to 6 of Algorithm 12 are performed until, for the resulting augmenting flow F^h_A w.r.t. the resulting multi-commodity flow $\mathcal{F}^h$, there is no edge $e^* \in E^*$ with $F^h_A(e^*) > 0$. Note that, due to Lemma 16, it holds that $h \leq m$. Thus, by Lemmas 15 and 17, every one of the h iterations of the steps 1 to 6 corresponds to a consistent migration update. Moreover, $d_{F^h_y} = d_{F_y}$ for all $1 \leq y \leq k$, by Lemma 15, and $F^h_A(e) + \sum_{F^h \in \mathcal{F}^h} F^h(e) \leq c(e)$ for all $e \in E$, by Lemma 12. Therefore, increasing $F^h_x(e)$ by $F^h_A(e)$ for all $e \in E$ corresponds to a consistent migration update and the resulting multi-commodity flow $\mathcal{F}^*$ satisfies the conditions given in Theorem 21. □

7.5 Consistent Flow Migration with Augmentation

A standard approach in the single-commodity case for increasing the size of a flow is to compute augmenting paths, apply them to the network, and

[3]We note that other mechanisms such as, e.g., *SWAN* [37] or *zUpdate* [50], do not give *any* bound on the number of updates needed for consistent migration.

iterate this process until the desired demand is reached, if possible. However, this method requires a lot of updates in the network itself, as the number of augmenting paths needed can be linear in the number of edges. Due to the fact that we augment our multi-commodity flow in Section 7.4 with a *flow* instead of a *path*, in our framework just one augmenting flow per commodity suffices to satisfy any possible new demands, as we show in this section.

While a linear programming solution does not show how to migrate the network consistently, we can use LPs to compute the augmenting flows needed for the consistent migration. For our method we will first need the notion of *difference flows*, which are flows obtained by "subtracting" a multi-commodity flow from another.

Definition 30. *Let N be a network, let $\mathcal{K} = (K_1, \ldots, K_k)$ be a multi-commodity, and fix some $i \in \mathbb{N}$ with $1 \leq i \leq k$. Let $\mathcal{F} = (F_1, \ldots, F_k), \mathcal{F}' = (F'_1, \ldots, F'_k)$ be multi-commodity flows for $\mathcal{K}$ with $d_{F_i} < d_{F'_i}$ and $d_{F_j} = d_{F'_j}$ for all $1 \leq j \leq k$, $j \neq i$.*

We define a difference flow $Z^{\mathcal{F},\mathcal{F}'}$ *for $\mathcal{F}$ and $\mathcal{F}'$ in $\overline{N}$ as follows: First, for all $e \in E$: i) If $\sum_{y=1}^{k} F'_y(e) - \sum_{y=1}^{k} F_y(e) \geq 0$, then $Z^{\mathcal{F},\mathcal{F}'}(e) := \sum_{y=1}^{k} F'_y(e) - \sum_{y=1}^{k} F_y(e)$ and $Z^{\mathcal{F},\mathcal{F}'}(e^*) := 0$. ii) If $\sum_{y=1}^{k} F'_y(e) - \sum_{y=1}^{k} F_y(e) < 0$, then set $Z^{\mathcal{F},\mathcal{F}'}(e^*) := -\left(\sum_{y=1}^{k} F'_y(e) - \sum_{y=1}^{k} F_y(e)\right)$ and $Z^{\mathcal{F},\mathcal{F}'}(e) := 0$. Second, remove cycles until $Z^{\mathcal{F},\mathcal{F}'}$ is cycle-free.*

A difference flow is also an augmenting flow:

Lemma 18. *Let $Z^{\mathcal{F},\mathcal{F}'}$ be a difference flow for $\mathcal{F}$ and $\mathcal{F}'$ with $d_{F_i} < d_{F'_i}$. Then, $Z^{\mathcal{F},\mathcal{F}'}$ is an augmenting flow w.r.t. $\mathcal{F}$ for the commodity K_i of size $d_{F'_i} - d_{F_i} > 0$.*

Proof. Recall that $\mathcal{F}$ and $\mathcal{F}'$ are multi-commodity flows in N.

We start by checking the conditions for an augmenting flow given in Definition 29. For all $e \in E$, we have $Z^{\mathcal{F},\mathcal{F}'}(e) \leq \sum_{y=1}^{k} F'_y(e) - \sum_{y=1}^{k} F_y(e) \leq c(e) - \sum_{y=1}^{k} F_y(e)$ in case i), and $Z^{\mathcal{F},\mathcal{F}'}(e) = 0 \leq c(e) - \sum_{y=1}^{k} F_y(e)$ in case ii). For all $e^* \in E^*$, we have $Z^{\mathcal{F},\mathcal{F}'}(e^*) = 0 \leq \sum_{y=1}^{k} F_y(e)$ in case i), and $Z^{\mathcal{F},\mathcal{F}'}(e^*) \leq -\left(\sum_{y=1}^{k} F'_y(e) - \sum_{y=1}^{k} F_y(e)\right) \leq \sum_{y=1}^{k} F_y(e)$ in case ii). Note that removing cycles can never increase the flow on any edge.

As $Z^{\mathcal{F},\mathcal{F}'}$ is cycle-free, it is only left to show that $Z^{\mathcal{F},\mathcal{F}'}$ is a single-commodity flow for K_i in $\overline{N}$ of size $d_{F'_i} - d_{F_i}$. Condition (7.1) follows directly from the previous considerations. The definition of $Z^{\mathcal{F},\mathcal{F}'}$ ensures that Condition (7.2) is satisfied for all nodes except s_i and t. Note that removing cycles does not change the difference between the amount of outgoing and incoming flow for a node.

As $d_{F'_i} - d_{F_i} > 0$ holds due to the construction of $Z^{\mathcal{F},\mathcal{F}'}$ and as all cycles were removed from $Z^{\mathcal{F},\mathcal{F}'}$, we obtain that $\sum_{e \in \mathtt{out}(s_i)} Z^{\mathcal{F},\mathcal{F}'}(e) = d_{F'_i} - d_{F_i} = \sum_{e \in \mathtt{in}(t)} Z^{\mathcal{F},\mathcal{F}'}(e)$, i.e., Condition (7.3) holds and $Z^{\mathcal{F},\mathcal{F}'}$ is an augmenting flow for the commodity K_i of size $d_{F'_i} - d_{F_i} > 0$. □

In the following Algorithm 13, we will show how any desired demands can be obtained by consistent migration, if there is a multi-commodity flow satisfying these demands.

Let N be a network and let $\mathcal{F}$ be a multi-commodity flow for the multi-commodity $\mathcal{K}$. Let $(d_1, \ldots, d_k)$ be a vector of demands s.t. *i*) there exists a multi-commodity flow for $\mathcal{K}$ satisfying these demands, and *ii*) $d_1 \geq d_{F_1}, \ldots, d_k \geq d_{F_k}$.

1. Compute a multi-commodity flow $\mathcal{F}'_1$ with a demand vector of $(d_1, d_{F_2}, \ldots, d_{F_k})$ using an LP.
2. Compute the difference flow $Z^{\mathcal{F},\mathcal{F}'_1}$.
3. Augment $\mathcal{F}$ with $Z^{\mathcal{F},\mathcal{F}'_1}$ by using Algorithm 12 iteratively until no more back-edges exist for the resulting augmenting flow F_A. Then, replace F_A by a flow of commodity K_1 and eliminate all cycles for commodity K_1, thereby obtaining some flow $\mathcal{F}_1$ with a demand vector of $(d_1, d_{F_2}, \ldots, d_{F_k})$.
4. Iterate the above three steps for the remaining commodities $K_2, \ldots, K_k$, thereby obtaining the flows $\mathcal{F}_2, \ldots, \mathcal{F}_k$ with the demand vectors of $(d_1, d_2, d_{F_3}, \ldots, d_{F_k})$, $\ldots$, $(d_1, d_2, d_3, \ldots, d_{k-1}, d_{F_k})$, $(d_1, \ldots, d_k)$.

Algorithm 13: Performing consistent migration to new demands via augmenting flows

Corollary 17. *Algorithm 13 performs a consistent migration from $\mathcal{F}$ to some multi-commodity flow with a demand vector of $(d_1, \ldots, d_k)$, using only k augmenting flows.*

We note that Algorithm 13 can be used for any imaginable purpose, as long as the respective desired demand vector (for which some flow exists) can be computed. Common examples in practice are maximizing the sum of all commodities or reaching max-min fairness. The respective desired demand vectors can be computed with an LP, cf., e.g., [1] [14]. If the computation time is an issue as well, one can also resort to approximation algorithms with a better runtime [32].

