Consistent Network Management for Software-defined Networking based Multicast

Thomas Kohler, Frank Dürr, and Kurt Rothermel

***Abstract*—Updating a network is an essential and continual task in the management of today's softwarized networks. When applying updates on distributed network elements, desired network properties, such as drop- and loop-freeness, might be transiently violated. Although being crucial, update consistency has yet been less considered in network management.**

In this article, we argue for incorporating the particularities of update consistency into the reconfiguration process of continuous network management. We present a generic management architecture allowing for an appropriate selection of an update mechanism and its parameters based on expected inconsistency effects. We investigate update consistency for the case of multicast routing and show in an extensive analysis why simultaneous drop- and duplicate-freeness is not possible. We present an update procedure for multicast routing updates that identifies critical update steps, which are fed back into the reconfiguration process, along with a lightweight approach that allows for the selection of an update strategy, preventing either drops or duplicates. Furthermore, we present an optimization of an existing powerful, but resource-intensive update approach as well as an approach for in-network filtering of duplicates.

***Index Terms*—Software-defined networking, configuration management, update consistency, quality-of-service management, multicast communication**

I. Introduction

SOFTWARE-DEFINED NETWORKING (SDN) has been gaining substantial attention from both, academia and industry. Large internet companies like Google and Facebook have been adopting the SDN paradigm both within data center networks (DCNs) [1] and their interconnections in global wide area networks (WANs) [2]. A large internet service provider has set the goal of controlling 75% of their network with software by 2020 [3].

One key element of SDN is the logical centralization of network control in form of a logically centralized controller with global knowledge over SDN-enabled switches, their interconnections as well as end hosts. Thus, SDN has fostered the evolution of network management [4], [5], to allow for complex management of heterogeneous network elements and possibly overlapping network functions. Furthermore, today's softwarized networks are in constant flux, continuously adapting to changes in the network topology, load and network functions, e.g., through migration of virtual machines or dynamic scaling processes in Network Function Virtualization (NFV). This high degree of dynamics has blurred the borders of network operation and network management and has led to their convergence.

The authors are with the Institute for Parallel and Distributed Systems (IPVS), University of Stuttgart, 70569 Stuttgart, Germany
e-mail: {firstname.lastname}@ipvs.uni-stuttgart.de.

To implement adaptations, the network has to be reconfigured inevitably. Updating the data plane of an SDN-based network can be described as transition from an old to a new global network state. Centralized control allows for consistent updates, such that certain network properties (invariants), e.g., *no packet should be dropped*, hold during a transition. For instance, in consistent route updates, invariants such as avoidance of black-holes, which occur when packets are dropped on-route due to a missing rule, or avoiding micro-loops due to transient cycles, hold not only before and after the update but also during route transition [6], [7]. Network functions are encoded in a set of rules, installed in inherently distributed elements of the network, i.e., the switches. To advance to a new network state, the switches have to be updated individually. Since the delay until a rule update has been successfully applied varies among the switches, packets may be processed according to a mixture of new and old rules as they traverse the network, while it is being updated. Thus, properties that hold in the old and new network state my not hold in an intra-update state, possibly violating network invariants.

The violation of invariants in the best case reduces the network's efficiency, possibly violating soft requirements such as Quality of Experience (QoE). In the worst case, it may severely jeopardize stipulated hard invariants, e.g., when black-holes occur in routing, which is a network function of utmost importance. Since the management of networks includes definition and realization of requirements on the network operation, we argue to create awareness about update consistency in network management in order to be able to circumvent effects that stem from update inconsistencies. This implies the selection of suitable update methods based on the network specifications to maintain.

To achieve update consistency, two classes of approaches have been proposed: a state-based approach and stateless update ordering. The state-based approach is very powerful. It guarantees maintenance of for arbitrary consistency properties [6]. However, it relies on state information, injected in the packets and encoded in vacant header fields. Thus its applicability is limited, in particular, when such header fields are used for the specification of flows or by the application layer. In practice, the VLAN tag field is typically used in this approach, which is only feasible when the network configuration does not use VLANs itself. Furthermore, it requires to store both, the old and new rules at the same time in each switch. Hardware switches store their flow table in ternary content-addressable memory (TCAM), which is a power-hungry and limited resource, with typical capacities of only up to 2000 rules for current hardware switches [8], [9]. Therefore, stateless approaches based on an appropriate ordering of updates received a lot of attention.

Although update ordering approaches already solve many problems for unicast route updates and even minimal procedures for some invariants exist [7], there are no thorough investigations of update ordering approaches for multicast, which offers efficient many-to-many communication, although in particular multicast could benefit very much from consistent updates to increase performance and user experience. Furthermore, we will show in this article that update consistency entails numerous crucial particularities which are to be considered in the management of multicast networks. Multicast is one network paradigm that tremendously benefits from SDN [10]. Opposed to the distributed creation and maintenance of multicast distribution trees, logical control centralization tremendously reduces complexity and allows for optimized routing, as shown in [11] and in this article. Membership management and control naturally benefit from centralization alike. Multiple types of applications already rely on multicast, including large-scale media streaming in WANs, like distribution of live television broadcasts using IPTV. To provide resiliency, both, data and services in data centers are typically distributed and replicated, using e.g., Apache Hadoop or Infinispan, heavily relying on multicast. We thus see SDN as an enabler for multicast and expect to see a significant boost in the adoption of multicast in both, DCNs and WANs.

In this article, we make the case for update consistency awareness and perform an in-depth analysis for the concrete case of multicast networks. We propose an update ordering and hybrid approach that tackles the update problem of avoiding dropped messages and duplicate messages in multicast networks. The relevance of drops is obvious. For instance, in video streams, dropped packets result in dropped frames, which may severely reduce the Quality of Experience (QoE). Duplicate messages waste bandwidth and might lead to bandwidth bottlenecks during updates, which again might degrade the QoE of the application (e.g., a video application not receiving sufficient bandwidth anymore). Furthermore, duplicates might confuse the application if it is not prepared to handle them. As an overall implication of drops or duplicates, temporary degradation of network performance and thus application QoE degradation can be stated.

In detail, our contributions are as follows:

First, we propose a generic system architecture for network management, incorporating knowledge about update consistency to allow for the selection of an appropriate update mechanism and its parameters.

Second, based on our previous work [12], we specify a network update correctness property specific to multicast—*duplicate-freeness*—which has been formerly unconsidered in the context of network update consistency. In our extensive analysis, we then prove that in general it is impossible to avoid violation of two invariants, drop- and duplicate-freeness, for arbitrary multicast network updates using a stateless approach.

Third, we identify and define necessary conditions on multicast tree updates, leading to undesired effects that possibly break invariants. We show that update ordering is a degree of freedom, resulting in maintenance of drop-freeness, while sacrificing duplicate-freeness and vice versa. This allows for an *update strategy* that avoids drops at the cost of duplicates and vice versa. Either behavior can be achieved by a deliberate selection of a respective strategy. We extend [12] by the specification of the *loop-freeness* invariant, which our approach additionally maintains.

Fourth, we conduct a detailed analysis of packets traversing the network while an update is being applied and show that the update order as perceived by the packets may differ, depending on their propagation delay. We show the implications of this reordering and present a method to prevent it.

Fifth, we propose a generic multicast update procedure. We introduce the *path update algorithm*, which decomposes a global multicast network transition into update steps for which invariant maintenance can be guaranteed, leveraging the degrees of freedom identified in the analysis. We outline the involved algorithms and procedures which translate tree changes to SDN rule updates and executes these updates in guaranteed order, maintaining desired invariants. As an extension of our prior work, we present a mechanism to mitigate update-caused duplications through in-network filtering. Furthermore, we present a novel alternative update approach as an optimized state-based approach which maintains arbitrary invariants, while rule space consumption, i.e., TCAM space, is minimized. Overall, we particularly highlight feedback of the prevailing network state and update situation to the network management, as well as its decisions about the approach selection and parameter determination.

The remainder of this article is thus organized as follows: Section II states related work. In Section III a generic system architecture for network management, incorporating update consistency is presented. Henceforth, this scheme is applied to the concrete case of multicast networks, where update consistency is shown to be of particular relevance. Section IV formally introduces the multicast model. Section V provides the problem statement and an in-depth analysis. Section VI describes our flexible update approach for multicast trees utilizing update ordering, with optional duplicate filtering, or an optimized hybrid approach. To investigate the implications of inconsistency effects, in Section VII their frequencies and impact for real WAN scenarios are evaluated, both, analytically and empirically, through direct measurement in the data plane under update. Finally, conclusions are provided in Section VIII.

II. Related Work

SDN constitutes a perfect match to solve key deployment problems that have been impeding the adoption of multicast in large-scale real-world scenarios so far. Especially for data center networks, a management method of multicast in overlay networks is described in [13]. Avalanche enables secure and bandwidth-efficient multicast in DCNs [10]. Scaling and routing of multicast in data-center topologies is investigated in [14]. For multicast-based streaming with multiple simultaneous streams among multiple WAN sites under real-time requirement (3D teleimmersion), [11] introduces an SDN-based control protocol allowing for seamless reconfiguration of the network. Bandwidth and connectivity invariants are maintained using a state-based update procedure. Stateless network updates are not considered in these works. However, the number of recent

works indicate a clear trend for SDN enabled adoption of multicasting in both, DCNs and WANs.

As mentioned in the introduction, in the domain of network update consistency, a state-based mechanism guaranteeing arbitrary invariants [6] as well as stateless approaches, based on update ordering have been proposed. A minimal stateless procedure as well as an overview of correctness properties and their interdependencies are presented in [7]. With the aim to improve update speed, dynamic scheduling of consistent updates respecting these interdependencies is proposed in [15]. A stateless search-based approach is presented in [16]. Neither approach considers the peculiarities of updating multicast networks though.

In the domain of network management, a rather early contribution from the year 2013 explores SDN's ability to ease management and configuration of a variety of networks [4]. The authors propose a network control architecture focusing on the interdependency of high-level network policies, declared in a functional programming language, and low-level network configuration, including its deployment in a campus network. More recent and detailed work on the SDN-enabled management of large-scale networks propose a layered and distributed control plane. The authors of [17] focus on intra-domain control and management of large scale networks and present revised algorithms for hierarchical routing and local link-failure recovery in the context of management distribution, exploiting global network knowledge. In [18], a layered management and control framework for fixed backbone networks is proposed, along with a placement algorithm for control entities. This allows for adaptive resource management operations as demonstrated on two exemplary use cases, adaptive load-balancing and energy management. Neither of these approaches considers efficient incremental updates or identifies the relevance of update consistency.

