On Optimal Topology Verification and Failure Localization for Software Defined Networks

Ulaş C. Kozat, *Senior Member, IEEE,* Guanfeng Liang, *Member, IEEE,* Koray Kökten, and János Tapolcai *Member, IEEE*

***Abstract*—We present a new set of solutions for topology verification and failure localization in Software Defined Networks (SDNs). Our solutions are targeted towards offloading the control plane as much as possible and bringing more resilience against congestion or partitioning in the control plane. The core idea is to define control flows for network diagnosis and utilize a fraction of the forwarding table rules on the switches to serve these control flows. For topology verification, we present provably optimal or order-optimal solutions in total number of static forwarding rules and control messages. For single link failure localization, we present a solution that requires at least $3|\mathbf{E}|$ but at most $6|\mathbf{E}|$ forwarding rules using at most $1+\log_2|\mathbf{E}|$ control messages, where $|\mathbf{E}|$ denotes the number of bidirectional links in the forwarding plane. We analyze the latency vs. rule and control message optimality trade-offs showing that sub-second failure localization is possible even in data center scale networks without significant additional overhead in the number of static rules and control messages. We further simulate the performance of failure localization in identifying multiple link failures.**

***Index Terms*—Software Defined Networks, OpenFlow, Network Diagnosis, Failure Localization.**

I. Introduction

SDNs promise to change the landscape of communication networks by extracting the control functions and applications distributed over the individual forwarding elements and placing them on top of logically centralized network controllers. In this view, the forwarding elements are simplified and their internal pipelines are exposed to the external controllers as programmable abstractions. For instance, if the forwarding elements are OpenFlow switches, then the internal pipeline is a series of flow tables, where the external controllers can dictate which packets are processed by which tables in what order by specifying match-action sequences based on the packet headers ultimately specifying the outgoing port for the packet in the last table it is processed. When a packet cannot be matched at an OpenFlow switch, a default action is applied such as *drop packet* or *send to controller*. It is controller's responsibility to collect the network state and install flow forwarding and processing rules.

For high network availability and performance, it is essential for a controller to be able to verify the forwarding plane topology and identify link failures at forwarding plane speeds. In pure SDN architectures such as the ones based on OpenFlow, controllers can be notified about various network events by the switches such as new flow events, per flow or per port packet counters, interface up/down events, master/slave configurations, new or expired forwarding table rules, etc. However, switches do not support any protocols for neighborhood discovery, topology discovery, bidirectional forwarding detection (BFD) [1], or other solutions that require control signaling between the switches. Thus, for network diagnosis in the form of verifying topological connectivity, detecting network partitions, locating link failures, etc., controllers must define specific control flows and install the corresponding forwarding rules onto the switches. To this end, many modern controller frameworks take advantage of the existing control plane protocols such as Link Layer Discovery Protocol (LLDP) [2]. By sending LLDP packets to network switches and installing static forwarding rules to handle them properly, network controllers discover and diagnose the topology of the forwarding plane.

However, LLDP based approach suffers from excessive number of messages that must be handled by the software agents (i.e., slow-path handling) on network switches that can be substantially slower (e.g., by $1000\times$) than the forwarding speed supported by hardware rules (i.e., fast-path handling). Frequently performing topology discovery would result in control plane congestion. Thus, continuous monitoring via LLDP is not a feasible approach in a relatively well-connected network. Furthermore, LLDP based topology discovery and diagnosis fail to work under control plane failures/partitions that disrupt the communication among the controllers as well as the communication between the individual switches and controllers.

Our paper focuses on finding efficient solutions for topology verification and failure localization in OpenFlow networks that are fast and also tolerant against control plane partitions. To achieve higher speeds and partition tolerance in the control plane, one must offload most of the control plane processing onto the forwarding plane in the form of hardware-based forwarding rules installed on the forwarding elements. These static forwarding rules combined with dynamic forwarding rules allow each controller to inject control packets into the forwarding plane such that the packets traverse all or a subset of network links before coming back to the controller. Inspecting which of its control packets are looped back from the forwarding plane and which are not, the controller can identify the forwarding plane problems. Since the control flows must be statically installed and the total number of hardware

U.C. Kozat is with Argela USA, Sunnyvale, CA. G. Liang is with LinkedIn, Mountain View, CA.

K. Kökten is with NETAŞ, Istanbul.

János Tapolcai is with MTA-BME Future Internet Research Group, Budapest University of Technology, Hungary.

U.C. Kozat is the contact author. E-mail: kozat@alumni.bilkent.edu.tr.

The conference version of the paper is presented in IEEE Infocom'14.

forwarding rules supported by a switch can be quite limited, realizing network diagnosis using minimum number of static forwarding rules becomes the most critical objective.

Our main contributions can be listed as follows:

(i) We show that topology verification can be done optimally in terms of using minimum number of static forwarding rules and control messages. With a counter example, we also show that minimizing the total number of static forwarding rules is not equivalent to finding the shortest walk that visit every link in the forwarding plane at least once.

(ii) For topology verification problem, we present a solution based on shortest possible walk that visits every link in the forwarding plane at least once. The solution iterates over an initially computed shortest walk to reduce the number of forwarding rules while preserving the walk length using a greedy heuristic. We can show that the total number of forwarding rules required by our solution is upper bounded by $2|\mathbf{E}|$. It is also delay optimal when only one control message is used. For special topology cases or failure scenarios (referred to as asymmetric failures later in the paper), the proposed solution also becomes provably optimal in number of static forwarding rules. Simulations over real world topologies indicate that the overhead remains within 14% of the optimum and matches the optimum for 60% of the topologies.

(iii) We present order optimal solutions (in number of forwarding rules and control messages) to locate an arbitrary single link failure. With a counter example, we show that locating multiple arbitrary link failures over arbitrary network topologies is not solvable.

(iv) We present how one can trade off the number of control messages and/or static forwarding rules to speed up failure localization in data-center scale networks. Even when the number of links is in the order of 100K, our analysis indicate that by slightly using more bandwidth and rules, failure localization can be done at sub-second latencies.

(v) As our solution can probabilistically locate multiple failed links, we evaluate the performance of multiple link failure detection over real world topologies. Our results indicate that even when only one switch is accessible by a controller, that controller can detect more than two failures on average over the majority of the topologies.

The paper is organized as follows. In Section II, we summarize the related works. In Section III, we describe the system model. In Section IV, we focus on verification of topology connectivity and establish optimality results as well as performance bounds. In Section V, we turn our attention to locating a single but arbitrary link failure. We provide an order optimal solution and present performance bounds on important metrics. In Section VI, we extend the results to multiple link failures. In Section VII, we show performance results using publicly available, real world topologies. In Section VIII, we discuss several variations from the original problem set up. Finally, we conclude in Section IX.

II. Related Work

There are several works both for classical networks and SDNs that are closely related to ours. In all-optical networks, fault diagnosis (or failure detection) is done by using monitoring trails (m-trails) [3]–[8]. An m-trail is a pre-configured optical path. Supervisory optical signals are launched at the starting node of an m-trail and a monitor is attached to the ending node. When the monitor fails to receive the supervisory signal, it detects that some link(s) along the trail has failed. The objective is then to design a set of m-trails with minimum cost such that all link failures up to a certain level can be uniquely identified at some pre-determined nodes or by all nodes in the forwarding plane. Once the failure is localized, it can be reported to an external controller or even a fast path failover/rerouting can be done as a function of the optical transport without requiring any action by an external controller. The fundamental difference in OpenFlow-based SDN architectures is that switches do not have any control signaling between them. Therefore, controllers should serve as the start and end points of the monitoring signals.

The more salient differences from the optical networks can be listed as: (i) In all-optical networks, the main resources are wavelengths (i.e., bandwidth) allocated for monitoring. In our set up, the resources are the control packets and the forwarding rules. (ii) In our problem, a given static forwarding rule can be (and are in fact) reused by different walks. This creates an interesting situation where shortest walks do not always lead to the minimum number of forwarding rules. (iii) In optical networks, the monitoring is performed in a non-adaptive fashion, e.g., the monitoring trails are fixed and the control signals should be continuously sent over all the trails. In contrast, SDNs enable adaptive group testing, where each group test can depend on the results of the previous tests. Moreover, signals are initiated by the network controllers as event based signals (e.g., if a controller observes poor state synchronization or partition, it injects the control packets). These differences also holds for works on graph-constrained group testing [9].

In various SDN solutions, LLDP-based topology discovery is used to periodically monitor the changes in the network topology [2]. LLDP has specific field values: EtherType field set to 0x88cc, destination MAC address is set to a designated multicast address[1]. In OpenFlow networks, network controllers send LLDP packets encapsulating them in *packet-out* messages over the transport layer control sessions maintained with individual switches to flood all the links in the network. Controller has two main options to perform flooding: (1) Send one *packet-out* message for each link of a given switch, which requires sending $2|E|$ *packet-out* messages for the whole network. (2) Send one *packet-out* message for each switch instructing the switch to multicast the encapsulated LLDP packet on all ports, which requires sending $|S|$ *packet-out* messages for the whole network. When the next hop switch receives LLDP packets from its neighboring switches, it must either have a specific rule for LLDP packets so that it is forwarded to the controller or there should not be any forwarding rules matched to LLDP packets and the default action for unmatched flows must then be *send to controller*. In both cases, the next hop switch forwards the LLDP packet

[1] 01:80:c2:00:00:0e, 01:80:c2:00:00:03, or 01:80:c2:00:00:00

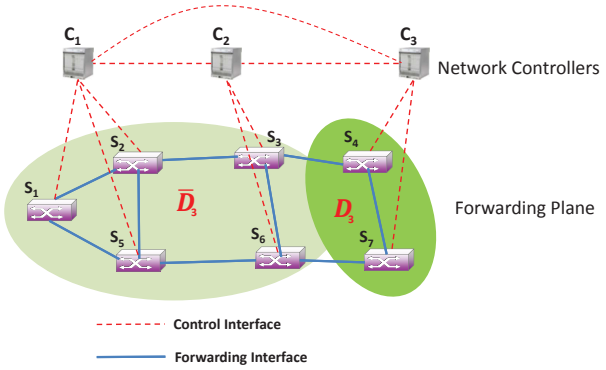


Fig. 1: System Model

as *packet-in* message over the control session established with the controller. Hence, $2|E|$ *packet-in* messages are generated in total. Note that the default rule *send to controller* is not a preferred option for carrier-grade networks as it can be used to launch denial of service attacks. In general, the default rule for handling unmatched flows is to drop the packets of these flows. Therefore, controllers must put 1 static rule per switch to handle LLDP packets incoming from other switches by matching on EtherType and/or destination MAC address. Overall complexity of LLDP based approach then requires $|S|$ static rules and at least $|S| + 2|E|$ control messages are sent to/from controllers in the form of *packet-in* and *packet-out* messages. Processing speed of these control messages can be as much as three orders of magnitude slower than the forwarding speed between the switches using hardware rules. As a result LLDP based topology discovery cannot be done too frequently, or otherwise it congests the control plane. Another major disadvantage of this approach is that it is highly intolerant to partitions in the control plane, e.g., in its simplest form if a controller looses its control session with any switch or another controller, the process fails. Our solutions set forth do not suffer from these issues at the expense of using more static rules than the LLDP approach.