Furthermore, the actual updates performed in the network itself are expensive, while "off-line" computations are cheap regarding the execution time in SDNs, rendering the computation overhead induced by the LPs to be bearable in practice.

7.6 Strongly Consistent Flow Migration

In Definition 11 and Definition 28 we defined the consistency model for capacity constraints as proposed in [37]. A main motivation behind this model is that changes in the flow should not violate the capacity of any edge, no matter what "mix" of old and new flow rules is currently in place due to asynchrony. However, the model only considers specific discrete points in time: Either a flow F_i has changed completely or it has not. Thus, the impact of latency on the different routes is neglected, as shown in Chapter 6.

Subsequently, we developed a framework in Chapter 6 to obtain lossless updates, differentiating between fixed and arbitrary latencies. We now propose an even stronger consistency model: When a (part of a) flow changes its route at some node due v to a network update, we consider the node v to be a new virtual source that duplicates the flow along the old and the new route. Only if both the old and the new flow fit into the network at the same time, we allow the network update.

In other words, a part of the old flow that coincides with a part of the new flow is left in the network unchanged. The remaining part B of the old flow is migrated to the remaining part B' of the new flow, but we only consider this migration to be *strongly consistent* if B' can be added to the complete old flow without violating any edge capacity constraints. This ensures that there cannot be any congestion due to latency. What is required for the migrated part B' of the new flow is that any packets in B' stay in B' until

they reach their destination (otherwise, again, congestion can occur due to latency). In other words, we require B' to be a *half flow* as given by the following definition:

Definition 31. *A* half flow *is a function $H : E \to \mathbb{R}_{\geq 0}$ s.t. for all nodes $v \in V \setminus \{s\}$ it holds that $\sum_{e \in in(v)} H(e) \leq \sum_{e \in out(v)} H(e)$. A* multi-commodity half flow *is a tuple $\mathcal{H} = (H_1, \ldots, H_k)$ of half flows.*

By formalizing the above considerations, we obtain a stronger consistency model:

Definition 32. *Let N be a network and let $\mathcal{F} = (F_1, \ldots, F_k)$ and $\mathcal{F}' = (F'_1, \ldots, F'_k)$ be multi-commodity flows for the multi-commodity $\mathcal{K}$ s.t. $d_{F_i} \leq d_{F'_i}$, $1 \leq i \leq k$. A consistent migration update from $\mathcal{F}$ to $\mathcal{F}'$ is called a* strongly consistent migration update *if there exists a multi-commodity half flow $\mathcal{H} = (H_1, \ldots, H_k)$ s.t. the following three conditions hold:* 1) *$H_i(e) \leq F'_i(e)$ for all $e \in E$ and all $1 \leq i \leq k$,* 2) *$F'_i(e) - H'_i(e) \leq F_i(e)$ for all $e \in E$ and all $1 \leq i \leq k$, and* 3) *$\mathcal{F}(e) + \sum_{i=1}^{k} H_i(e) \leq c(e)$ for all $e \in E$.*

A strongly consistent migration *is a sequence of strongly consistent migration updates.*

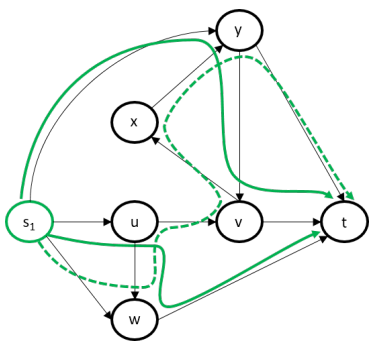
Condition 1) ensures that each half flow is contained in the respective new flow, Condition 2) ensures that when subtracting the half flow from the respective new flow, the result is contained in the respective old flow (thus, it can remain in the network unchanged) and Condition 3) ensures that even if the old flow is still completely present in the network, the multi-commodity half flow can be added without violating any edge capacity constraints.

Note that every strongly consistent update is lossless as well. However, as we will show, we do not need to weaken our strong consistency model to completely cover the space of lossless updates, both for fixed and arbitrary latencies: As long as the new demands fit into the network, we can *always* migrate to these demands in a strongly consistent fashion.

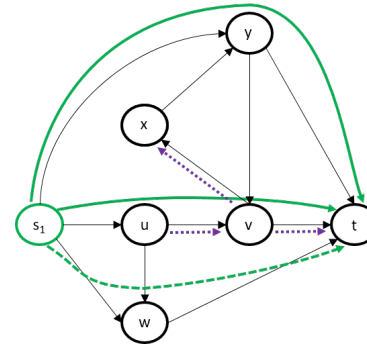
Extending our previous work to strong consistency

As Figure 7.3 shows, the migration updates performed by Algorithm 12 are not necessarily strongly consistent.

In the following, we show how to adapt Algorithm 12 in order to make the performed migration update strongly consistent. In a way, the point where the strong consistency breaks in Algorithm 12 is the replacement/removal



(a) Initial network with a solid green flow F of size 2 and a dashed augmenting flow F_A for F of size 1 via s_1, w, u, v, x, y, t. All edges have a capacity of 1.



(b) Network with F' after applying a consistent update via Algorithm 12. The purple arrows depict where strong consistency cannot be maintained.

Figure 7.3: In this small example, Algorithm 12 updates directly from F in Subfigure 7.3a to F' in Subfigure 7.3b. While the update is consistent, it is not strongly consistent: The purple arrows in Subfigure 7.3b from u to v, from v to x, and from v to t depict where strong consistency cannot be maintained: Assume that the update is strongly consistent and a half flow as required exists. Due to Condition 1), the half flow on (v, x) needs to be zero. Due to Condition 2), the half flow on (u, v) needs to be positive. Lastly, due to Condition 3), the half flow on (v, t) needs to be zero as well. Hence the half flow definition is violated at node v, since the incoming half flow is positive, but all outgoing half flow is zero.

of cycles in steps 5 and 6. More precisely, if there are no cycles in any flow in $\mathcal{F}$ after step 4, then the performed update is strongly consistent as the flows added in step 3 can be taken as the multi-commodity half flow whose existence is required. The design of Algorithm 12 asserts immediately that the flow added in step 3 indeed satisfies the half flow conditions. Note that cycles occurring in the augmenting flow F_A do not change this assessment and therefore do not have to be avoided specifically.

Our adapted algorithm proceeds as follows: Starting from the destination t, find a first back-edge along the augmenting flow. However, now, starting from this farthest back-edge, we will consider each commodity indi-

vidually, and augment along the farthest cycle beyond this back-edge (seen from s), made up of the augmenting flow and the respective commodity.

If one arc of the cycle is constituted by the augmenting path and the remaining arc by the chosen commodity (i.e., the cycle consists only of two continuous segments of commodity or augmenting flow), then we can augment along a cycle analogously to augmenting w.r.t. a back-edge. As we will show, there is always a farthest cycle of this kind.

Initially starting from t, in each of these augmenting updates, we will progress closer towards the farthest back-edge, guaranteeing that the number of updates is polynomial. As such, we need a more fine-grained Algorithm 14, that handles these updates.

For ease of readability, we give a high-level description of the algorithm:

We will now show that Algorithm 14 performs a strongly consistent migration for a given multi-commodity flow and a corresponding augmenting flow F_A belonging to commodity K. Recall that all flows are cycle-free and observe that the design of the algorithm prevents the occurrence of flows for any commodity. The only cycles occurring are those containing augmenting flow, which are subsequently deleted.

Further observe that in step 4, F_a can be decomposed into at most m augmenting paths by, e.g., iteratively choosing a path F_a^j in F_a from u to t s.t. there exists some edge e'' with $F_a^j(e'') = F_a(e'')$ and removing F_a^j from F_a.

After every execution of step 7 (and also during every execution of 5), the respective augmenting flow F_a^j has the following property $\mathcal{Q}$: There is a node w (after step 7 we have $w = u'$) s.t. all edges beyond w are marked and all edges before w are not marked, with the flow F_a^j constituting the same simple path between u and w as in the last iteration of steps 5 to 7. Note that even though we start with an unsplitted flow F_a^j in step 5, the augmenting flow F_a^j beyond w may be splitted due to the performed augmentations.

