Ensuring Connectivity via Data Plane Mechanisms

Junda Liu[‡], Aurojit Panda[♮], Ankit Singla[†], Brighten Godfrey[†], Michael Schapira[◇], Scott Shenker[♮♠]
[‡]Google Inc., [♮]UC Berkeley, [†]UIUC, [◇]Hebrew U., [♠]ICSI

Abstract

We typically think of network architectures as having two basic components: a data plane *responsible for forwarding packets at line-speed, and a* control plane *that instantiates the forwarding state the data plane needs. With this separation of concerns, ensuring connectivity is the responsibility of the control plane. However, the control plane typically operates at timescales several orders of magnitude slower than the data plane, which means that failure recovery will always be slow compared to data plane forwarding rates.*

In this paper we propose moving the responsibility for connectivity to the data plane. Our design, called Data-Driven Connectivity (DDC) ensures routing connectivity via data plane mechanisms. We believe this new separation of concerns — basic connectivity on the data plane, optimal paths on the control plane — will allow networks to provide a much higher degree of availability, while still providing flexible routing control.

1 Introduction

In networking, we typically make a clear distinction between the *data plane* and the *control plane*. The data plane forwards packets based on local state (*e.g.,* a router's FIB). The control plane establishes this forwarding state, either through distributed algorithms (*e.g.,* routing) or manual configuration (*e.g.,* ACLs for access control). In the naive version of this two-plane approach, the network can recover from failure only after the control plane has computed a new set of paths and installed the associated state in all routers. The disparity in timescales between packet forwarding (which can be less than a microsecond) and control plane convergence (which can be as high as hundreds of milliseconds) means that failures often lead to unacceptably long outages.

To alleviate this, the control plane is often assigned the task of precomputing failover paths; when a failure occurs, the data plane utilizes this additional state to forward packets. For instance, many datacenters use ECMP, a data plane algorithm that provides automatic failover to another shortest path. Similarly, many WAN networks use MPLS's Fast Reroute to deal with failures on the data plane. These "failover" techniques set up additional, but static, forwarding state that allows the datapath to deal with one, or a few, failures. However, these methods require careful configuration, and lack guarantees. Such configuration is tricky, requiring operators to account for complex factors like multiple link failures, and correlated failures. Despite the use of tools like shared-risk link groups to account for these issues, a variety of recent outages [21, 29, 34, 35] have been attributed to link failures. While planned backup paths are perhaps enough for most customer requirements, they are still insufficient when stringent network resilience is required.

This raises a question: can the failover approach be extended to more general failure scenarios? We say that a data plane scheme provides *ideal forwarding-connectivity* if, for any failure scenario where the network remains physically connected, its forwarding choices would guide packets to their intended destinations.[1] Our question can then be restated as: can any approach using static forwarding state provide ideal forwarding-connectivity? We have shown (see [9] for a precise statement and proof of this result) that without modifying packet headers (as in [16, 18]) the answer is *no*: one cannot achieve ideal forwarding-connectivity with static forwarding state.

Given that this impossibility result precludes ideal forwarding-connectivity using *static* forwarding information, the question is whether we can achieve ideal forwarding-connectivity using state change operations that can be executed at data plane timescales. To this end, we propose the idea of *data-driven connectivity* (DDC), which maintains forwarding-connectivity via simple changes in forwarding state predicated only on the destination address and incoming port of an arriving packet. DDC relies on state changes which are simple enough to be done at packet rates with revised hardware (and, in current routers, can be done quickly in software). Thus, DDC can be seen as moving the responsibility for connectivity to the data plane.

The advantage of the DDC paradigm is that it leaves the network functions which require global knowledge (such as optimizing routes, detecting disconnections, and distributing load) to be handled by the control plane, and moves connectivity maintenance, which has simple yet crucial semantics, to the data plane. DDC can react, at worst, at a much faster time scale than the control plane, and with new hardware can keep up with the data plane.

DDC's goal is simple: ideal connectivity with data plane mechanisms. It does not bound latency, guarantee in-order packet delivery, or address concerns of routing

[1]Note that ideal forwarding-connectivity does not guarantee packet delivery, because such a guarantee would require dealing with packet losses due to congestion and link corruption.

policy; leaving all of these issues to be addressed at higher layers where greater control can be exercised at slower timescales. Our addition of a slower, background control plane which can install arbitrary routes safely even as DDC handles data plane operations, addresses the latency and routing policy concerns over the long term.

We are unaware of any prior work towards the DDC paradigm (see discussion of related work in §5). DDC's algorithmic foundations lie in *link reversal routing* (Gafni and Bertsekas [10], and subsequent enhancements [8, 26, 32]). However, traditional link reversal algorithms are not suited to the data plane. For example, they involve generating special control packets and do not handle message loss (*e.g.,* due to physical layer corruption). In addition, our work extends an earlier workshop paper [17], but the algorithm presented here is quite different in detail, is provably correct, can handle arbitrary delays and losses, and applies to modern chassis switch designs (where intra-switch messaging between linecards may exhibit millisecond delays).

2 DDC Algorithm

2.1 System Model

We model the network as a graph. The assumptions we make on the behavior of the system are as follows.

Per-destination serialization of events at each node. Each node in the graph executes our packet forwarding (and state-update) algorithm serially for packets destined to a particular destination; there is only one such processing operation active at any time. For small switches, representing the entire switch as a single node in our graph model may satisfy this assumption. However, a single serialized node is a very unrealistic model of a large high-speed switch with several linecards, where each linecard maintains a FIB in its ASIC and processes packets independently. For such a large switch, our abstract graph model has *one node for each linecard*, running our node algorithm in parallel with other linecards, with links between all pairs of linecard-nodes within the same switch chassis. We thus only assume each linecard's ASIC executes packets with the same destination serially, which we believe is an accurate model of real switches.