III. Update Consistency In Network Management

In this section, we describe a generic system architecture for SDN network management. Fig. 1 gives an overview of the proposed system architecture and control flow.

We build on a typical SDN architecture, where management and control are strictly separated from the actual packet forwarding in the data plane. Specifically, our architecture embeds a dedicated control loop that implements an adaptation mechanism providing network updates that are consistent with given consistency criteria and avoiding undesired inconsistency effects. We subsume the control and management plane in a logically centralized entity denoted as ***Network Manager*** (NM), which implements the network configuration logic and handles the communication with the switches to deploy the configuration. We assume the Network Manager to have global knowledge of the data plane. To achieve scalability, the NM might be transparently distributed. ***High Level Policies*** (HLPs) represent the definition of network functions as well as global network constraints (QoS, QoE), the data plane has to implement and adhere to. HLPs might be declaratively defined in a high level network programming language, such as Frenetic [19], [20], or be concretely implemented as SDN

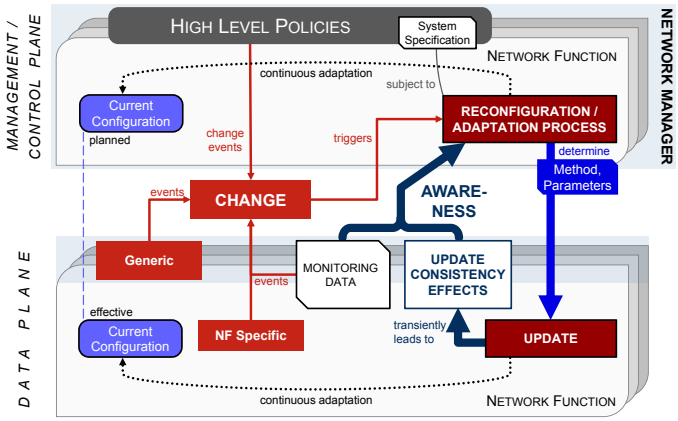


Fig. 1. Overview of the proposed SDN-based system architecture.

controller modules. Multiple network functions, such as routing, traffic monitoring, load balancing, or multicast, may co-exist. Their composition [21] is handled by the Network Manager. Initially, a concrete configuration of a network function NF_i is derived from its HLP_i (planned configuration) and pushed to the data plane where it eventually becomes effective, such that the **effective configuration** (data plane) reflects the **planned configuration** (management plane).

In this article, we focus on *change management*: based on the current configuration, the NM reacts to **changes** in both, the management plane, such as changes of the HLP, and the data plane. Data plane changes may be **generic**, such as topology changes, and thus affecting all network functions, or **network function specific**, such as a host joining a multicast group. In the configuration control loop, change events are interpreted as disturbance and trigger a **reconfiguration process**, in which the NM in reaction adapts to the changed conditions. This consists of two steps: 1) A set of rule updates U that change the current planned configuration of the NM to a new, adapted configuration, has to be calculated. This is naturally highly network function specific. In this article, we will show how incremental updates of multicast traffic distribution trees are calculated. 2) In order to implement the network state transition, the **update** is then to be applied to the data plane. Based on how this is done (**update method**), the update execution possibly leads to transient **inconsistency effects** that might also affect other NFs.

The selection of an **update method** and its **parameters** has severe implications not only on the type of occurring inconsistency effects, but also on their extent, the reconfiguration duration, and data plane resource consumption, i.e., switch rule space. Thus, the NM assesses different types of information, describing the prevailing update situation, in order to decide on an appropriate method and its parameters. This information includes expected update inconsistency effects, specific to the available update methods, characteristics of the triggering change event, the affected HLPs, and **monitoring data** from the data plane, such as statistics of NF-specific or generic flows. While the expected type of effects of an update method are known *a priori* to the NM, its extent can be estimated through a static analysis of U. The monitoring data, e.g., flow-associated packet rates [22], even allows the

NM to empirically estimate the expected number of affected packets. The NM is thus able to evaluate a method-parameter combination against the stipulated system specification or concrete HLP goals. This allows for a dynamic selection of an adequate update mechanism and determination of its parameters on the granularity of NFs and concrete updates. We leave the description of concrete algorithms for selection and quantitative parameter determination for later work, and rather focus on describing specific update methods.

IV. Multicast Model

In this section we describe preliminary assumptions and introduce multicast tree updates along with relevant consistency properties which are defining the problem statement.

Throughout this article, we follow the IP Multicast Service Model (RFC-1112[1]): there are possibly many senders and several receivers, denoted as *multicast group*. Logical addressing assigns a single class D IP address to each group. Group messages are sent to respective destination multicast IP addresses over a distribution tree which defines the routing of the multicast traffic through the network. In our analysis, we focus on multicast traffic distribution and thus only consider switches in the distribution tree, not actual member end hosts. While irrelevant for the analysis, our approach assumes group management to be handled using the Internet Group Management Protocol (IGMP, RFC-2236[1]) as end-host-to-switch protocol.

We call a switch with connected group members (hosts belonging to group) a member switch. Routing is entirely done by SDN switches, which snoop IGMP signaling packets and report group membership to an SDN controller, enabling logically centralized group and tree management. We furthermore assume a network consisting of a set of SDN switches $sw \in SW$ with associated ports $p \in P$ connected over bidirectional non-lossy links $l \in L$ with associated weight $w(l)$, forming a topology graph T. The distribution tree is denoted as a directed acyclic graph (DAG): $G(SW_{MC} \subseteq SW, L_{MC} \subseteq L)$. Switches are associated special roles as depicted in the legend of Fig. 2 and listed below:

- Exactly one root: $S = \{s\} \subset SW_{MC} : deg_{in}(s) = 0$
- At least two group members: $m \in M \subseteq SW_{MC}$
- Relays (single out-port): $rel \in Rel \subseteq SW_{MC} : deg_{out}(rel) = 1$
- Replicators (multiple out-ports): $r \in Rep \subseteq SW_{MC} : deg_{out}(r) > 1$
- Non-tree switches: $SW \setminus SW_{MC}$

Throughout the article, we use the terms switch and node as well as link and edge interchangeably, depending on the current focus on either the networking aspect or its graph-theoretical representation. Packets are intentionally *replicated* and sent out on multiple links of *replicator* switches. We differentiate between relays and replicators, since replicators are shown to be play a distinguished role for update consistency, as we will describe in the analysis. Note that group members, i.e., switches with connected member end hosts, may also relay or replicate, they are not necessarily leaves of the tree. For ease of illustration, we assume a single multicast tree, which either might be a source-based tree or shared tree. We consider only group traffic of one multicast group, i.e., one destination IP address. This simplification does not limit the generality of our approach. It is valid for multiple groups and thus multiple distribution trees as well.

[1]https://tools.ietf.org/rfc/rfc1112.txt; https://tools.ietf.org/rfc/rfc2236.txt

V. Problem Statement and Analysis

In this section, we first given the problem statement (Section V-A). Next, we state the impossibility of combined drop- and duplicate-freeness (Section V-B). We then refine conditions on updates which lead to violation of invariants (Section V-C). We introduce a central structure for both the analysis and approach (Section V-D). We then introduce the *loop-freeness* property (Section V-E). Finally, we conduct an analysis of the update order as seen by packets being in the network during an update (Section V-F).

A. Problem Statement: Multicast Tree Updates

Applying network updates to a multicast network, a global network state translates to a distribution tree instance. We assume distribution tree calculation to be a non-incremental process: in reaction to a topology or membership change, a tree is computed entirely anew, irrespective of the extent of the actual change that triggers that recalculation. Thus, even small changes may result in huge differences in the recalculated tree. Our goal is to advance from an old to a new distribution tree by finding an update order, while maintaining certain invariants. Particularly, we want to guarantee drop-freeness and duplicate-freeness, where the latter informally describes the reception of an unintended duplicate.

B. Impossibility Result

Claim: It is impossible to avoid violation of the two simultaneous invariants drop- and duplicate-freeness for arbitrary transitions using a stateless update method.

Proof: We assume that maintaining both invariants at any time is possible, i.e., there exists exactly one effective path from s to each m_i at any time. Consider a scenario as shown in Fig. 2, where a transformation from G (left) to G' (right), both correct multicast trees, is performed. In G, packets from a source s are sent to a replicator switch 2, which replicates

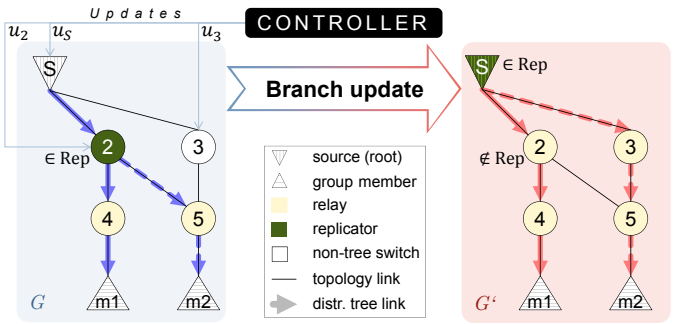


Fig. 2. Update of a branch $s \rightarrow m_2$ (dashed lines) within a multicast distribution tree (G), leading to a new network state, i.e., distribution tree (G').

the packets and forwards them to relay switches 4 and 5, where the respective replicas are forwarded to member m_1 and member m_2 respectively. In G', s takes over the replication and forwards to 2 and 3, while 2 only forwards to 4 and thus does not replicate anymore. To implement this transformation, the output port list of switches s, 2 and 3 have to be updated by rule updates u_i for switch i as follows (to improve readability, we denote updates that add out-ports, i.e., add new paths, as u_i^+, whereas updates that remove out-ports, i.e., remove existing paths, as u_i^-):