Before showing property $\mathcal{Q}$, we assume for now that $\mathcal{Q}$ holds, in order to show two things: First, we show that in step 6, when choosing a cycle, we can actually choose a cycle as described in step 6.

Property $\mathcal{Q}$ guarantees the existence of at least one such cycle: Just choose some arbitrary cycle and then follow the continuous part of the cycle consisting of commodity K_i from u' to the other end, given by some node x. Then there must be a path from u' to x consisting of marked edges, which

1. Choose a farthest back-edge $(u, v) = e^*$.
2. Choose a commodity K_i.
3. Choose a partial flow F_a of F_A starting from u of size $\min(F_i(e), F_A(e^*))$ s.t. $F_a(e^*) = \min(F_i(e), F_A(e^*))$.
4. Decompose F_a into $r \leq m$ augmenting unsplitted flows $F_a^1, \ldots, F_a^r$, i.e., $\sum_{j=1}^{r} F_a^j = F_a$.
5. Mark edges on F_a^1 successively from t towards u until a cycle composed of marked edges and F_i appears. Let $e' = (u', v')$ be the edge which was marked last.
6. Choose such a cycle where one continuous arc consists of marked edges and the remaining arc of commodity K_i. Augment along this cycle. Iterate until there is no cycle composed of marked edges and F_i left.
7. Delete cycles of F_a^1 until no more cycles of F_a^1 exist. For this, always choose cycles of the following kind: There exists a node y on the not re-routed part of F_a^1 s.t. the cycle consists of a part of F_a^1 from y to u' that was not re-routed in step 6 and a part of F_a^1 from u' to y that was re-routed. Of these cycles, choose one with the largest number of edges in the second part. From now on, consider just the edges of F_a^1 between u' and t (according to the order given by F_a^1) as marked.
8. Iterate steps 5 to 7, always starting from the farthest unmarked edge and proceeding towards u in step 5, until all edges in F_a^1 are marked.
9. Iterate steps 5 to 8 for the flows $F_a^2, \ldots, F_a^r$ successively.
10. Iterate steps 3 to 9 for all other commodities than K_i successively.
11. Iterate steps 2 to 10, always choosing a farthest back-edge, until no back-edge remains.
12. Iterate steps 3 to 9 for the commodity K to which the augmenting flow F_A "belongs", where we set $u := s$. However, after each execution of step 8, replace the respective augmenting flow F_a^j with a flow of commodity K.

Algorithm 14: Strongly consistent edge augmentation algorithm

together with the aforementioned path forms a cycle as described. After augmenting along this cycle, we may consider the re-routed augmenting flow as marked. Thus, property $\mathcal{Q}$ still holds and hence guarantees the existence of another cycle as described, under the condition that a cycle of marked edges and commodity K_i still exists. Note that in this context we still speak of marked edges, though technically some specific flow on the edges is marked (which is necessary because of potential cycles in the augmenting flow after augmenting along the first cycle).

Second, we show that in step 7, when choosing a cycle, we can actually choose a cycle as described in step 7. Again assuming property $\mathcal{Q}$, for the first cycle deletion we can choose a cycle as described due to the choice of u' (if a cycle exists).

According to the properties of the deleted cycle, the (deleted) flow going from y to u' must continue towards t only along edges which cannot be part of a cycle. Thus, if we ignore this deleted flow and its continuation towards t, all the (augmenting) flow re-routed in step 6 still arises from u'. Hence, after deleting the first (and any subsequent) cycle, we can again choose a cycle as described, should a cycle still exist. Note that the amount of flow in F_a^j going from u to u' is sufficient for all cycle deletions determined by the re-routed flow. Moreover, note that the considered cycles may use any edge only once, but are allowed to contain a node more than once in the corresponding cyclic order of nodes.

We now show that property $\mathcal{Q}$ holds: Observe that property $\mathcal{Q}$ holds in the beginning of each iteration of steps 5 to 8. If property $\mathcal{Q}$ holds in the beginning of step 5, then it holds during the whole step 5. Also, it holds after step 7, due to the last sentence in step 7 in conjunction with the above considerations regarding step 7. In particular, a part of the (unmarked) flow from u to u' must remain after the deletion of cycles in step 7, since all the flow in F_a^j starts along this path and some part of F_a^j must actually arrive at t.

Note that the edges marked at the end of any execution of step 7 together with the edges of commodity K_i cannot form a cycle, since these newly marked edges were used by commodity K_i before the augmentation in step 6 (while the reverse applies to the re-routed flow of commodity K_i).

All of the above applies analogously for the execution of step 12.

After these preliminaries regarding the correctness of Algorithm 14, we will now show the strong consistency. Observe that the only updates per-

formed in the physical network occur in step 6 and step 12. We first deal with step 6:

When flow of commodity K_i is re-routed in step 6, it replaces (parts of) the augmenting flow F_a^j, yielding a half flow H_a^j. Note that as this replaced flow did not form a cycle with itself or together with flow of the commodity K_i, adding flow F_i along these edges will not form a cycle with F_a^j or the commodity K_i either.

It remains to show for step 6 that H_a^j satisfies the strong consistency conditions given in Definition 32. As H_a^j is part of the new flow, it satisfies Condition 1). Furthermore, the new flow was just increased by H_a^j, thus guaranteeing Condition 2). Lastly, as the augmenting flow F_a^j together with the old flow did not violate any edge capacity constraints (and the augmenting flow is not actually inserted in the physical network), Condition 3) follows.

Again, analogous arguments apply for step 12, as the newly added flow just replaces an augmenting flow. Hence, the migration performed by Algorithm 14 is a strongly consistent migration.

We will now bound the number of performed strongly consistent migration updates from above: The only (physical) network updates performed occur in steps 6 and 12.

We first omit step 12, before adding the number of updates it induces at the end: As there are at most m back-edges, k commodities, and at most m flows in the decomposition of each F_a, steps 5 to 8 are iterated at most km^2 times. In each such iteration, the number of iterations of steps 5 to 7 is at most n, since F_a^j is a simple path in the beginning and the number of unmarked edges in F_a^j decreases by at least one in each iteration. Hence, step 6 is executed at most km^2n times.

For the analysis of 6, we go into further details of its execution, which we omitted above for the ease of readability: Observe that all cycles considered in step 6 pass through the edge (u', v'), and as thus, through the node u'. Furthermore, we can define a partial ordering of the nodes along the marked flow F_a^j, beginning with u'. Let z be a node of highest order s.t. there is a flow of commodity K_i from z to u'. Now, let d_z be the size of the flow of commodity K_i from z to u' and let d'_z be the size of the augmenting flow F_a^j from u' to z.

Imagine that we introduce a new edge from u' to z in our network with capacity $\min\{d_z, d'_z\}$, which is fully used by F_a^j, and a new edge from z to

u' with capacity $\min\{d_z, d'_z\}$, which is fully used by commodity K_i. At the same time, reduce the usage of F_a^j and F_i elsewhere in the network s.t. both F_i and F_a^j fulfill the flow constraints again. Now, augment along the new (imaginary) cycle between u' and z (which also deletes the previously added edges). The re-routed part of the augmenting path now leaves u' via edges that are different from e', as the re-routed part uses edges previously used by commodity K_i, and K_i does not use e' before or after the augmentation (otherwise, an e' "closer" to t would have been chosen). Therefore, the re-routed part is not part of any cycle consisting of F_i and F_a^j, and hence we do not get any new candidates for node z (i.e., a node along the marked edges of F_a^j from where there is a flow of commodity K_i to u') when repeating this augmentation technique. Thus, the number of candidates for such a z strictly decreases, meaning we need to augment at most n times in step 6.

We obtain at most km^2n^2 strongly consistent migration updates, before considering step 12. As step 12 invokes iterations of the steps 3 to 9, and performs a strongly consistent migration update after each invocation of step 8, the asymptotic number of updates does not change. Hence, we can bound the total number of updates by $O(km^2n^2)$.

However, in order to strongly consistently migrate to new desired demands, we still need a framework akin to Algorithm 13, which we will now provide with Algorithm 15.

