Computer Communications 116 (2018) 172–183

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

The show must go on: Fundamental data plane connectivity services for dependable SDNs

Michael Borokhovich[a], Clement Rault[b], Liron Schiff[c], Stefan Schmid[*,d,e]

[a] AT&T Labs - Research, USA
[b] TU Berlin, Department of Telecommunication Systems, Marchstrasse 23, D - 10587 Berlin, Germany
[c] GuardiCore Labs, Israel
[d] University of Vienna, Austria
[e] Aalborg University, Denmark

ARTICLE INFO

Keywords:
Local Fast Failover
Software-Defined Networking (SDN)
OpenFlow
Algorithms
Connectivity

ABSTRACT

Software-defined network (SDN) architectures raise the question of how to deal with situations where the indirection via the control plane is not fast enough or not possible. In order to provide a high availability, connectivity, and robustness, dependable SDNs must support basic functionality also in the data plane. In particular, SDNs should implement functionality for inband network traversals, e.g., to find failover paths in the presence link failures. This paper shows that robust inband network traversal schemes for dependable SDNs are feasible, and presents three fundamentally different mechanisms: simple stateless mechanisms, efficient mechanisms based on packet tagging, and mechanisms based on dynamic state at the switches. We show how these mechanisms can be implemented in today's SDNs and discuss different applications.

1. Introduction

1.1. Motivation

Software-Defined Network (SDN) architectures distinguish between the *data plane*, consisting of the forwarding switches, and the *control plane*, consisting of one or multiple software controllers. Out-sourcing and consolidating the control over the data plane elements to a software controller simplifies the network management, and introduces new flexibilities as well as optimization opportunities, for instance, in terms of traffic engineering [1,2].

However, indirections via the control plane can come at a cost, both in terms of communication overhead as well as latency. Indeed, the reaction time to data plane events in the control plane can be orders of magnitude slower compared to a direct reaction in the network [3]: especially for the recovery of failures, a slow reaction is problematic. Worse, the indirection via the control plane may not even be possible: a controller may be temporarily or permanently unreachable, e.g., due to a network partition, a computer crash, or even due to a malicious attack [4].

This is problematic today, as computer networks have become a critical infrastructure and should provide high availability. Over the last years, researchers and practitioners have put much effort into the design of more reliable and available SDN control planes. In these designs, redundant (and possibly also geographically distributed) controllers manage the network in a coordinated fashion [5–9].

Despite these efforts to improve the control plane performance, redundant controllers alone are not sufficient to ensure the availability of SDNs. First, the additional latency incurred by the redirection via the controller may still be too high, even if the controller is nearby. Moreover, if implemented inband, even with a distributed control plane, we face a bootstrap problem [10,11]: the communication channels between switches and controllers must be established and maintained via the data plane.

Accordingly, we in this paper argue that highly available and reliable Software-Defined Networks require basic connectivity services in the data plane. In particular, the data plane should offer functionality for *inband network traversals* or *fail-safe routing*: the ability to compute alternative paths after failures (a.k.a. failover). Moreover, it should support connectivity checks.

1.2. Challenges of inband mechanisms

We are not the first to observe the benefits of inband mechanisms [12–14]. Indeed, many modern computer networks already include primitives to support the implementation of *local fast failover*

* Corresponding author at: University of Vienna, Währinger Straße 29, 1090 Vienna, Austria.
E-mail address: stefan_schmid@univie.ac.at (S. Schmid).

https://doi.org/10.1016/j.comcom.2017.12.004
Received 3 December 2016; Received in revised form 15 July 2017; Accepted 11 December 2017
Available online 13 December 2017
0140-3664/ © 2017 Published by Elsevier B.V.

mechanisms: mechanisms to handle the failures in the data plane directly.

For instance, in datacenters, Equal-Cost Multi-Path (ECMP) routing is used to automatically failover to another shortest path; in wide-area networks, networks based on Multiprotocol Label Switching (MPLS) use Fast Reroute to deal with data plane failures [3]. In the SDN context, conditional rules whose forwarding behavior depends on the local state of the switch, have been introduced in recent OpenFlow versions [12,15]. Future OpenFlow versions are likely to include more functionality or even support maintaining dynamic network state, see for example the initiatives in the context of P4 [16] and OpenState [17].

However, implementing network traversals or computing failover paths is challenging, even with the possibility to define OpenFlow local fast failover rules. Mainly for two reasons:

1) The OpenFlow failover rules must be *pre-computed* and installed ahead of time, i.e., without knowledge of the actual failures.
2) Failover rules can only depend on the *local* state of the switch, i.e., the local link failures. A local rerouting decision may not be optimal, especially in the presence of additional failures occuring in other parts of the network.

1.3. The case for robust inband traversals

A local fast failover mechanism must essentially be able to perform a *network traversal* for failsafe routing: it should find a route from source to destination, despite failures. Such robust inband network traversals may also be a useful data plane service, e.g., to check network connectivity.

Ideally, a network traveral provides a *maximal robustness*, in the sense that any packet originating at s and destined to d will reach its destination independently of the location and number of link failures, as long as s and d belong to the same physically connected component.

Little is known today about the feasibility and efficiency of implementing robust inband network travesals in software-defined networks, the topic addressed in this paper. In particular, robust routing algorithms known from other types of networks such as MPLS, are sometimes impossible to implement without additional functionality at the switch, or inefficient (e.g., require large packet headers), and hence do not scale to large networks.

1.4. Our contributions

This paper studies the feasibility and efficiency of inband network traversals, a fundamental building block for more advanced data plane services of dependable SDNs, such as robust routing and connectivity testing.

We present a comprehensive approach, exploring conceptually different solutions which provide different tradeoffs in terms of overhead and performance:

1) *Stateless mechanisms:* We show that it is feasible to implement simple yet very robust data plane traversals using today's OpenFlow protocol. In particular, we present a simple stateless scheme which is maximally robust: a route from source to destination is found, whenever this is possible. The disadvantage of this scheme are the potentially high and unpredictable route lengths.
2) *Tagging mechanisms:* We present more efficient robust traversals in a more advanced network model, using *packet tagging*, as it is also supported in OpenFlow. Our OpenFlow model and approach may be of independent interest, as it introduces an interesting new graph exploration problem.
3) *Stateful mechanisms:* Given the benefits of maintaining state in the packets, we further explore means to introduce state in an OpenFlow network. We show that, maybe suprisingly, using packet tagging is not the only way state can be introduced in OpenFlow traversals. In fact, it is possible to implement simple state machines on the switches, using the standard OpenFlow protocol, and we will refer to the corresponding state as inband registers. Moreover, we present an interesting and novel mechanism to store state in the hosts attached to the network, in a completely transparent manner, using MAC addresses to encode paths.

Finally, we discuss applications for a robust inband network traversal, including robust routing and efficient connectivity checks.

1.5. Organization

The remainder of this paper is organized as follows. Section 2 introduces the necessary background on SDN and OpenFlow. In Section 3, we present and discuss different robust traversal algorithms, using packet tagging, and in Section 5 we show how to introduce state in switches and hosts. In Section 6, we discuss applications. After reviewing related work in Section 7, we conclude with a discussion in Section 8.

2. Background and model

2.1. SDN and OpenFlow

Our work is motivated by the Software-Defined Networking paradigm, and especially OpenFlow, the predominant SDN protocol today. This section provides the necessary background on OpenFlow (focusing on the commonly used version 1.3). OpenFlow is based on a match-action concept: OpenFlow switches store rules (installed by the controller) consisting of a match and an action part. For example, an action can define a port to which the matched packet should be forwarded or change a header field (e.g., add or change a *tag*).

A new flow entry can either be installed *proactively* or *reactively*. In the reactive case, when a packet of a flow arrives at a switch and there is no matching rule, the *table miss* entry will be used. By default, upon a table miss, a packet is forwarded to the controller. Given such a *packet-in* event, the controller will create a new rule and push the new flow entry to this switch. The switch will then apply this rule to the packet. In the proactive case, flow entries are pushed to the switches ahead of time.