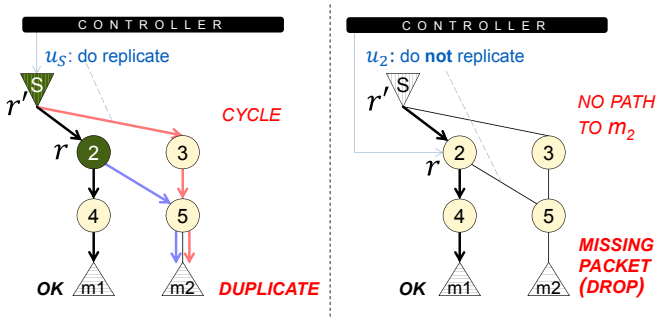
Switch i	Out-port set		
	old (PO_i)	new (PO_i')	update
s	{2}	{2, 3}	$u_s^+ = PO_s \cup \{3\}$
2	{4, 5}	{4}	$u_2^- = PO_2 \setminus \{5\}$
3	$\emptyset$	{5}	$u_3^+ = PO_3 \cup \{5\}$

Path $s \longrightarrow 5$ has to be installed (u_s^+, u_3^+), whereas path $2 \longrightarrow 5$ has to be removed (u_2^-). This implies the "shift" of the replication from 2 to s. The execution order of these updates is crucial: Fig. 3 shows intermediate states of two update order permutations, where u_3^+ and u_s^+ (a) and u_3^+ and u_2^- (b) have been executed. In other words, path $3 \longrightarrow 5$ has been installed first, followed by an update of the new replicator r' to do replication in (a) and, respectively, followed by an update of the old replicator r to stop replication in (b). We use the *happens-before* relation [23] to express an order of events: event e_1 happens before event e_2 iff $e_1 \lhd e_2$. We do not consider delay aspects for now, hence we subsume in an update event u_i the sending of an update message from the controller as well as the reception and the execution at a switch i. A detailed analysis including additional timing aspects, such as propagation delay, is given in Section V-F. Formally, those two cases depict intermediate states of update orders $u_3^+ \lhd u_s^+ (\lhd u_2^-)$ (a) and $u_3^+ \lhd u_2^- (\lhd u_s^+)$ (b), where the respective last update has not yet been executed (in parentheses).

Obviously, a cycle has been introduced in (a) through a new effective path $s \longrightarrow m_2$. A packet p entering the network at s, denoted as event $switch(p, s)$, at that point in time will get replicated twice and follow both paths which results in two replicas reaching m_2, which we call a *duplicate* at m_2. In (b), with $u_3^+ \lhd u_2^- \lhd switch(p, s)$, neither s nor 2 are replicating. Hence, there is no effective path $s \longrightarrow m_2$ at all, resulting in a missing packet at m_2. Note that for (a) it does not matter whether u_3^+ is executed before or after u_s^+. Both orders, $u_3^+ \lhd u_s^+ (\lhd u_2^-)$ (a) and $u_s^+ \lhd u_3^+ (\lhd u_2^-)$, result in an intermediate state with multiple effective paths to m_2. Analogously, there exists an equivalence class of update orders causing drops due to an intermediate state with no effective path to m_2: $u_3^+ \lhd u_2^- (\lhd u_s^+)$ (b), $u_2^- \lhd u_3^+ (\lhd u_s^+)$, $u_s^+ \lhd u_2^- (\lhd u_3^+)$ and $u_2^- \lhd u_s^+ (\lhd u_3^+)$. Hence, in any of all six possible update order permutations, either *duplicate-freeness* or *drop-freeness* is violated which contradicts our assumption. We thus have proven that there exists no correct update procedure w.r.t. both, drop- and duplicate-freeness. □

C. Conditions for Violation of Invariants

Although we have proven that in general we cannot guarantee both desired properties at the same time, we will describe the conditions for the violation in the following.



(a) $u_3^+ \lhd u_s^+ \lhd switch(p, s)$ (b) $u_3^+ \lhd u_2^- \lhd switch(p, s)$

Fig. 3. Traces of packets (arrows), entering the network in two intra-update states, where u_3^+ and u_s^+ (a) and u_3^+ and u_2^- (b) have been executed, resulting in duplicates and drops, respectively.

***Branch update, replicator pair*:** We refer to the kind of update as shown above, where there is a change in the path from s to some branch member, as *branch update*. We define a *replicator pair* (r, r') as a pair of old and new replicator, where $r \in G$, $r' \in G'$. In the example, r is switch 2 and r' is s. Note that, as described, postponing u_3^+ in (a), i.e., $u_s^+ \lhd u_3^+ (\lhd u_2^-)$, does not solve but only defers the problem: as long as $3 \longrightarrow 5$ is missing, duplicates are stopped at 3, however, eventually, u_3^+ and u_2^- have to be executed. Anyway, pushing a critical update, i.e., that update that finally establishes a new path and thus would cause duplicates, from a replicator downstreams along unicast paths constitutes an additional degree of freedom.

This reveals the reason for the impossibility result: replicator updates inherently involve updating a pair of distinct switches, affecting common subsequent switches downstreams of them. Ignoring m_1, this transition would be a trivial unicast path update, were after $3 \longrightarrow 5$ has been installed, the single switch, s, would be updated through a single non-competing update to forward to 3, instead of 2. In our multicast scenario, this is not feasible, since 2 still has to forward packets downstream via 2 to m_2 and thus has to receive packets from s.

***Dependency of replicators*:** In the depicted type of replicator change, both, r and r' of a replicator pair are common elements of at least one path from s to all $m \in M$. Hence, there exists a dependency among (r, r') in forwarding packets on such paths. In such cases, we denote replicators of a pair to be *dependent*. We refine this definition in the following.

***Replicator move (downstream/upstream)*:** If a dependency among (r, r') is present, we call the two associated updates $(u_r, u_{r'})$ a *replicator move*. A replicator move is directed. It is denoted an *upstream move* if r' is upstreams of r, i.e., r' is a (transitive) predecessor of r on at least one path from s to all m. In this case, as in the example, r is dependent on r'. This dependency stems from the dependency of packets, being forwarded on such a path: The events of r receiving a packet p from r' ($e_{r,rec}$) and subsequently sending p further downstreams ($e_{r,snd}$) are causally dependent on the event of r' sending a message to r ($e_{r',snd}$). Thus: $e_{r',snd} \lhd e_{r,rec} \lhd e_{r,snd}$. Vice versa, if r is a predecessor of r', it is denoted a *downstream move*. A minimal illustration of both, a downstream and an upstream move is show in Fig. 6.

***Replicator swap*:** If replicators are not dependent, we call the update a *replicator swap*.

In conclusion, we state that simultaneous drop- and duplicate-freeness is not possible for updates that involve a change of a replicator pair (r, r'). However, depending on the update ordering, one of the inconsistency effects, either drops or duplicates, can be prevented.

This degree of freedom can be leveraged by a deliberate selection of an inconsistency effect that is tolerable, while the other one is prevented. We denote the exploitation of this particular degree of freedom within multicast network update as ***update strategy***. As explained in the introduction, it depends on the concrete application that uses multicast whether violating one property is preferred over violating the other. In general, this decision is to be made by the network manager. Henceforth, we assume drops more fatal than duplicates, and thus, without loss of generality, argue from this perspective.

D. Central Analysis Structure: The Delta Graph

In the following, we introduce a central structure for both, our analysis and approach. As seen in the examples so far, inconsistency effects due to replicator pair updates do not necessarily affect all members. Furthermore, a network update might consist of several replicator pair updates as well as of uncritical updates. To be able to identify relevant nodes of a network update as well as to identify affected nodes of individual replicator moves, we introduce the *delta graph* (G^Δ). It captures differences of two distribution trees G and G'. Informally, it can be constructed by merging G and G', removing all common edges, followed by removing all unconnected vertices. Formally, we define G^Δ as follows:

Def. G^Δ**:** $G^\Delta = \left[(G - L'_{MC}) + (G' - L_{MC})\right] - SW^0_{MC}$, where L_{MC} is the set of links and SW_{MC} is the set of switches of G. Primes indicate reference to G'. SW^0_{MC} denotes unconnected nodes, i.e., $\forall sw \in SW^0_{MC} : deg_{total}(sw) = 0$.

Fig. 4 illustrates an intermediate step after merging exemplary G and G', where ellipsis denote unicast paths of arbitrary length, to provide generalization. Thin, blue edges (bottom paths) are exclusively in G and thus represent old paths, to be removed as part of a network update, whereas thick, red edges (top paths) are exclusively in G' and thus represent new paths, to be installed respectively. Common edges (dashed), both in G and G', and respective nodes (dashed) are not to be changed and thus removed from G^Δ in a subsequent step. We further define a set of *join nodes* $N^>$ as follows:

Def. ***join node*:** $\forall j \in N^> \subset SW^\Delta_{MC} : deg_{in}(j) = 2$. Each j is associated with a replicator move. The effects of a corresponding replicator move affect all downstream nodes of j.

Def. P**,** P'**,** ***split node*** $s^<$**:** We further denote the old path from s to j in G as $s \xrightarrow{P} j$, and the new path from s to j in G' as $s \xrightarrow{P'} j$, analogously. To identify a replicator pair (r, r'), we back-traverse P and P' in G^Δ, starting from j, until no further predecessor exists or a common predecessor in both P and P', denoted as split node $s^<$, is reached. The end nodes on P and P' define r and r', respectively. If $s^<$ exists, a non-competing update, which is performed by an update of a single node, $s^< = r = r'$, is present. Non-competing updates arise, e.g., when a single edge, connecting $s^<$ and j is replaced by a path $s^< \xrightarrow{P'} j$. Since they are not critical in terms of update consistency, we do not consider them to be replicator updates. In the exemplary G^Δ in Fig. 5, four join nodes j_i define replicator moves with respective replicator pairs (r_i, r'_i) and paths (P_i, P'_i).

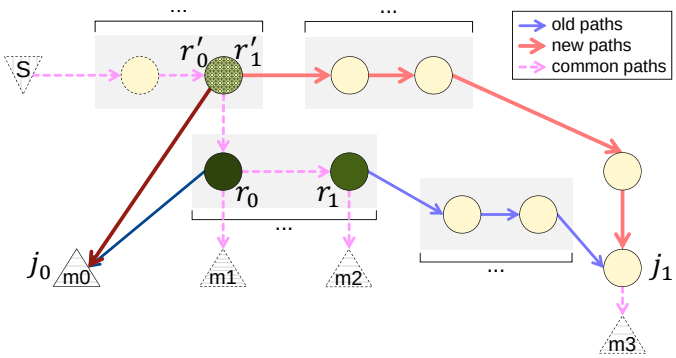


Fig. 4. Intermediate step of G^Δ-construction, after exemplary G, G' have been merged. Dashed edges and vertices are removed in a subsequent step.