Let N be a network and let $\mathcal{F}$ be a multi-commodity flow for the multi-commodity $\mathcal{K}$. Let $(d_1, \ldots, d_k)$ be a vector of demands s.t. *i*) there exists a multi-commodity flow for $\mathcal{K}$ satisfying these demands, and *ii*) $d_1 \geq d_{F_1}, \ldots, d_k \geq d_{F_k}$.

1. Compute a multi-commodity flow $\mathcal{F}'_1$ with a demand vector of $(d_1, d_{F_2}, \ldots, d_{F_k})$ using an LP.
2. Compute the difference flow $Z^{\mathcal{F},\mathcal{F}'_1}$.
3. Augment $\mathcal{F}$ with $Z^{\mathcal{F},\mathcal{F}'_1}$ by using Algorithm 14.
4. Iterate the above three steps for the remaining commodities $K_2, \ldots, K_k$, thereby obtaining the flows $\mathcal{F}_2, \ldots, \mathcal{F}_k$ with the demand vectors of $(d_1, d_2, d_{F_3}, \ldots, d_{F_k})$, $\ldots$, $(d_1, d_2, d_3, \ldots, d_{k-1}, d_{F_k})$, $(d_1, \ldots, d_k)$.

Algorithm 15: Strongly consistent migration to new demands via augmenting flows

Note that all calculations performed by Algorithm 15 (and by its invoked

Algorithm 14) can be performed in polynomial time.

Corollary 18. *Algorithm 15 performs a strongly consistent migration from $\mathcal{F}$ to some multi-commodity flow with a demand vector of $(d_1, \ldots, d_k)$, using only $O(k^2m^2n^2)$ strongly consistent migration updates.*

7.7 Hardness of Flow Migration to new Demands

So far we considered splittable multi-commodity flows as defined in Section 7.3, however this is just one side of the coin. As already defined in the previous chapters, an unsplittable flow is defined as a flow only taking one simple path from its source to its destination:

Definition 33. *A single-commodity flow F is called an* unsplittable single-commodity flow *if the set of edges $e \in E : F(e) > 0$ forms a simple (i.e., cycle-free) path from s to t. A multi-commodity flow $\mathcal{F}$ is called an* unsplittable multi-commodity flow *if all of its single-commodity flows are unsplittable single-commodity flows.*

Similarly, as in the previous chapters, we define an unsplittable consistent migration to be a consistent migration using only unsplittable flows:

Definition 34. *A consistent migration update $(N, \mathcal{F}, \mathcal{F}')$ is called an* unsplittable consistent migration update *if both $\mathcal{F}, \mathcal{F}'$ are unsplittable multi-commodity flows. A consistent migration is called an* unsplittable consistent migration *if it consists of unsplittable consistent migration updates.*

Hardness of Deciding if Demands can be Satisfied

Even et al. [18] showed for general[4] multi-commodity flow problems that it is NP-hard to decide if even the demand of two commodities can be met by integral flows in graphs with unit capacities; this case is equivalent to unsplittable multi-commodity flows. Kleinberg showed the NP-hardness also for the single-source unsplittable multi-commodity flow case [47], which is equivalent to the model used in this article (by reversing all edge directions). From the results of Baier et al. [6], it can be inferred that the NP-hardness also holds for just two unsplittable flows with common source s and destination t. The case of just one unsplittable single-commodity flow can be solved

[4] I.e., with each flow having a possibly distinct source and destination respectively.

in polynomial time by, e.g., finding the path with the highest capacity from s to t.

Consistently Migrating Unsplittable Flows to new Demands

As it is already NP-hard to decide if the demands of some commodities can be met, it is also an NP-hard problem to consistently migrate to a multi-commodity flow meeting these demands.

However, what happens if we know that the desired demands of the commodities can be met? I.e., if we are given the current multi-commodity flow and a multi-commodity flow meeting the desired demands (both unsplittable), is unsplittably consistently migrating NP-hard as well? As it turns out, the answer is yes:

Theorem 22. *Let N be a network and let $\mathcal{F} = (F_1, \ldots, F_k)$, $\mathcal{F}' = (F'_1, \ldots, F'_k)$ be unsplittable multi-commodity flows for the multi-commodity $\mathcal{K}$ s.t. $d_{F_i} \leq d_{F'_i}$, $1 \leq i \leq k$. It is NP-hard to decide if there is an unsplittable consistent migration from $\mathcal{F}$ to some unsplittable multi-commodity flow satisfying the demands of $\mathcal{F}'$.*

Note that for multi-commodity flows $\mathcal{F}, \mathcal{F}'$ with identical demands, the problem is trivial as zero updates are required to meet the demands of $\mathcal{F}'$.

We will prove Theorem 22 by reduction from the classic NP-hard problem *Partition* (also known as number partitioning). We note that our proof is similar in concept to the proof of Theorem 14, we still state the Partition problem again and give the proof in full length for ease of readability.

Definition 35 (*Partition* [29])**.** *Let $\mathcal{A}$ be a multiset of k positive real-valued elements $a_1, \ldots, a_k$, and set $A := \sum_{i=1}^{k} a_i$. Is it possible to partition $\mathcal{A}$ into two sets $\mathcal{A}_1, \mathcal{A}_2$ s.t. the sums $A_1 := \sum_{a_i \in \mathcal{A}_1} a_i$, $A_2 := \sum_{a_i \in \mathcal{A}_2} a_i$ of their respective elements are identical, i.e., $A_1 = A_2 = \frac{A}{2}$?*

Theorem 23 ([29])**.** *The* Partition *problem from Definition 35 is NP-hard.*

We can now prove Theorem 22:

Proof of Theorem 22. For every instance I of the Partition problem we will construct in polynomial time an instance I' of the problem described in

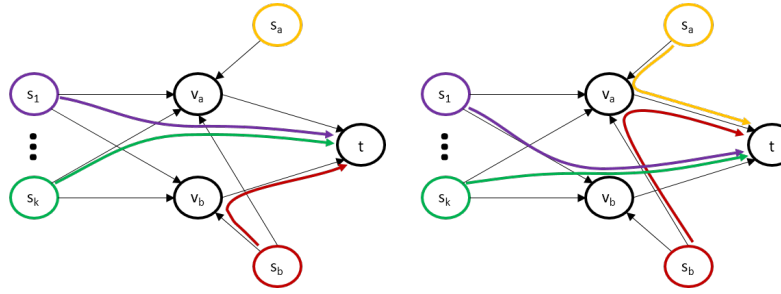
Theorem 22 s.t. I is a yes-instance if and only if I' is a yes-instance.

Construction of the new Instance I'
Given an instance I of the Partition problem, we create a network $N = (G = (V, E), c)$ as follows: V consists of k sources $s_1, \ldots, s_k$, two sources s_a, s_b, two nodes v_a, v_b, and a destination t, i.e., $k + 5$ nodes in total. E is composed of an edge from each $s_1, \ldots, s_k$ to both v_a and v_b with capacity of a_i, $1 \leq i \leq k$ respectively. Furthermore, there are edges from v_a, v_b to t with a capacity of A each, and edges from s_a to v_a and s_b to v_a, v_b with capacities of $A/2$. In total, there are $2k + 2 + 1 + 2 = 2k + 5$ edges.

The unsplittable multi-commodity flow $\mathcal{F}$ composed of the unsplittable flows $F_1, \ldots, F_k, F_b, F_a$ is defined as follows: $d_{F_1} = a_1, \ldots, d_{F_k} = a_k$, with each of these k flows F_i being routed from its source s_i via v_a to t, $1 \leq i \leq k$. Furthermore, $d_{F_b} = A/2$, with it being routed from its source s_b via v_b to t. Lastly, $d_{F_a} = 0$. The unsplittable multi-commodity flow $\mathcal{F}'$, composed of the unsplittable flows $F'_1, \ldots, F'_k, F'_b, F'_a$, is defined as follows: $d_{F'_1} = a_1, \ldots, d_{F'_k} = a_k$, but with each of these k flows F_i being routed from their source s_i via v_b to t, $1 \leq i \leq k$. Lastly, $d_{F_b} = d_{F_a} = A/2$, with both being routed via v_a. The construction can be performed in polynomial time and is depicted in Figure 7.4.

If I is a *yes*-instance, then I' is a *yes*-instance
Let us start by assuming that the Partition instance I is solvable, i.e., it is a yes-instance. Then, we can select the flows corresponding to $\mathcal{A}_1$ and unsplittably consistently migrate them to the path via v_b, as their combined size is exactly $A/2$. In the next step, we can add an unsplittable flow F'_a of size $A/2$ from s_a via v_a to t, showing that I' is a yes-instance as well.