SDN era has also generated many recent works on network debugging, fault diagnosis and detection, policy verification, dynamic and static state analysis [10]–[14]. As far as we are aware of, all these works are complementary to our work in terms of the problem spaces they specifically target. In [11], for instance, authors piggyback on existing rules installed for data flows to identify which header space can locate link failures given these rules. The obvious drawback of such solutions is that the currently installed forwarding rules for data flows might not be sufficient for diagnosing the forwarding plane (e.g., in the extreme if there are no data traffic, there may be no forwarding rules installed!). Furthermore, the controllers should be able to reach any switch (i.e., all control interfaces are operational) and be able to inject data packets mimicking the end hosts. Installing static forwarding rules to be used for later forwarding plane diagnosis via control flows and optimizing the associated costs are the main features we have that are also absent in prior art on SDNs.

III. System Model

A. Network Architecture

The main features of the network architecture are captured in Fig. 1. Our system model follows the OpenFlow model [15]. Network consists of a forwarding plane and a control plane. The forwarding plane consists of forwarding elements (will use the term *switch* interchangeably), each supporting a finite number of $\{match, action\}$ rules. A *match* pattern is defined using incoming port ID and packet headers using ternary values 0, 1, or * (i.e., don't care). In essence, each match pattern defines a network flow. The following *actions* on a particular flow match are allowed: *forward to outgoing port ID*, *drop packet*, *pop outer header field*, *push outer header field*, *overwrite header field*. Regardless of match length and number of actions taken per matching (e.g., switch can first rewrite a particular field in the packet header, then push an MPLS label, and finally forward to a particular interface), we will count the cost of each flow matching rule as one forwarding rule. The forwarding plane in Fig. 1 has seven switches (s_1 through s_7) and nine interfaces/links between them. The set of switches is denoted by $\mathbf{S}$. We assume that each link is bidirectional, i.e., a switch can both receive and send over the same link. The model is applicable to cases where multiple interfaces exist between the same pair of switches, there are logical interfaces (e.g., a preconfigured tunnel with a tunnel ID configured on both ends), or pairs of directional links in opposite directions interconnect the same pair of switches.

The control plane consists of one or more controllers. In OpenFlow, a switch can be programmed (i.e., forwarding rules are installed) by multiple controllers by assigning these controller *equal roles*. A common deployment scenario however allows each switch to be programmed only by one controller (called *master* controller). Controllers typically install forwarding rules on multiple switches using control interfaces (depicted by red dotted lines in the figure). The control interfaces can be in-band (i.e., the forwarding plane is used for inter-controller and controller-switch communication) or out-of-band (i.e., a separate physical network interconnects controllers with each other and switches) or hybrid of both. The subset of switches that a controller C_i can send packets to and install rules on forms its control domain $\mathbf{D}_i \subseteq \mathbf{S}$. The rest of the switches constitutes its complementary domain $\overline{\mathbf{D}}_i = \mathbf{S} \setminus \mathbf{D}_i$. For C_3, $\mathbf{D}_3 = \{s_4, s_7\}$ and $\overline{\mathbf{D}}_3 = \{s_1, s_2, s_3, s_5, s_6\}$, i.e., C_3 can send control packets to and install forwarding rules on s_4 and s_7. To install a rule on a switch in its $\overline{\mathbf{D}}_3$ (e.g., s_1), C_3 must send its request to any controller (e.g., C_1) which includes that switch in its control domain.

We assume that the core functions such as topology discovery, inter-controller communications, master/slave selection, routing, etc., are supported as part of the controller framework and they are implementation specific. As a pre-requisite, we assume that the controllers know the network topology as it is used by at least the routing applications. LLDP can be used infrequently as part of the planned topology updates or network reboots. In some cases, the controllers can extract topology information directly from a database updated manu-

ally or using a network planning tool.

In OpenFlow model, switches monitor the interface up/down events for their local ports and report status changes to the controllers over their control interfaces. Although it is not part of the OpenFlow specification, suppose each switch monitors the health of their local links by exchanging periodic heartbeat messages with their neighboring switches. Even in that case, a controller cannot receive direct notification from the switches outside its $\overline{\mathbf{D}}$. E.g., C_3 would be unaware of a link failure between s_2 and s_3. One way of receiving failure notification is via other controllers. E.g., both C_1 and C_2 can receive failure notification for link between s_2 and s_3. Then, they can notify C_3. Note that this series of events are dependent on the assumption that switches locate their local link failures via heartbeat messages. As already stated, this is not supported by current OpenFlow switches. Even if a future OpenFlow switch specification indeed makes this a standard functionality, the failure reporting itself might fail due to problems in the control plane. For instance the communication path between pairs of controllers might be slow, congested, or disrupted (e.g., configuration mistakes, controller overload, software/hardware failures on the control plane, DOS attacks). Thus, in the absence of switch support on locating local link failures or proper reporting, the only other viable option is to install static forwarding rules onto the forwarding plane for global network monitoring.

Once these static forwarding rules are in place, a controller can learn about the topology failures within its $\overline{\mathbf{D}}$ by sending control messages from its $\mathbf{D}$ into its $\overline{\mathbf{D}}$ and listen to the responses. These messages constitute control flows. Since the controller cannot install new rules in $\overline{\mathbf{D}}$, a priori static forwarding rules must be installed on switches in $\overline{\mathbf{D}}$ for the control flows. Finding these static rules given the forwarding plane topology for optimal network diagnosis is the focus of this paper. Once the forwarding plane topology is learned and consistently shared across the controllers, any controller can compute these static rules, distribute to other controllers, and each controller installs them in their current $\mathbf{D}$. When the static rules are placed into the switch forwarding tables, controllers can start using them for network diagnosis.

Our system model does not assume that $\mathbf{D}$ and $\overline{\mathbf{D}}$ are fixed or known a priori. C_3 for instance might have had functional control interfaces to other switches in $\overline{\mathbf{D}}_3$, but they may have been lost. A priori C_3 does not have any clue on which control interfaces would be failed. In another case, there might be a dynamic partitioning of the forwarding plane such that a controller becomes in charge of different forwarding elements over time based on some optimization logic. Not assuming fixed $\mathbf{D}$ and $\overline{\mathbf{D}}$ may be useful even when they are actually fixed. If multiple controllers coexist each with a different $\mathbf{D}$ and $\overline{\mathbf{D}}$, instead of setting static rules with respect to each controller, it would be more efficient (i.e., requires less forwarding rules for control flows) to set up static rules independent of what $\mathbf{D}$ and $\overline{\mathbf{D}}$ are actually.

Note that a controller can dynamically install new forwarding rules on its $\mathbf{D}$ for control flows. As it will be clear in the later sections, we will make use of this advantage to loopback the control packets to the original controller.

B. Failure Scenarios

We will differentiate between two different failure scenarios: *(i) Symmetric failures*, where a bidirectional link is either functional in both directions or non-functional in either direction. *(ii) Asymmetric failures*, where a bidirectional link can fail in one direction but not necessarily in the other direction.

For symmetric failures, it suffices to visit an interface in any direction to check its health. Thus, if symmetric failures are the most common scenarios, then one can specifically plan network diagnosis for such more probable incidents and save costs. We will model the forwarding plane as an undirected graph $\mathbf{G_u}(\mathbf{S}, \mathbf{E})$ in this case. Here, $\mathbf{S}$ is the set of vertices, where there is a 1-1 mapping from the forwarding elements to the vertices in $\mathbf{S}$ and $\mathbf{E}$ is the set of edges, where there is 1-1 mapping from the bidirectional links to the edges in $\mathbf{E}$.

For asymmetric failures, both directions of the link must be examined. The problem is more constrained and imposes higher diagnosis costs. However, as we will see later in the paper, shortest walks become optimal in the number of forwarding rules. We will model the forwarding plane as a directed graph $\mathbf{G_d}(\mathbf{S}, \vec{\mathbf{E}})$ in this case. Here, $\vec{\mathbf{E}}$ is the set of arcs, where there is 1-1 mapping from each direction of the links to the arcs in $\vec{\mathbf{E}}$. Note that we use the terms *directed edge* and *arc* interchangeably throughout the paper. We denote the directed edge from vertex i to vertex j in $\mathbf{S}$ by $e_{ij} \in \vec{\mathbf{E}}$.

C. Problem Statement and Cost Metrics

Given the forwarding plane topology and without any information on $\mathbf{D}$, we would like to compute and install static forwarding rules such that as long as controller C_i is master of at least one switch (i.e., $|\mathbf{D}_i| \geq 1$) it can run network diagnostics. We investigate two related but distinct problems for network diagnosis: (1) verification of topology connectivity and (2) localization of link failures. Observe that solving failure localization problem can also serve as topology verification, but not vice versa.

Static forwarding rules can be interpreted as one or more walks on the undirected ($\mathbf{G_u}(\mathbf{S}, \mathbf{E})$) or directed ($\mathbf{G_d}(\mathbf{S}, \vec{\mathbf{E}})$) topology graph of the forwarding plane. Each walk in essence a sequence of consecutively visited directed edges in the form $[e_{ij}, e_{jm}, e_{ml}, \ldots, e_{pz}, e_{zr}]$, where the tail of the current directed edge is the head of the next directed edge. We also use the form $s_i \rightarrow s_j \rightarrow s_m \rightarrow \ldots \rightarrow s_p \rightarrow s_z \rightarrow s_r$ to denote the same walk. Without loss of generality, we only consider closed walks (i.e., walk starts and ends at the same vertex) since such walks are the only ones that can satisfy the constraint $|\mathbf{D}| \geq 1$. Let $\mathbf{\Omega}$ represent the set of all closed walks on $\mathbf{G_u}$ (or $\mathbf{G_d}$). Suppose we picked the subset of walks $\mathbf{\Omega}_j = \{\mathbf{W}_{j1}, \ldots, \mathbf{W}_{jK}\} \subseteq \mathbf{\Omega}$ to diagnose the forwarding plane. Then, we can measure the cost of $\mathbf{\Omega}_j$ as follows.

1) Number of control messages: Each walk is traversed by a different control packet as otherwise we cannot differentiate among the walks. Thus, the overhead in terms of the number of control messages is equal to the number of walks we picked (i.e., K).