Each OpenFlow switch stores one or multiple *flow tables*, each of which contains a set of rules (a.k.a. flow entries). Flow tables form a *pipeline*, and flow entries are ordered according to *priorities*: A packet arriving at a switch is first checked by the rule of the *highest priority* in *table 0*: the fields of the data packet are compared with the match fields of that rule, and if they fit, some instructions (the actions) are executed; subsequently, lower priority rules are checked. Depending on the outcome of the table 0 processing, the packet may be sent to additional flow tables in the pipeline. Concretely, instructions can be used to define additional tables to be visited (*goto* instruction), to *modify* the set of to-be-applied actions (either by appending, deleting, or modifying actions), or *immediately apply* some actions to the packet. A meta-data field can be used to exchange information between tables. Part of the header can be inserted or removed from a packet via pushing and popping of labels and tags, e.g., of MPLS and Virtual Local Area Network (VLAN) fields.

In general, a packet can be matched against any of its header fields, and fields can be *wildcarded* and sometimes *bitmasked* (e.g., the meta-data field is maskable). If no rule matches, the packet is dropped. The use of multiple flow tables (compared to a single one) can simplify management and also improve performance.

Our robust traversal algorithms make use of *Group Tables*, and especially the Fast Failover (*FF*) concept introduced in OpenFlow 1.3. The group table consists of group entries, and each group entry contains one or more *action buckets*. For the group entries of the fast failover type, each bucket is associated with a specific port (or group), and only

the buckets associated with a live port (or group) can be used. As we will see, we can exploit this mechanism to control the forwarding behavior of the switch, depending on the liveness of ports. Another important group type for us are *select groups*: the select group provides the possibility to link an action bucket to a port (or a group), and define different selection types (e.g., *round-robin* or *all*).

2.2. Model

Robust inband network traversals are a fundamental building block and useful data plane network service for dependable SDNs. In this paper, we explore the feasibility and efficiency of implementing very robust inband network traversals: We say that a traversal mechanism is *maximally robust* if it tolerates an arbitrary number of link failures: a route between source and destination is always found, whenever this is possible, i.e., as long as the underlying network is connected.

In order to implement robust traversals in OpenFlow, we make a smart use of the OpenFlow local fast failover mechanism. Computing such local failover tables for robust traversals however is non-trivial, as the forwarding rules need to be allocated *before* failure(s) happen, and as the rules can depend on local liveness information only.

We will sometimes refer to the initial network (before the failures occur) by G_0, and to the remaining "sub-graph" after the failures by G_1. The problem of how to implement robust traversals and compute failover tables for G_0 *without knowing the actual failure scenario*, i.e., G_1, is an algorithmic one.

As we will show in this paper, it is possible to implement maximally robust traversals in today's OpenFlow protocol. However, different techniques come with different tradeoffs, in terms of *route length* (how long is the longest forwarding path in G_1, in terms of hops?), *rule and table complexity* (how many additional tables and rules per node are required to implement the robust traversal mechanism?) or *tag complexity* (how much header space is required for *tagging*?).

In the following, n will denote the total number of network switches (the same in G_0 and G_1), E will denote the set of links (total number: $m = |E|$), and Δ will refer to the maximal node degree in G_0. We will use subscripts, e.g., Δ_i, to denote the corresponding value for the given node v_i, and assume that node's ports are indexed from 1 to Δ_i.

3. Stateless robust inband network traversal

We start by observing that a simple yet robust network traversal can be implemented in OpenFlow, using (pseudo-) random walks. Random walks are attractive as they do not require to store or maintain information about the current network configuration, neither at the switches, nor in the packets.

The main advantage of our scheme, henceforth called *RWalk*, is that it can provide an alternative path for *any* link failure (as long as the remaining graph is connected). We can implement *RWalk* in standard OpenFlow using *fast failover* and *select group type* features. We set the bucket selection method to random. Similarly to the fast failover type FF, the select group type provides the possibility to link an action bucket to a port (or a group), and the bucket can be selected only if this port is live. When the packet that triggered the traversal arrives at a switch, the switch applies Gr 1 to it from the group table (cf. Table 1). In this group entry (of the select type) the switch will randomly select a bucket linked to a live switch port. Hence, at each switch the traversal packet is forwarded to a random live port.

Table 1
Random walker: Group table for switch i.

	Group table	
Gr ID	Gr type	Action buckets
Gr 1	SELECT	$\langle \text{Fwd j} \rangle_{j=1..\Delta_i}$

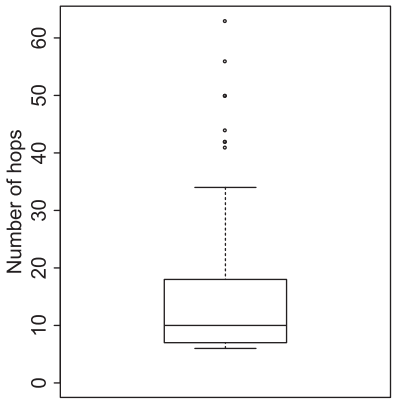

Fig. 1. Number of hops taken by packets using the random walk failover scheme, after the failure took place.

Note that only one group table entry is needed to implement the traversal mechanism. This is important as switch memory is a critical resource. However, the route lengths are relatively high and subject to a high variance, see the qualitative results in Fig. 1: the plot is based on a small IGEN topology (delaunay triangulated), consisting of 20 switches and hosts connected to them, each link having a delay of 2 ms. Using *pings*, 100 packets each time, we analyze the RTT between two hosts connected at both ends of the network, before and after the failure (one link is broken along the default path between the hosts) occurs. Before the failure, the shortest path is used between the hosts (and it consists of 5 hops).

This variance motivates the more deterministic failover schemes developed in this paper.

4. Predictable traversals with tagging

While attractive for its simplicity and robustness, the random network traversal scheme presented in the previous section can result in long and unpredictable traversal paths. Worse, different packets from the same microflow may take different paths, potentially introducing many packet reorderings, which can negatively affect the Transmission Control Protocol's (TCP's) throughput. In the following, we show that more efficient traversals can be implemented by storing network traversal information in packet tags.

In the following, we first present a simple Depth-First Search (DFS)-like traversal scheme called $ineff-DFS$. While it is deterministic, the disadvantage of $ineff-DFS$ is that it requires much header space (linear in n, the network size). We will later present a more header-space efficient algorithm, $eff-DFS$, requiring as little as$O(D \cdot \log n)$ space, where D is the diameter of the network: D is typically small.

The $ineff-DFS$ algorithm traverses the network in a depth-first fashion, implicitly constructing a spanning tree. Towards this goal, for each node v_i, a certain part of the packet header $pkt.v_i.par$ is reserved to store the parent $p(v_i)$ of v_i: the node from which the packet was received for the first time (indicated by the incoming port *in*). There is also a reserved place to store the $pkt.v_i.cur$ variable which represents the output port of the switch being currently traversed by the algorithm (Fig. 2).

The INEFF-DFS algorithm is summarized in Algorithm 1 in pseudo-code. Here, $pkt.start = 0$ denotes that the traversal was not started yet. (Depending on the use case, see later, traversals can be started explicitly upon request, or are triggered for an existing packet hitting a failed link.) For example, a switch can be programmed to match a specific "codeword" in a packet and then initiate the traversal. Alternatively, a switch can use regular routing rules until a failure of the outgoing port is detected, which will trigger the traversal.

Upon reception of a traversal-triggering packet, a node starts the

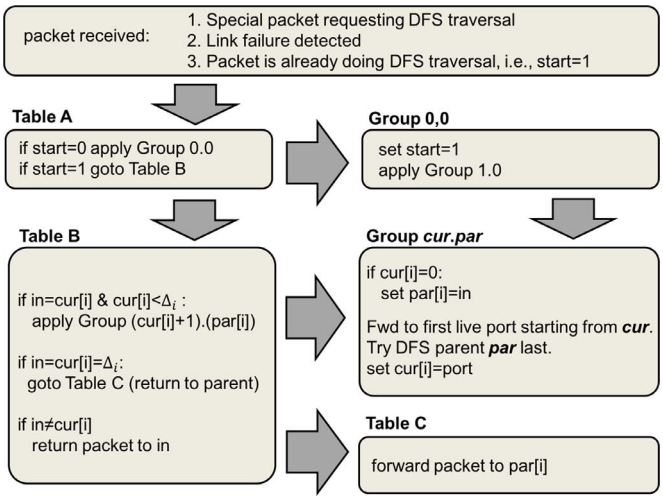

Fig. 2. Flowchart for the *ineff − DFS* algorithm. When a packet received by a switch, it decides whether to apply the *ineff − DFS* algorithm to it. The algorithm may be applied, for example, under one of the three conditions in the upper block. The *start* means *pkt.start* bit, *in* means the input port through which the packet was received, the *cur[i]* means $pkt.v_i.cur$ and *par[i]* is the $pkt.v_i.par$ field.

algorithm by setting *pkt.start* to 1 and trying to send the packet (i.e., initiate the traversal) from the next live port, beginning with port 1. A node tries to forward the packet to each neighbor, and only when all the neighbors (connected to live ports) were traversed, the node returns the packet to its parent.