If I is a *no*-instance, then I' is a *no*-instance
To conclude the proof, let us assume that the Partition instance I is not solvable, i.e., it is a no-instance. Observe that currently, the edge from v_a to t has no free capacity, and the edge from v_b to t has a free capacity of exactly $A/2$. Unless we can move a subset of the set of flows $F_1, \ldots, F_k$ of combined size exactly $A/2$ to the path via v_b, neither the edge from v_a to t nor the edge from v_b to t will have a free capacity of at least $A/2$. As the Partition instance is not solvable, this is not possible, meaning neither F_b can be moved consistently nor F_a can be added to the network, as both their sizes are $A/2$. Hence, I' is a no-instance as well. □



(a) Unsplittable multi-commodity fl. $\mathcal{F}$ **(b)** Unsplittable multi-commodity fl. $\mathcal{F}'$

Figure 7.4: The old unsplittable multi-commodity flow $\mathcal{F}$ is depicted in Subfigure 7.4a on the left, while the new unsplittable multi-commodity flow $\mathcal{F}'$ is depicted in Subfigure 7.4b on the right. In Subfigure 7.4a, the edge from v_a to t is used at full capacity. Similarly, in Subfigure 7.4b both edges from v_a, v_b to t are used at full capacity. In order to consistently migrate to a multi-commodity flow with the same demands as $\mathcal{F}'$, flows from $F_1, \ldots, F_k$ of combined size exactly $A/2$ need to migrate to the edge from v_b to t. However, this is equivalent to solving the corresponding Partition instance.

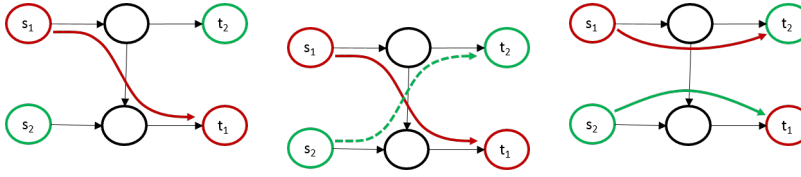
7.8 Augmenting Flows beyond a Single Destination

Besides the case of unsplittable flows as covered in Section 7.7, there is another natural extension of our model: Namely, the case of multi-commodity flows with multiple sources and multiple destinations.

As a simple example shows (cf. Figure 7.5), applying an augmenting path in a straightforward way to a network with multiple sources and destinations will not even necessarily result in a correct multi-commodity flow. The outgoing flow can end up being re-routed to a wrong destination.

A logical consequence is to admit only augmenting flows which re-route correctly, i.e., each outgoing flow of a source is still routed to its assigned destination. However, as Hu noted [38], it is unlikely that the technique of augmenting paths can be extended to a general multi-commodity setting (cf. Section 7.1). Nonetheless, what would happen if we could develop an augmenting path approach that results in correct multi-commodity flows?

Sacrificing polynomial runtime, one could check all possible flows be-



(a) Initial network with just one flow from s_1 to t_1.

(b) An augmenting flow is found from s_2 to t_2.

(c) The resulting flows are not feasible in the network!

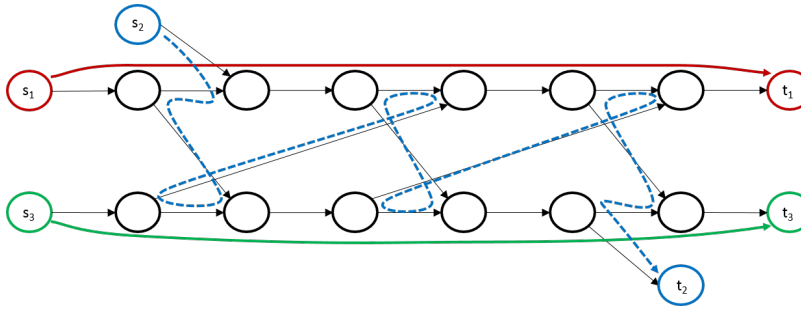
Figure 7.5: The existence of an augmenting flow does not guarantee feasible flows for multiple sources and destinations. E.g., the flow from s_1 might end up in t_2.

tween source-destination pairs in the residual networks to see if there is an augmenting flow that respects each flow ending at its correct destination.

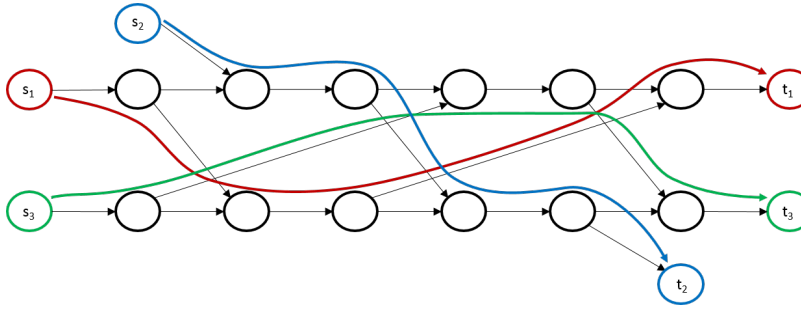
However, a more intricate example (cf. Figure 7.6) shows that, even in this case, it is not always possible to migrate consistently from the initial flow to the augmented flow.

7.9 Summary

In this chapter, we extended the notion of augmenting flows to the setting of multi-commodity flows for a single logical destination, providing algorithms to efficiently tackle the problem of consistent migration in Software Defined Networks. We also showed that augmenting flows can guarantee stronger consistency properties in this setting, and that consistent migration is NP-hard for unsplittable flows, even if both initial and desired demands are satisfiable. A natural question arises: Can we generalize the concept of an augmenting path to the general multi-commodity setting? As it turns out, even if we could efficiently find augmenting paths respecting the source-destination pairs, they will break consistency during migration. We thus believe that fundamentally different techniques are required to apply the method of augmenting flows for consistent migration updates beyond the anycast setting.



(a) There is an augmenting flow from s_2 to t_2 that results in a proper multi-commodity flow.



(b) The resulting new flow after the augmenting flow from above is applied to the network.

Figure 7.6: Neither (part of) the red nor the green flow can consistently migrate to any imaginable flow in the network: Moving any part of the red flow to the bottom path (or any part of the green flow to the top path) in Subfigure 7.6a will violate the consistency condition. Still, there is an augmenting flow moving both flows to other edges – which also respects the assignment of the sink-destination pairs, see Subfigure 7.6b for the resulting network. Hence, even an augmenting flow resulting in a proper multi-commodity flow does not guarantee a consistent migration for multiple sources and destinations.

Part III

Outlook

8

Future Directions

Research on loop freedom has a rich history in network reconfigurations, but there is still a plethora of algorithmic challenges to be tackled. Older work focuses mostly on reliability and feasibility, while SDN allowed to aim for optimization of various performance metrics: While it is known that any dynamic/greedy update schedule will succeed for a single destination, finding optimal or even non-trivial approximation algorithms seems to be out of reach right now. Nearly all work, including ours in this thesis, explores the hardness of loop-free updates, similar for its combination with blackhole freedom.

However, inapproximability results are still rare, implicitly posing the question of where loop freedom falls on the complexity scale. Algorithmic progress was made for the specific case of dynamic/greedy updates, but scheduling loop-free updates has the difficulty of it being a dynamic problem as well in some way: Each set of forwarding rules that gets updated changes the problem graph in the next iteration, leading to an explosion of the state

space.

We believe that fundamentally different approaches will have to be developed for the problem of efficiently scheduling loop-free updates, especially when also considering the additional restriction of limited memory in the case of prefix-based rules. Beyond strengthening other complexity results, we consider it to be the most crucial research question in the field of loop freedom.

Similarly to loop freedom, flow migration has a well-studied background, in the form of bandwidth throughput optimization and allocation. Consistent migration of flows in its current state uses the mechanisms provided by flow tagging protocols, which in turn provide packet coherence. An interesting concept beyond the scope of this thesis would be to tackle congestion-free migration on the basis of forwarding rules, without tagging. Still, there are many open problems within the reach of the current state of the art to be considered.

