Latency and Consistent Flow Migration: Relax for Lossless Updates

Klaus-Tycho Foerster
Faculty of Computer Science
University of Vienna, Austria

Laurent Vanbever
Networked Systems Group
ETH Zurich, Switzerland

Roger Wattenhofer
Distributed Computing Group
ETH Zurich, Switzerland

***Abstract*—Consistency in network updates is a nascent research area, especially in the context of traffic engineering or Software Defined Networks. Various approaches have been proposed and implemented in the problem space of flow migration and congestion, primarily focusing on different flows not breaking the bandwidth capacities of the used links during updates.**

However, current network update techniques overlook the effect of flows congesting their own path during a network update due to latency on the links. 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 that these effects occur already in a small topology in practice, causing packet loss due to overfull buffers. Furthermore, we prove that finding a lossless flow migration is NP-hard, already for a single (splittable) flow on directed acyclic graphs.

Nonetheless, we can relax latency requirements to still obtain lossless flow migration. To this end, we show how to adapt current systems such as *SWAN* or *Dionysus* [SIGCOMM'13/'14], also developing our own polynomial time schedule algorithm, and discussing future consistent flow migration technique adaptations.

I. Introduction

Network operators continuously strive for increased performance, especially to improve end-to-end latency or increase bandwidth utilization [49]. As such, the paths of network flows (i.e., the forwarding rules) are always in a state of flux, either by automated or manual network updates.

Network updates (flow migration) may temporarily destabilize a network, as mixing old and new forwarding rules may contradict. 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, congestion is highly problematic especially in contexts such as WAN and data-centers. As such, there has been (not only) recent interest in consistency mechanisms for network updates [6], [7], [10], [11], [17], [31], [34], [35], [36], [37], [41], [46], [54], [53], notably against bandwidth violations [8], [24], [27], [47].

However, understanding network updates has issues besides packet dropping. We would like to motivate these issues with a simple example. In a small gigabit ethernet network, 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 Fig. 1.

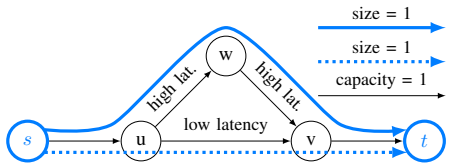

Fig. 1. In this network with just one flow, the flow along the solid path is moved to the dashed path, causing congestion at v due to packets in flight. Current methods overlook this issue of flows congesting "themselves", as they just consider the flows before and after the update, but not the transient behavior of packets in flight – causing packet loss due to overfull buffers.

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*.

UDP and TCP effects 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.

Packet loss can occur in every non-trivial topology As noted above, packets are dropped if the remaining buffer size is not sufficient enough to capture the additional flow caused by the network update. Thus, as long as there are two different paths to an egress router, switching to the path with lower latency causes extra packets to appear at some router for time equivalent to the latency difference. Therefore, previous flow consistency formulations are *not lossless* as they ignore packets in flight: not just in specific cases, but in any non-trivial topology.

Being lossless is a desirable property As pointed out by [2], already small losses and additional latencies can cause drastic changes in data-center performance metrics [30]. Service level agreements impose an additional challenge [39]. Ideally, the network's packet loss rate should be oblivious to flow migration.

ISBN 978-3-903176-08-9

Buffering packet loss Packet buffers could be used to temporarily "store" congestion. Taking Fig. 1 as an example, already a 1ms latency difference in a gigabit network would require a 1Mbit buffer, 10Mbit for 10ms etc. However, commodity gigabit switches usually just have 32KB buffer size per port [50]. Similarly, optical switches have small buffers in general, though optical networks have in turn also lower latencies. Still, in hybrid networks [26], both effects can collide.

A. Background

To the best of our knowledge, a study of the effects described for the network in Fig. 1 has not been considered in the (network update) literature.[1] Flow theory from a mathematical point of view [1] rather focuses on maximizing the flows.

Traffic engineering has a long history in networks, but maybe it was not before the rise of Software Defined Networks (SDNs) that consistency during network updates became a field of wider interest [10]. Complex network updates in SDNs are common nowadays, e.g., Microsoft's *SWAN* [24] and Google's *B4* [49] run updates every few minutes, hundreds of times per day. Moreover, updates for rerouting can also be needed in everyday situations, e.g., taking a link offline for repair [58].

In the area of congestion and flows, staged partial moves of flows are the current state of the art to prevent packet loss during updates [17], [18], [37]. However, these methods currently overlook the effects of edge latencies during the migration of flows. For example, *zUpdate* [30] denotes an update like our example from Fig. 1 as *lossless*, as their model does not take edge latencies or packets in flight into account.

The methods of *two-phase updates* [46] and *timed-based consistent updates* [42], [41] also do not protect against this issue. First, changing the routing rule in u is a valid operation in [46]. Second, as only one router needs to change its behavior (u), there are no timing problems either [42], [41].

B. Contribution

Motivation: Packets are dropped in reality Current consistent flow migration models overlook the impact of latency in the network, already in the small example of Fig. 1. To prove that the described packet loss and congestion effects actually occur in reality, we build a small testbed and show that our conjecture holds up in practice, see Sec. II for details.