Simple operations on packet time scales. Reading and writing a handful of FIB bits associated with a destination and executing a simple state machine can be performed in times comparable to several packet processing cycles. Our algorithm works with arbitrary FIB update times, but the performance during updates is sub-optimal, so we focus on the case where this period is comparable to the transmission time for a small number of packets.

In-order packet delivery along each link. This assumption is easily satisfied when switches are connected physically. For switches that are separated by other network elements, GRE (or other tunneling technologies) with sequence numbers will enforce this property. Hardware support for GRE or similar tunneling is becoming more common in modern switch hardware.

Unambiguous forwarding equivalence classes. DDC can be applied to intradomain routing at either layer 2 or layer 3. However, we assume that there is an unambiguous mapping from the "address" in the packet header to the key used in the routing table. This is true for routing on MAC addresses and MPLS labels, and even for prefix-based routing (LPM) as long as every router uses the same set of prefixes, but fails when aggregation is nonuniform (some routers aggregate two prefixes, while others do not). This latter case is problematic because a given packet will be associated with different routing keys (and thus different routing entries). MPLS allows this kind of aggregation, but makes explicit when the packet is being routed inside a larger Forwarding Equivalence Class. Thus, DDC is not universally applicable to all current deployments, but can be used by domains which are willing to either (a) use a uniform set of prefixes or (b) use MPLS to implement their aggregation rather than using nonuniform prefixes.

For convenience we refer to the keys indexing into the routing table as destinations. Since DDC's routing state is maintained and modified independently across destinations, our algorithms and proofs are presented with respect to one destination.

Arbitrary loss, delay, failures, recovery. Packets sent along a link may be delayed or lost arbitrarily (*e.g.,* due to link-layer corruption). Links and nodes may fail arbitrarily. A link or node is not considered recovered until it undergoes a control-plane recovery (an AEO operation; §3). This is consistent with typical router implementations which do not activate a data plane link until the control plane connection is established.

2.2 Link Reversal Background

DDC builds on the classic Gafni-Bertsekas (**GB**) [10] link reversal algorithms. These algorithms operate on an abstract directed graph which is at all times a directed acyclic graph (**DAG**). The goal of the algorithm is to modify the graph incrementally through **link reversal** operations in order to produce a "destination-oriented" DAG, in which the destination is the only node with no outgoing edges (i.e., a **sink**). As a result, all paths through the DAG successfully lead to the destination.

The GB algorithm proceeds as follows: Initially, the graph is set to be an arbitrary DAG. We model link directions by associating with each node a variable *direction* which maps that node's links to the data flow direction (In or Out) for that link. Throughout this section we describe our algorithms with respect to a particular destination. Sinks other than the destination are **activated** at arbitrary times; a node v, when activated, executes the following algorithm:

```
GB_activate(v)
  if for all links L, direction[L] = In
      reverse all links
```

Despite its simplicity, the GB algorithm converges to a destination-oriented DAG in finite time, regardless of the pattern of link failures and the initial link directions, as long as the graph is connected [10]. Briefly, the intuition for the algorithm's correctness is as follows. Suppose, after a period of link failures, the physical network is static. A node is *stable* at some point in time if it is done reversing. If any node is unstable, then there must exist a node u which is unstable but has a stable neighbor w. (The destination is always stable.) Since u is unstable, eventually it reverses its links. But at that point, since w is already stable, the link $u \rightarrow w$ will never be reversed, and hence u will always have an outgoing link and thus become stable. This increases the number of stable nodes, and implies that all nodes will eventually stabilize. In a stable network, since no nodes need to reverse links, the destination must be the only sink, so the DAG is destination-oriented as desired. For an illustrated example of GB algorithm execution, we refer the reader to Gafni-Bertsekas' seminal paper [10].

Gafni-Bertsekas also present (and prove correct) a *partial reversal* variant of the algorithm: Instead of reversing all links, node v keeps track of the set S of links that were reversed by its neighbors since v's last reversal. When activated, if v is a sink, it does two things: (1) It reverses $N(v) \setminus S$—unless *all* its links are in S, in which case it reverses all its links. (2) It empties S.

However, GB is infeasible as a data plane algorithm: To carry out a reversal, a node needs to generate and send special messages along each reversed link; the proofs assume these messages are delivered reliably and immediately. Such production and processing of special packets, potentially sent to a large number of neighbors, is too expensive to carry out at packet-forwarding timescales. Moreover, packets along a link may be delayed or dropped; loss of a single link reversal notification in GB can cause a permanent loop in the network.

2.3 Algorithm

DDC's goal is to implement a link reversal algorithm which is suited to the data plane. Specifically, all events are triggered by an arriving data packet, employ only simple bit manipulation operations, and result only in the forwarding of that single packet (rather than duplication or production of new packets). Moreover, the algorithm can handle arbitrary packet delays and losses.

The DDC algorithm provides, in effect, an emulation of GB using only those simple data plane operations. Somewhat surprisingly, we show this can be accomplished without special signaling, using only a *single bit* piggybacked in each data packet header—or equivalently, *zero bits*, with two virtual links per physical link. Virtual links can be implemented as GRE tunnels.

The DDC algorithm follows. Our presentation and implementation of DDC use the partial reversal variant of GB, which generally results in fewer reversals [4]. However, the design and proofs work for either variant.

State at each node:

- `to_reverse`: List containing a subset of the node's links, initialized to the node's incoming links in the given graph G.

Each node also keeps for each link `L`:

- `direction[L]`: In or Out; initialized to the direction according to the given graph G. Per name, this variable indicates this node's view of the direction of the link `L`.
- `local_seq[L]`: One-bit unsigned integer; initialized to 0. This variable is akin to a version or sequence number associated with this node's view of link `L`'s direction.
- `remote_seq[L]`: One-bit unsigned integer; initialized to 0. This variable attempts to keep track of the version or sequence number at the neighbor at the other end of link `L`.

All three of these variables can be modeled as booleans, with increments resulting in a bit-flip.

State in packets: We will use the following notation for our one bit of information in each packet:

- `packet.seq`: one-bit unsigned integer.