The implementation of the *ineff − DFS* includes three flow tables and a group table (see Table 2). Table *A* checks whether the traversal has already started and if not, starts it. All groups in the group table are of type *Fast-Failover*, which means that each action bucket is coupled to a port (forwarding port in our case), and the first bucket with a working port is applied to the packet.

Each of the groups Gr *cur.par*, $cur \in [1, ..., \Delta_i]$, $par \in [0, ..., \Delta_i - 1]$, tries to find the next working port: it starts from port *cur*, skips the parent port *par*, and forwards back to the parent *par* only if all the previous ports have failed. Notice that the case when $pkt.v_i.par = \Delta_i$ is handled using the group Gr *cur*.0.

Once a packet was forwarded by the group Gr 1.0, its *pkt.start* bit will be set to 1, thus, when it is received by the next switch, it is passed to Table B; Table B will apply the corresponding group according to the $pkt.v_i.cur$ and $pkt.v_i.par$ fields. If $pkt.v_i.cur$ is 0, Table B will save the input port to the $pkt.v_i.par$ field. When all the neighbors are traversed, the packet is returned to the parent using Table C. The detailed OpenFlow tables for this algorithm appear in Table 2.

Notice that the traversal algorithm does not include the triggering mechanism since it depends on the specific application. For example, in the failover application, a switch can use an additional Group Table entry that will first try the default route for the given destination and if that port is failed, the next bucket in this entry will send the packet to the Gr 0.0 entry that actually starts the traversal algorithm. Another application may request a network traversal (e.g., for some data/statistics collection or connectivity check) by sending a packet with a specific predefined "codeword". Once the switch matches this codeword it can initiate the traversal by sending the packet to Table A (i.e., if *match(codeword)* then goto Table A).

4.1. More efficient traversals with tagging

Given this basic scheme, we now present a more efficient traversal. We propose a distributed, DFS-like traversal in which a packet is allowed to perform DFS up to a maximum depth *maxdist*. Assuming *maxdist* is at least the diameter of the network (i.e., in G_1, after the failures had occurred), such a traversal is guaranteed to traverse all the nodes.

The pseudocode of our algorithm is shown in Algorithm 2. The packet header includes the tag *T* consisting of *maxdist* many *cells*, as well as the global parameters: *pkt.start* and *pkt.dist*. Each cell of the tag *T* consists of three fields: $T[\cdot].ID$, the ID of a node which is currently using this cell, $T[\cdot].par$, the port connected the node's parent in the traversal, and $T[\cdot].cur$, the port connected to the node's successor which is currently being traversed. Note that only *maxdist* cells need to be allocated in this scheme.

The *pkt.start* field indicates whether the traversal procedure has started. It is set by the node which first received the packet triggering the traversal. This node is called the *root* of the traversal. The *pkt.dist* field keeps track of how far from the traversal's root the packet is located.

When a switch v_i receives a packet, it first inspects the *pkt.start* field to check whether the traversal has started, and if not, starts it: First, the switch initializes the global parameters *pkt.start* and *pkt.dist*. Then it indicates that it will use the first cell in the tag *T*, sets its port to the parent to 0 (the root has no parent), and sets $T[pkt.dist].cur$ and the output port to 1 (i.e., traversing successors starting with port 1).

Then we check whether the output port set by the switch is working or whether it is connected to its parent. In both cases the switch will try the next port (*while loop* on Line 25). If all potential ports (from *out* to Δ_i, the last port of switch v_i) have been tried, the switch has finished traversing all its successors, and the packet is returned to the parent. Before returning the packet to the parent, the switch removes itself from the tag *T*, by setting the $T[pkt.dist].ID$ field to 0, and decrementing the current path length by 1 (Lines 32,33). When a packet is sent to the next successor, the distance from the root increases (Line 35).

If a switch receives a packet which is already performing the traversal (i.e., *pkt.start* is 1), it checks whether the packet's distance from the root reached its maximum or whether the switch's ID is already present in *T* somewhere before the last cell $T[pkt.dist]$ (see Line 9). The first part of the condition implies that the packet had reached its maximum permitted distance from the root, and the second part implies that the switch is already in the current traversal path, but not at the last position. In both cases the packet is returned back to the port from which it has been received.

Next, a switch checks whether it received this packet for the first time, in which case it initializes the $T[pkt.dist]$ cell and uses its fields to save the port which is connected to the switch's parent (Line 13). Then, on Lines 16 and 17, the switch sets the output port to the next successor it needs to traverse, and saves this port in the $T[pkt.dist].cur$ field. In case the next port is larger than Δ_i, this implies that all the successors were traversed, and the packet is sent to the parent (Lines 19–24). Notice that when the root needs to return the packet to its parent (which does not exist), this implies that the traversal is finished and the packet should be dropped (Lines 21 and 31).

Implementing the *eff − DFS* traversal in OpenFlow is non-trivial. Overall, there are five flow tables and a group table. In order to make the group table representation more compact, we use the following notation to indicate a sequence of *k* action buckets: $\langle action(j) \rangle_{j=j_1,j_2,...,j_k}$. That is, in each bucket in the sequence, *j* is replaced by the next number in the sequence $j_1, j_1, ..., j_k$. The group IDs have the form *cur.par* where *cur* is the value of $T[pkt.dist].cur$ in the packet, i.e., the port which is currently traversed, and *par* is the value of $T[pkt.dist].par$. The detailed OpenFlow tables for this algorithm appear in Table 3.

Table A checks whether the traversal procedure has already started and if not, starts it by invoking group Gr 0.0 from the group table. Gr 0.0 initializes the required fields in the packet and invokes Gr 1.0 (i.e., chains it), which in turn will try to forward the packet to the next working port, starting from port 1. Tables B and C implement the condition on Line 9 in Algorithm 2. Table E implements Lines 19–24, i.e., sends the packet to its parent. Lines 29–33 are used to send the message to the parent, using the group tables in groups $\Delta_i + 1$. *par*: in this case, the port(s) preceding the parent port failed.

Input: current node: v_i, input port: *in*, traversal global parameter in packet: $pkt.start$, packet tag array: $\{pkt.v_j\}_{j\in[n]}$
Output: output port: *out*

```
1:  if pkt.start = 0 then
2:    pkt.start ← 1
3:    out ← 1
4:  else
5:    if pkt.v_i.cur = 0 then
6:      pkt.v_i.par ← in
7:    else if pkt.v_i.cur ≠ in then
8:      out ← in
9:      goto 20
10:   out ← pkt.v_i.cur + 1
11:   if out = Δ_i + 1 then
12:     out ← pkt.v_i.par
13:     goto 20
14: while out failed or out = pkt.v_i.par do
15:   out ← out + 1
16:   if out = Δ_i + 1 then
17:     out ← pkt.v_i.par
18:     goto 20
19: pkt.v_i.cur ← out
20: return out
```

Algorithm 1. Algorithm INEFF-DFS.

Table 2
ineff − DFS: Flow and group tables of switch i.