2) Latency: Each control packet k traverses the walk $\mathbf{W}_{jk}$ and thus experiences a latency of $L_k(\tau_s + \tau_p) + 2\tau_c$. Here, L_k is the length of walk $\mathbf{W}_{jk}$ in hops, τ_s is the switching delay for a control packet in the forwarding plane, τ_p is the average propagation delay between the switches along the walk, and τ_c is the delay between the controller and switch. Depending on whether the walks can be executed in parallel or not, the overall latency figure would vary. If walks can be done in parallel, then the total latency is given by $(\tau_s+\tau_p)\max_k L_k+(K+1)\tau_c$. If walks must be done sequentially, then the total latency becomes $(\sum_k L_k)(\tau_s+\tau_p)+2K\tau_c$. Any hybrid solution (i.e., walks are grouped together, within a group they are executed in parallel while across groups they are executed in sequence) would have a latency between these two extremes.

3) Number of static rules: Let $\vec{\mathbf{E}}(\mathbf{W}_{jk})$ represent the set of directed edges traversed by $\mathbf{W}_{jk}$. If a directed edge is traversed multiple times within the same walk or across walks, it can share the same forwarding rule at the head of directed edge (i.e., sending switch). The easiest way of proving this is to let controllers use source based routing. In this brute-force approach, each arc is assigned a unique label. The whole route is specified in the packet header by concatenating the labels of the links in the order they are visited. Each switch has a matching rule for the label and the action consists of popping the outermost label and forwarding. However, for data-center scale topologies, this would create very large control packets. A better way is to let switches do the packet labeling/tagging where necessary. Large control packets can be avoided by pushing and popping labels on the forwarding plane rather than stacking per hop labels at the source. Each duplicated sequence of arcs on a walk are treated as a single *tunnel* or *trunk*. Labels are pushed before the tunnel and popped at the end of the tunnels (see Section IV-D for details). For each unique arc (i.e., directional edge) in any $\mathbf{W}_{jk}$, there must be a distinct forwarding rule as without such a rule traversing the corresponding link in the specific direction is not possible. Accordingly, we can express the total number of static forwarding rules as $|\bigcup_k \vec{\mathbf{E}}(W_{jk})|$, i.e., the cardinality of union of arc sets belonging to distinct walks used for diagnosis. Thus, in SDNs minimization of $|\bigcup_k \vec{\mathbf{E}}(W_{jk})|$ becomes the relevant resource optimization criterion as opposed to $\sum_k L_k$ used in optical networks.

For an arbitrary topology, it is not possible to optimize each of these cost metrics simultaneously. In the rest of the paper, our focus will be mainly on minimizing the total number of static forwarding rules. We will also quantify the costs in terms of latency and number of control messages. As it will be clear, our solutions will be optimal for both the static forwarding rules and the number of control messages simultaneously for several network topologies.

IV. Verifying Topology Connectivity

Topology verification amounts to finding a set of walks that collectively visit every edge in $\mathbf{G_u}$ or arc in $\mathbf{G_d}$ at least once. We set our main objective as the minimization of the total number of forwarding rules. Then, a natural choice for solving this problem is to pick a single closed walk $\mathbf{W_{opt}}$ that has the shortest length across all walks that visit each edge in $\mathbf{G_u}$ or arc in $\mathbf{G_d}$ at least once. In general there are multiple shortest walks that can be picked as $\mathbf{W_{opt}}$. In the following sections we will establish that (i) it is not necessarily true that any of these candidate $\mathbf{W_{opt}}$'s is the optimum solution for the number of forwarding rules, (ii) not all shortest paths are equivalent in terms of the objective, (iii) one can always find a single walk that achieves the objective in polynomial time. We also present a heuristic algorithm that tries to pick the best walk within the class of shortest walks. Our evaluations over real topologies indicate that this heuristic solution is within 14% of the optimum solution.

A. Symmetric Failure Scenarios

Finding the shortest length across all closed walks that visit each edge in $\mathbf{G_u}$ or arc in $\mathbf{G_d}$ at least once is known as *Chinese Postman Problem*. Thus, as a starting point, we pick $\mathbf{W_{opt}}$ as any solution of *Chinese Postman Problem*.

For undirected connected graphs such as $\mathbf{G_u}(\mathbf{S},\mathbf{E})$, $\mathbf{W_{opt}}$ can be computed in polynomial time [16]. A trivial lower bound on the number of forwarding rules is $|\mathbf{E}|$ since we need to forward the control packet for topology verification onto each link at least once and without a rule such forwarding action would not occur. Then, if there is a cycle in $\mathbf{G_u}$ that visits every edge in $\mathbf{E}$ exactly once (i.e., an *Euler Cycle* exists), we can state that $\mathbf{\Omega}^* = \{\mathbf{W_{opt}}\}$ is the optimum solution for topology verification. $\mathbf{\Omega}^*$ minimizes both the total number of static forwarding rules and the total number of control messages. Specifically, $\mathbf{\Omega}^*$ requires $|\mathbf{E}|$ static rules and one control message. The latency of $\mathbf{\Omega}^*$ becomes $|\mathbf{E}|(\tau_s + \tau_p) + 2\tau_c$. A well-known necessary and sufficient condition for existence of an Euler cycle in a connected undirected graph is to have every vertex to have even number of edges.

Unfortunately, not all forwarding topologies have Euler cycles. E.g., the forwarding plane in Fig. 1 has no Euler cycle. Remember that when we want to optimize the number of static rules, it is not the length of $\mathbf{W_{opt}}$ (denoted as L_{opt}), but the cardinality of $\vec{\mathbf{E}}(\mathbf{W_{opt}})$ that must be minimized. Denote the total number of duplicate arcs for a given walk $\mathbf{W}$ as $\kappa(\mathbf{W})$. Then, we can express the total number of static rules by $\mathbf{W_{opt}}$ as $(L_{opt} - \kappa(\mathbf{W_{opt}}))$. After stating the next lemma, we can at least claim that $\mathbf{\Omega}^*$ has a worst case *competitiveness ratio* of two in number of forwarding rules, i.e., $|\mathbf{E}| \leq (L_{opt} - \kappa(\mathbf{W_{opt}})) \leq 2|\mathbf{E}|$. The solution is obviously optimum in number of control messages. The latency can be written as $T_k = L_{opt}(\tau_s + \tau_p) + 2\tau_c$.

Lemma 4.1: In $\mathbf{W_{opt}}$, no edge is traversed more than twice. In other words: $L_{opt} \leq 2|\mathbf{E}|$.

Proof: Suppose an edge is traversed more than twice. Then, we can construct an undirected (multi)graph $\mathbf{G_u}(\mathbf{S}',\mathbf{E}')$ from $\mathbf{G_u}(\mathbf{S},\mathbf{E})$ such that $\mathbf{S} = \mathbf{S}'$ and $\mathbf{E} \subset \mathbf{E}'$ with each occurrence of an edge in $\mathbf{W_{opt}}$ has a 1-1 mapping to $\mathbf{E}'$. Note that although $\mathbf{W_{opt}}$ may not necessarily be an Euler cycle on $\mathbf{G_u}(\mathbf{S},\mathbf{E})$, it is always an Euler cycle on $\mathbf{G_u}(\mathbf{S}',\mathbf{E}')$ by construction. Hence, every vertex in $\mathbf{G_u}(\mathbf{S}',\mathbf{E}')$ must have an even degree as this is a necessary and sufficient condition

for connected graphs [16]. However, if there are more than two edges between two vertices as in this case (meaning that the corresponding edge in $\mathbf{G_u}(\mathbf{S}, \mathbf{E})$ is traversed more than twice), one can remove two edges at a time between these vertices until there remains one or two edges between the same vertices. Since we started with vertices that have even degrees, removing an even number of edges would preserve the same property. In other words, an Euler path exists after these edge deletions in $\mathbf{G_u}(\mathbf{S}', \mathbf{E}')$ such that it is strictly shorter than $\mathbf{W_{opt}}$ and visited every edge in $\mathbf{G_u}(\mathbf{S}, \mathbf{E})$ at least once. Hence, $\mathbf{W_{opt}}$ cannot be the shortest cycle for topology verification, which is a contradiction. ■

We can express a tighter lower bound for the number of static rules than $|\mathbf{E}|$ by counting *bridge* links.

Definition An edge (if omitted) that partitions an undirected graph into two disconnected sub-graphs is called a ***bridge***.

When a walk on a graph starts on one side of a bridge, if it crosses the bridge in one direction, it has to cross the same bridge in the reverse direction to come back to the starting point. Using this trivial observation, one can state the following lower bound on total number of forwarding rules.

***Lemma** 4.2 (Lower Bound):* Topology verification needs at least $|\mathbf{E}|+|\mathbf{B}|$ forwarding rules, where $\mathbf{B}$ is the set of bridges.

Proof: For topology verification, each edge on $\mathbf{G_u}$ must be traversed in at least one direction. Moreover, each bridge must be crossed in both directions as otherwise we cannot loop back to the starting point. Thus, there are at least $|\mathbf{E}| + |\mathbf{B}|$ unique arcs that must be visited. Each uniquely visited arc requires at least one forwarding rule at the head node. Thus, we need at least $|\mathbf{E}| + |\mathbf{B}|$ forwarding rules. ■

All the edges in tree, star, and linear topologies are bridges rendering the lower bound $2|\mathbf{E}|$. Hence, $\mathbf{\Omega}^*$ is indeed the optimum solution in number of static rules over such topologies.

Note that the lower bound given by Lemma 4.2 may not be achievable by $\mathbf{W_{opt}}$ in general. Though, it can be achievable by a longer walk. An example is depicted in Figure 2. The forwarding plane topology represented by the leftmost graph has no bridges and has $|\mathbf{E}| = 8$. Lemma 4.2 indicates that we need at least 8 forwarding rules. Solving Chinese Postman Problem however leads to an optimal walk $\mathbf{W_{opt}}$ with $L_{opt} = 10$. The corresponding logical ring is shown at the center of Figure 2. The walk traverses the arc e_{41} twice, leading to $\kappa = 1$. Hence, $\mathbf{W_{opt}}$ requires $(L_{opt} - \kappa) = 9$ static forwarding rules. This is strictly larger than the lower bound. The difference is due to the edge between s_2 and s_3 as it must be traversed in both directions requiring installation of one rule at s_2 and one rule at s_3. Other candidate solutions for $\mathbf{W_{opt}}$ suffer from a similar non-bridge link reversal. This situation is avoided over the logical ring constructed by a longer walk as shown by the rightmost ring topology in the figure. Instead of moving directly from s_3 to s_2 (as in the optimal walk), a longer path $s_3 - s_4 - s_1 - s_2$ is taken. As a result, the ring has 12 hops with $\kappa = 4$ (e_{12}, e_{34}, e_{41} occur two, two, and three times, respectively). Thus, this longer walk requires $(L - \kappa) = 8$ static rules achieving the lower bound. Next, we formalize the achievability result.