A comparison of an arriving packet's sequence number with `remote_seq[L]` provides information on whether the node's view of the link direction is accurate, or the link has been reversed. We note that `packet.seq` can be implemented as a bit in the packet header, or equivalently, by sending/receiving the packet on one of two virtual links. The latter method ensures that *DDC requires no packet header modification*.

Response to packet events: The following two routines handle received and locally generated packets.

```
packet p generated locally:
   update_FIB_on_departure()
   send_on_outlink(any outlink, p)

packet p received on link L:
   update_FIB_on_arrival(p, L)
   update_FIB_on_departure()
   if (direction[L] = Out)
      if (p.seq != remote_seq[L])
         send_on_outlink(L, p)
   send_on_outlink(any outlink, p)

send_on_outlink(link L, packet p)
   p.seq = local_seq[L]
   send p on L
```

These algorithms are quite simple: In general, after updating the FIB (as specified below), the packet can be sent on any outlink. There is one special case which will allow us to guarantee convergence (see proof of Thm. 2.1): if a packet was *received* on an outlink without a new sequence number, indicating that the neighbor has stale information about the direction of this link, it is "bounced back" to that neighbor.

FIB update: The following methods perform local link reversals when necessary.

```
reverse_in_to_out(L)
   direction[L] = Out
   local_seq[L]++

reverse_out_to_in(L)
   direction[L] = In
   remote_seq[L]++

update_FIB_on_arrival(packet p, link L)
   if (direction[L] = In)
      assert(p.seq == remote_seq[L])
   else if (p.seq != remote_seq[L])
      reverse_out_to_in(L)

update_FIB_on_departure()
   if there are no Out links
      if to_reverse is empty
         // 'partial' reversal impossible
         to_reverse = all links
      for all links L in to_reverse
         reverse_in_to_out(L)
      // Reset reversible links
      to_reverse = {L: direction[L] = In}
```

The above algorithms determine when news of a neighbor's link reversal has been received, and when we must locally reverse links via a *partial reversal*. For the partial reversal, `to_reverse` tracks what links were incoming at the last reversal (or at initialization). If a partial reversal is not possible (i.e., `to_reverse` is empty), all links are reversed from incoming to outgoing.

To understand how our algorithms work, note that the only exchange of state between neighbors happens through `packet.seq`, which is set to `local_seq[L]` when dispatching a packet on link `L`. Every time a node reverses an incoming link to an outgoing one, it flips `local_seq[L]`. The same operation happens to `remote_seq[L]` when an outgoing link is reversed.

The crucial step of detecting when a neighbor has reversed what a node sees as an outgoing link, is performed as the check: `packet.seq` $\stackrel{?}{=}$ `remote_seq[L]`. If, in the stream of packets being received from a particular neighbor, the sequence number changes, then the link has been reversed to an incoming link.

It is possible for a node v to receive a packet on an outgoing link for which the sequence number has not changed. This indicates that v must have previously reversed the link to outgoing from incoming, but the neighbor hasn't realized this yet (because packets are in flight, or no packets have been sent on the link since that reversal, or packets were lost on the wire). In this case, no new reversals are needed; the neighbor will eventually receive news of the reversal due to the previously-discussed special case of "bouncing back" the packet.

Response to link and node events: Links to neighbors that fail are simply removed from a node's forwarding table. A node or link that recovers is not incorporated by our data plane algorithm. This recovery occurs in the control plane, either locally at a node or as a part of the periodic global control plane process; both use the AEO operation we introduce later (§3).

FIB update delay: For simplicity our exposition has assumed that the FIB can be modified as each packet is processed. While the updates are quite simple, on some hardware it may be more convenient to decouple packet forwarding from FIB modifications.

Fortunately, DDC can allow the two `update_FIB` functions to be called after some delay, or even skipped for some packets (though the calls should still be ordered for packets from the same neighbor). From the perspective of FIB state maintenance, delaying or skipping `update_FIB_on_arrival()` is equivalent to the received packet being delayed or lost, which our model and proofs allow. Delaying or skipping `update_FIB_on_departure()` has the problem that there might be no outlinks. In this case, the packet can be sent out an inlink. Since `reverse_out_to_in()` is not called, the packet's sequence number is not incremented, and the neighbor will not interpret it as a link reversal.

Of course, delaying FIB updates delays data plane convergence, and during this period packets may temporarily loop or travel on less optimal paths. However, FIB update delay is a performance, rather than a correctness, issue; and our experiments have not shown significant problems with reasonable FIB update delays.

2.4 Correctness and Complexity

Our main results (proofs in Appendix A) show that DDC (a) provides ideal forwarding-connectivity; and (b) converges, *i.e.,* the number of reversal operations is bounded.

Theorem 2.1. *DDC guides*[2] *every packet to the destination, assuming the graph remains connected during an arbitrary sequence of failures.*

Theorem 2.2. *If after time t, the network has no failures and is connected, then regardless of the (possibly infinite)*

[2]By guide we mean that following the instructions of the forwarding state would deliver the packet to the destination, assuming no packet losses or corruption along the way.

sequence of packets transmitted after t, DDC incurs $O(n^2)$ reversal operations for a network with n nodes.

We note one step in proving the above results that may be of independent interest. Our results build on the traditional Gafni-Bertsekas (GB) link reversal algorithm, but GB is traditionally analyzed in a model in which reversal notifications are immediately delivered to all of a node's neighbors, so the DAG is always in a globally-consistent state.[3] However, we note that these requirements can be relaxed, without changing the GB algorithm.

Lemma 2.3. *Suppose a node's reversal notifications are eventually delivered to each neighbor, but after arbitrary delay, which may be different for each neighbor. Then beginning with a weakly connected DAG (i.e., a DAG where not all nodes have a path to the destination) with destination d, the GB algorithm converges in finite time to a DAG with d the only sink.*

2.5 Proactive Signaling

The data plane algorithm uses packets as reversal notifications. This implies that if node *A* reverses the link connecting it to *B*, *B* learns of the reversal only when a packet traverses the link. In some cases this can result in a packet traversing the same link twice, increasing path stretch. One could try to overcome this problem by having *A* proactively send a packet with TTL set to 1, thus notifying *B* of this change. This packet looks like a regular data packet, that gets dropped at the next hop router.