Flow Table A (Start)

Match	Instructions
$pkt.start$	
0	Gr 0.0
1	Table B

Flow Table C (Send-Parent)

Match	Instructions
$pkt.v_i.par$	
0	Drop
1	Fwd 1
2	Fwd 2
...	...
Δ_i	Fwd Δ_i

Group Table

Group ID	Type	Action buckets
0.0	FF	$\langle pkt.start \leftarrow 1, \text{Gr } 1.0\rangle$
1.0	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=1,2,...,\Delta_i}$
2.0	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=2,3,...,\Delta_i}$
...	...	...
$\Delta_i.0$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=\Delta_i}$
...	...	...
$1.x$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=1,2,...,x-1,x+1,...,\Delta_i,x}$
$2.x$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=2,3,...,x-1,x+1,...,\Delta_i,x}$
...	...	...
$\Delta_i.x$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=x}$
...	...	...
$1.\Delta_i - 1$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=1,2,...,\Delta_i-2,\Delta_i,\Delta_i-1}$
$2.\Delta_i - 1$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=2,3,...,\Delta_i-2,\Delta_i,\Delta_i-1}$
...	...	...
$\Delta_i.\Delta_i - 1$	FF	$\langle pkt.v_i.cur \leftarrow j, \text{Fwd } j\rangle_{j=\Delta_i-1}$

Flow Table B

Match			Instructions
in	$pkt.v_i.cur$	$pkt.v_i.par$	
1	0	*	$pkt.v_i.par \leftarrow 1$, Gr 1.1
2	0	*	$pkt.v_i.par \leftarrow 2$, Gr 1.2
...	...	...	...
Δ_i	0	*	$pkt.v_i.par \leftarrow \Delta_i$, Gr 1.$\Delta_i$
1	1	0	Gr 2.0
2	2	0	Gr 3.0
...	...	...	...
$\Delta_i - 1$	$\Delta_i - 1$	0	Gr Δ_i.0
Δ_i	Δ_i	0	Table C
...	...	...	...
1	1	x	Gr 2.x
2	2	x	Gr 3.x
...	...	...	...
$\Delta_i - 1$	$\Delta_i - 1$	x	Gr Δ_i.x
Δ_i	Δ_i	x	Table C
...	...	...	...
1	1	$\Delta_i - 1$	Gr 2.$\Delta_i - 1$
2	2	$\Delta_i - 1$	Gr 3.$\Delta_i - 1$
...	...	...	...
$\Delta_i - 2$	$\Delta_i - 2$	$\Delta_i - 1$	Gr Δ_i.$\Delta_i - 1$
Δ_i	Δ_i	$\Delta_i - 1$	Table C
1	1	Δ_i	Gr 2.0
2	2	Δ_i	Gr 3.0
...	...	...	...
$\Delta_i - 2$	$\Delta_i - 2$	Δ_i	Gr $\Delta_i - 1$.0
$\Delta_i - 1$	$\Delta_i - 1$	Δ_i	Table C
*	*	*	Fwd in

Each of the groups Gr *cur.par*, $cur \in [1, ..., \Delta_i]$, $par \in [0, ..., \Delta_i - 1]$, tries to find the next working port: it starts from the port *cur* and skips the parent port *par*. If all the ports have failed, the additional bucket Gr $\Delta_i + 1$. *par* will forward the packet back to the parent *par*. Notice that in the case the parent is 0, there are no additional buckets: the root of the traversal does not have a parent.

4.2. Alternative: Iterative Depth-DFS

It is easy to extend our approach to header-space efficient "breadth-first" traversals, or more specifically: Iterative Depth-DFS (IDDFS) traversals. The basic idea is to increase *maxdist* one-by-one. We start the DFS algorithm with $maxdist = 1$: only the 1-hop neighborhood of the traversal root will be explored. Once the packet returned back to the root, instead of discarding it (the default behavior of our DFS scheme), the root will increment *maxdist* and restart the DFS traversal. This time the 2-hops neighborhood of the root will be traversed. In the same manner, we continue to increase *maxdist* up to the diameter of the network G_1 (the resulting network after the failures). Eventually, the whole network will be traversed while the nodes are discovered in the BFS-like order.

5. Short routes with state

We have shown that the possibility to store state in the packet headers, using tagging, can be useful to overcome some of the downsides of the random walk scheme. In this section, we further explore the possibility of leveraging state information for the traversals. In particular, we show that, maybe suprisingly, using packet tagging is not the only way state can be introduced in OpenFlow traversals. In fact, it is possible to implement simple state machines on the switches, using the standard OpenFlow protocol, and we will refer to the corresponding state as *inband registers*. Moreover, we present an interesting and novel mechanism to store state in the hosts attached to the network, in a completely transparent manner, using MAC addresses to encode paths. We will refer to this state as *host-based registers*. For example, such state can be exploited to store shortest paths, which have recently been discovered during network traversals.

5.1. In-band registers

Our OpenFlow in-band register implements multiple (small) counters which are stored in the switch and support fetch-and-increment operations while processing different packets. In the following, we will use the terms register and counter interchangeably. The in-band register can be read and updated while packets are processed in the OpenFlow pipeline. In-band registers can be implemented using a group table with $\log_2 k$ groups of the *round-robin bucket selection* policy (an optional feature of OpenFlow 1.3), and each group contains 2 action buckets (see Table 4). Each group represents a bit of the counter, i.e., for a counter that counts from 0 to $k - 1$ we need $\log_2 k$ bits (groups). Fetching a counter is done by applying all the $\log k$ groups Gr x, and the value of the counter will be written to the packet's tag *pkt.cnt.x* ($x \in [1, ..., \log k]$).

In order to preserve the value of the counter after the fetch, the following steps need to be implemented. After fetching the counter, all its bits will be flipped (due to the round-robin type of the groups), thus, we need now to flip them one more time in order return the counter to its original value. This can be done by applying all the groups Gr x ($x \in [1, ..., \log k]$) once again.

Let us now describe how to set a new value to the in-band register. Assume that the fetched counter's value is stored bitwise in the packet's fields *pkt.cnt.x* and the desired counter's value is stored in the metadata *new.x*, where $x \in [1, ..., \log k]$. Now we create a table that matches all possible combinations of *pkt.cnt.x* and *new.x* (this will require k^2 entries). The action will consist of applying groups Gr y where $y \in \{i | pkt.\ cnt.\ i = new.\ i\}$. The latter is true since the first fetch already flipped the value of the bit y and thus we need another flip to make the counter's bit y be equal to *new.y*.

This implementation uses $2\log k$ bits for matching in the flow tables (flow tables are used to preserve or set counter values), and requires up to $\log k$ actions per entry.

Note that the register implementation (fetching and setting) requires to access the group table twice, interleavingly with flow tables;

Input: current node: v_i, input port: *in*, traversal global parameters in packet: ($pkt.start$, $pkt.dist$), preallocated packet tag array of length $maxdist$: T
Output: output port: *out*

1: **if** $pkt.start = 0$ **then**
2: $pkt.start \leftarrow 1$
3: $pkt.dist \leftarrow 0$
4: $T[pkt.dist].ID \leftarrow i$
5: $T[pkt.dist].par \leftarrow 0$
6: $T[pkt.dist].cur \leftarrow 1$
7: $out \leftarrow 1$
8: **else**
9: **if** $pkt.dist = maxdist$ **or** $\exists k < pkt.dist$ for which $T[k].ID = i$ **then**
10: $pkt.dist \leftarrow pkt.dist - 1$
11: $out \leftarrow in$
12: **return** *out*
13: **if** $T[pkt.dist].cur = 0$ **then**
14: $T[pkt.dist].par \leftarrow in$
15: $T[pkt.dist].ID \leftarrow i$
16: $out \leftarrow T[pkt.dist].cur + 1$
17: $T[pkt.dist].cur \leftarrow out$
18: **if** $out = \Delta_i + 1$ **then**
19: $out \leftarrow T[dist].par$
20: **if** $out = 0$ **then**
21: Drop packet and exit
22: $T[pkt.dist].cur \leftarrow 0$
23: $pkt.dist \leftarrow pkt.dist - 1$
24: **return** *out*
25: **while** *out* failed **or** $out = T[pkt.dist].par$ **do**
26: $out \leftarrow out + 1$
27: $T[pkt.dist].cur \leftarrow out$
28: **if** $out = \Delta_i + 1$ **then**
29: $out \leftarrow T[pkt.dist].par$
30: **if** $out = 0$ **then**
31: Drop packet and exit
32: $T[pkt.dist].cur \leftarrow 0$
33: $pkt.dist \leftarrow pkt.dist - 1$
34: **return** *out*
35: $pkt.dist \leftarrow pkt.dist + 1$
36: **return** *out*

Algorithm 2. Algorithm $eff - DFS$.