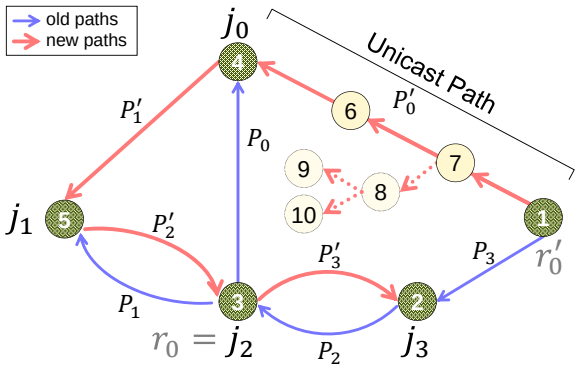


Fig. 5. Traversal of an exemplary delta graph, leading to a correct ordering among the updates. The dotted subtree represents joining group members. Replicator pairs (gray labels) other than (r_0, r'_0) are omitted for brevity.

E. Maintaining Loop-freeness

A special update case that has shown to occur very frequently in our evaluation scenarios arises, when old and new paths are interleaved, leading to *swap paths*, i.e., edges both in G and G' but with opposite direction, as illustrated in Fig. 5. Consider the following naive but drop-freeness-maintaining update: after processing of $j_1 = 5$ (install $4 \xrightarrow{P'_1} 5$, remove $3 \xrightarrow{P_1} 5$), a transient loop $3 \to 4 \to 5 \to 3$ is introduced during processing of $j_2 = 3$, while $5 \xrightarrow{P'_2} 3$ has been installed and $3 \xrightarrow{P_2} 4$ has not yet been removed. Note that if duplicate-freeness is to be maintained, i.e., P_i is removed before P'_i is installed, naturally, loops and duplicates do not appear. As the occurrence of loops is dependent on the update strategy, we define a third correctness property, *loop-freeness*, which is relevant for competing replicator move updates when drop-freeness is to be guaranteed. Avoiding cycles can however be achieved through a suitable update ordering. We derive a general update ordering, maintaining also loop-freeness in Section VI.

F. Effective Update Order

While we have focused on the formal structural analysis of multicast network updates so far, in the following we will incorporate further aspects of time to complement the prior analysis. In this section will describe the situation for packets traversing the network while an update is being applied. We will show that propagation delay of group messages may lead to an inversion of inconsistency effects and thus has to be included in the approach in order to be able to guarantee an invariant as selected by the update strategy. On the other hand it can be used to mitigate inconsistency effects.

Analysis: We differentiate between two types of events: 1) Events due to the packet forwarding process in the data plane: message receipt (packet ingress) and processing (forwarding), which possibly leads to multiple message sending. We consider processing delays negligible compared to link propagation delay and thus only consider the latter in this analysis. Therefore we subsume all named sub-events in a single *switch event*. 2) Events due to update messages from the controller: update message receipt and processing (execution). We assume that the network manager is aware of the control channel latencies and handles timing of control messages accordingly. The discussion over this assumption is taken up again at the end of this subsection. We depict *update event*s to mark the end of the update execution, i.e., when the update has become effective on the data plane.

For ease of demonstration, we simplify the example from Fig. 2 by omitting nodes which are solely relay nodes in both G and G', i.e., switches 3 to 5, as shown in Fig. 6.

We start with the upstream replicator move (cf. Fig. 6 with $r = s$ and $r' = 2$), where r is a successor of r' and thus dependent on r', as in our running example. We assume the drop prevention update strategy, at the cost of duplicates, and thus an update order of $u^+_{r'} \lhd u^-_r$. Fig. 7(a) shows a space-time diagram, depicting packet traversal with varying propagation delays and in intermediate states of an upstream move. A time axis for each node is given, where arrows depict events. A packet p is entering the network at the source node s and is forwarded on ingress by each switch i in a switch event $switch(p, i)$ according to the rules installed at i at the time of the switch event. The traversal of each packet is captured by a respective trace, reflecting the history of update events as seen by the traversing packet. As shown in Fig. 7(a), packets always reach m_1 correctly, i.e., exactly once, such that we do not include them in traces (dotted arrows).

Packet and trace names are binary coded according to the state that was present at network ingress or switch events, respectively. For instance, packet pk_{11} (rightmost) enters the network after all updates have already been executed and become effective (irrespective of their order): $\cdots \lhd switch(pk_{11}, r')$. Packet pk_{11} results in a correct trace tr_{11} (rightmost), since all switch events happen after the update of the respective switches. A packet pk_{00} is entering the network before u^-_r and $u^+_{r'}$ and is thus switched at the ingress switch $s = r'$ before $u^+_{r'}$: $switch(pk_{00}, r') \lhd u^+_{r'}$. Depending on the propagation delay of $r' \longrightarrow r$, denoted as $T_{r',r}$, packet pk_{00} might reach r before its update or in an intermediate state after its update. In the following we assume two cases of a small and large $T_{r',r}$ respectively. The first case, $switch(pk_{00}, r') \lhd switch(pk_{00}, r) \lhd \cdots$, results in a correct trace tr_{00} (leftmost trace). In the second case, $switch(pk_{00}, r') \lhd u^-_r \lhd switch(pk_{00}, r)$, however, r' did not yet replicate, whereas r has, due to its update, already stopped replicating, resulting in a drop at m_2 in the trace tr_{01}. Due to propagation delay, the effective update order, i.e., the update order as seen by the traversing packet, is inverted from $u^+_{r'} \lhd u^-_r$ to $u^-_r \lhd u^+_{r'}$.

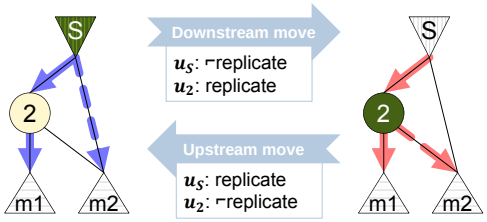


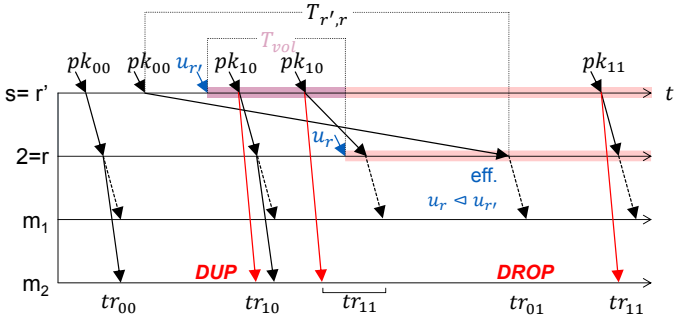
Fig. 6. Simplified example for the analysis of intermediate states, showing downstream and upstream replicator moves and corresponding updates.

As shown in the motivating example, a packet ingress in an intra-update state, i.e., within the volatile phase, pk_{10} with $u^+_{r'} \lhd switch(pk_{10}, r') \lhd switch(pk_{10}, r) \lhd \cdots$, results in a duplicate (tr_{10}). However, analogously, through propagation delay, pk_{10} might see a different update order, $u^+_{r'} \lhd switch(pk_{10}, r') \lhd u^-_r \lhd switch(pk_{10}, r)$, which leads to a correct trace tr_{11}, although the packet ingress has happened at an intra-update state.

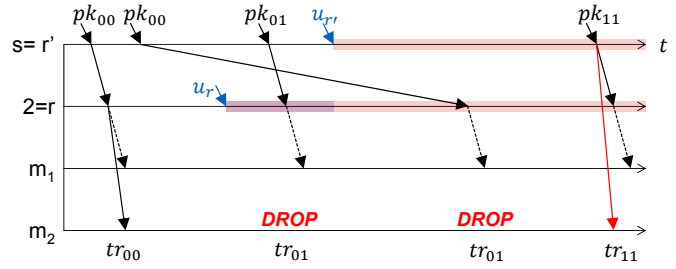
For upstream replicator moves with an update order of $u^-_r \lhd u^+_{r'}$ (cf. Fig. 7(b)), inversion through propagation delay cannot happen, since the dependent replicator r is updated before its predecessor r' and thus, irrespective of $T_{r',r}$, tr_{10} cannot occur. Thus, pk_{00} either leads to tr_{00} (correct) or tr_{01} (drop), where pk_{01} necessarily leads to tr_{01}.

In the case of downstream moves, where r' is a successor of and thus dependent on r, the situation is inverted. Downstream moves with $u^-_r \lhd u^+_{r'}$, i.e., the update strategy to prevent duplicates at the cost of drops, pk_{10} may lead to drops, as can be verified in Fig. 7(c). Through propagation delay, this effect may be inverted, such that duplicates instead of drops occur. Analogously, downstream moves with $u^+_{r'} \lhd u^-_r$, may result in duplicates in case of intermediate states, also due to propagation delay.

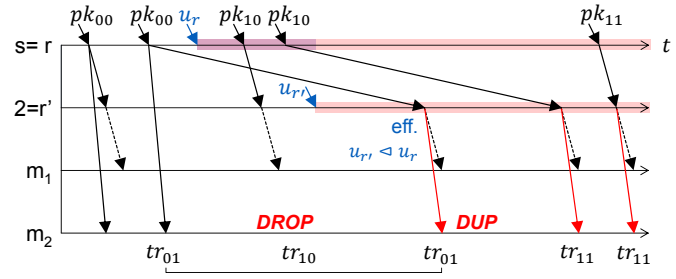
We thus conclude: in general, two conditions for perceived update reordering through propagation delay between (r, r') can be stated: dependency among (r, r') and the upstream node being updated before the downstream node. Due to the independence of replicator swaps, they are not prone to reordering. Replicator moves present dependency but only two out of four possible cases, those fulfilling the second condition, are prone to reordering: upstream moves with $u^+_{r'} \lhd u^-_r$, possibly leading to drops through propagation delay, instead of duplicates, and downstream moves with $u^-_r \lhd u^+_{r'}$, possibly leading to duplicates through propagation delay, instead of drops.