Note that proactive signaling is an entirely optional optimization. However, such signaling does not address the general problem of the data plane deviating from optimal paths to maintain connectivity. The following section addresses this problem.

3 Control Plane

While DDC's data plane guarantees ideal connectivity, we continue to rely on the *control plane* for path optimality. However, we must ensure that control plane actions are compatible with DDC's data plane. In this section, we present an algorithm to guide the data plane to use paths desired by the operator (*e.g.,* least-cost paths). The algorithm described does not operate at packet timescales, and relies on explicit signaling by the *control plane*. We start by showing how our algorithm can guide the data plane to use shortest paths. In §3.3, we show that our method is general, and can accommodate any DAG.

We assume that each node is assigned a *distance* from the destination. These could be produced, for instance, by a standard shortest path protocol or by a central coordinator. We will use the distances to define a *target* DAG on the graph by directing edges from higher- to lower-distance nodes, breaking ties arbitrarily but consistently (perhaps by comparing node IDs). Note that this target DAG may not be destination-oriented — for instance the shortest path protocol may not have converged, so distances are inconsistent with the topology. Given these assigned distances, our algorithm guarantees the following properties:

- **Safety:** Control plane actions must not break the data plane guarantees, even with arbitrary simultaneous dynamics in both data and control planes.
- **Routing Efficiency:** After the physical network and control plane distance assignments are static, if the target DAG is destination-oriented, then the data plane DAG will match it.

These guarantees are not trivial to provide. Intuitively, if the control plane modifies link directions while the data plane is independently making its own changes, it is easy for violations of safety to arise. The most obvious approach would be for a node to unilaterally reverse its edges to match the target DAG, perhaps after waiting for nodes closer to the destination to do the same. But dynamics (e.g., link failures) during this process can quickly lead to loops, so packets will loop indefinitely, violating safety. We are attempting to repair a running engine; our experience shows that even seemingly innocuous operations can lead to subtle algorithmic bugs.

3.1 Control Plane Algorithm

Algorithm idea: We decompose our goals of safety and efficiency into two modules. First, we design an all-edges-outward (**AEO**) operation which modifies all of a single node's edges to point outward, and is guaranteed not to violate safety regardless of when and how AEOs are performed. For example, a node which fails and recovers, or has a link which recovers, can unilaterally decide to execute an AEO in order to rejoin the network.

Second, we use AEO as a subroutine to incrementally guide the network towards shortest paths. Let $v_1, \ldots, v_n$ be the (non-destination) nodes sorted in order of distance from the destination, i.e., a topological sort of the target DAG. Suppose we iterate through each node i from 1 to n, performing an AEO operation on each v_i. The result will be the desired DAG, because for any undirected edge (u, v) such that u is closer to the destination, v will direct the edge $v \to u$ *after* u directed it $u \to v$.

Implementation overview: The destination initiates a heartbeat, which propagates through the network serving as a trigger for each node to perform an AEO. Nodes ensure that in applying the trigger, they do not precede neighbors who occur before them in the topological sort order. Note that if the target DAG is not destination-oriented, nodes may be triggered in arbitrary order. However, this does not affect *safety*; further, if the target DAG

[3] For example, [19] notes that the GB algorithm's correctness proof "requires tight synchronization between neighbors, to make sure the link reversal happens atomically at both ends ... There is some work required to implement this atomicity."

is destination-oriented, the data plane will converge to it, thus meeting the routing efficiency property.

We next present our algorithm's two modules: the all-edges-outward (AEO) operation (§3.1.1), and Trigger heartbeats (§3.1.2). We will assume throughout that all signals are sent through reliable protocols and yield acks/nacks. We also do not reinvent sequence numbers for heartbeats ignoring the related details.

3.1.1 All edges outward (AEO) operations

A node setting all its edges to point outward in a DAG cannot cause loops[4]. However, the data plane dynamics necessitate that we use caution with this operation—there might be packets in flight that attempt to reverse some of the edges inwards as we attempt to point the rest outwards, potentially causing loops. One could use distributed locks to pause reversals during AEO operations, but we cannot block the data plane operations as this would violate the ideal-connectivity guarantee provided by DDC.

Below is an algorithm to perform an AEO operation at a node *v* safely and without pausing the data plane, using virtual nodes (vnodes). We use virtual nodes as a convenient means of versioning a node's state, to ensure that the DAG along which a packet is forwarded remains consistent. The strategy is to connect a new vnode *vn* to neighbors with all of *vn*'s edges outgoing, and then delete the old vnode. If any reversals are detected during this process, we treat the process as failed, delete *vn*, and continue using the old vnode at *v*. Bear in mind that this is all with respect to a specific destination; *v* has other, independent vnodes for other destinations. Additionally, although the algorithm is easiest to understand in terms of virtual nodes, it can be implemented simply with a few extra bits per link for each destination[5].

We require that neighboring nodes not perform control plane reversals simultaneously. This is enforced by a simple lock acquisition protocol between neighbors before performing other actions in AEO. However, note that these locks only pause other control-plane AEO operations; all data plane operations remain active.

```
AEO algorithm:
   Get locks from {neighbors, self} in
                increasing ID order
   Create virtual node vn
   run(thread_watch_for_packets)
   run(thread_connect_virtual_node)

thread_watch_for_packets:
   if a data packet arrives at vn
      kill thread_connect_virtual_node
      delete vn
      exit thread_watch_for_packets

thread_connect_virtual_node:
   For each neighbor u of v
      Link vn, u with virtual link
      Signal(LinkDone?) to u

   After all neighbors ack(LinkDone):
      For each neighbor u of v
         Signal(Dir: vn->u?) to u
      After all neighbors ack(Dir: vn->u):
         kill thread_watch_for_packets
         delete old virtual nodes at v
   exit thread_connect_virtual_node

When all threads complete:
    release all locks
```

The algorithm uses two threads, one to watch for data packets directed to the destination (which would mean the neighbor has reversed the link), and the other to establish links with the neighbors directed towards them, using signal-ack mechanisms. The second set of signals and acks might appear curious at first, but it merely confirms that no reversals were performed before *all* acks for the *first* set had been *received* at *vn*. There may have been reversals since any of the second set of acks were dispatched from the corresponding neighbor, but they are inconsequential (as our proof will show).