***Theorem** 4.3 (Achievability of Lower Bound):* Lower bound

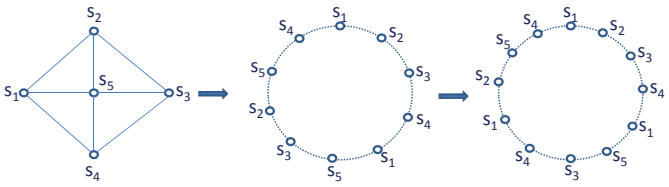


Fig. 2: Example of a topology, where $\mathbf{W_{opt}}$ cannot achieve the lower bound in Lemma 4.2, but a longer walk achieves it.

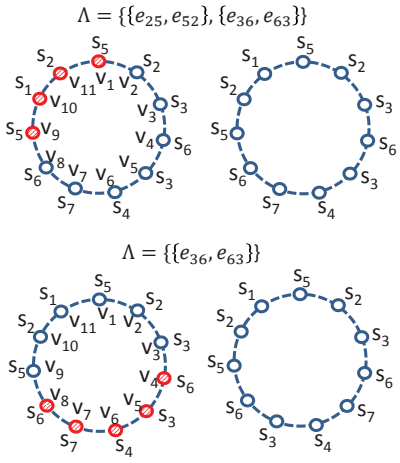


Fig. 3: Algorithm 1 in action.

in Lemma 4.2 is achievable.

Proof: We start the proof by first removing all the bridges from $\mathbf{G_u}$, which will generate $|\mathbf{B}|+1$ disconnected subgraphs G_i ($i = 0, \ldots, |\mathbf{B}|$). Let S_i and E_i represent the vertex set and edge set of G_i. Robbins Theorem [17] states that any two-connected undirected graph can be strongly oriented. Since all G_i's are two-connected undirected graphs, we can pick any one direction of an edge in G_i and generate a directed graph G_i^d such that we can find a directed path between any two vertices of G_i^d. Next, starting from any vertex on G_i^d we can construct a closed walk W_i that visits every arc on G_i^d at least once. The same closed walk visits every edge in E_i at least once and always in the same direction, i.e., $L(W_i) - \kappa(W_i) = |E_i|$. Therefore, each walk requires exactly E_i forwarding rules.

In the final part, we need to stitch each W_i's together to form a larger closed walk that also visits all the bridges in $\mathbf{G_u}$. This can be done easily as i-th bridge connects the cycle W_{i-1} to W_i. Thus, we can start the walk W_0, moving to W_1 on the first bridge, to W_2 on the second bridge, and so on up to walk $W_{|\mathbf{B}|}$. Then, crossing each bridge in the reverse direction after completing each walk W_k, we move back to W_{k-1}, to W_{k-2}, and so on until we arrive at W_0 and complete it. This combined walk ends up traversing each non-bridge edge only in one direction and each bridge edge in both directions. Therefore, it requires exactly $|\mathbf{E}| + |\mathbf{B}|$ forwarding rules. ■

Although optimality in number of static rules and number of control messages can be achieved by a longer walk, we may pay a substantial penalty in delay for large topologies as the walk might be much longer. Thus, it would be more desirable to pay a small penalty in number of static rules and do not in-

Algorithm 1 Heuristic for reducing $(L_{opt} - \kappa(\mathbf{W_{opt}}))$

Step 1: Find a solution $\mathbf{W_{opt}}$ to Chinese Postman Problem. Let v_k denote the absolute position of each hop on $\mathbf{W_{opt}}$ and $f(v_k)$ is the actual switch at that position.
Step 2: Construct set Λ such that a pair of arcs $\{e_{ij}, e_{ji}\} \in \Lambda$ iff both e_{ij}, e_{ji} appear in $\mathbf{W_{opt}}$ and the corresponding edge between s_i and s_j in $\mathbf{G_u}$ is not a bridge.
while $\Lambda \neq \varnothing$ **do**
  Step 3: Remove the pair of arcs $\{e_{ij}, e_{ji}\}$ from Λ that are closest to each other on $\mathbf{W_{opt}}$. Without this pair, $\mathbf{W_{opt}}$ divides into two parts. Denote the part that keeps s_i as W_1 and the part that keeps s_j as W_2.
  Step 4: Construct set Γ such that $\{v_k, v_l\} \in \Gamma$ iff $f(v_k) = f(v_l)$, v_k is on W_1 and v_l is on W_2.
  while $\Gamma \neq \varnothing$ **do**
    Step 5: Remove a pair $\{v_k, v_l\}$ from Γ.
    Step 6: Denote the cycle that starts from v_k on W_1 and ends at v_l on W_2 as W_3. Denote the cycle that starts from v_l on W_2 and ends at v_k on W_1 as W_4. Construct a new walk $\mathbf{W'_{opt}}$ by stitching W_3 to the reverse of walk W_4 (or equivalently stitching the reverse walk of W_3 to W_4).
    if $\kappa(\mathbf{W'_{opt}}) > \kappa(\mathbf{W_{opt}})$ **then**
      Step 7: $\mathbf{W_{opt}} := \mathbf{W'_{opt}}$ and break.
    end if
  end while
  Step 8: Remove any pair $\{e_{ij}, e_{ji}\}$ from Λ if either of its arcs is not on $\mathbf{W_{opt}}$.
end while

cur additional delays by sticking with $\mathbf{W_{opt}}$. In general, there is more than one solution to Chinese Postman Problem (CPP). Then, we should search for a solution of CPP that achieves the minimum $(L_{opt} - \kappa(\mathbf{W_{opt}}))$ (or equivalently maximum $\kappa(\mathbf{W_{opt}})$) to attain lower forwarding rule costs. Algorithm 1 provides a simple transformation on top of an initial $\mathbf{W_{opt}}$ that iterates over the non-bridge links that have both of their arcs in $\vec{\mathbf{E}}(\mathbf{W_{opt}})$. An example of these iterative transformations is shown in Fig. 3 based on the forwarding plane topology given in Fig. 1. In Step 1, initial solution of CPP returns a walk of length 11 with $\kappa = 0$. In Step 2 of the algorithm, we have $\Lambda = \{\{e_{25}, e_{52}\}, \{e_{36}, e_{63}\}\}$. In Step 3 and Step 4, we inspect the pair $\{e_{25}, e_{52}\}$ and construct $\Gamma = \{\{v_1, v_9\}\}$, respectively. In Step 6, $W_3 = s_5 \rightarrow s_1 \rightarrow s_2 \rightarrow s_5$ is reversed and the rest of the walk remains the same. The top-right ring in clockwise direction depicts newly constructed walk of same length as before but $\kappa = 1$. Hence, it requires one less forwarding rules than the initial walk. Next iteration starts with this new walk (bottom-left ring in the figure). There is only one candidate pair $\{e_{36}, e_{63}\}$ to consider with $\Gamma = \{\{v_4, v_8\}\}$. By reversing the part of the ring $s_6 \rightarrow s_3 \rightarrow s_4 \rightarrow s_7 \rightarrow s_6$ in Step 6, we obtain a new walk (bottom-right ring) with $\kappa = 2$. Algorithm terminates at this stage. The newly constructed walk is still a solution of CPP. Moreover it requires $(L_{opt} - \kappa(\mathbf{W_{opt}})) = 9$ static rules. Since $|E| = 9$ in $\mathbf{G_u}$, this is an optimum walk in number of static forwarding rules. Our evaluations over real topologies (see Section VII) indicate that Algorithm 1 is optimum in number of static rules for 60% of the topologies, and for the rest it stays within 14% of the optimum.

The complexity of Algorithm 1 is $O(|E|)$ if an Euler cycle exists, otherwise $O(|V|^3)$ steps needed to solve CPP. Finally, $O(2|E|)$ new rules should be installed in the network. The delay of pushing these rules would be dominant in practice rather than the computation latency of the rules. In the event that the network topology is updated (e.g., due to maintenance or upgrades), in the worst case we have to recompute the Algorithm 1 and update all the rules. If the topology changes are constrained, we do not have to re-compute the whole walk, but the walk can be incrementally updated.

B. Special Case: Fat-Tree Topology

Fat-tree topology [18] is a popular choice in data centers. A k-ary fat-tree topology is organized as k pods, each comprised of two layers of $k/2$ switches. Each switch has k ports. The switch in the lowest layer in the pod is connected to $k/2$ hosts and the remaining $k/2$ ports are each connected to a distinct switch in the upper layer in the pod. On top of the pods, there are $(k/2)^2$ k-port core switches, each connected to exactly one switch in each pod and each switch in the upper layer in the pod is connected to exactly $k/2$ core switch. Therefore, including the end hosts, each switch has k links. Clearly, k must be an even number to render $k/2$ an integer. The architecture supports $k^3/4$ end hosts connected to each switch over a single link. Overall, a k-ary fat-tree topology has $3k^3/4$ edges of which $k^3/4$ are bridges and k is an even number. Thus, the lower bound on forwarding rules becomes $|E| + |B| = 3k^3/4 + k^3/4 = k^3$.

When $k/2$ is even, this lower bound is achieved. For the proof, it suffices to augment the original topology graph into another one that has an Euler cycle. Note that, in fat-tree topology, only the end hosts have odd number of links and all the switches have even number of edges. When $k/2$ is itself even, we can add an extra edge between each host and the switch it is already connected to. This newly constructed topology graph has exactly k^3 edges with all switches having an even degree. Therefore, it has an Euler cycle of length $k^3 = |E| + |B|$, which can be computed in $O(|E| + |B|)$ steps. Thus, $\mathbf{W_{opt}}$ becomes an Euler cycle of this augmented topology and achieves the lower bound in Lemma 4.2. Many data-center grade switches have ports in multiples of four and for all practical purposes, to constrain $k/2$ to be an even number is not a major obstacle.

If $k/2$ is not even, we can augment the graph into another graph with Euler cycle as follows:. We add an additional edge to $k/2-1$ hosts from each switch at the bottom layer. The remaining hosts with odd number of edges are paired together to minimize the total length of the shortest paths interconnecting the pairs. Each shortest path is of length 4 if paired hosts are in the same pod or of length 6 if they are in different pods. Extra edges are added along these shortest paths. The resulting graph has all even-degree nodes and hence it has an Euler cycle. The total number of edges in the resulting graph can be computed as $3k^3/4 + (k/2-1)(k^2/2) + (k-2)k + 3k$, where the first

terms is the number of edges in fat-tree topology, the second term is the number of edges added for even number of hosts under the same switch, the third term is the number of edges added for hosts paired under the same pod, and the fourth term is the number of edges added for paired hosts between different pods. The total number of edges in the resulting graph becomes $k^3+k^2/2+k$ rendering $L_{opt} \leq k^3+k^2/2+k$. In other words, the number of static forwarding rules are at most within a factor of $(1+\frac{1}{2k}+\frac{1}{k^2})$ of the optimum in Theorem 4.3. For 3-port, 5-port, 7-port, 23-port switches, the overhead is within 28%, 14%, 9.2%, 2.4% of the minimum number of rules. As data centers employ even denser port counts per switch, we can claim near-optimality in number of static forwarding rules for fat-tree topologies.