Table 3
eff − DFS: Flow and Group tables of switch i.

Flow Table A (Start)

Match $pkt.start$	Instructions
0	Gr 0.0
1	Table B

Flow Table C

Match $T[0].ID$	Match $T[1].ID$	Match $\cdots$	Match $T[maxdist-1].ID$	Instructions
i	$*$	$\cdots$	$*$	$pkt.dist \leftarrow pkt.dist - 1$, Fwd in
$*$	i	$\cdots$	$*$	$pkt.dist \leftarrow pkt.dist - 1$, Fwd in
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$pkt.dist \leftarrow pkt.dist - 1$, Fwd in
$*$	$*$	$\cdots$	i	$pkt.dist \leftarrow pkt.dist - 1$, Fwd in
$*$	$*$	$\cdots$	$*$	Table D

Flow Table B

Match $pkt.dist$	Instructions
$maxdist$	$pkt.dist \leftarrow pkt.dist - 1$,Fwd in
$*$	Table C

Flow Table E (Send-Parent)

Match $pkt.v_i.par$	Instructions
0	Drop
1	$T[pkt.dist].cur \leftarrow 0, pkt.dist \leftarrow pkt.dist - 1$, Fwd 1
2	$T[pkt.dist].cur \leftarrow 0, pkt.dist \leftarrow pkt.dist - 1$, Fwd 2
$\cdots$	$\cdots$
Δ_i	$T[pkt.dist].cur \leftarrow 0, pkt.dist \leftarrow pkt.dist - 1$, Fwd Δ_i

Flow Table D

Match in	Match $T[pkt.dist].cur$	Match $T[pkt.dist].par$	Instructions
1	0	$*$	$T[pkt.dist].par \leftarrow 1$, $T[pkt.dist].ID \rightarrow i$, Gr 1.1
2	0	$*$	$T[pkt.dist].par \leftarrow 2$, $T[pkt.dist].ID \rightarrow i$, Gr 1.2
$\cdots$	$\cdots$	$\cdots$	$\cdots$
Δ_i	0	$*$	$T[pkt.dist].par \leftarrow \Delta_i$, $T[pkt.dist].ID \rightarrow i$, Gr 1.$\Delta_i$
1	1	0	Gr 2.0
2	2	0	Gr 3.0
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\Delta_i - 1$	$\Delta_i - 1$	0	Gr Δ_i.0
Δ_i	Δ_i	0	Table E
$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	1	x	Gr 2.x
2	2	x	Gr 3.x
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\Delta_i - 1$	$\Delta_i - 1$	x	Gr Δ_i.x
Δ_i	Δ_i	x	Table E
$\cdots$	$\cdots$	$\cdots$	$\cdots$
1	1	Δ_i	Gr 2.Δ_i
2	2	Δ_i	Gr 3.Δ_i
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\Delta_i - 2$	$\Delta_i - 2$	Δ_i	Gr $\Delta_i - 1$.Δ_i
$\Delta_i - 1$	$\Delta_i - 1$	Δ_i	Table E

Group Table

Gr ID ($cur.par$)	Type	Action buckets
0.0	FF	$\langle pkt.start \leftarrow 1, pkt.dist \leftarrow 0, T[pkt.dist].ID \leftarrow i, T[pkt.dist].par \leftarrow 0$, Gr 1.0$\rangle$
1.0	FF	$\langle T[pkt.dist].cur \leftarrow j, pkt.dist \leftarrow pkt.dist + 1$, Fwd $j\rangle_{j=1,2,\ldots,\Delta_i}$
2.0	FF	$\langle T[pkt.dist].cur \leftarrow j, pkt.dist \leftarrow pkt.dist + 1$, Fwd $j\rangle_{j=2,3,\ldots,\Delta_i}$
$\cdots$	$\cdots$	$\cdots$
Δ_i.0	FF	$\langle T[pkt.dist].cur \leftarrow \Delta_i, pkt.dist \leftarrow pkt.dist + 1$, Fwd $\Delta_i\rangle$
$\cdots$	$\cdots$	$\cdots$
1.x	FF	$\langle T[pkt.dist].cur \leftarrow j, pkt.dist \leftarrow pkt.dist + 1$, Fwd $j\rangle_{j=1,2,\ldots,x-1,x+1,\ldots,\Delta_i}$ $\langle$Gr $\Delta_i + 1.x\rangle$
2.x	FF	$\langle T[pkt.dist].cur \leftarrow j, pkt.dist \leftarrow pkt.dist + 1$, Fwd $j\rangle_{j=2,3,\ldots,x-1,x+1,\ldots,\Delta_i}$ $\langle$Gr $\Delta_i + 1.x\rangle$
$\cdots$	$\cdots$	$\cdots$
$\Delta_i.x$	FF	$\langle T[pkt.dist].cur \leftarrow \Delta_i, pkt.dist \leftarrow pkt.dist + 1$, Fwd $\Delta_i\rangle$ $\langle$Gr $\Delta_i + 1.x\rangle$
$\Delta_i + 1$.x	FF	$\langle T[pkt.dist].cur \leftarrow 0, pkt.dist \leftarrow pkt.dist - 1$, Fwd $x\rangle$
$\cdots$	$\cdots$	$\cdots$
1.Δ_i	FF	$\langle T[pkt.dist].cur \leftarrow j, pkt.dist \leftarrow pkt.dist + 1$, Fwd $j\rangle_{j=1,2,\ldots,\Delta_i-1}$ $\langle$Gr $\Delta_i + 1.\Delta_i\rangle$
2.Δ_i	FF	$\langle T[pkt.dist].cur \leftarrow j, pkt.dist \leftarrow pkt.dist + 1$, Fwd $j\rangle_{j=2,3,\ldots,\Delta_i-1}$ $\langle$Gr $\Delta_i + 1.\Delta_i\rangle$
$\cdots$	$\cdots$	$\cdots$
$\Delta_i.\Delta_i$	FF	$\langle T[pkt.dist].cur \leftarrow \Delta_i, pkt.dist \leftarrow pkt.dist + 1$, Fwd $\Delta_i\rangle$ $\langle$Gr $\Delta_i + 1.\Delta_i\rangle$
$\Delta_i + 1.\Delta_i$	FF	$\langle T[pkt.dist].cur \leftarrow 0, pkt.dist \leftarrow pkt.dist - 1$, Fwd $\Delta_i\rangle$

however common OpenFlow switches support group table access only at the end of the flow tables pipeline. We can overcome this limitation by making each packet to be processed twice in the switch. This can be implemented by physically interconnecting two of the switch ports (i.e., creating a physical loopback), or by asking a neighboring switch to return the packet back if a certain bit in the packet is set. Moreover, we expect that future OpenFlow switches would support more flexible group table accesses and more field actions.

Table 4
In-band register: Group Table that implements an in-band register which counts from 0 to $k-1$.

Gr id	Type	Action buckets
1	RR	$\langle pkt.cnt.1 \leftarrow 0\rangle$, $\langle pkt.cnt.1 \leftarrow 1\rangle$
2	RR	$\langle pkt.cnt.2 \leftarrow 0\rangle$, $\langle pkt.cnt.2 \leftarrow 1\rangle$
...	...	...
$\log k$	RR	$\langle pkt.cnt.\log k \leftarrow 0\rangle$, $\langle pkt.cnt.\log k \leftarrow 1\rangle$

5.2. Host-based registers

We next present an interesting alternative scheme to store state in the network: host-based registers. Host-based registers leverage the hosts' ARP caches to store per-destination information (e.g., efficient routing paths learned during a traversal), encoded in terms of MAC addresses. The solution is completely transparent to the hosts (and IP layer).

The registers are written by sending the host a (gratuitous) ARP reply message with specific (destination) host as source-IP and the register value as source-MAC (src_mac). Reading the registers is automatically performed for every transmitted packet, as the register value is set as destination-MAC (dst_mac), and can be used by the OpenFlow switches.