(a) Upstream replicator move with $u^+_{r'} \lhd u^-_r$.



(b) Upstream replicator move with $u^-_r \lhd u^+_{r'}$.



(c) Downstream replicator move with $u^-_r \lhd u^+_{r'}$.

Fig. 7. Intermediate states of replicator moves as perceived by traversing packets, considering propagation delay, which may invert the effects.

Effective update reordering due to propagation delay obviously counters the update strategy. However, first, typically the network manager would select one static update strategy for the whole multicast group traffic. Thus, there would only be one case left where drop-freeness cannot be fully guaranteed. Second, typical flow modification processing delay of SDN hardware switches [24], [25] are at least one order of magnitude larger than typical one-hop latencies of LAN or WAN links. Anyway, since we enable the network manager to be aware of this effect and its conditions, it is able to estimate the effect's extent and evaluate its criticality. The network manager has global knowledge of the topology and the changes to be applied to the multicast network, including the type of replicator change. This allows for a static analysis of the effect's extent (cf. Section VII, Stage I). Along with empirical measurement of data rates, the network manager can even estimate the expected number of drops (cf. Section VII, Stage II). Based on this evaluation the network manager may either simply choose to tolerate the reordering or apply a method to eliminate effective update reordering, which we present in the following.

Mitigation approach: In the analysis, we have identified a crucial measure for the reordering: the propagation delay between the replicators, $T_{r',r}$, which, in a real-world network, would typically include multiple hops and might thus accumulate to the order of tens to hundreds of milliseconds for WANs. The network manager's global knowledge also allows for a measurement of path latencies and thus the determination of $T_{r',r}$. To eliminate effective update reordering, for cases where these would occur, the update of the downstream replicator is artificially delayed by $max(T_{r',r}) + t_s$, where t_s denotes a safety margin which may possibly be added to handle outliers of $T_{r',r}$. For instance, in the example of Fig. 7(a), this guarantees that the last packet pk_{00} with ingress before $u^+_{r'}$ does not reach a yet updated r, such that drops are prevented. On the downside, however, this approach increases the volatile phase T_{vol} such that it constitutes a trade-off between potentially increasing the extent of tolerable effects through decreasing the extent of unwanted effects. Furthermore it is prone to jitter, i.e., variation of link delay. While the jitter in LANs is typically small enough to be safely ignored here, in WANs it might have to be considered in the determination of $T_{r',r}$. With small jitter, however, the extent of tolerable effects is small, since packets entering the network within the volatile phase (pk_{10}), would, through the artificial delay and small propagation jitter, most probably reach an yet updated downstream replicator, leading to a drop and duplicate free trace (tr_{11}). An optimization of this delay-based method, is to minimize T_{vol} through the incorporation of the network manager's knowledge about the update rates of the involved SDN switches, similarly to [15]. This would allow to create exactly timed update schedules, which could be precisely executed by the network manager, utilizing the timed network updates approach [26]. However, this approach relies on precisely synchronized switch clocks, which is reasonable assumption for LANs, whereas clock synchronization in WANs cannot safely be assumed. Since we enable awareness of the network manager also for the stated trade-off and dependencies, it is able to reason and decide on a concrete method, including parameters and optimizations. In Section VI-D, we present another update approach which guarantees the maintenance of arbitrary invariants by combining stateless updates with optimized state-based updates to minimize the rule space consumption.

VI. Flexible Approach for Multicast Tree Updates

Next, we present a flexible update approach for multicast distribution trees that feeds back the prevailing update situation to the network manager, which dynamically decides on an update mechanism to be used. In Sections VI-A to VI-B we describe a stateless update mechanism which allows for the selection of one primary invariant to be guaranteed (*update strategy*), where optional duplicate filtering (Section VI-C) may be applied. In Section VI-D we present a hybrid, i.e., both, stateless approach and state-based approach, which maintains arbitrary invariants, while rule space consumption is minimized. An overview of the proposed update procedure is given in Fig. 8, relevant notations for this section are listed in Table I.

TABLE I. Table of relevant notations for Section VI.

sw	switch (node)	$s_i^<$	split node i	l	link
s	source node	$G^{(')}$	distribution tree	pk	packet
j_i	join node i	G^Δ	delta graph	tr	packet trace

$r_i^{(')}$; (r_i, r_i')	replicator node $\in G^{(')}$, associated with j_i; r. pair
$P^{(')}$, $sw_1 \xrightarrow{P^{(')}} sw_2$	path in $G^{(')}$, from sw_1 to sw_2
L^- / L^+	paths to be removed/installed in one update step
u_{sw}^- / u_{sw}^+	rule update (removal/installation) at sw
T_{sw_1,sw_2}	propagation delay between sw_1, sw_2

The update procedure comprises five steps (cf. Figure 8):

S1) We capture differences in the distribution tree, which was recalculated due to events in the network, such as topology changes, e.g., link and node failures or utilization changes, as well as changes in the group membership due to joining or leaving nodes (churn). The ***change analyzer*** constructs the delta graph (G^Δ), representing all changes.

S2) The *change analyzer* returns all join nodes, their update type, i.e., (up-/downstream) replicator move, replicator swap, non-competing, as well as the number and identity of affected nodes to the network manager's **reconfiguration process**. There, an appropriate update mechanism is selected along with its parameters, based on the prevailing update situation.

S3) Based on the selected mechanism and parameters, the necessary data plane updates are calculated by the ***path update algorithm*** that decomposes the tree changes into incremental edge updates. To this end, it traverses G^Δ in order to identify and classify all changes into branch updates, replicator updates, and added and removed edges due to member dynamics. Then it defines update sequences by building a partial ordering over sets of edge updates, associated with an update type. The order depends on the given update strategy.

S4) These sets of edge updates are translated into SDN rule updates by the ***rule update generator***.

S5) Lastly, the updates are applied to the data plane in guaranteed order by the ***update executor***, which executes all updates based on the update mechanism, selected by the network manager.

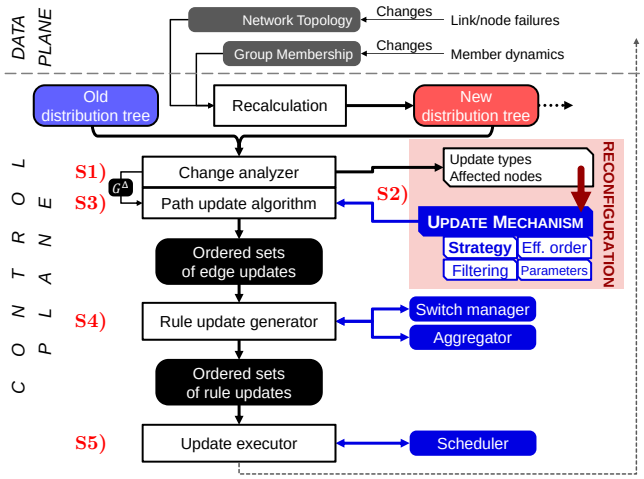


Fig. 8. Overview of the proposed multicast update procedure, where the specification of the update mechanism to be used is determined in the network manager's reconfiguration process.

A. Processing Graph Changes

The change analyzer creates G^Δ (S1), which contains all changed edges of a pair of old and new distribution tree (G, G'), reflecting their transition. Thus, every edge l is associated with either G or G'. An edge l is in G^Δ iff $l^- \in G \wedge l^- \notin G'$ or $l^+ \in G' \wedge l^+ \notin G$. Within the transition, all $l^- \in L^-$ are to be removed and all $l^+ \in L^+$ are to be installed.

3-phase G^Δ-decomposition: Through constructive traversal, the path update algorithm decomposes G^Δ into path segments (S3), i.e., ordered sets of edges, and defines an order of these steps. One step contains maximum-length path segments, while still maintaining invariants, e.g., unicast paths of arbitrary length as depicted in Fig. 5 (running example) can be combined in one update. We refer to paths in G consisting of l^- as old paths and vice versa for new paths. An update step is defined as a pair (L^-, L^+). On the one hand, the execution order within one update pair is determined by the update strategy: in general, if drops are to be prevented, edges are installed before edges are removed and vice versa, when preventing duplicates. On the other hand, the order among update pairs is crucial: intuitively, the general "remove-before-add" procedure implies careful removal, such that only edges that are independent, i.e., not needed by downstream nodes, are removed. This is, e.g., to verify, that an old path $sw_1 \xrightarrow{P} sw_2$ is removed only, when it either has been replaced by a new path $sw_1 \xrightarrow{P'} sw_2$ or $sw_2 \notin G'$. The first case would be caused by a replicator move, whereas the second case would be caused by a leaving member or a relay node in G becoming a non-tree node. To ensure this behavior, i.e., to determine a proper inter update pair order, we traverse G^Δ, starting at s.

Phase 1 - Determining join node sequence: In the first traversal step, we employ a depth-first search (DFS) to obtain an ordered list of all join nodes $j_i \in G^\Delta$ (indices depict sequence in Fig. 5), to be processed in order later. Note that s is not necessarily an element of G^Δ and furthermore, G^Δ might be a forest. We thus traverse G' starting from the closest (most upstream) node of s for each possibly isolated tree of G^Δ which is in G' (node 1 (rightmost) in Fig. 5).

Phase 2 - Join node processing: Then, join nodes are processed in order, where for each j_i, the number of affected nodes is assessed, as is a pair (L_i^-, L_i^+). The pairwise update step is determined by a backward DFS, traversing G^Δ backwards (upstreams), starting from j_i. Edges from G are appended to L_i^-, analogously edges from G' are appended to L_i^+. The traversal stops when another join node or the end of the path is reached. In the example of Fig. 5 (ignoring the dotted subtree), processing of j_0 would yield $(L_0^- = P_0,\ L_0^+ = P_0')$ with $P_0 = [3 \to 4]$ and $P_0' = [6 \to 4,\ 7 \to 6,\ 1 \to 7]$. As described, the ends of the respective paths define the replicator pair (r_i, r_i'), with $r_0 = 3$, $r_0' = 1$ in the example (gray label). If $r_i = r_i' = s_i^<$, the replicator move is classified as non-competing. Finally, L_i^- is reversed, such that the most upstream edge is removed first, continuing in downstream direction. Individual update steps are stored in an ordered set, in order of the join node processing.