C. Asymmetric Failure Scenarios

To verify topology against asymmetric failures, one or more control packets must visit each link in both directions. Since there are exactly $2|\mathbf{E}|$ arcs to be visited and each arc must have a distinct forwarding rule, the number of static forwarding rules is $\geq 2|\mathbf{E}|$. Due to our bi-directional link assumption on $\mathbf{G_d}(\mathbf{V}, \vec{\mathbf{E}})$, every vertex has equal in-degree (i.e., the number of incoming arcs) and out-degree (i.e., the number of outgoing arcs). In connected graphs, this is a sufficient condition for the existence of an Euler cycle [16]. An Euler cycle can be found in $\Theta(2|E|)$ steps over $\mathbf{G_d}$. As Euler cycles visit every arc in a directed graph exactly once, $\mathbf{\Omega}^* = \{\mathbf{W_{opt}}\}$ is the optimum solution both in terms of static forwarding rules and control messages. The latency of this solution, however, becomes $2|\mathbf{E}|(\tau_s + \tau_p) + 2\tau_c$.

D. Mapping the Walk to Static Forwarding Rules

Once a walk W is determined, we need to install forwarding rules at switch s_i for each arc e_{ij} in $\vec{\mathbf{E}}(W)$. The walk and the set of corresponding forwarding rules must uniquely define a control flow for topology connectivity. For this purpose, one of the packet headers is used to identify that this packet is used to verify topology connectivity. Suppose source IP address field is used to this end and a unique local IP address IP_A is assigned. Then, all static forwarding rules for W must match source IP address to IP_A. Table I shows how the closed walk $s_1 \rightarrow s_5 \rightarrow s_2 \rightarrow s_3 \rightarrow s_6 \rightarrow s_7 \rightarrow s_4 \rightarrow s_3 \rightarrow s_6 \rightarrow s_5 \rightarrow s_2 \rightarrow s_1$ is realized using 9 forwarding rules. Let us use this walk example below to describe the basic steps to convert a walk to a set of static forwarding rules. Note that there are many other alternatives to define matching rules and actions to realize the same walk.

If a switch s_i is visited only once by W, the static rule does not need to match any other packet field than the one that identifies the control packet. In our example, s_1, s_4, and s_7 are visited only once and these switches only need to match source IP address against IP_A to take the correct forwarding action. If a switch s_i is visited multiple times but always forwards to the same link, then again the static rule does not need to match any other packet field. E.g., s_3 and s_5 occurs twice in the example walk yet their forwarding action is the same: s_3 forwards to s_6 and s_5 forwards to s_2. If a switch s_i is visited multiple times and forwards to different links at some of these visits, it requires a separate forwarding rule for each of these links. In many occasions outgoing link has a one-to-one mapping to the incoming switch port and thus a forwarding rule that matches incoming switch port in addition to the source IP address is sufficient to identify the outgoing link. When incoming switch port is not sufficient, an additional header field must be used to identify the outgoing link. One can use VLAN tagging or MPLS labeling or a custom header field using an extension to OpenFlow protocol. Suppose we use VLAN tags as it is a standard feature in most switches including OpenFlow. Then, we assign a unique VLAN tag for each outgoing link of s_j. In the example, s_2 and s_6 receive the same control packet twice from the same incoming interface and yet must forward to different outgoing links at each time. Since there is no 1-1 mapping to an incoming interface, incoming switch port cannot be used as a differentiating field. Consider first s_2. We first assign each outgoing interface a VLAN tag, $vlan_1$ to e_{23} and $vlan_2$ to e_{21}. Table I shows the matching rules with these tags. Now, the question is which switches as a forwarding action should tag the control packet. The walk example corresponds to clockwise direction over the logical ring depicted at the bottom-right corner of Fig. 3. Iterating back from the tagged interface, we inspect the ring in the counter clockwise direction to identify a switch that can reuse its existing matching rule and add VLAN tagging to the action set of that matching. For instance, starting from e_{23} tagged with $vlan_1$ and traversing the ring in counter clockwise direction, we first reach s_5. But, s_5 is visited twice on the ring and has only one forwarding rule. Thus, we rule it out as a candidate. Moving counter clockwise direction further, we hit s_1 that occurs once in the ring and has one forwarding rule. Hence, adding VLAN tagging with $vlan_1$ into its action set would not possibly contradict with another forwarding decision taken at the same switch. We repeat the process with e_{21} tagged with $vlan_2$. Walking in counter clockwise direction, we reach s_5, s_6, and s_3 that cannot add a tagging action either because they require an additional forwarding rule (s_5, s_3) or their forwarding rules are not yet specified (e.g., s_6). Taking one more step counter clockwise, s_4 occurs only once on the ring and without modifying its matching rule we can add an additional VLAN tagging action. The only outstanding switch with no forwarding rules specified is s_6 at this point. We first check if we can piggyback on existing VLAN tags $vlan_1$ and $vlan_2$. Indeed, we can reuse $vlan_1$ for e_{67} and $vlan_2$ for e_{65} as shown in Table I.

TABLE I: Sample Forwarding Rules

SW	Matching Rules	Actions
s_1	src IP == IP_A	set VLAN to $vlan_1$ forward to e_{15}
s_2	src IP == IP_A $\wedge$ VLAN == $vlan_1$	forward to e_{23}
s_2	src IP == IP_A $\wedge$ VLAN == $vlan_2$	forward to e_{21}
s_3	src IP == IP_A	forward to e_{36}
s_4	src IP == IP_A	set VLAN to $vlan_2$ forward to e_{43}
s_5	src IP == IP_A	forward to e_{52}
s_6	src IP == IP_A $\wedge$ VLAN == $vlan_1$	forward to e_{67}
s_6	src IP == IP_A $\wedge$ VLAN == $vlan_2$	forward to e_{65}
s_7	src IP == IP_A	forward to e_{74}

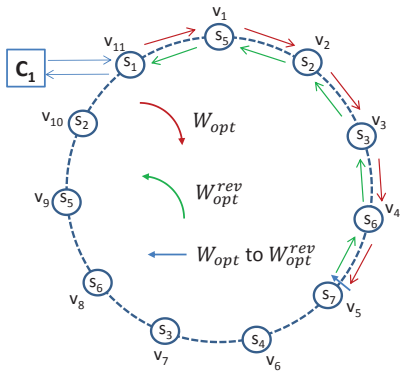


Fig. 4: An example of bidirectional logical ring topology.

E. How do controllers verify the topology connectivity?

By constructing a single closed walk that visits each edge (or arc) at least once, we formed a logical ring topology where all switches in the forwarding plane are part of. Thus, any controller C_i can use any $s_j \in \mathbf{D}_i$ to inject a control packet for topology verification. In our example in Fig. 1 and using forwarding rules in Table I, C_3 can use s_4 or s_7 to inject a packet with its source IP address set to IP_A. Similarly C_1 can use s_1, s_2, or s_5 and C_2 can use s_3 or s_6 to inject the same control packet. Controllers must initialize the control packet header properly. E.g., if s_2 is the injection point, according to Table I, not only the source IP address but also the VLAN tag must be assigned a valid value. In OpenFlow protocol, controllers can either tell the switch to which outgoing interface the control packet should be sent to or tell the switch to treat the packet the same as a packet coming from a particular incoming interface. In either case, the injection point is where the walk starts and once a packet is injected unless there is another rule specified, it would be indefinitely looped around the logical ring topology. Therefore, controllers must break the loop by defining a loopback rule at the injection point so that when the packet completes the walk, the packet is forwarded back to the controller that injected the packet. This loopback rule can be dynamically installed to any switch in a controller's current control domain. Naturally, loopback rules must have priority over the static rules installed for the closed walk. Since multiple controllers might be simultaneously inspecting the topology, this loopback rule must uniquely identify the controller. A simple solution is to use destination MAC address field and install a forwarding rule at the injection point that matches this field to MAC address of the injecting controller. Thus, each controller must also set this field in the packet header before it injects it. An alternative is to use TTL field in a matching rule (i.e., check if $TTL == 0$) and set the initial value of TTL in the control packet header to the length of the walk at the controller. The action set then must include a "decrement TTL" action in every hop.

V. Locating an Arbitrary but Single Link Failure

The solution to locate an arbitrary link failure (when one or more link failures occur) is based upon constructing a bidirectional logical ring topology. At the high level, controllers inject multiple control packets each inspecting a different segment of the ring. Depending on which control messages are received back or not by the originating controller, the controller not only detects link failures but also guarantees to locate one of the failed links.

We use the solution $\mathbf{W_{opt}}$ in the previous section as a clockwise walk on this logical ring. We also define a counter clockwise walk $\mathbf{W_{opt}^{rev}}$ by reverting the arcs. E.g., for $\mathbf{W_{opt}} = s_1 \rightarrow s_5 \rightarrow s_2 \rightarrow s_3 \rightarrow s_6 \rightarrow s_7 \rightarrow s_4 \rightarrow s_3 \rightarrow s_6 \rightarrow s_5 \rightarrow s_2 \rightarrow s_1$, $\mathbf{W_{opt}^{rev}} = s_1 \leftarrow s_5 \leftarrow s_2 \leftarrow s_3 \leftarrow s_6 \leftarrow s_7 \leftarrow s_4 \leftarrow s_3 \leftarrow s_6 \leftarrow s_5 \leftarrow s_2 \leftarrow s_1$. Since we assume bidirectional links, $\mathbf{W_{opt}^{rev}}$ is a valid closed walk over both $\mathbf{G_u}$ and $\mathbf{G_d}$. Further, $\mathbf{W_{opt}^{rev}}$ is also a solution to CPP (i.e., $\mathbf{W_{opt}^{rev}}$ has length L_{opt}) and it requires the same number of forwarding rules as $\mathbf{W_{opt}}$ (i.e., $\kappa(\mathbf{W_{opt}}) = \kappa(\mathbf{W_{opt}^{rev}})$). Once the distinct forwarding rules for $\mathbf{W_{opt}}$ and $\mathbf{W_{opt}^{rev}}$ are installed, any controller can attach to this logical ring from any switch in its $\mathbf{D}$ and inject control packets that traverse it in either clockwise or counter clockwise direction. Fig. 4 shows an example of the logical ring constructed for the forwarding plane topology in Fig. 1. We label each node on the logical ring uniquely as $v_i, i = 1, \ldots, L_{opt}$. A switch on the forwarding plane can map to multiple logical nodes on this ring. Define $f(v_i)$ as the surjective function that maps virtual nodes on the logical ring onto the switches in the forwarding plane. In Fig. 4, actual switch IDs are shown within the circles and virtual node labels are indicated next to them.