Hardness strikes already for a single flow Motivated by the findings in our testbed, we provide a model for lossless flow migration in Sec. III. However, we prove in Sec. IV that the problem is already NP-hard for a single flow for many variations: for both splittable and unsplittable flows, on DAGs, for unit latency and unit size flows, but also for latency ranges.

Relaxation leads to tractability Nonetheless, we can show how to "bake" losslessness into current systems such as *Dionysus* [27] or *SWAN* [24] in Sec. V, by making use of the fact that these systems require restricted problem scenarios.

To obtain a polynomial technique for the general case, we relax the lossless property in such a way that algorithms have to operate consistently even if the latency changes, see Sec. VI. We provide details and properties of our algorithm in Sec. VI-B.

Outlook and further work In Sec. VII, we discuss a multitude of further mitigation techniques on how to obtain a lossless migration in practice, which we hope to be useful for future (heuristic) improvements. E.g., how to incorporate flowlets into dependency graph approaches or the feasibility of buffering to avoid packet loss. Lastly, we give an overview and comparison of related work in Sec. VIII, concluding in Sec. IX.

II. Motivation

We show that the congestion effects highlighted in Fig. 1 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.

Model We start with some notations and assumptions. Consider the network in Fig. 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:
$$\min\left(B(v), \max((d_F + d_{F'} - c) \cdot \Delta, 0)\right), \qquad (1)$$
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, but contain at least 0 packets.
- The non-negative amount of data dropped at v will be:
$$\min\left((d_F + d_{F'} - c) \cdot \Delta - B(v), 0\right), \qquad (2)$$
i.e., the amount of surplus data not fitting into the buffer.
- The buffer at v will be drained after the following time:
$$\left((d_F + d_{F'} - c) \cdot \Delta\right) / \left(c - d_{F'}\right), \qquad (3)$$
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 3 implies that the buffer at v will never be drained.

Methodology & Testbed To validate our theoretical model in practice, we replicated Fig. 1 in a testbed composed of five servers directly connected through five Gigabit links. Each of the machines has the following specification: Ubuntu 64bit server 14.04 with kernel 3.16, 8GB of RAM, 2x Intel Xeon quadcore 2.4GHz. We configure a 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

[1] Some preliminary results of our work can be found in [13, §6].

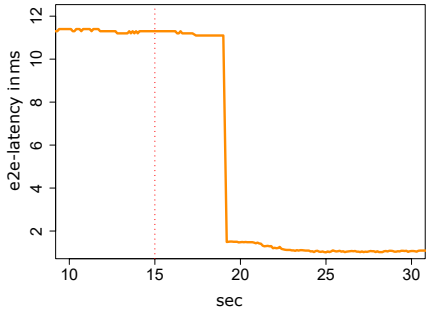

Fig. 2. Median end-to-end latency over 30 UDP experiments. The dotted vertical line denotes the update time. It takes seconds for the congestion induced by latency to disappear.

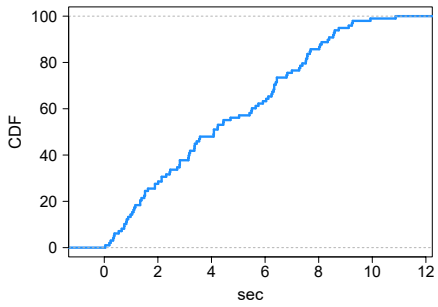

Fig. 3. CDF of the duration length. In 50% of the experiments, congestion (at the buffer at v) lasted for more than 4 seconds, in 10%, for more than 8 seconds.

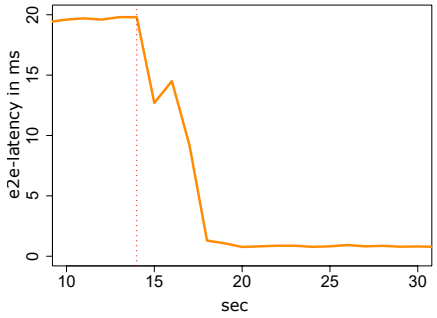

Fig. 4. Median end-to-end latency over 30 TCP experiments. The dotted vertical line denotes the update time. Latency-induced congestion is also detrimental for TCP flows, albeit for shorter time.

In the experiments depicted above, we used (combined) gigabit flow sizes in a Gigabit Ethernet; The buffer sizes were set (vastly over-provisioned) to one gigabit as well, to rule out any packet losses due to overfull buffers. By setting the buffer sizes to being smaller than the combined size of the packets in flight induced by the latency difference, packet loss occurs: E.g., by setting the buffer to near zero, the latency drops almost immediately at the price of lost packets.

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 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% [24]. 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. Fig. 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 latency from the packets in flight along the slow path $u-w-v$ which was moved into the buffer at v after the switch. Fig. 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 [24] or zUpdate [30]) 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. Fig. 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, to fill the pipe. 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 effects measured. In all our experiments, the maximum buffer size at v was consistent with the terms 1, 2, with delays ranging from 2ms to 40ms. Similarly, when we set the buffers to be too small to "store" all extra packets (i.e., to realistic sizes, e.g., 32KB), the remaining packets were dropped, causing packet loss. Thus, motivated by our findings, we will now develop a lossless flow framework.