On a conceptual basis, most other work uses linear or integer programming for consistent migrations, while we used the toolkit of flow augmentation for many of our results. It would be interesting to see if another mathematical paradigm allows for better algorithmic results. Next, for the complexity of unsplittable migration with intermediate paths, it is not even clear if the decision problem in general is in NP, or what the ballpark for the optimal number of updates is. If it is polynomial, the decision problem would be in NP as well, but we see no strong evidence for either possibility. Analogously, the maximum number of updates needed for splittable flows is not completely classified yet. Some trivial bounds can be deducted based on the free capacity in the network, but we suspect a combinatorial classification to be possible.

Lastly, it is not clear how our fast augmenting flow methods can be extended to multiple destinations. It seems to be inherently connected to developing augmenting flow techniques for multi-commodity flows in general, but research results have mostly ceased since the prevalence of (polynomial) linear programming methods starting decades ago.

Beyond the Consistent Network Update Problem

We have provided multiple new algorithmic concepts and complexity classifications for fundamental consistent network updates in this thesis, but

we believe this to be just one part of the greater puzzle. In the standard consistent update problemat, the task of finding intermediate states is decoupled from the task of providing a new network configuration. As thus, it can be that the same performance goals can be met with, e.g., another new routing table, but that the number of intermediate states needed to reach this routing table in a loop-free way is greatly reduced. The ongoing enhancement of the programmability of SDNs seems to be a strong factor for this decoupling: Driven by the desire to provide tools for the manual fine-tuning of networks, researchers all over the world jumped on the bandwagon of studying algorithms and complexities for consistent updates – and this thesis is no exception to that.

Nonetheless, we started to work in the previous chapter of this thesis towards the unification of both problems, by just targeting that new demands can be reached in a consistent way, without (over)specifying where the flows should be in the network, allowing for greatly improved bounds on the number of updates.

Going even a step further, we envision that SDNs manage all relevant optimization properties on their own without manual input. Especially smaller companies cannot (and do not want to) constantly manage their network by an (expensive) expert, they want a (cheap) off-the-shelf controller that just works. The current lines of work on consistent network updates, including the results of this thesis, would then be a building block of the ongoing optimization algorithms inside the logically centralized controller, which would act independently – akin to a *Skynet* for computer networking.[1]

[1] Hopefully, without the unpleasant side-effects depicted in the *Terminator* movies.

Bibliography

[1] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows - theory, algorithms and applications. Prentice Hall (1993)

[2] Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM **42**(4) (July 1995) 844–856

[3] Amiri, S., Ludwig, A., Marcinkowski, J., Schmid, S.: Transiently consistent sdn updates: Being greedy is hard. In: Structural Information and Communication Complexity - 23rd International Colloquium, SIROCCO 2016. (2016)

[4] Anderson, E., Anderson, T.E.: On the stability of adaptive routing in the presence of congestion control. In: Proceedings IEEE INFOCOM 2003, The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, San Franciso, CA, USA, March 30 - April 3, 2003, IEEE (2003)

[5] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V., Swallow, G.: RFC 3209, RSVP-TE: Extensions to RSVP for LSP Tunnels. Network Working Group, Category: Standards Track (December 2001)

[6] Baier, G., Köhler, E., Skutella, M.: The k-splittable flow problem. Algorithmica **42**(3-4) (2005) 231–248

[7] Bennett, C., Tseitlin, A.: NetFlix. Chaos Monkey Released Into The Wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

[8] Björklund, A., Husfeldt, T., Khanna, S.: Approximating longest directed paths and cycles. In Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D., eds.: Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings. Volume 3142 of Lecture Notes in Computer Science., Springer (2004) 222–233

[9] Bodlaender, H.: On linear time minor tests with depth-first search. Journal of Algorithms **14**(1) (1993) 1 – 23

[10] Borokhovich, M., Schmid, S.: How (not) to shoot in your foot with SDN local fast failover - A load-connectivity tradeoff. In Baldoni, R., Nisse, N., van Steen, M., eds.: Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings. Volume 8304 of Lecture Notes in Computer Science., Springer (2013) 68–82

[11] Casado, M., Foster, N., Guha, A.: Abstractions for software-defined networks. Commun. ACM **57**(10) (2014) 86–95

[12] Claburn, T.: Google Vs. Zombies – And Worse. InformationWeek - Network Computing (2013) http://ubm.io/1ftfjxA.

[13] Comer, D., ed.: Internetworking with TCP/IP - Principles, Protocols, and Architectures, Fourth Edition. Prentice-Hall (2000)

[14] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)

[15] Dinitz, Y.: Dinitz' algorithm: The original version and even's version. In Goldreich, O., Rosenberg, A.L., Selman, A.L., eds.: Theoretical Computer Science, Essays in Memory of Shimon Even. Volume 3895 of Lecture Notes in Computer Science., Springer (2006) 218–240

[16] Dudycz, S., Ludwig, A., Schmid, S.: Can't touch this: Consistent network updates for multiple policies. In: Proc. 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). (2016)

[17] Even, G., Naor, J., Schieber, B., Sudan, M.: Approximating minimum feedback sets and multicuts in directed graphs. Algorithmica **20**(2) (1998) 151–174

[18] Even, S., Itai, A., Shamir, A.: On the complexity of timetable and multicommodity flow problems. SIAM J. Comput. **5**(4) (1976) 691–703

[19] Flammini, M., Gambosi, G., Salomone, S.: Boolean Routing. In Schiper, A., ed.: Distributed Algorithms. Volume 725 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1993) 219–233

[20] Foerster, K.T., Luedi, T., Seidel, J., Wattenhofer, R.: Local checkability, no strings attached. In: Proceedings of the 17th International Conference on Distributed Computing and Networking. ICDCN '16, New York, NY, USA, ACM (2016) 21:1–21:10

[21] Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Canad. J. Math **8** (1956) 399–404

[22] Ford, L.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press, Princeton, NJ, USA (1962)

[23] Förster, K.T., Mahajan, R., Wattenhofer, R.: Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes. In: Proceedings of the 15th IFIP Networking Conference, Networking 2016, Vienna, Austra, 17-19 May, 2016, IEEE (2016)

[24] Fraigniaud, P., Gavoille, C.: A characterization of networks supporting linear interval routing. In: PODC. (1994)

[25] François, P., Bonaventure, O.: Avoiding transient loops during the convergence of link-state routing protocols. IEEE/ACM Trans. Netw. **15**(6) (2007) 1280–1292

[26] François, P., Filsfils, C., Evans, J., Bonaventure, O.: Achieving subsecond IGP convergence in large IP networks. Computer Communication Review **35**(3) (2005) 35–44

[27] Fuller, V., Li, T.: RFC 4632, classless inter-domain routing (cidr): The internet address assignment and aggregation plan. Network Working Group (August 2006)

[28] Gao, R., Blair, D., Dovrolis, C., Morrow, M., Zegura, E.W.: Interactions of intelligent route control with TCP congestion control. In Akyildiz, I.F., Sivakumar, R., Ekici, E., de Oliveira, J.C., McNair, J., eds.: NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet, 6th International IFIP-TC6 Networking Conference, Atlanta, GA, USA, May 14-18, 2007, Proceedings. Volume 4479 of Lecture Notes in Computer Science., Springer (2007) 1014–1025

[29] Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)

[30] Gavoille, C.: A survey on interval routing. Theor. Comput. Sci. **245**(2) (2000) 217–253

[31] Ghorbani, S., Caesar, M.: Walk the line: Consistent network updates with bandwidth guarantees. In: HotSDN. (2012)

[32] Goldberg, A.V., Oldham, J.D., Plotkin, S.A., Stein, C.: An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z., eds.: Integer Programming and Combinatorial Optimization, 6th International IPCO Conference, Houston, Texas, USA, June 22-24, 1998, Proceedings. Volume 1412 of Lecture Notes in Computer Science., Springer (1998) 338–352

[33] Hartman, T., Hassidim, A., Kaplan, H., Raz, D., Segalov, M.: How to split a flow? In Greenberg, A.G., Sohraby, K., eds.: Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012, IEEE (2012) 828–836

[34] Håstad, J.: Some optimal inapproximability results. J. ACM **48**(4) (2001) 798–859

[35] He, J., Rexford, J.: Toward internet-wide multipath routing. IEEE Network **22**(2) (2008) 16–21