To be able to locate a link failure, we should be able to inspect any segment of the logical ring. For this purpose, we also install bounce back rules at each node of the logical ring. Suppose we use source IP address field to differentiate between the directions on the ring, e.g., IP_A is used for $\mathbf{W_{opt}}$ and IP_B is used for $\mathbf{W_{opt}^{rev}}$. The bounce back rule at a node reverses the clockwise walk to the counter clockwise walk. Suppose, we assign each virtual node v_i a unique IP address IP_{v_i}. A bounce back rule can be specified by using various fields in the packet header. Let us fix the destination IP address for this purpose. For each logical node v_i, a static bounce back rule is installed on $f(v_i)$ in the following form

If source $IP == IP_A \wedge$ destination $IP == IP_{v_i}$ then set source IP to IP_B and send back to incoming port.

Note that a bounce back rule must always have a higher priority than a forwarding rule for $\mathbf{W_{opt}}$ when a packet has matching fields for both rules. When a controller (e.g., C_1 in Fig. 1) injects a packet at $f(v_{in})$ (e.g., when $v_{in} = v_{11}$, $f(v_{in}) = s_1$) with source IP address set to IP_A and destination IP address set to IP_{v_k} (e.g., IP_{v_5}), the packet travels in clockwise direction from v_{in} (e.g., v_{11}) to v_k (e.g., v_5) and travels back to v_{in} in counter clockwise direction over the constructed logical ring. As before, the controller must install a loopback rule at injection point $f(v_{in})$ (e.g., s_1) so that the switch $f(v_{in})$ forwards the packet back to the controller rather than forwarding it along $\mathbf{W_{opt}^{rev}}$.

Once the static rules for $\mathbf{W_{opt}}$, $\mathbf{W_{opt}^{rev}}$ and bounce back as well as the dynamic loopback rules are in place, each controller can fix an injection point on the logical ring and perform a binary search over the ring by eliminating half of the links from consideration at each iteration. For instance,

if the link between s_4 and s_7 fails, C_1 can learn about this using the ring topology in Fig.4 as follows. First C_1 determines whether all the connections are healthy or not by injecting a control packet for topology verification (e.g., source IP is set as IP_A). The packet never comes back to C_1 indicating one or more link failures. Next, C_1 targets half of the logical link topology by sending a control packet with source IP as IP_A and destination IP as IP_{v_5}. The packet is received back as there are no failures in this segment. C_1 expands the search up to v_8 by setting the source IP as IP_A and destination IP as IP_{v_8}. As this part of the ring includes the failed link, C_1 does not receive the packet back. C_1 shrinks the search up to v_6 and injects a fourth control packet with source IP set to IP_A and destination IP set to IP_{v_6}. Since C_1 does not receive this fourth packet, but it received back the second packet, C_1 can conclude that the link between s_4 and s_7 has failed.

Note that when there are multiple link failures, the binary search mechanism would locate the first failure in the clockwise direction from the injection point of the logical ring. E.g., if e_{25} and e_{47} fail and C_1 uses v_{11} as the injection point, the procedure above would be able to locate only e_{25}.

A. Cost of Locating a Single Link Failure

Counting the rules for $\mathbf{W_{opt}}$, $\mathbf{W^{rev}_{opt}}$, and bounce backs, the total number of static rules to be installed can be written as $3L_{opt} - 2\kappa(\mathbf{W_{opt}})$. Since $\kappa \geq 0$ and $L_{opt} \leq 2|\mathbf{E}|$, the total number of static rules is upper bounded by $6|\mathbf{E}|$ for both symmetric and asymmetric failures. As locating a link failure trivially verifies the topology and topology verification requires at least $|\mathbf{E}|$ and $2|\mathbf{E}|$ rules for symmetric and asymmetric cases, respectively, we can easily establish the order optimality of our solution.

Including the topology verification stage, the solution requires at most $K = 1 + \lceil \log_2(L_{opt}) \rceil$ control messages to be injected. Interpreting each control packet as a binary letter, $|\mathbf{E}|$ edges and $2|\mathbf{E}|$ arcs cannot be all checked with less than $\log_2(|\mathbf{E}|)$ and $\log_2(2|\mathbf{E}|) = 1 + \log_2 |\mathbf{E}|$ messages, respectively. Hence, we also have optimality in number of control packets for both symmetric and asymmetric cases.

We can find the best case and worst case time delays as follows. For simplification, suppose L_{opt} is a power of two. The best case delay happens when each subsequent control message travels a shorter distance (i.e., exactly half of the previous one). Hence, the best case failure scenario is when the failed link occurs on the logical ring next to the injection point in the clockwise direction. Brute-force summation over these paths including the topology verification stage amounts to $(3L_{opt} - 2)$ hops in total. The worst case delay happens when each subsequent control message travels a longer distance (i.e., increase exactly by half of the not inspected part of the ring). In other words, the worst case failure scenario happens when the failed link occurs on the logical ring next to the injection point in the counter clockwise direction. Again a brute-force summation over the path lengths of each control message including the topology verification stage results with total hop count of $L_{opt}(2\log_2(L_{opt}) - 1) + 2$. With per hop delay of $\tau = \tau_s + \tau_p$, the latency becomes $T + 2K\tau_c$, where T is bounded as:

$$(3L_{opt} - 2) \times \tau \leq T \leq [L_{opt}(2\log_2(L_{opt}) - 1) + 2] \times \tau$$

B. Speeding Up Search Time

To reduce the latency of failure location, we can trade off latency with more control messages and/or forwarding rules.

1) More Control Messages: Instead of just performing a sequential search on the logical ring, we can use more control packets to parallelize the search. If we allow m control message to be injected in parallel, we can inspect $(m + 1)$ segments of the ring at once. We can then reduce the number of iterations to $\alpha = \lceil \log_{m+1}(L_{opt}) \rceil$. Including the topology verification stage, the total number of control messages M become $1 + m\alpha$. In exchange, a trivial upper bound on latency (T_{UB}) can be expressed as $(1+2\alpha)L_{opt}\tau + [2 + (m+1)\alpha]\tau_c$.

2) More Static Rules: Instead of starting the search only in the clockwise direction, we can use both directions on the ring. To enable this, we can install bounce back rules at each v_i on the ring topology to reverse $\mathbf{W^{rev}_{opt}}$ onto $\mathbf{W_{opt}}$. For this we can assign a second unique IP address $IP^{(2)}_{v_i}$ to each v_i and install a rule at each $f(v_i)$ as follows:

If source IP == $IP_B \wedge$ destination $IP == IP^{(2)}_{v_i}$ then set source IP to IP_A and send back to incoming port.

After determining which half of the ring has a faulty part, provided that the fault is closer to the injection point in the counter clockwise direction, we can switch the search direction to shorten the walk distance. For large topologies, this would cut down the worst case latency of locating a single link failure roughly by one half. Note that adding a second set of bounce back rules still preserves order optimality in number of static rules, which is now $4L_{opt} - 2\kappa(\mathbf{W_{opt}}) \leq 8|\mathbf{E}|$. The new upper bound T^*_{UB} becomes $(1 + \alpha)L_{opt}\tau + [2 + (m+1)\alpha]\tau_c$.

VI. Locating Multiple Link Failures

For arbitrary topologies, one cannot guarantee to locate multiple link failures, e.g., when forwarding plane itself is partitioned due to link failures and the controller cannot reach one of the partitions with multiple link failures.

Nevertheless, by building on the solution in the previous section, we can detect more than one link failure in a probabilistic sense. First, a controller may have a switch in its $\mathbf{D}$ in multiple locations on the logical ring. Further, it has typically more than one switch in its $\mathbf{D}$ at any given time. Thus, it can tap on the logical ring at multiple points and locate the first failure in the clockwise direction from each of these points. If the second set of bounce back rules are also installed, it can also detect potentially other failures that are closest to each injection point in the counter clockwise direction. Note that there are no guarantees that these detected failures map actually to the same link. Thus, with probability one, we locate at least one link failure and at non-zero probabilities up to $2 \times \beta(\mathbf{D}, \mathbf{W_{opt}})$ failures might be located. Here, $\beta(\mathbf{D}, \mathbf{W_{opt}})$ counts the total multiplicity of switches in $\mathbf{D}$ on $\mathbf{W_{opt}}$.

VII. Evaluations over Real Topologies

In the previous sections, we already presented bounds on the number of forwarding rules and latency of the proposed solutions, as well as the exact number of control messages. Below, we provide more evaluations and contrast our solution against existing alternatives such as LLDP approach used in SDNs and Network-Wide Local Unambiguous Failure Localization (NWL-UFL) proposed for optical networks [7].

We first present our evaluations over publicly available topologies posted in Internet Topology Zoo with their sizes varying from a few links to more than one hundred links. We mainly investigate two things: (i) How close do we get to the achievable lower bounds established for symmetric failure cases? (ii) If we install bounce back rules for both $\mathbf{W_{opt}}$ and $\mathbf{W_{opt}^{rev}}$, how many failed links do we locate?

Fig. 5 plots the ratio $(L_{opt} - \kappa(\mathbf{W_{opt}}))/(|\mathbf{E}| + |\mathbf{B}|)$ as a function of the topology size in number of edges $|\mathbf{E}|$. We label our solution in Algorithm 1 as Logical Ring. When the ratio is one, Algorithm 1 becomes an optimum solution in the number of forwarding rules for topology verification. This is indeed the case for almost 60% of the topologies, indeed our solution is optimum. Our solution remains within at most 14% of the optimum and 10% of the optimum for 98% of the topologies. The lower bound is relevant only for solutions that are tolerant against the partitions in the control domain. LLDP based topology discovery is not partition-tolerant and hence free of such constraints. In Fig. 5, LLDP is 36% less costly on average in the number of static forwarding rules.

For the latency performance comparison, we model LLDP's performance with respect to the bottleneck switch $s*$ with the most number of links ($|E_{s*}|$). Since this switch has to process $2|E_{s*}|$ control plane messages over its slow path and each LLDP packet crosses one link on the forwarding plane, LLDP discovery period cannot be faster than $2|E_{s*}|\tau_c + \tau$. At $\tau = 1\mu s$, $\tau_c = 1ms$, LLDP can achieve 14 ms discovery period averaged over all topologies with the worst delay reaching 58 ms. Our solution requires at most 2.13 ms for topology verification. When we set $\tau = 0.1ms$ and $\tau_c = 1ms$, mean and worst delays become 14.1 ms and 58.1 ms for LLDP, whereas they become 5.9 ms and 14.9 ms for the logical ring. Independent of τ and τ_c, LLDP requires 82 and 276 (i.e., *packet-in* + *packet-out*) messages between the controller and the switches on average and in the worst case topology. In contrast, logical ring requires 1 *packet-in* and 1 *packet-out* message per discovery period independent of the topology. Thus, LLDP is in general slower and more heavyweight on the control plane than the logical ring method for the topologies in the Internet Topology Zoo. In reality, we cannot allocate the whole control plane bandwidth of the bottleneck switch for LLDP packets, rendering the actual discovery period for LLDP an integer multiple of the presented delay values.