Next, we describe how these registers can be used to store and follow paths. First we show how short paths can be aggregated and packed into an ARP reply message. Later we show how a path encoded in the dst_macfield can be followed. For simplicity, we assume paths of maximum length 12 for 16-port nodes, representable with 48 bits.

5.2.1. Aggregating paths for host-based registers

In order to aggregate a path for host-based registers, every node along the path appends the in-port to the current path array, thereby allowing backward replay of the path. As described in Table 5, the appending operation is performed according to the current hop number stored as VLAN tag: it includes copying the src_macto a metadata field (using the OpenFlow 1.5 copy field action), setting the 4 bits at offset *hopnumber* to the 4 bits encoding of the in-port, and copying the metadata back to src_mac.

Table 5
Aggregating paths for host-based registers. VLAN tag indicates the hop number and thereby the offset to append the in-port inside the src_mac. Writing to $metadat[i]$ is actually setting the four bits at offset $4i$, Assuming a 4bit representation of in-port.

Match		Instructions
pkt.vlan	*in*	
0	1	$metadata \leftarrow pkt.srcmac$, $metadata[0] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow 1$
1	1	$metadata \leftarrow pkt.srcmac$, $metadata[1] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow 2$
...	...	...
D	1	$metadata \leftarrow pkt.srcmac$, $metadata[D] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow D+1$
0	2	$metadata \leftarrow pkt.srcmac$, $metadata[0] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow 1$
1	2	$metadata \leftarrow pkt.srcmac$, $metadata[1] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow 2$
...	...	...
D	2	$metadata \leftarrow pkt.srcmac$, $metadata[D] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow D+1$
...	...	...
0	Δ_i	$metadata \leftarrow pkt.srcmac$, $metadata[0] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow 1$
1	Δ_i	$metadata \leftarrow pkt.srcmac$, $metadata[1] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow 2$
...	...	...
D	Δ_i	$metadata \leftarrow pkt.srcmac$, $metadata[D] \leftarrow 1$, $pkt.srcmac \leftarrow metadata$, $pkt.vlan \leftarrow D+1$

Table 6
Following paths stored in host-based registers. VLAN tag indicates the hop number and thereby the offset to match inside the dst_macwhich indicates the output port.

Match		Instructions
pkt.vlan	*pkt. dstmac*	
0	$*^{44}\cdot 1$	$pkt.vlan \leftarrow 1$, forward 1
1	$*^{40}\cdot 1\cdot *^{4}$	$pkt.vlan \leftarrow 2$, forward 1
2	$*^{36}\cdot 1\cdot *^{8}$	$pkt.vlan \leftarrow 3$, forward 1
...	...	...
$D-1$	$1\cdot *^{44}$	$pkt.vlan \leftarrow 0$, forward 1
0	$*^{44}\cdot 2$	$pkt.vlan \leftarrow 1$, forward 2
1	$*^{40}\cdot 2\cdot *^{4}$	$pkt.vlan \leftarrow 2$, forward 2
2	$*^{36}\cdot 2\cdot *^{8}$	$pkt.vlan \leftarrow 3$, forward 2
...	...	...
$D-1$	$2\cdot *^{44}$	$pkt.vlan \leftarrow 0$, forward 2
...	...	...
0	$*^{44}\cdot \Delta_i$	$pkt.vlan \leftarrow 1$, forward Δ_i
1	$*^{40}\cdot \Delta_i\cdot *^{4}$	$pkt.vlan \leftarrow 2$, forward Δ_i
2	$*^{36}\cdot \Delta_i\cdot *^{8}$	$pkt.vlan \leftarrow 3$, forward Δ_i
...	...	...
$D-1$	$\Delta_i\cdot *^{44}$	$pkt.vlan \leftarrow 0$, forward Δ_i

5.2.2. Following paths stored in host-based registers

When a host sends a packet to some destination host (represented by its IP), the packet holds the path as the dst_mac. In order to follow the path, each node along the path extracts the current port and forwards to this port. As described in Table 6, the extraction of each hop is performed backward (reversed to the aggregation order), and the VLAN tag is used to indicate the current offset in the dst_macto consider as next hop port.

6. Applications

Our traversal schemes come with interesting applications. In the following, we present a robust routing application and a connectivity testing application. We discuss different implementation variants, based on packet tagging as well as based on the registers introduced in the previous section.

6.1. Failover routing

In legacy networks, distributed protocols such as the Layer-2 Spanning Tree Protocol (STP) are used to update the configuration of each network element (switch/router) in response to network changes, for example link failures. Typically, a link failure is noticed by an incident network element, which then propagates the information to neighbors, etc. As such, these protocols ensure that hosts stay connected, as long as the physical network is connected.

In SDNs, events are dispatched to a logically centralized controller, which seeks to maintain a global view of the network, and pushes new forwarding rules to all switches, in response to network changes. In addition, SDN switches can be configured with inband failover rules which react to local failures (without controller intervention).

In the following, we discuss how to use traversals to implement robust routing in the presence of (possibly many and simultaneous) failures. The inband approach is relevant as it can also operate in cases where the control plane is unavailable. While our failover scheme could in principle be applied to the *entire traffic*, in practice, it can make sense to reroute only the flows of some critical applications. Indeed, the traffic engineering flexibilities of OpenFlow and the possibility to match not only Layer-2 but also Layer-3 and Layer-4 header fields, as well as the possibility to tag packets at ingress ports (e.g., using an intrusion detection system), allows for a fine-grained failover, where uncritical

flows or very heavy flows are dropped when links fail, while critical and light flows are rerouted. For example, it can make sense to prioritize (and reroute) only flows containing control messages, or flows providing connectivity between the switch and any controller. Time uncritical and/or large flows such as flows for Dropbox synchronization, do not necessarily have to be rerouted inband.

6.1.1. Using plain traversals

Given our inband traversals, it is straight-forward to implement failover routing applications: whenever a packet encounters a failed link along its path, it is diverted to an alternative link (according to Random Walk, DFS and BFS), eventually reaching the destination. While a random walk approach is simple and does not require any resources of the packet, the DFS and BFS schemes require some header space but also result in shorter and deterministic route lengths.

6.1.2. Improved version: Spanning trees with inband registers

With the concept of inband registers, we can construct a spanning tree. Within a spanning tree, simple forwarding rules can help any packet to traverse the network without using additional (packet) space. The main idea is to use inband registers to save the in-port to out-port mapping according to the one DFS traversal while performing a single DFS traversal. Then any packet can follow this mapping and perform a DFS traversal without storing any state in the packet.

6.1.3. Shortest paths with inband and host registers

Using inband registers, we can also implement inband shortest path routing. This routing can be established upon request, or triggered by an event (such as a path failure), and once computed, it can assist every packet. The main idea is to use an inband register to save the parent port ID which emerges during a network traversal, using the IDDFS traversal algorithm. Each traversal initiated from some root node i will store the parent port IDs in every node. These parent ports are later used to route packets for destination i. To support routing to every node, we will initiate a traversal from each node, keeping unique parent IDs per destination, in every node. Following the parent ports of a traversal, we will route a packet along the path to the root. Since the resulting paths were created by the IDDFS traversal algorithm, they are the shortest paths.

We can achieve a similar result also by using per-destination host registers. When detecting a shortest path with IDDFS traversal, we can maintain the current path to the root in the packet and store it in the registers of the hosts attached to the current node (where the destination is the root), rather than in the node's inband registers. Later, when one of the hosts tries to reach that root, the host register value is read and attached to the packets; hence, the switches can route accordingly.

In Table 7, we compare the performance of the above failover schemes. We can see the tradeoff between the maximum number of hops and the amount of the state used. While the Random Walk approach uses no state, the traversal paths can be very long. We can also see the benefit of the $eff-DFS$ over the $ineff-DFS$ in terms of a packet header space used. The Spanning Tree and the Shortest Paths approaches do not require any packet header space once the in-band registered were initialized. For example, in a small Clos datacenter [18], we may have $\Delta = 48$ port switches. Thus, in our implementation of *RWalk*, we will have one group table with one entry. $ineff-DFS$ requires three flow tables, one with two entries, one with $\Delta = 48$ entries and one with one with Δ^2 entries, as well as one gorup table with Δ^2 entries. $eff-DFS$ requires five flow tables, two with two entries, two with Δ entries, and one with Δ^2 entries; moreover, we need a group table with Δ^2 entries.