To consider member changes, within the backward DFS, a forward DFS is started when a node sw has $deg_{out}(sw) > 1$.

The subtree branching from 7 in the example (dotted lines), is traversed by a forward DFS, started when the backward DFS reaches 7, and is inserted as a nested sublist within P'_0.

Phase 3 - Processing residuals: After processing join nodes, there might be residual, unhandled parts in G^{Δ}, which are subsequently handled in a second traversal step. These can be isolated trees which do not contain a join node. Root nodes with $deg_{in}(sw) = 0$ are determined and processed in order by employing a forward DFS to identify residual paths and create update steps accordingly.

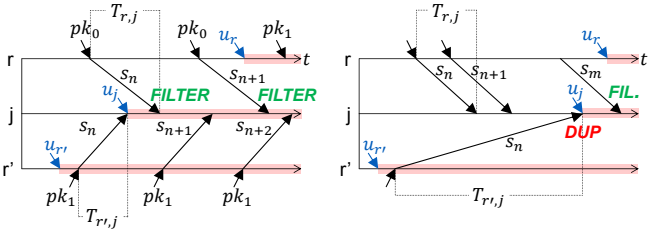
B. Translating Graph Changes to the Data Plane

After the path update algorithm has decomposed the (G, G') transition into a partially ordered set of edge update steps, the update steps have to be translated into SDN flow rule updates on a per-switch basis. Since distribution tree calculation is executed by the logically centralized network manager and thus both, topology data and switch management data is present, this step is straightforward: the *rule update generator* (S4) translates the directed edges l of the distribution tree into an associated switch-switchport pair (sw, p_{sw}), where sw is the source of l, in the order given by the update steps. The *update aggregator* component keeps track of rules installed in sw's flow table and determines an appropriate incremental rule update action[2] for each (sw, p_{sw}).

Lastly, the *update executor* executes all rule updates (S5) given by the rule update generator using an update mechanism as determined by the network manager (S2). For the stateless update mechanism, it coordinates the execution on multiple switches and guarantees total execution order. Thus the invariant, selected by the update strategy is guaranteed to be maintained throughout the whole update process. Update scheduling and execution has been subject to intensive research, e.g., [15]. We thus consider scheduling to be out of the scope of this article. Moreover, we identified several opportunities to leverage parallelization of update execution in our approach, which, due to space restrictions, we also did not include here.

C. In-network Duplicate Filtering

In this section, we describe an approach to mitigate duplicates which may occur in our stateless update mechanism with *drop-prevention* strategy, i.e., $u^+_{r'} \lhd u^-_r$. This approach is applicable to both replicator moves and replicator swaps. We tackle the symptoms of this update inconsistency by installing an additional rule at the join node j, associated with a replicator pair (r, r'), which aims to detect and filter duplicates after the actual replication (post-filtering). Consider a replicator swap, where the path from the split node $s^<$ to j over r ($s^< \xrightarrow{r} j$) and over r' ($s^< \xrightarrow{r'} j$) are completely disjoint, as in Fig. 5 with $j = j_1 = 5,\ s^< = 1,\ r' = 4,\ r = 3$. The base filter principle is illustrated in Fig. 9(a): when the first packet pk_1 reaches j over the (new) path P' we consider all consequent packets reaching j over the (old) path P to be duplicates. Thus, on ingress of pk_1 at j, we install a rule to drop all packets, reaching



(a) Total: $T_{r',j} \approx T_{r,j}$ (b) Partial: $T_{r',j} > T_{r,j}$

Fig. 9. In-network duplicate filtering: the ingress of a packet from r' triggers the installation of the drop rule at j, which identifies subsequent packets from r as duplicates and drops them accordingly. The filter effectiveness is dependent on $T_{r',j} - T_{r,j}$.

j over the ingress port that is associated with the link to j's predecessor in P. For sake of illustration only, we assume each packet to be identifiable over a sequentially increasing sequence number s_i. In the illustration, the ingress of pk_1 with s_n from r' triggers the installation of the drop rule at j (u_j), which identifies subsequent packets from r (pk_0 with s_n, s_{n+1}) as duplicates and drops them accordingly. In practice, flow rules typically cannot change the state or content of other rules, which is mandatory here. However, even in early releases of commonly used SDN switch software implementations, such as Open vSwitch[3], local switch logic has been enabled by implementing the *Nicira Extensions*[4]. They implement, *inter alia*, a MAC learning switch, where packet ingress triggers the installation of new forwarding rules, completely based on local logic, without controller involvement. Incorporating local switch logic thus, analogously, allows us to pre-install the described drop rules that are activated by local logic, i.e., packet ingress, by the switch, without time-consuming controller involvement. However, depending on differences in the accumulated propagation delay of $r \xrightarrow{P} j$ and $r' \xrightarrow{P'} j$, this approach can only guarantee partial duplicate filtering. As illustrated in Fig. 9(b), duplicates pass unfiltered, when $T_{r',j} > T_{r,j}$. Vice versa, when $T_{r',j} < T_{r,j}$, yet unreceived packets from r may be filtered, leading to effective drops. However, similar to the effective update order method, effective drops can be avoided by deferring u_j such that filtering becomes effective after the last non-duplicated pk_0 from r reached j.

In conclusion, the effectivity of the filtering approach is anti-proportional to the difference of $T_{r',j}$ and $T_{r,j}$, however removing any (extent of) unnecessary load from a network is beneficial. Awareness of this dependency allows the network manager to gauge the costs and benefit and dynamically decide whether to apply duplicate filtering (S2).

D. Hybrid Update Mechanism

While effective order elimination and duplicate filtering mechanisms are able to partially decrease the extent of invariant violation by tackling the symptoms of inevitable update inconsistency, in this section we present an optimized state-based approach, eliminating the reason of inconsistency. We present an optimized version of the prominent *two-phase update*

[2]When, e.g., a flow table entry in sw for group messages already exists, it is either deleted or its out-port action is changed to include or exclude p_{sw}.

[3]http://openvswitch.org/

[4]https://git.io/vgTKL

approach of Reitblatt *et al.* [6], combined with a stateless update ordering as an alternative update approach for multicast networks. In a two phase update, a version tag is appended to each incoming packet at the ingress switch, s in our case, while each network configuration, i.e., coherent set of rules installed in all switches, matches only one particular version. Reconfiguration is achieved by installing a new set of rules with an increased version match field on all switches, in parallel to the old rule set, before updating the ingress switch as to tag newly incoming packets with the new version number. A packet, traversing the network, thus is processed according to a coherent configuration (*per-packet consistency*). Increasing the tagged version number at the ingress switch effectively switches the configuration that packet is processed according to simultaneously, such that there exists no effective intra-update state and arbitrary invariants are maintained. Through the parallel installation of old and new configuration, the rule space consumption is doubled. In modern networking, multiple network functions are executed in parallel, leading to a high total number of rules. Furthermore, the two-phase approach has a technical dependency on the ability to encode state information in the packet. State of the art SDN switches can only process the packet header efficiently, such that the version information is stored in an unused header field, typically the VLAN tag. However, it cannot safely be assumed that a vacant header field is present at every packet, such that the approach's applicability is limited.

While the technical dependency remains, we optimize the original approach as described in the following. The conducted extensive analysis on the concrete problem of update consistency of multicast networks has identified the critical parts of reconfiguration, where drop- and duplicate-freeness breaks, as the replicator pair (r, r'). In order to maintain both—in fact, arbitrary invariants—we employ a two-phase update, but limited to the replicator pairs. While the calculation of the necessary updates stays unchanged, their execution order (S3) is changed: $P'_i \setminus r'_i$, denoted as P'^0_i, is installed, before a two-phase update, limited to (r_i, r'_i) is conducted. Note, that the installation order of P'^0_i is even irrelevant, since the associated path is not used until the two-phase update of (r_i, r'_i) has been executed. After (r_i, r'_i) has been updated, P' is effective, while P is ineffective due to the update of r, such that P^0_i can be safely removed in arbitrary order. The update order of $u_{P'^0_i} \lhd u_{(r,r')} \lhd u_{P^0_i}$ has to be guaranteed though.

Depending on the severity of the update inconsistency effects, the network manager might decide (S2) to prefer rule space capacity and use this approach only if effective update order and duplicate filtering mechanisms would yield bad results, rather than blindly apply it whenever the technical condition is met.

VII. Evaluation

Our evaluation consists of two stages: first, we analytically evaluate the impact, i.e., the number of replicator moves and affected nodes, for a varying degree of network dynamics (random member and link changes) for a small and large WAN topology. Second, we apply our approach to these graph changes to transform them into rule changes and corresponding network updates, which we execute on an SDN network emulated with Mininet. We empirically measure the occurrence of drops and duplicates directly on the data plane, while varying the update strategy and the degree of the random reordering of the given update sequences. All stages were executed on a dual-socket Intel Xeon E5-2687Wv3 (10 physical cores at 3.10GHz per socket) with 128GB RAM, running CentOS 7.

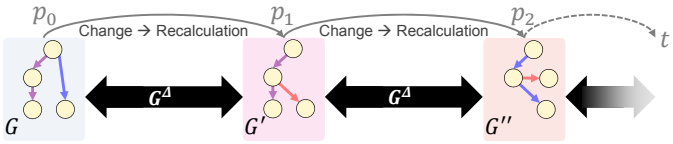


Fig. 10. Scenario Generator, stepwisely recalculating the distribution tree, triggered by simulated link and member changes.

Stage I: Impact Analysis of Network Dynamics

Methodology: For Stage I, we implemented a *scenario generator* which first creates a random initial state (source node and members selection) for a given number of members m and a given topology, along with an according distribution tree. The scenario generator then simulates random data-plane events (member and link changes), which trigger a recalculation, leading to a new tree. This change-recalculate process is repeated in a stepwise execution model, where one step is denoted by a time *period* p_i, as illustrated in Fig. 10. All trees of one scenario generator run are called a *scenario*. Distribution trees are calculated using a C++-implementation of a Steiner Tree approximation algorithm[5].

For membership changes, we employ a probabilistic model, where p defines the probability of a node changing its membership state. Link over-utilization is simulated by a temporary significant increase of the respective edge weight, where c denotes the number of over-utilized links, per period.