Fig. 6 plots the number of static forwarding rules per link required by our logical ring based failure localization as well as NWL-UFL that uses multiple spanning trees, and LLDP-based topology discovery. For our solution, we plot $(3L_{opt} - 2\kappa(\mathbf{W_{opt}}))/|\mathbf{E}|$ after finding the logical ring by running Algorithm 1. As expected, the per link costs are between

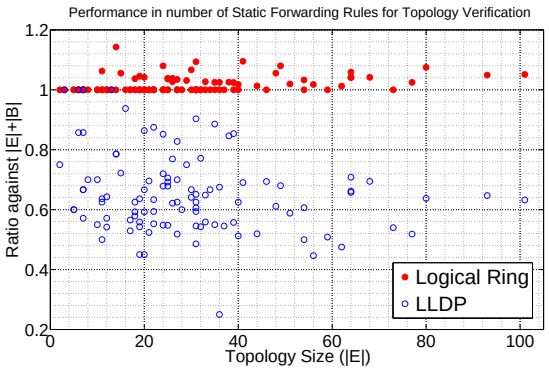


Fig. 5: Performance of Algorithm 1 in number of static forwarding rules against LLDP.

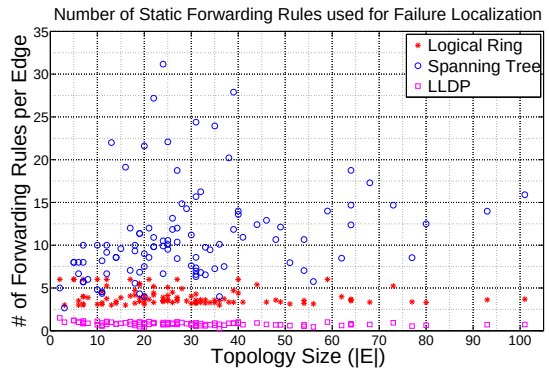


Fig. 6: Cost comparison for failure localization in number of static forwarding rules.

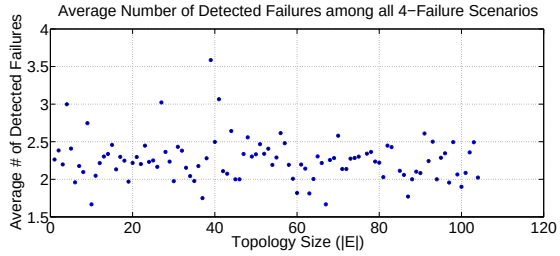


Fig. 7: Average number of detected failures per topology across all 4-failure patterns with $|\mathbf{D}| = 1$.

3 and 6 for all the evaluated topologies as $1 \leq L_{opt} \leq 2$ and $1 \leq (L_{opt} - \kappa) \leq 2$. 5.8% of the topologies have per link cost of 3, 46% have per link cost ≤ 3.5, and 70% have per link cost ≤ 4. Quite surprisingly NWL-UFL approach requires substantially more forwarding rules than the known lower bounds for using multiple trails (see Section VIII-A for further discussion). LLDP has exactly the same cost for topology verification and failure localization as it relies on periodic topology discovery. LLDP is clearly a winner in terms of the number of static forwarding rules it utilizes. Note that neither NWL-UFL nor LLDP are designed with respect to our design constraints, the first one violating control plane and data plane decoupling while the latter violating the requirements on

TABLE II: Failure Localization Latency (Logical Ring)

m	1	5	9	13	17
M	18	36	55	66	86
$T_{UB}, \tau_c = 100\tau_s$	3.87	1.66	1.44	1.22	1.23
$T_{UB}, \tau_c = 1000\tau_s$	3.9	1.7	1.5	1.29	1.31
$T^*_{UB}, \tau_c = 100\tau_s$	1.99	0.89	0.78	0.67	0.67
$T^*_{UB}, \tau_c = 1000\tau_s$	2.03	0.93	0.84	0.74	0.76

partition tolerance in the control domain.

Logical ring based solution uses periodic topology verification, and once topology verification stage fails, it attempts failure localization. Including the last topology verification stage and assuming no parallelization, no extra rules, at $\tau = 1\mu s$, $\tau_c = 1ms$, logical ring based localization 13.44 ms of average latency over 104 investigated topologies and 20.2 ms of latency in the worst case topology attaining better performance than LLDP. These mean and worst latency figures become 64.5 ms and 237.3 ms, respectively, at $\tau = 0.1ms$, $\tau_c = 1ms$. In other words, when the forwarding plane latencies increase, LLDP starts performing better and eventually beating the logical ring approach.

Fig. 7 quantifies how many failures we can actually locate. Provably, we already know that we can locate at least one. For evaluations, we fixed the number of failures, but exhaustively iterated over every failure combination on a given topology. We then assumed that there is only one switch in $\mathbf{D}$ of a particular controller and iterated over each switch as a possible injection candidate to find out as a function of injection point how many failures could be located. In the figure, we plot the average number of detected failures where the average is computed over all failure patterns and injection points for a given topology. We fixed the number of failures to four as larger number of failures was computationally quite prohibitive for us. As it can be seen, for a great majority of cases, we could actually locate two or more failures on average. For one topology with as much as 39 edges, the average was at 3.6 (i.e., for many failure patterns we could detect all four failed links). Note that although we consider only one switch in the control domain of the inspecting controller, the same switch can occur multiple times on the constructed logical ring. If a switch occurs twice, using both $\mathbf{W_{opt}}$ and $\mathbf{W^{rev}_{opt}}$, directions we can locate up to four failures. In our evaluations, we observed that a switch can occur three times for some topologies. Clearly, not all occurrences lead to locating new failures. We also have not observed any strong correlation between detecting more than one failure and the topology size.

Next, we present numerical results for fat-tree topology used commonly in data centers. For a fat-tree topology with 48-port switches (i.e., $k = 48$, $|\mathbf{E}| = 82,944$), logical ring requires $\frac{|\mathbf{E}|+|\mathbf{B}|}{|\mathbf{E}|} = 4/3$ and $\frac{3L_{opt}-2\kappa(\mathbf{W_{opt}})}{|\mathbf{E}|} = 3\frac{|\mathbf{E}|+|\mathbf{B}|}{|\mathbf{E}|} = 4$ rules per edge for topology verification and failure localization, respectively. LLDP requires merely 0.37 rules per edge.

For the same fat-tree topology, at $\tau = 1\mu s$, $\tau_c = 1ms$, the logical ring solution can complete topology verification in 112.6 ms consuming 1 packet-out and 1 packet-in message in the control plane. Logical ring utilizes 1.8% of the control plane capacity for one switch and 0% of the control plane capacity in the remaining switches. LLDP can complete topology discovery in 96 ms consuming 30,528 packet-out and 165,888 packet-in messages, utilizing 100% of the interfaces between the controller and each switch only for carrying LLDP messages. Note that our assumptions are favoring LLDP as we assume that distinct switches do not have shared bottlenecks (e.g., network bandwidth, controller CPU, etc.). If LLDP packets are limited to for instance 1.8% of the control interface capacity to match the logical ring, its discovery period increases to 5.3 seconds.

To quantify the worst case latency of failure localization (including the preceding topology verification period), we tabulate the upper-bounds T_{UB}, $T*_{UB}$ derived in Section V-B in Table II for the fat-tree topology with 48-port switches and for small m values and different τ_c. We set $\tau = \tau_s = 1\mu s$ as before. Sub-second failure localization becomes possible using $m = 5$ parallel messages ($M = 36$ total messages) and using 5.3 rules per edge. LLDP performance is the same as in topology verification. Unlike LLDP, failure localization in our solution is performed only when topology verification fails, which is in general a rare event. Thus, failure localization using logical ring has negligible impact on the long-term control plane utilization, whereas LLDP must periodically perform its heavy weight solution. As a compromised solution, one could use logical ring for continuous topology verification and upon failure detection, LLDP-based topology discovery can be run. Since logical ring in data centers can be quite long, packet losses become more probable. As logical ring is lightweight on the control plane, control packets can be sent more frequently to distinguish random packet losses from link failures.

VIII. Discussion

Next, we discuss several important issues such as fully offloading failure localization to the switches, further improving the number of forwarding rules for failure localization, and handling dynamic topologies.

A. Offloading Failure Localization to Switches

One possible alternative to controller initiated and terminated control flows is to make this control signaling an explicit function of switches. This implies that we deviate from SDN principles and have switches support control plane functions. Then, pairs of switches serve as source and destination points of control signaling. If the {source, destination} pairs and the paths in between are selected carefully, one or more switches that can tap into these control signals along the path can locate link failures and report to their respective controllers. Note that in our problem we require that as long as a controller can reach to any arbitrary switch, the controller should be able to diagnose the forwarding plane. In other words, all the switches should be able to locate any failure event in the forwarding plane when they tap into the control signals crossing them. This problem is referred to as Network-Wide Local Unambiguous Failure Localization (NWL-UFL) [7] in optical network literature.

For NWL-UFL, the total number of links traversed by any set of monitoring cycles is lower bounded by $2|\mathbf{E}|(1 - 1/|\mathbf{S}|)$ [7]. Without considering aggregation possibilities across trails,

this also constitutes a lower bound on the number of forwarding rules. If each monitoring trail is a spanning tree (e.g., as in [7]), then each link has to be traversed in both directions requiring the total number of forwarding rules (without using any rule aggregation) to be at least $4|\mathbf{E}|(1-1/|\mathbf{S}|)$. In Section VII, our evaluations indicate that NWL-UFL using the spanning tree solutions are substantially more costly than our solution in the number of forwarding rules and also far off from this lower bound.

For large enough $|\mathbf{S}|$, the lower bound without any rule aggregation simplifies to 2 forwarding rules per link in general (or 4 when trails are picked as spanning trees). Single logical ring requires ≥ 3 but ≤ 6 rules per link. In comparison to the lower bounds: logical ring is competitive against any spanning tree based solutions that does not perform rule aggregation, yet in general there is significant gap to fill or otherwise need tighter lower bounds for general cycles.

Next, we show that 2 static rules per link is actually sufficient for failure localization without offloading this function to the forwarding plane. The caveat is though we will take full advantage of rule aggregation at the expense of using longer control messages.