III. Model & Problem Setting

We first define some common notation, such as network, latency, or flow, before describing network updates. Afterwards, in Sec. IV, we show that lossless migration is NP-hard.

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

Definition 1. *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 $out(v)$ and the set of incoming edges (u, v) by $in(v)$. A* network *N is a pair (G, c), with $c : E \to \mathbb{R}^+$ assigning each edge $e \in E$ a* capacity *of $c(e)$.*

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

Definition 2. *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 besides not dropping packets due to edges being over capacity, we also want to avoid congestion in the form of buffer build-ups, as seen in the example of Fig. 1.

As thus, we effectively model the available buffer sizes as zero (i.e., packets cannot wait), 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 paper.

Definition 3. *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.,*

$$\forall v \in V \setminus \{s, t\} : \sum_{e \in out(v)} F(e) = \sum_{e \in in(v)} F(e), \quad (4)$$

$$\sum_{e \in out(s)} F(e) = d_F = \sum_{e \in in(t)} F(e), \quad (5)$$

$$\forall e \in E : F(e) \leq c(e), \quad (6)$$

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

Definition 4. *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 We 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 of a flow F'.

Definition 5. *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 instantaneous operation on the ingress router s. In practice, this can be achieved by a two-phase protocol as described in, e.g., [46]: 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'. 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.

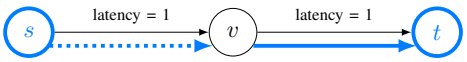

Fig. 5. This picture depicts the network one time unit after the ingress router s switches from F (solid) to F' (dashed). Note that atomicity can only be guaranteed per router, not for the whole network.

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

Definition 6. *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 called* **latency-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 6 being applied, consider the network update in Fig. 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 latency-consistent.

IV. *Latency*-Consistency is Intractable

We will now show that *latency*-consistency is an NP-hard problem, already for a single splittable flow on a DAG. The hardness also holds if the latencies are not known exactly.

***Latency*-consistent migration is NP-hard** Our first reduction is 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 1. *Deciding if a latency-consistent migration from F to F' exists is NP-hard.*

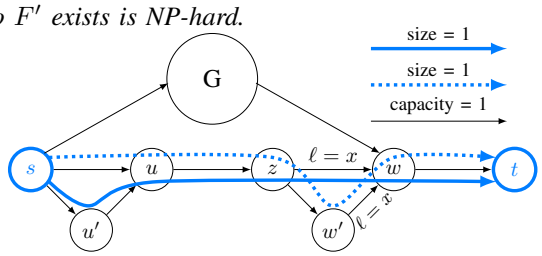

Fig. 6. In this network, the task is to migrate in an *latency*-consistent way from the flow F, depicted in solid blue, to the flow F', depicted as a blue dashed path. All edges have a capacity of 1 and a latency of 1, except for (z, w) and (w', w) with a latency of x. The only way to migrate in an *latency*-consistent 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 *latency*-consistently migrate F to P, w, t, then to F'.

Proof: Let $I = (G = (V, E))$ be an instance of the Hamiltonian path [21] problem, with $|V| = x + 2$ nodes. We construct an instance I' for *latency*-consistent migration, which is solvable if and only if I contains a Hamiltonian path (of length $x + 1$). An illustration is depicted in Fig. 6.

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.

Observe that a network update from F to F' is not *latency*-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 exists if and only if G contains a Hamiltonian path $\mathcal{H}$ of length $x + 1$ Then, one can update *latency*-consistent 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 1 still works even if the flows are splittable (during the migration), as the problem of finding a longer path remains. Furthermore, as $x \in O(n^2), x \in \mathbb{N}$ in the above proof, we can turn all latencies to 1 by replacement with paths of length x.

Corollary 1. *Deciding if a latency-consistent migration from F to F' exists is NP-hard, for both splittable and unsplittable flows, even if all latencies, capacities, and demand sizes are 1.*

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

Corollary 2. *Deciding if any flow F' exists, F' being different to F, but of same or larger size, s.t. the network update (N, F, F') is latency-consistent is NP-hard, even if the flows are splittable and all latencies, capacities, demand sizes are 1.*

It is not just about finding the longest path While the longest path problem is intractable on general graphs, it can be solved in linear time on DAGs [12]. Nonetheless, *latency*-consistency remains NP-hard, via reduction from PARTITION [21]: in the proof of Theorem 1, we can replace the general graph G with a DAG s.t. the task of finding a path through the DAG of a certain length is equivalent to solving PARTITION. The specific graph construction is detailed in [25, Fig. 9, p. 285].

Corollary 3. *Latency-consistent migration is NP-hard on DAGs, already for a single splittable flow of unit size.*

Hardness also holds for latency ranges From a practical point of view, it is unrealistic to know the latency values of every edge with 100% accuracy. Much rather, one can assume that the latency of an edge e falls into a certain (uncertainty) range $[a, b]$, but the specific value is unknown (or might shift).