In each period, the change analyzer creates a G^Δ and assesses three metrics: the number of replicator moves, the total number of affected nodes, and the distribution of replicator update types within that period. Note that non-competing replicator moves are not considered here. Furthermore, recall that (affected) nodes do not consider potentially connected end hosts, which would be affected as well. Also, the path update algorithm calculates all edge update sequences, which are dumped and used in the Stage II.

In Stage I, we use two real WAN topologies of different scales (number of vertices *v*, number of edges *e*): the European National Research & Education Networks (NREN)[6] with $v = 440,\ e = 599$, as well as the IP-backbone of the German Research & Education Network (DFN X-WiN) with $v = 50,\ e = 76$. Link latencies are used as initial edge weights. For NREN, latencies missing in the obtained data set where interpolated (with an added Gaussian-distributed random error),

[5] Enabled by the global knowledge, optimality is shown to be reached, constructing a minimum Steiner tree [27]. The *Steiner tree problem* asks for a tree, spanning a set of terminals $(S \cup M)$ at minimal cost $(\sum_{l \in L_{MC}} w(l))$, possibly including additional non-terminal elements, called Steiner nodes.

[6] obtained from http://www.topology-zoo.org/eu_nren.html

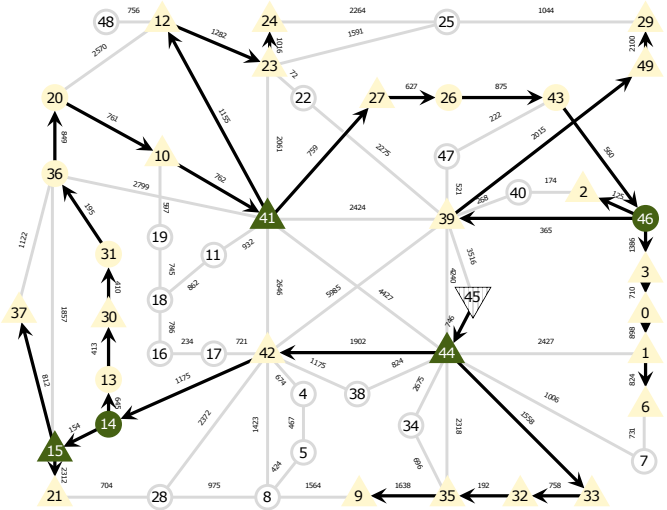


Fig. 11. DFN X-WiN base topology (light edges, light and unfilled vertices; edge labels: avg. link delay in µs) with superimposed exemplary multicast distribution tree (dark edges; downward triangle: source (*node 45*); upward triangle: member; dark vertex: replicator; light vertex: relay).

whereas latencies of the X-WiN were obtained from the web service of its active probing system[7]. The number of initial members m is given as a fixed ratio of $m/v = 0.4$. An exemplary Steiner tree based on the DFN X-WiN underlay topology is shown in Fig. 11. The degree of dynamics is gradually increased in 6 levels, ranging in $p \in [0.005, \ldots, 0.1]$, $c \in [0, \ldots, 10]$ (p: member change probability; c: number of simulated link over-utilizations). To reduce structural dependency on the initial state, each level is evaluated by a common set of 5 scenarios. Each scenario consists of 200 periods.

Results: Fig. 12 shows period- and scenario-aggregated mean ratios of replicator moves. Error bars in the figure and stated variances henceforth refer to the standard deviation of the inter-scenario aggregation, i.e., among scenarios. As expected, a strong correlation between degree of dynamics and extent of effects can be inferred. The number of replicator moves directly reflects the degree of dynamics. Constantly low variance indicates low dependency on both topology and scenario.

The number of affected nodes naturally reflects the number of replicator nodes. Even small/few changes in the network already cause a significant extent of effects, mostly stemming from simulated link over-utilization. For instance, for $(p = 0.005, c = 1)$ the number of affected nodes is 24.21 ± 13.72 (NREN) and 13.06 ± 2.72 (X-WiN). Respective statistical ratios[8] are 5.5% of all NREN nodes and 26.1% of all X-WiN nodes.

However, a high variance and thus a high dependency on the underlying topology and its initial conditions can be stated. Naturally, the specific position of the respective replicator pairs in the graph along with initial conditions and scenario parameters, such as average distance to the source or graph diameter, strongly influence the number of affected nodes.

The distribution of replicator update types, excluding non-competing replicator updates, is shown in Fig. 13. The *churn*

[7]http://pallando.rrze.uni-erlangen.de:8090/services/MA/HADES/DFN

[8]Note that this ratio does not imply the coverage of actual nodes, since a single node can be affected by multiple replicator moves within a period.

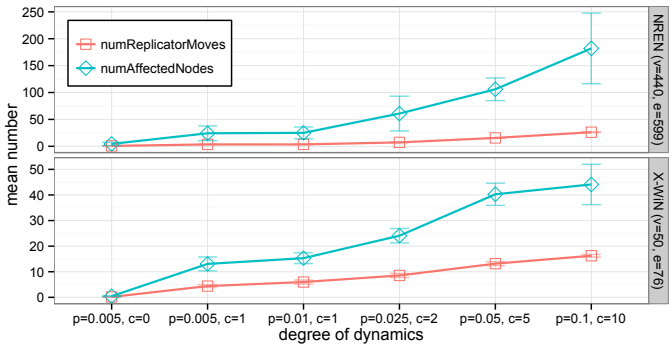


Fig. 12. Analytic evaluation (Stage I): period and scenario aggregated mean values of number of replicator moves and affected nodes for varying degree of dynamics. Even small dynamics cause a significant extent of network changes.

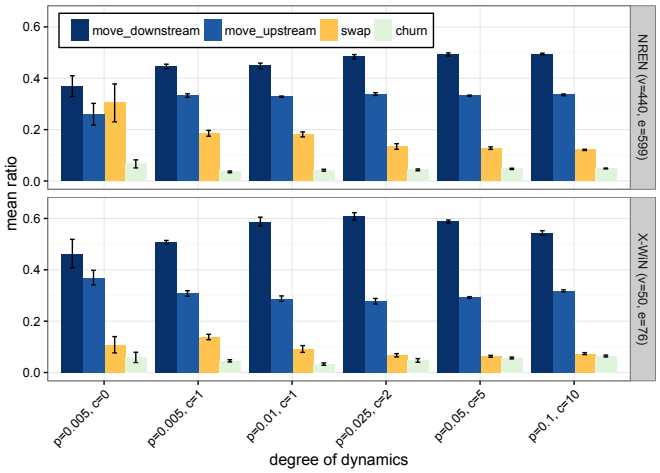


Fig. 13. Distribution of replicator update types.

type depicts moves, where either $r \notin G'$ or $r' \notin G$. In both, NREN and X-WiN, $\approx 80\%$ of the moves are replicator moves, where, on average, downstream moves are twice as likely as upstream moves for X-WiN with moderate and high dynamics and about 25% more likely for NREN, irrespective of the degree of dynamics. For *drop-prevention*, the effective update reordering, happening at upstream moves, is thus shown to be of potential practical relevance, if HW switches' flow update rates are unconsidered, as discussed. Reorder-free replicator swaps are more significant for NREN with $\approx 17\%$ on average than on X-WiN with $\approx 10\%$, both showing anti-proportionality wrt. the degree of dynamics. The churn type ratio is rather constant at $\approx 5\%$ for both base topologies.

We summarize the observations in stating that even moderate degree of dynamics lead to significant extent of tree changes and thus to a significant amount of nodes, affected by update inconsistency.

Stage II: Empirical Validation

Methodology: In Stage II, we translate the edge updates from the scenarios of Stage I into corresponding FlowMod messages (OF 1.3) and execute them in an emulated network with X-WiN topology and given link characteristics (latency). We use Mininet version 2.2.1 with Linux Traffic Control (TC) enabled links, which allow for link latency emulation, in

combination with Open vSwitch (OVS) version 2.3.2. The rule update generator and the update executor are implemented as a module for the Python-based Ryu SDN controller. Since the update executor has to guarantee total update execution order on a switch basis, OF *Barrier Messages* are used as flow-modification feedback mechanism, i.e., updates, sent to a switch for execution, block sending updates to other switches, until the execution of all former updates are acknowledged.

Recent SDN hardware switches have a limited flow modification capacity of around 40 flows per second (update rate) [24], [25], which we simulate through an artificial delay in the update executor. This assumed lower bound is probably exceeded in the evolution of upcoming SDN hardware switches. Thus, we present an extended evaluation with update rates up to 1000 flows per second at the end of this section. However, other work, including [15], suggest a high volatility of the flow update rate and strong dependency on a number of factors, such as control-plane load, number of installed rules, rule priority and complexity (actions). The update executor processes all translated updates of a scenario period, before progressing to the next period. This results in a typical execution time of $1.1s \pm 0.3s$ per period.

We measure the occurrence of drops and duplicates within one period directly on the data plane, while updates are being executed. Therefore, the sender node is added as a Mininet host that constantly sends group messages, containing a sequence number as payload, at a rate of 250pps (packets per second), which is a realistic number, e.g., for media streaming, and a packet size of 50 Bytes. We capture the complete traffic on all Mininet network interfaces, i.e., switchports. The number of captured packets is denoted by n. Through evaluating the sequence numbers of captured packets on a switchport, we directly measure the number of duplicates du. To assess the number of dropped packets dr, we evaluate sequence number gaps between two consecutively captured packets on a switchport, respecting period-borders. We then aggregate duplicate and drop values to the switch level by summation and evaluation of possibly overlapping sequence number and their gaps, respectively.

Here, edge updates of the 5 scenarios with moderate degree of dynamics ($p = 0.005,\ c = 1$) from Stage I are used. The update strategy is varied between drop prevention (`ADD_F`) and duplicate prevention (`REM_F`), to validate the effectiveness of our approach. As a baseline, we provide a completely random update order. To show the implications of deviation from the pure strategies up to complete randomness, we employ a gradually increasing extent of random reordering: p_s denotes the probability of a message within an ordered list of messages to be chosen for reordering. The set of chosen messages are then randomly reordered, using Fisher–Yates Shuffling. While unchosen messages stay at their list position, chosen messages are replaced by shuffled messages. Note that only the ordered elements within an update pair (L_i^-, L_i^+) are shuffled, whereas the order among update pairs (strategy) is maintained.