B. Two Rules per Edge for Failure Localization

In Section III-C3, we briefly covered how rule aggregation can be achieved via label switching and source based routing when the same link is traversed in the same direction repeatedly. In the most verbose form, each direction of all the edges are assigned a unique label. Each switch has a unique forwarding table rule per each of its interfaces in the form:

If outmost label == label(x), then pop the outmost label and forward to interface x (*)

Then, a control packet would have a header with concatenation of labels <*label(a), label(b), label(a), label(c), label(d)*> to realize the walk $[a, b, a, c, d]$. Note that since the links are bidirectional, we will have a total of $2|\mathbf{E}|$ forwarding rules in the form of (*). As a result, controllers can realize any walk as long as it is feasible to initiate within their respective control domains. For instance, walks on the logical ring that perform binary search or parallel search can be all realized leading to the same failure localization capability. In fact if one has constructed a set of walks $\mathbf{\Omega}$ that is optimal in some sense under a given set of constraints, these static rules together with source based routing are sufficient to realize them as long as walks start and end within the domain of a controller.

For data-center scale topologies though, brute-worse application of source based routing would lead to substantial sizes for control packets. E.g., 2 bytes per label for a walk of length 64K would lead to control packets larger than 128 Kbytes. Any control packet that has to go through packet fragmentation/defragmentation due to MTU limits in the network would be a poor design choice with sluggish delay performance. Therefore, practical constraints on control packet sizes and walk length require carefully crafting a set of static walks that allows forwarding rule aggregation with a few label concatenations. Existing spanning tree based constructions may **not** be particularly well suited for this purpose. It remains as an open problem how to craft these walks with constraints on packet sizes and walk lengths while getting close to the performance of two rules per edge for large topologies.

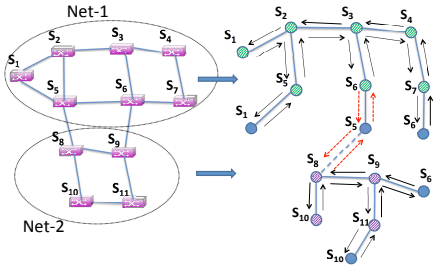


Fig. 8: Example of expanding the existing forwarding plane with a connected component.

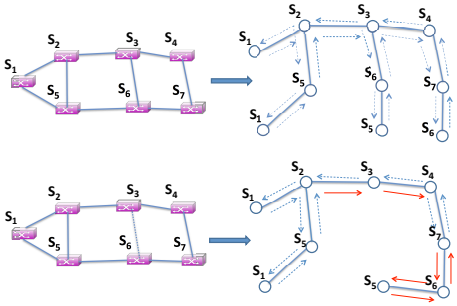


Fig. 9: Example of how a removed link is handled over spanning tree based walks.

C. Dynamic Topologies

To increase the capacity or for maintenance purposes, the physical network topology can be altered by adding or removing new links and switches. In general, this requires that we compute a new set $\mathbf{\Omega}_{new}$ of walks for network diagnosis. A new set of walks typically requires changing the existing forwarding rules in a subset of switches. One exception is when we use source based routing with unique labeling of each directional edge: adapting against the topology changes amounts to controllers recomputing the walks and injecting new control messages with updated packet header. No rule modification messages are needed. If source based routing is prohibitive, however, we need to migrate some or all of the existing static forwarding rules to the new set $\mathbf{\Omega}_{new}$ depending on topology changes and how much we want to preserve optimality (e.g., in number of static rules).

We can provide meaningful solutions that lead to minimal updates for the existing forwarding rules yet keep the total number of forwarding rules below or at 2 rules per link for topology verification and 6 rules per link for failure localization using a spanning tree approach.

Spanning Tree-based Logical Ring: When we construct the closed walk using a spanning tree, we visit each direction of the link exactly once leading to 2 rules per link to realize the clockwise walk, counter clockwise walk, and bounce back rules. Since no directional edge is visited twice, a forwarding rule solely based on the incoming port and packet identifier would be sufficient.

The benefit of settling with this upper bound is that adding an arbitrary set of new nodes and links to the forwarding topology can be accommodated by simply expanding the existing spanning tree. The border switches relative to the new set of links are the only ones that require one rule modification and one rule addition. In Figure 8, the original network topology (Net-1) is expanded by a new set of switches and links (collectively referred to as Net-2). The spanning tree based backbone of Net-1 and Net-2 are shown with green and purple patterned circles, respectively. The links that are not part of the spanning tree should be traversed as well. For this reason with solid blue colored circles we terminate such *dangling* links by duplicating the switches that comprise the other end of the link. The arrows on both side of the links indicate the direction of the closed walk. When Net-2 is added, s_5 must modify the matching rule for control packets arriving over interface e_{65} (i.e., instead of sending the control packets onto e_{56} it now sends to e_{58}). Also, as no forwarding rule exists for packets coming over e_{85}, a new matching rule for incoming interface e_{85} should be inserted with forwarding action towards interface e_{56}. These modifications and additions are shown by dashed red arrows. The rest of the forwarding rules in Net-1 remains intact.

Removal of some links can be handled by pruning these links from the existing spanning tree and finding a minimum number of edges to maintain spanning tree for the remaining links. We need up to three rule changes per removed link (two rule changes at one end of the interface and one rule change at the other end). In Figure 9, network administrator wants to bring down the link between s_3 and s_6. To orchestrate this alteration, the forwarding rule on s_3 for the incoming interface e_{23} must be modified to forward the control packets onto e_{34} instead of e_{36}. On s_6, we need two rule changes as the control packets coming from interface e_{76} are now forwarded towards e_{65} and packets coming from e_{56} are switched onto e_{67}. In the special case of removing a bridge link, one or two rule changes would suffice. E.g., in Fig. 9 the removal of link between s_6 and s_5 leads to the modification of one forwarding rule at s_6.

IX. Conclusion

We presented new results on how to diagnose the forwarding plane of SDNs using static forwarding rules. Our results are provably either optimal or order-optimal in terms of the number of static forwarding rules and control messages. For topology verification, the evaluations over real topologies revealed that our solution stayed within 14% of the lower bound and for more than half the topologies matched the lower bound in number of control messages. We also presented latency performance. At the expense of slight increase in bandwidth usage and forwarding rules, sub-second delays in locating link failures even at data-center scale topologies are achievable. Our solutions guarantee locating a single link failure, but can also probabilistically locate multiple link failures as dictated by the topology and failure patterns.

References

[1] R. Aggarwal, et al., "Bidirectional Forwarding Detection (BFD) for MPLS Label Switched Paths (LSPs)," *RFC 5884*, 2010.

[2] F. Pakzad, et al., "Efficient topology discovery in software defined networks," in *Signal Processing and Communication Systems (ICSPCS), 2014 8th International Conference on*, Dec 2014, pp. 1–8.

[3] S. Ahuja, et al., "SRLG failure localization in all-optical networks using monitoring cycles and paths," in *Proc. of IEEE INFOCOM*, 2008.

[4] J. Tapolcai, et al., "On monitoring and failure localization in mesh all-optical networks," in *Proc. of IEEE INFOCOM*, 2009.

[5] B. Wu and K. Yeung, "Monitoring cycle design for fast link failure detection in all-optical networks," in *Proc. of IEEE GLOBECOM*, 2007.

[6] N. Harvey, et al., "Non-adaptive fault diagnosis for all-optical networks via combinatorial group testing on graphs," in *Proc. of INFOCOM*, 2007.

[7] J. Tapolcai, et al., "Network-wide local unambiguous failure localization (nwl-ufl) via monitoring trails," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1762–1773, Dec. 2012.

[8] J. Tapolcai, et al., "On achieving all-optical failure restoration via monitoring trails," in *Proc. of IEEE INFOCOM*, 2013, pp. 380–384.

[9] M. Cheraghchi, et al., "Graph-constrained group testing," in *IEEE ISIT*, 2010.

[10] N. Handigol, et al., "Where is the debugger for my software-defined network?" in *Proc. of HotSDN*, 2012, pp. 55–60.

[11] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. of NSDI*, 2012.

[12] A. Khurshid, et al., "VeriFlow: Verifying network-wide invariants in real time," in *Proc. of NSDI*, 2013.

[13] R. McGeer, "Verification of switching network properties using satisfiability," in *Proc. of IEEE ICC*, June 2012, pp. 6638 –6644.

[14] M. Reitblatt, et al., "Abstractions for network update," in *Proc. of the ACM SIGCOMM*, 2012, pp. 323–334.

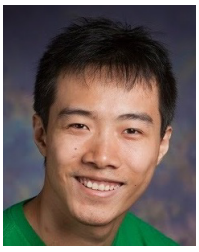
[15] N. McKeown, et al., "OpenFlow: enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[16] J. Edmonds and E. L. Johnson, "Matching, Euler tours and the Chinese postman," *Mathematical Programming*, vol. 5(1), pp. 88–124, 1973.

[17] H. Robbins, "A theorem on graphs, with an application to a problem of traffic control," *American Mathematical Monthly*, pp. 281–283, 1939.

[18] M. Al-Fares, et al., "A scalable, commodity data center network architecture," *ACM SIGCOMM CCR*, vol. 38, no. 4, pp. 63–74, 2008.

Ulaş C. Kozat (S'97-M'04-SM'10) received his B.Sc. degree in Electrical and Electronics Engineering from Bilkent University, Ankara, Turkey, in 1997, M.Sc. degree in Electrical Engineering from the George Washington University, Washington, DC, in 1999, and Ph.D. degree in Electrical and Computer Engineering from the University of Maryland, College Park, in 2004. He is a principal scientist at Argela USA, Sunnyvale, CA and an adjunct faculty at Ozyegin University, Istanbul, Turkey. Previously, he was at DOCOMO Innovations (formerly DOCOMO USA Labs) as principal research engineer and project manager.



Guanfeng Liang (S'06-M'12) received his B.E. degree from University of Science and Technology of China, Hefei, Anhui, China, in 2004, M.A.Sc. degree in Electrical and Computer Engineering from University of Toronto, Canada, in 2007, and Ph.D. degree in Electrical and Computer Engineering from the University of Illinois at Urbana-Chanpaign, in 2012. He currently works at LinkedIn, Mountain View, CA. Formerly, he worked at DOCOMO Innovations, Palo Alto, CA, as a Research Engineer.

Koray Kökten Koray Kokten received his B.Sc. degree from Istanbul Technical University and his M.Sc. degree from Ozyegin University, Istanbul, all in Electrical and Electronics Engineering, in 2010 and 2012, respectively. Between 2012 and 2013, he worked at DOCOMO Innovations Inc., Palo Alto, CA, as a visiting Researcher mainly on Software Defined Networks. Currently, he is working as a Software Design Engineer at Netas, Istanbul, in LTE eNodeB Development Department.

János Tapolcai received his M.Sc. ('00 in Technical Informatics), Ph.D. ('05 in Computer Science) degrees from Budapest University of Technology and Economics (BME), Budapest, and D.Sc. ('13 in Engineering Science) from Hungarian Academy of Sciences (MTA). Currently he is an Associate Professor at the Department of Telecommunications and Media Informatics at BME. His research interests include applied mathematics, combinatorial optimization, optical networks and IP routing, addressing and survivability.