However, this assumption does not make the migration problem easier from a theoretical point of view, as we can transform the fixed latencies of hardness instances into non-empty latency ranges, retaining the NP-hardness property: e.g., in the proof for Theorem 1, replace the latency from z to w with $[1, x-3]$, from w' to w with $[2x+3, 3x+3]$, and all other latencies with $[1, 2]$. As such, we still need to find an intermediate Hamiltonian path in G from s to w with a latency range of $[x+3, 2x+6]$. Similar examples can be constructed where all edges have an identical latency range, e.g., of $[1, 2]$.

V. Adapting Current Systems

To our knowledge, current network update mechanisms that aim at lossless migration of live flows overlook the effects of latency in the network. 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: the network is still a distributed system, where the individual routers could execute their updates in any ordering, cf. the studies in [23], [27], [29], unless orchestrated in e.g., rounds.

For example, *zUpdate* defines a network update of multiple flows $F_1, \ldots, F_k$ to be *lossless* if the following condition holds:

$$\forall e \in E : \sum\nolimits_{1 \leq i \leq k} \max\left(F_i\left(e\right), F_i'\left(e\right)\right) \leq c\left(e\right) \ , \qquad (7)$$

which we call $\max$-*consistent*. 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 congesting their own path due to latency. When considering Fig. 1, all edges satisfy condition (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. Nonetheless, systems such as *SWAN* [24] or *Dionysus* [27] are custom-tailored, i.e., they cannot decide if a consistent flow migration exists in general. As such, we will be able to "bake" losslessness into them.

A. A Dependency Graph Approach: Dionysus

For the consistent migration of unsplittable flows, *Dionysus* [27] builds a dependency graph based on the old and new flow paths. E.g., in order to move flow F_1, first move F_2 or F_3. No intermediate flow paths are considered. Their approach relies on turning the dependency graph into a DAG during the runtime, which can be traversed in a topological order. Cycles are broken via rate-limiting flows to avoid deadlocks.

Adapting *Dionysus* to latency can be done in a straightforward way. Observe that a single path change of an unsplittable flow can be simulated in time-steps, based off the latency difference between intersections of the old and the new path. To come back to the above F_1, F_2, F_3 example, if all three flows can be updated in a *latency*-consistent way, then we have two possible update orders for F_1: first F_2 or F_3, then F_1.

Corollary 4. *Let $\mathcal{G}$ be a dependency graph where individual updates are latency-consistent. If $\mathcal{G}$ is a DAG,* Dionysus *will perform a latency-consistent flow migration.*

B. Slack Based Approaches: SWAN

A key concept used by *SWAN* [24] 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* is that consistent flow migration is easy if there is relative slack s_R on *every* edge, a technique which we will call the *slack method*: e.g., if s_R is $1/10$, one can migrate in 9 *relaxed*-consistent updates by moving 10% of the flow each time to the new flow rules, never breaking capacity constraints[2].

As free capacity on every edge cannot be assumed in general, *SWAN* proposes to rate-limit background flows of low priority, creating a suitable environment for consistent flow migration. Should rate-limiting not be possible, *SWAN* proposes an LP-based approach, which aims at finding a fast migration, but cannot decide infeasibility. In particular, for each flow change, they enforce Condition 7, which we showed to be not lossless.

Enforcing enough space for old and new flow A straightforward way to "fix" Condition (7) would be to replace the maximum operation with the addition operation [19], i.e.,

$$\forall e \in E : \sum\nolimits_{1 \leq i \leq k} \left(F_i\left(e\right) + F_i'\left(e\right)\right) \leq c\left(e\right) \ , \qquad (8)$$

which we call $\oplus$-*consistent*. Now, the network update in Fig. 1 is no longer detected as lossless. In fact, the lossless property always holds: if there is enough capacity to accommodate the flow twice, the latency on the edges is irrelevant for congestion.

Corollary 5. *The slack-method is latency-consistent. Replacing* $\max$*-consistency with* $\oplus$*-consistency in* SWAN*'s LP makes it (and analogous approaches) latency-consistent.*

Completeness? However, Condition (8) is "too strong", cf. Fig. 7: 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 Fig. 7 violates condition (8)!

[2]As proposed by Alizadeh et al. [2] with the *HULL* architecture, leaving some slack capacity can also be leveraged for greatly reducing latencies in DCNs. Slack based approaches can thus trade some migration speed for latency.

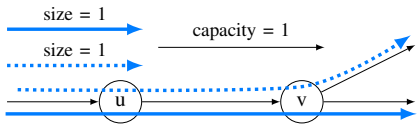

Fig. 7. When changing from the solid (F) to the dashed path (F'), Condition (7) is satisfied (max-consistency) on all edges, but Condition (8) ($\oplus$-consistency) is violated on the incoming edges of u and v.

This is no contradiction to the Corollaries 4 and 5: they only apply to special cases (*latency*-consistent dependency-DAG, slack everywhere) or sophisticated heuristics (LPs).

Hence, we are in a bit of a conundrum, having to choose between computational intractability (*latency*-consistency, Thm. 1), accepting packet loss (max-consistent, [8], [24], [27], [30]), or only being able to cover special scenarios (Cor. 4, 5).