Table 7
Comparison between the failover routing methods. The number of nodes is denoted by n, the number of links by m, the degree by Δ and the diameter by D. We compare the maximum number of hops a packet will travel until the destination will be reached; size of the header space used in the packet; per-switch space used by the in-band registers.

Technique	#hops	Packet space	Per switch space
Random walk	$O(n^3)$	0	0
$ineff-DFS$	$O(m)$	$O(n\log\Delta)$	0
$eff-DFS$	$O(m)$	$O(D\log n)$	0
$eff-IDDFS$	$O(Dm)$	$O(D\log n)$	0
Spanning tree	$O(n)$	0	$\Delta^2\log\Delta$
Shortest paths	$O(D)$	0	$n\log n$

6.2. Connectivity queries

A query mechanism to check the connectivity status between hosts is crucial for dependable networked systems, e.g., for emergency or disaster handling, but for example also in the context of distributed SDN control applications in datacenters [19].

6.2.1. One time queries

Our traversal algorithms (Random Walk, DFS, BFS) not only provide a guaranteed message delivery as long as the underlying network is physically connected, but they can also be used to indicate disconnectivity. A one time connectivity query can be implemented in the following way: A traversal is used to search the destination, up to a certain hop limit. If the destination is reached, another traversal is used to inform the source. If the hop limit is reached before reaching the destination, a second traversal is issued to inform the source. Depending on the traversal technique, it may be possible to keep the path information during the first traversal, such that the response can be sent backward, without the need for a second traversal.

By using per-destination host registers, we can make the reply to a query follow the detected path (in case it is short enough), or at least to assist its traversal, without any modification to the queried host.

6.2.2. Connectivity service

Traversals can be also used to implement connectivity service for hosts [19], in the following way. Each host, upon request or by configuration, actively initiates new traversals, multicasting its liveness status. All interested subscribers can be informed, in a publish/subscribe manner, using rules on their attached OpenFlow switch. Traversals are numbered using increasing IDs, to avoid multiple visits of the same traversal (e.g., in case of network loops or the random walk based traversal).

Similar to failover routing, inband registers can vastly improve the performance of the queries. For example, in the connectivity service, each switch, before it forwards the current traversal to the subscribers, can verify whether the current traversal uses a larger ID than the previous one. By performing the connectivity test from all switches, switches keep track of the last traversal IDs of each other, speeding up one time queries further: it is then sufficient to check the attached switch for a change in the relevant traversal IDs.

Note that allowing ICMP ping packets to be sent using traversals, we turn them into (one time) physical network queries. Compared to a distributed application based on ICMP probing, requiring n^2 packets, our service uses a linear number of packets only.

7. Related work

While the benefits of more centralized network architectures enabled by protocols like OpenFlow [20] and segment routing [21] have attracting much interest from both researchers and operators, the question of which functionality can and should remain in the data plane is subject to ongoing discussions.

There exist several empirical studies showing that link failures, even simultaneous ones, do occur in different networks [22,23], including wide-area [24] and datacenter networks [25]. For example, it has been reported that in a wide area network, a link fails every 30 minutes on

average [26].

Commercial networks today usually rely on routing schemes such as OSPF, IS-IS, and MPLS reroute traffic, which however do not come with formal guarantees under multiple failures. Accordingly, backbone networks are usually largely over-provisioned.

Moreover, it is well-known that reactions to even a single link failure can be slow in traditional networks based on a distributed control plane: in the order of tens of milliseconds or even seconds, depending on the network; much higher than packet forwarding intervals (in the order of μsec in Gbps networks) [27].

More systematically, existing robust routing mechanisms can be classified according to whether a single link/node failure [28–30] or multiple ones can be tolerated [31]. Alternatively, they can be classified into static and dynamic ones. Dynamic tables and using link reversals [3,32,33] can yield very robust networks, but require dynamic tables. Finally, one can also classify existing mechanisms as basic routing schemes [34,35], schemes exploiting packet-header rewriting [36,37], and (randomized) routing with packet-duplication [38]. While packet-header rewriting can improve resiliency, it can be problematic in practice, especially under multiple failures, as header space (and rule space) is limited.

The works closest to ours are by Feigenbaum [27], Chiesa et al. [38,39], Stephens et al. [37,40], and Borokhovich and Schmid [41]. Feigenbaum [27] introduces the notion of *perfect resilience*, resilience to arbitrary failures. Chiesa et al. [39] focus on "scalable" static failover schemes that rely only on the destination address, the packets incoming link, and the set of nonfailed links incident to the router. The authors find that per-incoming link destination-based forwarding tables are a necessity as destination-based routing alone is unable to achieve robustness against even a single link failure, and, moreover, entails computationally hard challenges. In [38], Chiesa et al. study failover routing in different models with and without packet marking but also with duplication, and present several new algorithms, in contrast to our paper, also considering the stretch but also deriving impossiblity results.

Stephens et al. [37,40] present a new forwarding table compression algorithm called Plinko [37,40], which however cannot provide robustness guarantees in all possible failure scenarios. Chiesa et al. [42] recently proposed a systematic algorithmic study of the resiliency of immediate failover in a variety of models.

More generally, from an algorithmic perspective, our work is related to the field of graph exploration, see, e.g. [43–46] for an overview. In particular, the deterministic analogue of our stateless random walk is known as the rotor router (sometimes also called Propp machine or Eulerian walkers), and has been studied in various contexts before [47,48].

*Bibliographic note.*First ideas leading to this paper have been presented at the SIGCOMM HotSDN 2014 [49] workshop as well as at the HotNets 2014 workshop [50]. In [49], we presented two graph traversals: one based on Depth-First Search (DFS) and another one based on Breadth-First Search (BFS) resp. IDDFS. These algorithms provide guaranteed connectivity, however they require a non-trivial amount of header space for tagging (linear in n). In the current full version of the paper, we present more efficient failover schemes and a more comprehensive study of traversal mechanisms, also showing that stateful mechanisms are possible, and discussing different applications. In [50], we observed the possibility to implement more stateful network functions using existing OpenFlow versions; the latter has also been observed independently by Bianchi et al. [17].

8. Conclusion

According to a recent Communications of the ACM article [2], the possibility to render failover more predictable was one of the key reasons for Google to move to an SDN solution. An attractive solution to implement a fast failover relies on inband mechanisms: a local fast failover can serve as a first line of defense, before the controller subsequently can rigorously optimize the route allocation. However, not much is known today about how to exploit such inband mechanisms, nor what are their limitations.

This paper showed the feasibility of very robust inband network traversals in OpenFlow networks. Such traversals can form the basis for critical services of dependable networks, such as robust routing or connectivity testing. We have demonstrated traversals in three basic models: a simple stateless model (using a random walk scheme), a model with packet tagging (using spanning tree search) and a model with state at switches or in hosts (e.g., storing recently discovered shortest paths).

Our approach is optimized toward a worst-case: our network traversals ensure packet delivery even under a large number of link failures, as long as the source and the destination are still physically connected. In practice, such a rigorous approach may only be worthwhile for a subset of very critical flows (e.g., control flows between switches and controllers): non-critical flows can simply be dropped.

We hope that our paper can inform the community of what degree of robustness can be achieved in today's OpenFlow protocol and what tradeoffs exist, and also nourish the ongoing discussion of what can and should be implemented in the data plane.

Together with this paper, we will make available the source code of a canonical robust traversal algorithm presented in this paper.

Acknowledgments

The authors would like to thank Leszek Antoni Gasieniec for several discussions. Liron Schiff is supported by the European Research Council (ERC) Starting Grant no. 259085 and by the Israel Science Foundation Grant no. 1386/11. Stefan Schmid is supported by the Danish Villum Foundation project *ReNet*.