Results: Fig. 14 shows mean packet drop factor (quotient of num. of dropped and num. of expected packets: $\lambda_{dr} = {}^{dr}/_{dr+n}$) and duplication factor (quotient of num. of duplicates and num. of captured packets: $\lambda_{du} = {}^{du}/_{n}$) of affected nodes for

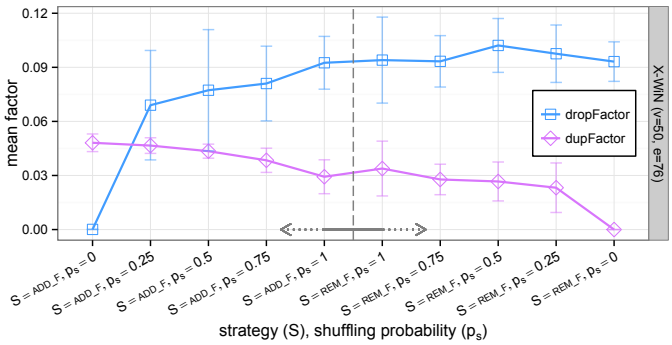


Fig. 14. Empirical data plane effect occurrence evaluation of affected nodes (Stage II): mean drop and duplication factors for varying strategy and degree of random update message reordering, validating the correctness of our approach.

varying degree of reordering, where the extremes (leftmost, rightmost) show pure strategies (no shuffling: $p_s = 0$), with a gradual 0.25 p-increase (partial shuffling) towards the center, which is fully shuffled ($p_s = 1$). As illustrated, our approach can be shown to be correct: the drop prevention strategy (left) successfully prevents drops $\lambda_{dr} = 0 \pm 0$, at the cost of duplicates $\lambda_{du} = 0.05 \pm 0.005$ and vice versa for duplicate prevention (right): $\lambda_{du} = 0 \pm 0,\ \lambda_{dr} = 0.09 \pm 0.01$.

Deviating from a pure strategy, e.g., drop prevention, is shown to result in introduced effects, actually to be prevented, e.g., drops, as expected. While the extent of the inverse effect, i.e., duplicates, decreases strong monotonically with increasing p_s in the case of drop prevention, curiously this is not the case, when deviating from pure duplicate prevention. One possible explanation is the small topology scale, typically resulting in short message lists, where the effectiveness of probabilistic shuffling is small, such that the extent of reordering is similar for a large range of p_s-gradations.

Another asymmetry is present in the average λ_{dr} being almost $2 * \lambda_{du}$. This asymmetry however naturally follows from the nature of removal and installation of unicast paths to a respective join node j: for the non-shuffled case of *add first*, the new path to j is first fully installed, before the old path is removed. Until the last edge in the installation phase has been installed, neither drops nor duplicates occur. In contrary, with *remove first*, the first removal of an edge immediately results in drops, lasting over the removal and installation phase, until the last edge of the new path has been installed. Thus, drops are much more likely than duplicates. Similarly, for shuffled cases, the probability to have the only unicast path to j broken by a reordered and thus premature remove-update is much higher than the probability to have a complete redundant unicast path installed, despite the mixing of installation and removal updates. For averagely larger path lengths, e.g., in larger topologies, this effect is expected to be of much higher extent.

In a last evaluation, we have increased the assumed flow update rate to $1000s^{-1}$—the order of magnitude we expect to see on upcoming SDN switch hardware. In order to reliably assess update inconsistency effects, we target a packet rate that is one order of magnitude higher than the update rate, i.e., 10000pps. On our evaluation machine, Mininet does not scale

further than approx. 3000pps for the X-WiN base topology, using default Linux virtual Ethernet (veth) pair links. Thus, we have to use OVS specific *OVS patch links*[9] to interconnect OVS bridge ports. Since these interfaces are not exposed to the OS, performing packet capture on them is not possible. We thus add a *Linux dummy interface* to each OVS bridge and mirror all packets from the other (regular) OVS ports to it. Capturing on those dummy interfaces thus allows for assessing the complete network traffic. The results shown above were confirmed for increased update and packet rates, however, the changed evaluation method introduces a packet reorder rate of about 1%. Furthermore, OVS patch links do not support link latency emulation, so that we could not consider timing aspects.

VIII. Conclusion

In this article, we have proposed a generic system architecture for network management, focusing on change management. We have proven that it is impossible in general to achieve drop-freeness and duplicate-freeness simultaneously just by ordering updates in a multicast network. We have presented a detailed formal analysis of this update problem. In order to alleviate this problem, we have proposed a flexible update approach, allowing for selecting a strategy that either prevents duplicate or drops. We have shown that update consistency is multifarious and comprises many degrees of freedom, spanning a large configuration space. In combination with the severity of impacts on the network performance, this has shown the relevance of update consistency in network management and argues for incorporating update consistency awareness in the network reconfiguration process.

Our evaluation has shown the relevance of the addressed problem even for small degree of network dynamics and has validated the correctness of our update order approach: drop-freeness can be achieved at the cost of as few as 5% introduced duplicates. Since duplicates, in contrast to drops, have been shown to be of less extent and certainly can be assumed less fatal for most applications, our approach is highly practical.

References

[1] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 183–197.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined Wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 3–14.

[3] J. Donovan, "Setting the Pace with Our Next-Gen Network," http://about.att.com/innovationblog/121514settingthepace, [Online; acc. 2016-02-11].

[4] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, Feb. 2013.

[5] R. Ahmed and R. Boutaba, "Design considerations for managing wide area software defined networks," *IEEE Communications Magazine*, vol. 52, no. 7, pp. 116–123, Jul. 2014.

[9] cf. manual page of `ovs-vswitchd.conf.db`

[6] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *Proceedings of the ACM SIGCOMM 2012*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012.

[7] R. Mahajan and R. Wattenhofer, "On Consistent Updates in Software Defined Networks," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII. NY, USA: ACM, 2013.

[8] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for Data Centers," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 49–60.

[9] M. Kuźniar, P. Perešíni, and D. Kostić, "What You Need to Know About SDN Flow Tables," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science, J. Mirkovic and Y. Liu, Eds. Springer International Publishing, Mar. 2015, no. 8995, pp. 347–359.

[10] A. Iyer, P. Kumar, and V. Mann, "Avalanche: Data center Multicast using software defined networking," in *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, Jan. 2014.

[11] A. Arefin, R. Rivas, R. Tabassum, and K. Nahrstedt, "OpenSession: SDN-based cross-layer multi-stream management protocol for 3d teleimmersion," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, Oct. 2013, pp. 1–10.

[12] T. Kohler, F. Dürr, and K. Rothermel, "Update consistency in software-defined networking based multicast networks," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, Nov. 2015, pp. 177–183.

[13] Y. Nakagawa, K. Hyoudou, and T. Shimizu, "A Management Method of IP Multicast in Overlay Networks Using Openflow," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 91–96.

[14] D. Li, J. Yu, J. Yu, and J. Wu, "Exploring efficient and scalable multicast routing in future data center networks," in *2011 Proceedings IEEE INFOCOM*, ser. INFOCOM '11, Apr. 2011, pp. 1368–1376.

[15] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic Scheduling of Network Updates," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 539–550.

[16] J. McClurg, N. Foster, and P. Cerny, "Efficient Synthesis of Network Updates," ser. PLDI '15, Portland, OR, USA, Jun. 2015.

[17] Y. Fu, J. Bi, Z. Chen, K. Gao, B. Zhang, G. Chen, and J. Wu, "A Hybrid Hierarchical Control Plane for Flow-Based Large-Scale Software-Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 117–131, Jun. 2015.

[18] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou, "Adaptive Resource Management and Control in Software Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 18–33, Mar. 2015.

[19] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 279–291.

[20] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *USENIX NSDI*, ser. NSDI'13, 2013, p. 2.

[21] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software Transactional Networking: Concurrent and Consistent Policy Composition," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA: ACM, 2013.

[22] N. van Adrichem, C. Doerr, and F. Kuipers, "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014.

[23] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[24] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for Openflow Switch Evaluation," in *Proceedings of the 13th International Conference on Passive and Active Measurement*, ser. PAM'12. Berlin, Heidelberg: Springer-Verlag, 2012.

[25] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity Switch Models for Software-defined Network Emulation," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013.

[26] T. Mizrahi, E. Saat, and Y. Moses, "Timed Consistent Network Updates," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 21:1–21:14.

[27] L.-H. Huang, H.-J. Hung, C.-C. Lin, and D.-N. Yang, "Scalable Steiner Tree for Multicast Communications in Software-Defined Networking," *arXiv:1404.3454 [cs]*, Apr. 2014.

Thomas Kohler received the M.Sc. degree in Computer Science from Augsburg University, Germany, in 2013. He is currently pursuing the Ph.D. degree at the Distributed Systems research group, University of Stuttgart, Germany. His research interests include consistency and determinism in Software-defined Networking as well as Whitebox networking hardware. In particular, his research focusses on update consistency, local switch logic and control plane distribution.

Frank Dürr is a senior researcher and lecturer at the Distributed Systems Department of the Institute of Parallel and Distributed Systems (IPVS) at University of Stuttgart, Germany. He received both his doctoral degree and diploma in computer science from University of Stuttgart. Frank Dürr is currently leading the mobile computing and the software-defined networking (SDN) & time-sensitive networking (TSN) groups of the Distributed Systems Department. He has given tutorials on SDN at several national and international conferences, and as a lecturer he has been giving lectures and practical courses on networked systems and SDN. Besides SDN and TSN, Frank Dürr's research interests include mobile and pervasive computing, location privacy, and cloud computing aspects overlapping with these topics like mobile cloud and edge computing, or datacenter networks.

Kurt Rothermel received his doctoral degree in Computer Science from University of Stuttgart in 1985. From 1986 to 1987 he was Post-Doctoral Fellow at IBM Almaden Research Center in San José, U.S.A., and then joined IBM's European Networking Center in Heidelberg. Since 1990 he is a Professor for Computer Science at the University of Stuttgart. From 2003 to 2011 he was head of the Collaborative Research Center Nexus (SFB 627), conducting research in the area of mobile context-aware systems. He is a Director of the Institute of Parallel and Distributed Systems. His current research interests are in the field of distributed systems, computer networks, and mobile systems.