VI. Relax for Lossless Updates

Given the (computationally) problematic situation pointed out at the end of the last section, we now devise a tractable model where we are not confined to the restricted scenarios of *Dionysus* and *SWAN*. Inspired by Ludwig et al.'s [16] approach to accelerating loop-free updates ("*It's Good to Relax!*"), we *relax* the latency requirements for flow migration. Specifically, we formulate the relaxed flow migration consistency in such a way that it is *oblivious* to the latency in the network. In other words, consistency has to hold for all latency settings.

Definition 7. *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 relaxed-*consistent*, if for all possible latencies ℓ holds: (N, F, F') is latency-consistent.*

To give some examples, the scenarios in Fig. 7 and Fig. 5 are *relaxed*-consistent, but not the one in Fig. 1.

Before presenting our polynomial *relaxed*-consistent migration algorithm in Sec. VI-B, we present some discussions and definitions in Sec. VI-A, motivating our algorithmic approach.

A. Further preparatory considerations

***Relaxed*-consistency for unsplittable flows: NP-hard** By using analogous arguments as the reductions in [8, 3-SAT] or [56, PARTITION], it is easy to show that *relaxed*-consistent flow migration is NP-hard if the flows always have to be unsplittable. Hence, we will allow the splitting of flows during the migration process to obtain tractability.

Splitting flows vs. n-splittable flows A natural approach, inspired from mathematical flow theory, is to let the flow packets arriving at routers 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. Even if the splitting were to be 100% exact, already moving a single splittable flow once can create exponentially many different utilizations on an edge. Thus, as we do not want to drop packets and desire a tractable decision process, we use another approach, as done by, e.g., Google's *B4* [49]: we split each flow into (unsplittable) flow paths, each having its own rules, eliminating probabilistic splitting.

***relaxed*-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 *relaxed*-consistent migration will begin and end with a set of unsplittable multi-commodity flows $\mathcal{F}, \mathcal{F}'$, but in intermediate updates, each flow F_i can consist of multiple paths:

Definition 8. *A sequence of r relaxed-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 relaxed-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 $d_{F_i} < d_{F'_i}$.

Augmentation is not lossless The only polynomial approach that can check general feasibility for the max-consistency model is by Brandt et al. [8] via augmenting flows. Specifically, for each flow, they consider the residual network and search for an augmentation that frees up some capacity on some edge at full capacity. However, their augmentation idea "interleaves" a flow with itself during updates, not being oblivious to latency.

Nonetheless, we are inspired by their work in the sense that their approach is *combinatorial*, and not via linear programming, as many other works such as, e.g., [30], [56].

We now present our *relaxed*-consistent flow migration:

B. A relaxed-consistent flow migration algorithm

Our algorithm construction has two phases: first, as in, [8], [24] we check for slack generation. However, the process is different, via the concept of so-called divergence nodes, introduced in the next paragraph (and the constraint that the intermediate updates have to be *relaxed*-consistent). Second, once the slack generation has finished, we can move the flows in partial steps, see Algorithm 2.

Slack and divergence nodes 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 7, 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 slack method.

In the following Algorithm 1 we will give a method to check how slack can be generated on as many edges as possible under the restriction of *relaxed*-consistent network updates:

Lemma 1. *Algorithm 1 creates slack on every edge where slack can be created via a relaxed-consistent migration (i.e., using only relaxed-consistent updates).*

Proof: The performed network updates by Algorithm 1 are *relaxed*-consistent, as flows are only split and rerouted via

Algorithm 1: *Slack creation*

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 w 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 multi-commodity flow $\mathcal{F}^* = (F_1^*, \ldots, F_k^*)$.

Algorithm 1: Recall that F_i is a collection of unsplittable flows $F_{i,j}$ from s_i to t_i, initially just $F_{i,1}$. The algorithm creates slack on all edges where slack can be created by a sequence of *relaxed*-consistent network updates. The slack generation is non-destructive in the sense that if an edge has slack at some point, it always keeps some slack.

paths of free capacity. Assume that there exists some *relaxed*-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 1 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 1 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 $O(nm^3)$.

Lemma 2. *Algorithm 1 has a runtime of $O(nm^3)$, performing at most m network updates, introducing $\leq m$ new flows.*

We can now use Algorithm 1 to check (and in the positive case, perform) for a *relaxed*-consistent migration. The idea is as follows: we see if we can create slack on all edges beyond 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. Else, no *relaxed*-consistent migration is possible.

Theorem 2. *Algorithm 2 decides the feasibility of relaxed-consistent flow migration. If the answer is positive, a relaxed-consistent migration schedule is provided.*

Proof: We begin by showing that the migration is *relaxed*-consistent. The migration via Algorithm 1 is *relaxed*-consistent due to Lemma 1. Note that every *relaxed*-consistent network update is also *relaxed*-consistent if applied in reverse. Hence, using Algorithm 1 in reverse is a *relaxed*-consistent migration as well. Lastly, when using the slack method, we only insert an amount of flow to edges that is at most the available slack.

We now show that else no *relaxed*-consistent migration is possible. Assume that for some i there is an edge e^* behind

Algorithm 2: *Flow migration*

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

1) Execute Algorithm 1 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 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 *relaxed*-consistent migration is possible.
   a) Migrate from $\mathcal{F}$ to $\mathcal{F}^*$ as described in Algorithm 1.
   b) Migrate from $\mathcal{F}^*$ to $\mathcal{F}'^*$ using the slack method.
   c) Migrate from $\mathcal{F}'^*$ to $\mathcal{F}'$ using Algorithm 1 in reverse.