References

[1] N. Feamster, J. Rexford, E. Zegura, The road to SDN, Queue 11 (12) (2013).
[2] A. Vahdat, D. Clark, J. Rexford, A purpose-built global network: Google's move to SDN, Queue 13 (8) (2015) 100.
[3] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, S. Shenker, Ensuring connectivity via data plane mechanisms, Proceedings of the Tenth USENIX Conference on Networked Systems Design and Implementation, (2013), pp. 113–126.
[4] N. Solomon, Y. Francis, L. Eitan, Floodlight openflow ddos, (2013) www.Slideshare.net.
[5] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, R. Kompella, Towards an Elastic Distributed SDN Controller, Proceedings of the THotSDN, (2013).
[6] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, S. Shenker, Onix: a distributed control platform for large-scale production networks, Proceedings of the Ninth USENIX Conference on Operating Systems Design and Implementation (OSDI), (2010).
[7] S.H. Yeganeh, Y. Ganjali, Beehive: towards a simple abstraction for scalable software-defined networking, Proceedings of the HotNets, (2014).
[8] M. Canini, P. Kuznetsov, D. Levin, S. Schmid, A distributed and robust SDN Control plane for transactional network updates, Proceedings of the IEEE INFOCOM, (2015).
[9] N. Katta, H. Zhang, M. Freedman, J. Rexford, Ravana: controller fault-tolerance in software-defined networking, Proceedings of the ACM SOSR, (2015).
[10] A. Akella, A. Krishnamurthy, A highly available software defined fabric, Proceedings of the HotNets, (2014).
[11] L. Schiff, S. Schmid, M. Canini, Medieval: towards a self-stabilizing, plug & play, in-band SDN control network, Proceedings of the ACM Sigcomm Symposium on SDN Research (SOSR), (2015).
[12] anonymous, Robust routing in openflow, (2014) http://tinyurl.com/z5lzdga.
[13] A.R. Curtis, et al., Devoflow: scaling flow management for high-performance networks, Proceedings of the SIGCOMM, (2011), pp. 254–265.
[14] S. Zhang, S. Malik, S. Narain, L. Vanbever, In-band update for network routing policy migration, Proceedings of the International Conference on Network Protocols (ICNP), (2014), pp. 356–361.
[15] O. Tilmans, S. Vissicchio, IGP-as-a-backup for robust SDN networks, Proceedings of the Tenth International Conference on Network and Service Management (CNSM), (2014).
[16] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: programming protocol-independent packet processors, SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95.
[17] G. Bianchi, M. Bonola, A. Capone, C. Cascone, Openstate: programming platform-

independent stateful openflow applications inside the switch, SIGCOMM Comput. Commun. Rev. (CCR) 44 (2) (2014) 44–51.
[18] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, ACM SIGCOMM Comput. Commun. Rev. 38 (4) (2008) 63–74.
[19] J.B. Leners, T. Gupta, M.K. Aguilera, M. Walfish, Taming uncertainty in distributed systems with help from the network, Proceedings of the Tenth EuroSys, (2015).
[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.
[21] C. Filsfils, N.K. Nainar, C. Pignataro, J.C. Cardona, P. Francois, The segment routing architecture, Global Communications Conference (GLOBECOM), IEEE, 2015, pp. 1–6.
[22] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, C. Diot, Characterization of failures in an ip backbone, Proceedings of the IEEE INFOCOM, 4 (2004), pp. 2307–2317.
[23] D. Turner, K. Levchenko, A.C. Snoeren, S. Savage, California fault lines: understanding the causes and impact of network failures, ACM SIGCOMM Comput. Commun. Rev. 41 (4) (2011) 315–326.
[24] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, R. Wattenhofer, Achieving High Utilization with Software-Driven WAN, Proceedings of the SIGCOMM, (2013).
[25] P. Gill, N. Jain, N. Nagappan, Understanding network failures in data centers: measurement, analysis, and implications, 41 (2011), pp. 350–361.
[26] H.H. Liu, S. Kandula, R. Mahajan, M. Zhang, D. Gelernter, Traffic engineering with forward fault correction, 44 (2014), pp. 527–538.
[27] J. Feigenbaum, Ba: On the resilience of routing tables, Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), (2012), pp. 237–238.
[28] G. Enyedi, G. Rétvári, T. Cinkler, A novel loop-free ip fast reroute algorithm, Dependable and Adaptable Networks and Services, Springer, 2007, pp. 111–119.
[29] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, C.-N. Chuah, Fast local rerouting for handling transient link failures, IEEE/ACM Trans. Netw. (ToN) 15 (2) (2007) 359–372.
[30] J. Wang, S. Nelakuditi, IP fast reroute with failure inferencing, Proceedings of the SIGCOMM Workshop on Internet Network Management, (2007), pp. 268–273.
[31] T. Elhourani, A. Gopalan, S. Ramasubramanian, IP fast rerouting for multi-link failures, Proceedings of the IEEE INFOCOM, IEEE, 2014, pp. 2148–2156.
[32] E. Gafni, D. Bertsekas, Distributed algorithms for generating loop-free routes in networks with frequently changing topology, IEEE Trans. Commun. 29 (1) (1981) 11–18.
[33] J. Liu, B. Yan, S. Shenker, M. Schapira, Data-driven network connectivity, Proceedings of the HotNets, (2011), pp. 8:1–8:6.
[34] A. Atlas, et al., U-turn alternates for IP/LDP fast-reroute, IETF Internet Draft (2006).
[35] B. Yang, J. Liu, S. Shenker, J. Li, K. Zheng, Keep forwarding: towards k-link failure resilient routing, Proceedings of the IEEE INFOCOM, (2014), pp. 1617–1625.
[36] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, I. Stoica, Achieving convergence-free routing using failure-carrying packets, Proceedings of the ACM SIGCOMM, (2007), pp. 241–252.
[37] B. Stephens, A.L. Cox, S. Rixner, Plinko: building provably resilient forwarding tables, Proceedings of the Twelfth ACM HotNets, (2013).
[38] M. Chiesa, A.V. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, M. Schapira, S. Shenker, On the resiliency of randomized routing against multiple edge failures, Proceedings of the Forty-Third International Colloquium on Automata, Languages, and Programming (ICALP), (2016), pp. 134:1–134:15.
[39] M. Chiesa, A. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, A. Panda, M. Schapira, S. Shenker, Exploring the limits of static resilient routing, Proceedings of the IEEE INFOCOM, (2016).
[40] B. Stephens, A.L. Cox, S. Rixner, Scalable multi-failure fast failover via forwarding table compression, SOSR. ACM (2016).
[41] M. Borokhovich, S. Schmid, How (not) to shoot in your foot with SDN local fast failover: a load-connectivity tradeoff, Proceedings of the Seventeenth International Conference on Principles of Distributed Systems (OPODIS), (2013).
[42] M. Chiesa, I. Nikolaevskiy, S. Mitrovic, A. Panda, A. Gurtov, A. Madry, M. Schapira, S. Shenker, The quest for resilient (static) forwarding tables, Proceedings of the IEEE INFOCOM, (2016).
[43] G. Barnes, W.L. Ruzzo, Deterministic algorithms for undirected s-t connectivity using polynomial time and sublinear space. Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing (STOC), (1991), pp. 43–53.
[44] O. Reingold, Undirected connectivity in log-space, J. ACM 55 (4) (2008) 17:1–17:24.
[45] M. Patrascu, M. Thorup, Planning for fast connectivity updates, Proceedings of the Forty-Eight Annual IEEE Symposium on Foundations of Computer Science (FOCS), (2007), pp. 263–271.
[46] S. Istrail, Polynomial universal traversing sequences for cycles are constructible, Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC), (1988), pp. 491–503.
[47] E. Bampas, L. Gasieniec, N. Hanusse, D. Ilcinkas, R. Klasing, A. Kosowski, Euler tour lock-in problem in the rotor-router model: I choose pointers and you choose port numbers, Proceedings of the Twenty-Third International Conference on Distributed Computing (DISC), (2009), pp. 423–435.
[48] V.B. Priezzhev, D. Dhar, A. Dhar, S. Krishnamurthy, Eulerian walkers as a model of self-organized criticality, Phys. Rev. Lett. 77 (25) (1996) 5079.
[49] M. Borokhovich, L. Schiff, S. Schmid, Provable data plane connectivity with local fast failover: introducing openflow graph algorithms, Proceedings of the ACM SIGCOMM HotSDN, (2014).
[50] L. Schiff, M. Borokhovich, S. Schmid, Reclaiming the brain: useful openflow functions in the data plane, Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets), (2014).