3.1.2 Trigger heartbeats

We make use of periodic heartbeats issued by the destination to trigger AEO operations. To order these operations, a node uses heartbeat H as a trigger after making sure all of its neighbors with lower distances have already responded to the H. More specifically, a node, *v*, responds to heartbeat H from neighbor *w* as follows:

```
If (last_heartbeat_processed >= H)
   Exit

rcvd[H,w] = true
If (rcvd[H,u] = true for all nbrs u with
                     lower distance)
   If (direction[u] = In for any nbr with
                     lower distance)
       AEO(v)
   Send H to all neighbors
   last_heartbeat_processed = H
```

With regard to the correctness of this algorithm, we prove the following theorem in Appendix B:

Theorem 3.1. *The control plane algorithm satisfies the safety and routing efficiency properties.*

3.2 Physical Implementation

A vnode is merely an abstraction containing the state described in §2.3, and allowing this state to be modified in the ways described previously. One can therefore

[4] This is why we designed the algorithm as a sequence of AEOs.

[5] Routers implement ECMP similarly: In a *k*-port router, *k*-way ECMP [7] stores state items per (destination, output-port) pair.

represent a vnode as additional state in a switch, and interactions with other switches can be realized using virtual links. As stated previously, the virtual links themselves may be implemented using GRE tunnels, or by the inclusion of additional bits in the packet header.

3.3 Control Plane Generality

It is easy to show that the resulting DAG from the AEO algorithm is entirely determined by the order in which AEO operations are carried out. While trigger heartbeats as described in §3.1.2 order these AEO operations by distance, any destination-oriented DAG could in fact be used. It is also easy to see that given a DAG, one can calculate at least one order of AEO operations resulting in the DAG.

Given these observations, the control plane described above is entirely general allowing for the installation of an arbitrary DAG, by which we imply that given another control plane algorithm, for which the resultant routes form a DAG, there exists a modified trigger heartbeat function that would provide the same functionality as the given control plane algorithm. DDC therefore does not preclude the use of other control plane algorithms that might optimize for metrics other than path length.

3.4 Disconnection

Detecting disconnection is particularly important for link-reversal algorithms. If our data plane algorithm fails to detect that a destination is unreachable, packets for that destination might keep cycling in the connected component of the network. Packets generated for the destination may continue to be added, while none of these packets are being removed from the system. This increases congestion, and can interfere with packets for other destinations.

Since DDC can be implemented at both the network and link-layer we cannot rely on existing TTL/hop-count fields, since they are absent from most extant link-layer protocols. Furthermore, failures might result in paths that are longer than would be allowed by the network protocol, and thus TTL-related packet drops cannot be used to determine network connectivity.

Conveniently, we can use heartbeats to detect disconnection. Any node that does not receive a heartbeat from the destination within a fixed time period can assume that the destination is unreachable. The timeout period can be set to many times the heartbeat interval, so that the loss of a few heartbeats is not interpreted as disconnection.

3.5 Edge Priorities

The algorithm as specified allows for the use of an arbitrary output link (*i.e.,* packets can be sent out any output link). One can exploit this choice to achieve greater efficiency, in particular the choice of output links can be driven by a control plane specified priority. Priorities can be chosen to optimize for various objective functions, for instance traffic engineering. Such priorities can also be used to achieve faster convergence, and lower stretches, especially when few links have failed.

Edge priorities are not required for the correct functioning of DDC, and are only useful as a mechanism to increase efficiency, and as a tool for traffic engineering. Priorities can be set by the control plane with no synchronization (since they do not affect DDC's correctness), and can either be set periodically based on some global computation, or manually based on operator preference.

4 Evaluation

We evaluated DDC using a variety of microbenchmarks, and an NS-3 [23] based macrobenchmark.

4.1 Experimental Setup

We implemented DDC as a routing algorithm in NS-3. (The code is available on request.) Our implementation includes the basic data plane operations described in §2, and support for assigning priorities to ports (*i.e.,* links, which appear as ports for individual switches), allowing the data plane to discriminate between several available output ports. We currently set these priorities to minimize path lengths. We also implemented the control plane algorithm (§3), and use it to initialize routing tables. While our control plane supports the use of arbitrary DAGs, we evaluated only shortest-path DAGs.

We evaluated DDC on 11 topologies: 8 AS topologies from RocketFuel [30] varying in size from 83 nodes and 272 links (AS1221), to 453 nodes and 1999 links (AS2914); and 3 datacenter topologies—a 3-tier hierarchical topology recommended by Cisco [3], a Fat-Tree [1], and a VL2 [12] topology. However, we present only a representative sample of results here.

Most of our experiments use a link capacity of 10 Gbps. Nodes use output queuing, with drop-tail queues with a 150 KB capacity. We test both TCP (NS-3's TCP-NewReno implementation) and UDP traffic sources.

4.2 Microbenchmarks

4.2.1 Path Stretch

Stretch is defined as the ratio between the length of the path a packet takes through the network, and the shortest path between the packet's source and destination in the current network state, *i.e.,* after accounting for failures.

Stretch is affected by the topology, the number of failed links, and the choice of source and destination. To measure stretch, we selected a random source and destination pair, and failed a link on the connecting path. We then sent out a series of packets, one at a time (*i.e.,* making sure there is no more than one packet in the network at any time) to avoid any congestion drops, and observed the

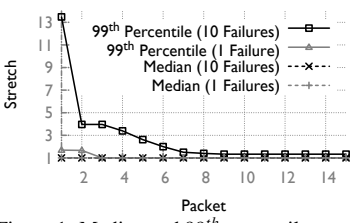

Figure 1: Median and 99^{th} percentile stretch for AS1239.