[36] He, K., Khalid, J., Gember-Jacobson, A., Das, S., Prakash, C., Akella, A., Li, L.E., Thottan, M.: Measuring control plane latency in sdn-enabled switches. In Rexford, J., Vahdat, A., eds.: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015, ACM (2015) 25:1–25:6

[37] Hong, C.Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., Wattenhofer, R.: Achieving high utilization with software-driven WAN. In Chiu, D.M., Wang, J., Barford, P., Seshan, S., eds.: SIGCOMM, ACM (2013) 15–26

[38] Hu, T.C.: Multi-commodity network flows. Operations Research **11**(3) (1963) 344–360

[39] Itai, A.: Two-commodity flow. J. ACM **25**(4) (1978) 596–611

[40] Ito, H., Iwama, K., Okabe, Y., Yoshihiro, T.: Avoiding routing loops on the internet. Theory Comput. Syst. **36**(6) (2003) 597–609

[41] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., Vahdat, A.: B4: experience with a globally-deployed software defined wan. In Chiu, D.M., Wang, J., Barford, P., Seshan, S., eds.: ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013, ACM (2013) 3–14

[42] Jin, X., Liu, H.H., Gandhi, R., Kandula, S., Mahajan, R., Zhang, M., Rexford, J., Wattenhofer, R.: Dionysus: Dynamic scheduling of network updates. In Bustamante, F.E., Hu, Y.C., Krishnamurthy, A., Ratnasamy, S., eds.: ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, USA, August 17-22, 2014, ACM (2014) 539–550

[43] Kandula, S., Katabi, D., Sinha, S., Berger, A.: Dynamic load balancing without packet reordering. SIGCOMM CCR (2007)

[44] Kandula, S., Menache, I., Schwartz, R., Babbula, S.R.: Calendaring for wide area networks. In: SIGCOMM. (2014)

[45] Katta, N.P., Rexford, J., Walker, D.: Incremental consistent updates. In: HotSDN. (2013)

[46] Khachian, L.G.: A polynomial algorithm in linear programming. Dokl. Akad. Nauk SSSR **244** (1979) 1093–1096 English translation in Soviet Math. Dokl. 20, 191-194, 1979.

[47] Kleinberg, J.M.: Single-source unsplittable flow. In: 37th Annual Symposium on Foundations of Computer Science, FOCS '96, Burlington, Vermont, USA, 14-16 October, 1996, IEEE Computer Society (1996) 68–77

[48] Knight, S., Nguyen, H.X., Falkner, N., Bowden, R.A., Roughan, M.: The internet topology zoo. IEEE Journal on Selected Areas in Communications **29**(9) (2011) 1765–1775

[49] Kuzniar, M., Peresíni, P., Kostic, D.: What you need to know about SDN flow tables. In Mirkovic, J., Liu, Y., eds.: Passive and Active Measurement - 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings. Volume 8995 of Lecture Notes in Computer Science., Springer (2015) 347–359

[50] Liu, H.H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R., Maltz, D.A.: zUpdate: updating data center networks with zero loss. In Chiu, D.M., Wang, J., Barford, P., Seshan, S., eds.: SIGCOMM, ACM (2013) 411–422

[51] Ludwig, A., Dudycz, S., Rost, M., Schmid, S.: Transiently secure network updates. In: Proc. ACM SIGMETRICS. (2016)

[52] Ludwig, A., Marcinkowski, J., Schmid, S.: Scheduling loop-free network updates: It's good to relax! In Georgiou, C., Spirakis, P.G., eds.: Proceedings of the 2015 ACM Symposium on Principles of Distributed

Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015, ACM (2015) 13–22

[53] Ludwig, A., Rost, M., Foucard, D., Schmid, S.: Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In Katz-Bassett, E., Heidemann, J.S., Godfrey, B., Feldmann, A., eds.: Proceedings of the 13th ACM Workshop on Hot Topics in Networks, HotNets-XIII, Los Angeles, CA, USA, October 27-28, 2014, ACM (2014) 15:1–15:7

[54] Lukovszki, T., Schmid, S.: Online admission control and embedding of service chains. In Scheideler, C., ed.: Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015. Volume 9439 of Lecture Notes in Computer Science., Springer (2015) 104–118

[55] Luo, L., Yu, H., Luo, S., Zhang, M.: Fast lossless traffic migration for SDN updates. In: 2015 IEEE International Conference on Communications, ICC 2015, London, United Kingdom, June 8-12, 2015, IEEE (2015) 5803–5808

[56] Luo, S., Yu, H., Luo, L., Li, L.: Arrange your network updates as you wish. In: Proc. of IFIP Networking. (2016)

[57] Mahajan, R., Spring, N., Wetherall, D., Anderson, T.: Inferring link weights using end-to-end measurements. In: Internet Measurement Workshop. (2002)

[58] Mahajan, R., Wattenhofer, R.: On consistent updates in software defined networks. In Levine, D., Katti, S., Oran, D., eds.: Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013, ACM (2013) 20:1–20:7

[59] McClurg, J., Hojjat, H., Cerný, P., Foster, N.: Efficient Synthesis of Network Updates. In Grove, D., Blackburn, S., eds.: PLDI, ACM (2015) 196–207

[60] Mizrahi, T., Rottenstreich, O., Moses, Y.: TimeFlip: Scheduling network updates with timestamp-based TCAM ranges. In: INFOCOM, IEEE (2015) 2551–2559

[61] Mizrahi, T., Moses, Y.: On the necessity of time-based updates in SDN. In Sherwood, R., ed.: Open Networking Summit 2014 - Research Track, ONS 2014, Santa Clara, CA, USA, March 2-4, 2014, USENIX Association (2014)

[62] Mizrahi, T., Moses, Y.: Software Defined Networks: It's About Time. In: INFOCOM. (2016)

[63] Mizrahi, T., Saat, E., Moses, Y.: Timed consistent network updates. In Rexford, J., Vahdat, A., eds.: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015, ACM (2015) 21:1–21:14

[64] Noyes, A., Warszawski, T., Cerný, P., Foster, N.: Toward synthesis of network updates. In: SYNT. (2013)

[65] Orlin, J.B.: Max flows in o(nm) time, or better. In Boneh, D., Roughgarden, T., Feigenbaum, J., eds.: Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013, ACM (2013) 765–774

[66] Paris, S., Destounis, A., Maggi, L., Paschos, G.S., Leguay, J.: Controlling flow reconfigurations in sdn. In: Proc. of INFOCOM. (2016)

[67] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., Fugal, H.: Fastpass. In: SIGCOMM. (2014)

[68] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. In Eggert, L., Ott, J., Padmanabhan, V.N., Varghese, G., eds.: ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012, ACM (2012) 323–334

[69] Reitblatt, M., Foster, N., Rexford, J., Walker, D.: Consistent updates for software-defined networks: change you can believe in! In Balakrishnan, H., Katabi, D., Akella, A., Stoica, I., eds.: Tenth ACM Workshop on Hot Topics in Networks (HotNets-X), HOTNETS '11, Cambridge, MA, USA - November 14 - 15, 2011, ACM (2011) 7

[70] Roskind, J., Tarjan, R.E.: A note on finding minimum-cost edge-disjoint spanning trees. Math. Oper. Res. **10**(4) (1985) 701–708

[71] Rothfarb, W., Shein, N.P., Frisch, I.T.: Common terminal multicommodity flow. Operations Research **16**(1) (1968) 202–205

[72] Santoro, N., Khatib, R.: Routing without routing tables. Technical report, SCS-TR-6, School of Computer Science, Carleton University, Ottawa (1982)

[73] Schmid, S., Suomela, J.: Exploiting locality in distributed SDN control. In Foster, N., Sherwood, R., eds.: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013, ACM (2013) 121–126

[74] Shelly, N., Tschaen, B., Förster, K., Chang, M.A., Benson, T., Vanbever, L.: Destroying networks for fun (and profit). In de Oliveira, J., Smith, J., Argyraki, K.J., Levis, P., eds.: Proceedings of the 14th ACM Workshop on Hot Topics in Networks, Philadelphia, PA, USA, November 16 - 17, 2015, ACM (2015) 6:1–6:7

[75] Siegel, J.: Azure's Search Chaos Monkey is wreaking havoc to find potential points of failure `http://bit.ly/1HPLtQ9`.