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

Algorithm 2: The algorithm first utilizes two executions of Algorithm 1 to check if a *relaxed*-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 *relaxed*-consistent network updates.

$v_{F_i,F_i'}$ in F_i s.t. Algorithm 1 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 *relaxed*-consistent updates), but that there is a *relaxed*-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 U, as *relaxed*-consistency requires some slack on e^*. Thereby, U could be used to create slack on e^*, a contradiction to the above assumption. ■

Complexity of *relaxed*-consistency Note that the number of network updates and new flow rules created by the executions of Algorithm 1 are at most $2m$, respectively. When migrating between $\mathcal{F}^*$ and $\mathcal{F}'^*$, no further flow rules are needed for the updates of the slack 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 slack method (perform $\lceil 1/s_R \rceil$ network updates of the same type) is dominated by executions ofAlgorithm 1 with runtimes of $O(nm^3)$ (Lemma 2).

Corollary 6. *Let s_R be the smallest relative slack created by Algorithm 1 on $\mathcal{F}, \mathcal{F}'$. Algorithm 2 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 We note that in the current form, our migration algorithm does not respect waypointing (e.g., firewalls) or service chains in general, cf. [5], [32]. However, observe that each intermediate (splitted) flow has its own forwarding rules, as we use the method of [46]. As thus, we can extend our migration algorithm to enforce such rules. E.g., intermediate paths are only allowed if they pass a firewall.

Relation to $\oplus$-consistency Lastly, as $\oplus$-consistent updates only move flows to edges that currently have sufficient capacity, they are also *relaxed*-consistent. The reverse is false though, there are *relaxed*-consistent updates that are not $\oplus$-consistent.

Observation 1. *All $\oplus$-consistent updates are relaxed-consistent. The reverse is not true, see for example Fig. 7.*

As thus, Algorithm 2 can be used to cover the feasibility space of $\oplus$-consistency, extending it to *relaxed*-consistency.

VII. Further Mitigation Techniques

In the previous sections, we discussed how to achieve a lossless flow migration under theoretical model assumptions. We can imagine that in practice, some of the following considerations can be useful for future (heuristic) improvements.

Scheduling ahead In specialized environments such as, e.g., single-tenant data-centers, one can also take an orthogonal approach to network updates, by scheduling all the traffic [22], [28], [44], often eliminating the need for the migration of flows. Should flows arrive online though, flow migration is unavoidable for optimal traffic utilization, as proven in [40].

Waiting for the gap Using the burstiness of TCP as an advantage, Sinha et al. [48] defined a *flowlet* as "*a burst of packets from the same flow followed by an idle interval*", allowing them to be switched independently. The latest generation of data-center load-balancers such as *LetFlow* [51] uses that observation to shift traffic during these frequent gaps without causing packet reordering. Burstiness varies between different types of TCP flows (e.g., porcupine and cheetah) [45], with burstiness being low when, e.g., controlling the sending rates [24]/using UDP.

We can imagine using flowlets in systems such as, e.g., *Dionysus*, Sec. V-A: in case a dependency-cycle has to be broken, wait for an eventual large gap to proceed consistently.

Accept losses By using failure-resilient protocols and efficient hardware, losses can often be recovered quickly [49]. Furthermore, the network asynchrony can be kept low by using timed protocols [41]. Nonetheless, even with perfect timing and synchronization, packet loss can occur, recall Fig. 1.

Don't migrate For cases such as remote surgery [38], where resilience is of much higher importance than any positive migration impact, link protection can come into play [52].

VIII. Related Work

The two-phase approach of Reitblatt et al. [46] is easy to deploy and efficient: the old and new network state exist in parallel until all packets from the old state are drained from the network, preventing the issues of blackholes, loops, and enforcing packet coherence. Their idea is not aimed at eliminating congestion, but still mitigates it in many aspects. We use their mechanism in this article to guarantee an atomic switch between flows from the same source (cf. Section III).

SWAN [24] and *zUpdate* [30] target splittable flows and propose to leave a fraction of free bandwidth ("slack") on every link, allowing for a network migration in staged partial moves, see also Sec. V-B. For example, should 25% of each link's capacity not be utilized, they finish in three moves. In case this slack cannot be guaranteed, even with shutting down background traffic, they employ an LP-based binary search to find a sequence of moves that complies with the different flows not violating bandwidth constraints. Zheng et al. [56] take an orthogonal approach and aim to minimize the congestion for a given sequence length, showing the NP-hardness of this problem for unsplittable flows. All three works cannot decide if such a consistent sequence exists however, but follow-up work showed that it can be decided for splittable flows in polynomial time using augmenting flows [8], respectively in exponential time for unsplittable flows [14]. As they all ignore the effect of edge latency, and model the edge utilization of e as $\max(F(e), F'(e))$, their network updates are not lossless.

The migrations described in [19], [33] are lossless, but their model is not complete, see Sec. V, furthermore not accounting for intermediate paths. Similarly, the model in [9] is lossless, but only covers one destination, not migrating to specified new paths, but rather placing new flows anywhere.