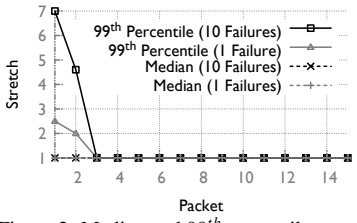

Figure 2: Median and 99^{th} percentile stretch for a FatTree

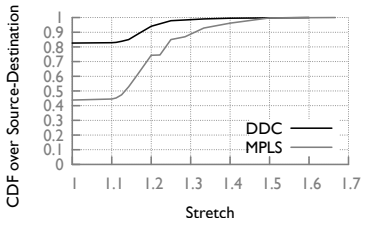

Figure 3: CDF of steady state stretch for MPLS FRR and DDC in AS2914.

length of the path the packet takes. Subsequent packets use different paths as DDC routed around the failures.

Figures 1 and 2 show stretch for a series of such packets, either with 1 or 10 links failed. We tested a wider range of link failures, but these graphs are representative of the results. As expected, the initial stretch is dependent on the number of links failed, for instance the 99^{th} percentile stretch for AS1239 with 10 link failures is 14. However, paths used rapidly converge to near-optimal.

We compare DDC's steady-state stretch with that of MPLS link protection [25]. Link protection, as commonly deployed in wide-area networks, assigns a backup path around a single link, oblivious of the destination[6]. We use the stretch-optimal strategy for link protection: the backup path for a link is the shortest path connecting its two ends. Figure 3 shows this comparison for AS2914. Clearly, path elongation is lower for DDC. We also note that link protection does not support multiple failures.

4.2.2 Packet Latency

In addition to path lengths, DDC may also impact packet latency by increasing queuing at certain links as it moves packets away from failures. While end-to-end congestion control will eventually relieve such queuing, we measure the temporary effect by comparing the time taken to deliver a packet before and after a failure.

To measure packet latencies, we used 10 random source nodes sending 1GB of data each to a set of randomly chosen destinations. The flows were rate limited (since we were using UDP) to ensure that no link was used at anything higher than 50% of its capacity, with the majority of links being utilized at a much lower capacity. For experiments with AS topologies, we set the propagation delay to 10ms, to match the order of magnitude for a wide area network, while for datacenter topologies, we adjusted propagation delay such that RTTs were $\sim$250μs, in line with previously reported measurements [5, 36].

For each source destination pair we measure baseline latency as an average over 100 packets. We then measure the latency after failing a set of links. Figure 4 shows the results for AS2914, and indicates that over 80% of packets encounter no increase in latency, and independent of the number of failures, over 96% of packets encounter only a modest increase in latency. Similarly, Figure 5 shows the same result for a Fat Tree topology, and shows that over 95% of the packets see no increased latency. In the 2 failure case, over 99% of packets are unaffected.

4.2.3 TCP Throughput and FIB Update Delay

Ideally, switches would execute DDC's small state updates at line rate. However, this may not always be feasible, so we measure the effects of delayed state updates. Specifically, we measure the effect of additional delay in FIB updates on TCP throughput in wide-area networks.

We simulated a set of WAN topologies with 1 Gbps links (for ease of simulation). For each test we picked a set of 10 source-destination pairs, and started 10 GB flows between them. Half-a-second into the TCP transfer, we failed between 1 and 5 links (the half-a-second duration was picked so as to allow TCP congestion windows to converge to their steady state), and measured overall TCP throughput. Our results are shown in Figure 6, and indicate that FIB delay has no impact on TCP throughput.

4.3 Macrobenchmarks

We also simulated DDC's operation in a datacenter, using a fat-tree topology with 8-port switches. To model failures, we used data on the time it takes for datacenter networks to react to link failures from Gill et al [11]. Since most existing datacenters do not use any link protection scheme, relying instead on ECMP and the plurality of paths available, we use a similar multipath routing algorithm as our baseline.

For our workload, we used partition-aggregate as previously described in DCTCP [2]. This workload consists of a set of background flows, whose size and interarrival frequencies we get from the original paper, and a set of smaller, latency sensitive, request queries. The request queries proceed by having a single machine send a set of 8 machines a single small request packet, and then receiving a 2 KB response in return. This pattern commonly occurs in front-end datacenters, and a set of such requests are used to assemble a single page. We generated a set of such requests, and focused on the percentage of these

[6]Protecting links in this manner is the standard method used in wide-area networks, for instance [6], states "High Scalability Solution—The Fast Reroute feature uses the highest degree of scalability by supporting the mapping of all primary tunnels that traverse a link onto a single backup tunnel. This capability bounds the growth of backup tunnels to the number of links in the backbone rather than the number of TE tunnels that run across the backbone."

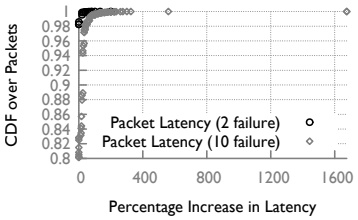

Figure 4: Packet latencies in AS2914 with link failures.

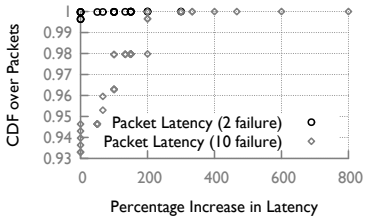

Figure 5: Packet latencies in a datacenter with a Fat-Tree topology with link failures.