[76] Tanenbaum, A.S., Wetherall, D.J.: Computer Networks. 5th edn. Prentice Hall Press, Upper Saddle River, NJ, USA (2010)

[77] Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2) (1972) 146–160

[78] Tarjan, R.E.: A note on finding the bridges of a graph. Inf. Process. Lett. **2**(6) (1974) 160–161

[79] Tseitlin, A.: The antifragile organization. Commun. ACM **56**(8) (August 2013) 40–44

[80] Van Leeuwen, J., Tan, R.B.: Interval routing. The Computer Journal **30**(4) (1987) 298–307

[81] Vanbever, L.: Methods and techniques for disruption-free network reconfiguration. PhD thesis, Université catholique de Louvain (2012)

[82] Vanbever, L., Vissicchio, S., Pelsser, C., Francois, P., Bonaventure, O.: Lossless migrations of link-state igps. IEEE/ACM Trans. Netw. **20**(6) (December 2012) 1842–1855

[83] Vissicchio, S., Cittadini, L.: Flip the (flow) table: Fast lightweight policy-preserving sdn updates. In: 2016 IEEE Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-15, 2016, IEEE (2016)

[84] Wang, W., He, W., Su, J., Chen, Y.: Cupid: Congestion-free consistent data plane update in software defined networks. In: Proc. of INFOCOM. (2016)

[85] Zhang, L., Wu, C., Li, Z., Guo, C., Chen, M., Lau, F.C.M.: Moving big data to the cloud: An online cost-minimizing approach. IEEE Journal on Selected Areas in Communications **31**(12) (2013) 2710–2721

[86] Zheng, J., Xu, H., Chen, G., Dai, H.: Minimizing transient congestion during network update in data centers. In: 23rd IEEE International Conference on Network Protocols, ICNP 2015, San Francisco, CA, USA, November 10-13, 2015, IEEE Computer Society (2015) 1–10

[87] Zhou, W., Jin, D.K., Croft, J., Caesar, M., Godfrey, P.B.: Enforcing customizable consistency properties in software-defined networks. In: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015, USENIX Association (2015) 73–85

Publications

The following lists the publications related to Computer Science, in the English language, during my time as a PhD student at ETH Zurich. In order to have a consistent subject, and for space constraints, only a subset of these papers is presented in this thesis, namely the ones focusing on Software Defined Networks. Part I is (partially) based on the articles at ICCCN 2016, IFIP Networking 2016, HotNets 2015, and 1. Part II is based on the articles at ICCCN 2016, INFOCOM 2016, ICDCN 2016, in Pervasive and Mobile Computing, and 2. Finally, some results of this thesis have not been published yet.

Manuscripts (partially) included in this thesis

1. *Local Checkability, No Strings Attached: (A)cyclicity, Reachability, Loop Free Updates in SDNs.* Klaus-Tycho Förster, Thomas Lüdi, Jochen Seidel and Roger Wattenhofer. Invited to a special issue of Theoretical Computer Science.

2. *Not so Lossless Flow Migration: The Impact of Latency on Network Updates.* Sebastian Brandt, Klaus-Tycho Förster, Laurent Vanbever and Roger Wattenhofer.

Accepted Publications

3. *Local Checkability in Dynamic Networks.* Klaus-Tycho Förster, Oliver Richter, Jochen Seidel and Roger Wattenhofer. 18th International Conference on Distributed Computing and Networking (ICDCN), Hyderabad, India, January 2017.

4. *Distributed Discussion Diarisation.* Pascal Bissig, Klaus-Tycho Förster, Simon Tanner and Roger Wattenhofer. 14th Annual IEEE Consumer and Networking Conference (CCNC), Las Vegas, NV, USA, January 2017.

5. *RTDS: Real-Time Discussion Statistics.* Pascal Bissig, Jan Deriu, Klaus-Tycho Förster and Roger Wattenhofer. 15th International Conference on Mobile and Ubiquitous Multimedia (MUM), Rovaniemi, Finland, December 2016.

6. *Reducing the Latency-Tail of Short-Lived Flows: Adding Forward Error Correction in Data Centers.* Klaus-Tycho Förster, Demian Jäger, David Stolz and Roger Wattenhofer. 15th IEEE International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, November 2016.

7. *Augmenting Flows for the Consistent Migration of Multi-Commodity Single-Destination Flows in SDNs.* Sebastian Brandt, Klaus-Tycho Förster and Roger Wattenhofer. accepted for publication in Pervasive and Mobile Computing, September 2016.

8. *A Concept for an Introduction to Parallelization in Java: Multithreading with Programmable Robots in Minecraft.* Klaus-Tycho Förster, Michael König and Roger Wattenhofer. 17th Annual Conference on Information Technology Education (SIGITE), Boston, MA, USA, September 2016.

9. *Integrating Programming into the Mathematics Curriculum: Combining Scratch and Geometry in Grades 6 and 7.* Klaus-Tycho Förster. 17th Annual Conference on Information Technology Education (SIGITE), Boston, MA, USA, September 2016.

10. *The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration.* Klaus-Tycho Förster and Roger Wattenhofer. 25th International Conference on Computer Communication and Networks (ICCCN), Waikoloa, Hi, USA, August 2016.

11. *Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes.* Klaus-Tycho Förster, Ratul Mahajan and Roger Wattenhofer. 15th IFIP Networking Conference (IFIP Networking), Vienna, Austria, May 2016.

12. *On Consistent Migration of Flows in SDNs.* Sebastian Brandt, Klaus-Tycho Förster and Roger Wattenhofer. 36th IEEE International Conference on Computer Communications (INFOCOM), San Francisco, California, USA, April 2016.

13. *Augmenting Anycast Network Flows.* Sebastian Brandt, Klaus-Tycho Förster and Roger Wattenhofer. 17th International Conference on Distributed Computing and Networking (ICDCN), Singapore, January 2016.

14. *Local Checkability, No Strings Attached.* Klaus-Tycho Förster, Thomas Lüdi, Jochen Seidel and Roger Wattenhofer. 17th International Conference on Distributed Computing and Networking (ICDCN), Singapore, January 2016.

15. *Lower and Upper Competitive Bounds for Online Directed Graph Exploration.* Klaus-Tycho Förster and Roger Wattenhofer. Accepted for publication in Theoretical Computer Science, November 2015.

16. *Destroying networks for fun (and profit).* Nick Shelly, Brendan Tschaen, Klaus-Tycho Förster, Michael Chang, Theophilus Benson and Laurent Vanbever. 14th ACM Workshop on Hot Topics in Networks (HotNets), Philadelphia, PA, USA, November 2015.

17. *Programming in Scratch and Mathematics: Augmenting Your Geometry Curriculum, Today!.* Klaus-Tycho Förster. 16th Annual Conference on Information Technology Education (SIGITE), Chicago, IL, USA, October 2015.

18. *Lower Bounds for the Capture Time: Linear, Quadratic, and Beyond.* Klaus-Tycho Förster, Rijad Nuridini, Jara Uitto and Roger Wattenhofer. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO), Montserrat, Spain, July 2015.

19. *SpareEye: A Smart Phone App that Enhances the Safety of the Inattentionally Blind.* Klaus-Tycho Förster, Alex Gross, Nino Hail, Jara Uitto and Roger Wattenhofer. The 13th International Conference on Mobile and Ubiquitous Multimedia (MUM), Melbourne, Australia, November 2014.

20. *Deterministic Leader Election in Multi-Hop Beeping Networks.* Klaus-Tycho Förster, Jochen Seidel and Roger Wattenhofer. 28th International Symposium on Distributed Computing (DISC), Austin, Texas, USA, October 2014.

21. *Approximating Fault-Tolerant Domination in General Graphs.* Klaus-Tycho Förster. SIAM Analytic Algorithmics and Combinatorics (ANALCO), New Orleans, Louisiana, USA, January 2013.

22. *Directed Graph Exploration.* Klaus-Tycho Förster and Roger Wattenhofer. 16th International Conference On Principles Of Distributed Systems (OPODIS), Rome, Italy, December 2012.

Further Manuscripts

23. *Survey of Consistent Network Updates.* Klaus-Tycho Förster, Stefan Schmid and Stefano Vissicchio. arXiv:1609.02305 [cs.NI] , September 2016.

24. *Multi-Agent Pathfinding with n Agents on Graphs with n Vertices: Combinatorial Classification and Tight Algorithmic Bounds.* Klaus-Tycho Förster, Linus Groner, Torsten Hoefler, Michael König, Sascha Schmid and Roger Wattenhofer.