Dionysus [27] builds a dependency graph for possible migrations and runs a greedy algorithm to find an ordering of network updates to reach the desired network state. Should the greedy algorithm not be able to proceed with the flow migration in a consistent manner, then blocking flows are rate-limited for progress. We adapt their work in Sec. V-A.

Mizrahi et al. tackle the problem of asynchrony: as updates are installed in a distributed fashion in the network's routers, the atomicity of a network update cannot be guaranteed. They propose the mechanisms *TIMEFLIP [42] / timed-based consistent updates [41]* to ensure that rules adhere to timing constraints. Their approach does not prevent losses in Fig. 1.

Order replacement updates for flows have also been studied in [3], [4] for multiple flows, but in an asynchronous setting without latencies. The concept of timed updates has been recently studied by Zheng et al. [55], [57] for unit latencies, where flows have to be migrated from an old to a new path. To perform the updates, the nodes along the path can change their forwarding behavior, from old to new, synchronously. In this context of so-called *order replacement updates*, for a single flow, the authors give a polynomial time algorithm for a capacity-respecting update order. Single updates for single flows have also been studied in [15]. The difference between the mentioned order replacement updates and the by us employed two-phase approach is at a conceptual level: order replacement updates are inherently less powerful as they can only shift between old and new rules at each node, but on the other hand, they do not require the overhead of tagging each flow-packet. In between are Nguyen et al. [43], using a two-phase approach, where flows may only use paths in the union of old/new routes. They propose a decentralization approach, see also [20, §5].

Lastly, we refer to the recent survey in [18] for a general overview and discussion on consistent updates in SDNs.

IX. Conclusions

Our experiments in a small testbed showed that current consistent flow migration techniques induce packet loss, due to ignoring latency. As thus, we gave a flow migration model that takes latency into account, proving that the problem of finding a lossless migration is NP-hard, already for a single splittable flow, for both fixed latencies and latency ranges.

Notwithstanding, we showed how to adapt state of the art systems such as *SWAN* or *Dionysus* to being lossless, as they only cover consistency in restricted flow migration scenarios.

In order to develop a general and polynomial lossless flow migration algorithm, we relaxed the problem setting, namely to being oblivious to the latencies in the network. In this new relaxed model, we presented a multi-commodity flow migration scheme, proving it to be polynomial, general, and lossless.

We also briefly discussed heuristic mitigation techniques.

We hope our work can be useful for adapting and developing further traffic engineering systems, especially those that take balancing a multitude of consistency properties into account.

Acknowledgements. The authors would like to thank Sebastian Brandt for valuable insights and discussions.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.*, 40(4):63–74, 2010.

[3] S. A. Amiri, S. Dudycz, M. Parham, S. Schmid, and S. Wiederrecht. On polynomial-time congestion-free software-defined network updates. In *IFIP Networking*, 2019.

[4] S. A. Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht. Congestion-free rerouting of flows on dags. In *ICALP*, 2018.

[5] S. A. Amiri, K.-T. Foerster, R. Jacob, and S. Schmid. Charting the algorithmic complexity of waypoint routing. *SIGCOMM Comput. Commun. Rev.*, 48(1):42–48, 2018.

[6] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid. Transiently consistent SDN updates: Being greedy is hard. In *SIROCCO*, 2016.

[7] A. Basta, A. Blenk, S. Dudycz, A. Ludwig, and S. Schmid. Efficient loop-free rerouting of multiple SDN flows. *IEEE/ACM Trans. Netw.*, 26(2):948–961, 2018.

[8] S. Brandt, K.-T. Foerster, and R. Wattenhofer. On consistent migration of flows in sdns. In *INFOCOM*, 2016.

[9] S. Brandt, K.-T. Foerster, and R. Wattenhofer. Augmenting flows for the consistent migration of multi-commodity single-destination flows in sdns. *Pervasive Mob. Comput.*, 36:134–150, 2017.

[10] M. Casado, N. Foster, and A. Guha. Abstractions for software-defined networks. *CACM*, 57(10):86–95, 2014.

[11] F. Clad et al. Computing minimal update sequences for graceful router-wide reconfigurations. *IEEE/ACM Trans. Netw.*, 23(5):1373–1386, 2015.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[13] K.-T. Foerster. *Don't disturb my Flows: Algorithms for Consistent Network Updates in Software Defined Networks*. Phd. thesis, ETH Zurich, Switzerland, September 2016.

[14] K.-T. Foerster. On the consistent migration of unsplittable flows: Upper and lower complexity bounds. In *IEEE NCA*, 2017.

[15] K.-T. Foerster. On the consistent migration of splittable flows: Latency-awareness and complexities. In *IEEE NCA*, 2018.

[16] K-T. Foerster, A. Ludwig, J. Marcinkowski, and Schmid S. Loop-free route updates for software-defined networks. *IEEE/ACM Trans. Netw.*, 26(1):328–341, 2018.

[17] K.-T. Foerster, R. Mahajan, and R. Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking*, 2016.

[18] K.-T. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Commun. Surveys Tuts.*, 2018.