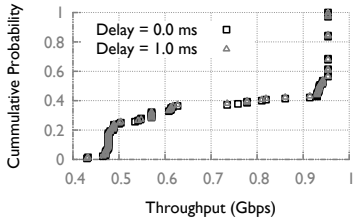

Figure 6: Distribution of TCP throughputs for varying FIB update delays.

that were satisfied during a "failure event", *i.e.,* a period of time where one or more links had failed, but before the network had reacted. On average, we generated 10 requests per second, spread evenly across 128 hosts. The hosts involved in processing a request were also picked at random. We looked at a variety of failure scenarios, where we failed between 1 and 5 links. Here, we present results from two scenarios, one where a single link was failed, and one where 5 links were failed. The links were chosen at random from a total of 384 links.

Figure 7 shows the percentage of requests served in every 10 second interval (*i.e.,* percent of request packets resulting in a response) in a case with 5 link failures. The red vertical line at $x = 50$ seconds indicates the point at which the links failed. While this is a rare failure scenario, we observe that, without DDC, in the worst case about 14% of requests cannot be fulfilled. We also look at a more common case in Figure 8 where a single link is failed, and observe a similar response rate. The response rate itself is a function of both the set of links failed, and random requests issued.

For many datacenter applications, response latency is important. Figure 9 shows the distribution of response latencies for the 5 link failure case described previously. About 4% of the requests see no responses when DDC is not used. When DDC is used, all requests result in responses and fewer than 1% see higher latency than the common case without DDC. For these 1% (which would otherwise be dropped), the latency is at most 1.5× higher. Therefore, in this environment, DDC not only delivers all requests, it delivers them relatively quickly.

5 Related Work

Link-Reversal Algorithms: There is a substantial literature on link-reversal algorithms [10, 8, 26, 32]. We borrow the basic idea of link reversal algorithms, but have extended them in ways as described in §2.2.

DAG-based Multipath: More recently there has been a mini-surge in DAG-based research [15, 28, 27, 14, 24]. All these proposals shared the general goal of maximizing robustness while guaranteeing loop-freeness. In most cases, the optimization boils down to a careful ordering of the nodes to produce an appropriate DAG. Some of this research also looked at load distribution. Our approach differs in that we don't optimize the DAG itself but instead construct a DAG that performs adequately well under normal conditions and rely on the rapid link reversal process to restore connectivity when needed.

Other Resilience Mechanisms: We have already mentioned several current practices that provide some degree of data plane resilience: ECMP and MPLS Fast Reroute. We note that the ability to install an arbitrary DAG provides strictly more flexibility than is provided by ECMP.

MPLS Fast Reroute, as commonly deployed, is used to protect individual links by providing a backup path that can route traffic around a specific link failure. Planned backups are inherently hard to configure, especially for multiple link failures, which as past outages indicate, may occur due to physical proximity of affected links, or other reasons [20]. While this correlation is often accounted for (*e.g.,* using shared risk link groups), such accounting is inherently imprecise. This is evidenced by the Internet outage in Pakistan in 2011 [21] which was caused by a failure in both a link and its backup, and other similar incidents [29, 35, 34] which have continued to plague providers. Even if *ideal connectivity* isn't an explicit goal, using DDC frees operators from the difficulties of careful backup configuration. However, if operators do have preferred backup configurations, DDC makes it possible to achieve the best of both worlds: Operators can install a MPLS/GRE tunnel (*i.e.,* a virtual link) for each desired backup path, and run DDC over the physical and virtual links. In such a deployment, DDC would only handle failures beyond the planned backups.

End-to-End Multipath: There is also a growing literature on end-to-end multipath routing algorithms (see [31] and [22] for two such examples). Such approaches require end-to-end path failure detection (rather than hop-by-hop link failure detection as in DDC), and thus the recovery time is quite long compared to packet transmission times. In addition, these approaches do not provide ideal failure recovery, in that they only compute a limited number of alternate paths, and if they all fail then they rely on the control plane for recovery.

Other Approaches: Packet Recycling [18] is perhaps the work closest in spirit to DDC (but quite different in approach), where connectivity is ensured by a packet forwarding algorithm which involves updating a logarithmic number of bits in the packet header. While this approach is a theoretical tour-de-force, it requires solving an NP-

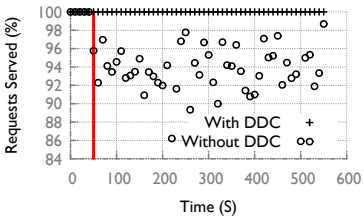

Figure 7: Percentage of requests satisfied per 10 second interval with 5 failed links.

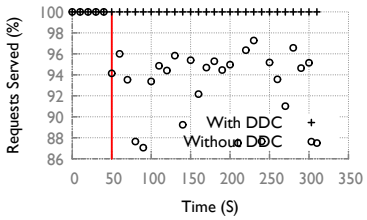

Figure 8: Percentage of requests satisfied per 10 second interval with 1 failed link

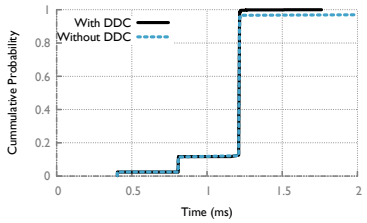

Figure 9: Request latency for the 5-link failure case in Figure 7.

hard problem to create the original forwarding state. In contract, DDC requires little in the way of precomputation, and uses only two bits in the packet header. Failure-Carrying Packets (FCP) [16] also achieves ideal connectivity, but data packets carry explicit control information (the location of failures) and routing tables are recomputed upon a packet's arrival (which may take far longer than a single packet arrival). Furthermore, FCP packet headers can be arbitrarily large, since packets potentially need to carry an unbounded amount of information about failures encountered along the path traversed.

6 Conclusion

In this paper we have presented *DDC*, a dataplane algorithm guaranteeing ideal connectivity. We have both presented proofs for our guarantees, and have demonstrated the benefits of DDC using a set of simulations. We have also implemented the DDC dataplane in OpenVSwitch[7], and have tested our implementation using Mininet [13]. We are also working towards implementing DDC on physical switches.