[19] K.-T. Foerster and R. Wattenhofer. The power of 2 in consistent network updates: Hard loop freedom, easy flow migration. In *ICCCN*, 2016.

[20] K.T-. Foerster, T. Luedi, J. Seidel, and R. Wattenhofer. Local checkability, no strings attached: (a)cyclicity, reachability, loop free updates in sdns. *Theor. Comput. Sci.*, 709:48–63, 2018.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[22] S. Ghorbani and M. Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *HotSDN*, 2012.

[23] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring control plane latency in sdn-enabled switches. In *SOSR*, 2015.

[24] C.-Y. Hong et al. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[25] A. Itai, Y. Perl, and Y. Shiloach. The complexity of finding maximum disjoint paths with length constraints. *Networks*, 12(3):277–286, 1982.

[26] J. Perelló et al. All-optical packet/circuit switching-based data center network for enhanced scalability, latency, and throughput. *IEEE Network*, 27(6):14–22, 2013.

[27] X. Jin et al. Dynamic scheduling of network updates. *In SIGCOMM'14*.

[28] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. In *SIGCOMM*, 2014.

[29] M. Kuzniar, P. Peresíni, and D. Kostic. What you need to know about SDN flow tables. In *PAM*, 2015.

[30] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate: updating data center networks with zero loss. *SIGCOMM 2013*.

[31] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. Transiently policy-compliant network updates. *IEEE/ACM Trans. Netw.*, 26(6), 2018.

[32] T. Lukovszki and S. Schmid. Online admission control and embedding of service chains. In *SIROCCO*, 2015.

[33] L. Luo, H. Yu, S. Luo, and M. Zhang. Fast lossless traffic migration for SDN updates. In *ICC*, 2015.

[34] L. Luo, H.-F. Yu, S. Luo, M. Zhang, and S. Yu. Achieving fast and lightweight SDN updates with segment routing. In *GLOBECOM*, 2016.

[35] S. Luo, Z. Li, J. Wang, and H.-F. Yu. Simplifying flow updates in software-defined networks using atoman. In *IEEE Access*, volume to appear, 2019.

[36] S. Luo, H.-F. Yu, L. Luo, and L. Li. Arrange your network updates as you wish. In *IFIP Networking*, 2016.

[37] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. In *HotNets*, 2013.

[38] J. Marescaux, J. Leroy, F. Rubino, M. Smith, M. Vix, M. Simone, and D. Mutter. Transcontinental robot-assisted remote telesurgery: feasibility and potential applications. *Annals of surgery*, 235(4):487, 2002.

[39] E. Marilly, O. Martinot, H. Papini, and D. Goderis. Service level agreements: a main challenge for next generation networks. *ECUMN*, 2002.

[40] T. Mizrahi and Y. Moses. On the necessity of time-based updates in SDN. In *ONS*, 2014.

[41] T. Mizrahi, E. Saat, and Y. Moses. Timed consistent network updates in software-defined networks. *IEEE/ACM TON*, 24(6):3412–3425, 2016.

[42] T. Mizrahi et al. Timeflip: Using timestamp-based TCAM ranges to accurately schedule network updates. *Trans. Netw.*, 25(2):849–863, 2017.

[43] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. Decentralized fast consistent updates. In *SOSR*, 2017.

[44] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *SIGCOMM*, 2014.

[45] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP revisited: a fresh look at TCP in the wild. In *IMC*, 2009.

[46] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[47] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill. RADWAN: rate adaptive wide area network. In *SIGCOMM*, 2018.

[48] S. Sinha, S. Kandula, and D. Katabi. Harnessing tcp's burstiness with flowlet switching. In *HotNets*, 2004.

[49] J. Sushant et al. B4: experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[50] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM*, 2009.

[51] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. *NSDI*, 2017.

[52] J.-P. Vasseur, M. Pickavet, and P. Demeester. *Network recovery: Protection and Restoration of Optical, SONET-SDH, IP, MPLS*. Elsevier, 2004.

[53] S. Vissicchio and L. Cittadini. Safe, efficient, and robust SDN updates by combining rule replacements and additions. *IEEE/ACM Trans. Netw.*, 25(5):3102–3115, 2017.

[54] S. Vissicchio et al. Safe update of hybrid SDN networks. *IEEE/ACM Trans. Netw.*, 25(3):1649–1662, 2017.

[55] J. Zheng, B. Li, C. Tian, K.-T. Foerster, S. Schmid, G. Chen, and J. Wu. Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In *ICDCS*, 2018.

[56] J. Zheng, Hong Xu, Guihai Chen, and Haipeng Dai. Minimizing transient congestion during network update in data centers. In *ICNP*, 2015.

[57] Jiaqi Zheng, Guihai Chen, Stefan Schmid, Haipeng Dai, Jie Wu, and Qiang Ni. Scheduling congestion- and loop-free network update in timed sdns. *IEEE J. Sel. Areas Commun.*, 35(11):2542–2552, 2017.

[58] D. Zhuo, M.Ghobadi, R. Mahajan, K.-T. Foerster, A. Krishnamurthy, and T. E. Anderson. Understanding and mitigating packet corruption in data center networks. In *SIGCOMM*, 2017.