7 Acknowledgments

We are grateful to Shivaram Venkatraman, various reviewers, and our shepherd Dejan Kostic for their comments and suggestions. This research was supported in part by NSF CNS 1117161 and NSF CNS 1017069. Michael Schapira is supported by a grant from the Israel Science Foundation (ISF) and by the Marie Curie Career Integration Grant (CIG).

A DDC Algorithm Correctness

Compared with traditional link reversal algorithms, DDC has two challenges. First, notifications of link reversals might be delayed arbitrarily. Second, the mechanism with which we provide notifications is extremely limited—piggybacking on individual data packets which may themselves be delayed and lost arbitrarily. We deal with these two challenges one at a time.

A.1 Link reversal with delayed notification

Before analyzing our core algorithm, we prove a useful lemma: the classic GB algorithms (§2.2) work correctly even when reversal notifications are delayed arbitrarily.

[7] Available at `https://bitbucket.org/apanda/ovs-ddc`

The subtlety in this analysis is that if notifications are not delivered instantaneously, then nodes have inconsistent views of the link directions. What we will show is that when it matters—when a node is reversing its links—the node's view is consistent with a certain canonical global state that we define.

We can reason about this conveniently by defining a global notion of the graph at time t as follows: In G_t the direction of an edge (u,v) is:

- $u \to v$ if u has reversed more recently than v;
- $v \to u$ if v has reversed more recently than u;
- otherwise, whatever it is in the original graph G_0.

It is useful to keep in mind several things about this definition. First, reversing is now a local operation at a node. Once a node decides to reverse, it is "officially" reversed—regardless of when control messages are delivered to its neighbors. Second, the definition doesn't explicitly handle the case where there is a tie in the times that u and v have reversed. But this case will never occur; it is easy to see that regardless of notifications begin delayed, two neighbors will never reverse simultaneously because at least one will believe it has an outgoing edge.

Often, nodes' view of their link directions will be inconsistent with the canonical graph G_t because they haven't yet received reversal notifications. The following lemma shows this inconsistency is benign.

Lemma A.1. *Consider the GB algorithm with arbitrarily delayed reversal notifications, but no lost notifications. If v reverses at t, then v's local view of its neighboring edge directions is consistent with G_t.*

Proof. The lemma clearly holds for $t = 0$ due to the algorithm's initialization. Consider any reversal at $t \geq 0$. By induction, at the time of v's previous reversal (or at $t = 0$ if it had none), v's view was consistent with G. The only events between then and time t are edge reversals making v's edges incoming. Before v receives all reversal notifications, it believes it has an outgoing edge, and therefore will not reverse. Once it receives all notifications (and may reverse), its view is consistent with G. □

It should be clear given the above lemma that nodes only take action when they have a "correct" view of their edge directions, and therefore delay does not alter GB's effective behavior. To formalize this, we define a **trace** of an algorithm as a chronological record of its link reversals. An algorithm may have many possible traces, due to

non-determinism in when nodes are activated and when they send and receive messages (and due to the unknown data packet inputs and adversarial packet loss, which we disallow here but will introduce later).

Lemma A.2. *Any trace of GB with arbitrarily delayed notification is also a trace of GB with instant notification.*

Proof. Consider any trace T produced by GB with delay. Create a trace T' of GB with instantaneous notification, in which nodes are initialized to the same DAG as in T and nodes are activated at the moments in which those nodes reverse edges in T. We claim T and T' are identical. This is clearly true at $t = 0$. Consider by induction any $t > 0$ at which v reverses in T. In the GB algorithm, notice that v's reversal action (i.e., which subset of edges v reverses) is a function only of its neighboring edge-states at time t and at the previous moment that v reversed. By Lemma A.1, local knowledge of these edge-states are identical to G_t, which is in turn identical at each step in both T and T' (by induction). □

LEMMA 2.3. *Suppose a node's reversal notifications are eventually delivered to each neighbor, but after arbitrary delay, which may be different for each neighbor. Then beginning with a weakly connected DAG with destination d, the GB algorithm converges in finite time to a DAG with d the only sink.*

Proof. Lemmas A.1, A.2 imply that if reversal notifications are eventually delivered, both versions of the algorithm (with arbitrary delay and with instant notifications) reverse edges identically. Convergence of the algorithm with arbitrary delay to a destination-oriented DAG thus follows from the original GB algorithm's proof. □

A.2 DDC

Having shown that GB handles reversal message delay, we now show that DDC, even with its packet-triggered, lossy, delayed messages, effectively emulates GB.

Lemma A.3. *Any trace of DDC, assuming arbitrary loss and delay but in-order delivery, is a prefix of a trace of GB with instantaneous reliable notification.*

Proof. Consider any neighboring nodes v, w and any time t_0 such that:

1. v and w agree on the link direction;
2. v's `local_seq` for the link is equal to w's `remote_seq`, and vice versa; and
3. any in-flight packet p has p.`seq` set to the same value as its sender's current `local_seq`.

Suppose w.l.o.g v reverses first, at some time t_1. No packet outstanding at time t_0 or sent during $[t_0, t_1)$ can be interpreted as a reversal on delivery, since until that time, the last two properties and the in-order delivery assumption imply that such an arriving packet will have its p.`seq` equal to the receiving node's `remote_seq`. Now we have two cases. In the first case, no packets sent $v \to w$ after time t_1 are ever delivered; in this case, w clearly never believes it has received a link reversal. In the second case, some such packet is delivered to w at some time t_2. The first such packet p will be interpreted as a reversal because it will have p.`seq` $\neq w$.`remote_seq`. Note that neither node reverses the link during (t_0, t_2) since both believe they have an outlink and, like GB, DDC will only reverse a node with no outlinks.

Note that the above three properties are satisfied at time $t_0 = 0$ by initialization; and the same properties are again true at time t_2. Therefore we can iterate the argument across the entire run of the algorithm. The discussion above implies that for any a, b such that v reverses at time a and does not reverse during $[a, b]$, w will receive either zero or one reversal notifications during $[a, b]$. This is exactly equivalent to the notification behavior of GB with arbitrary delay, except that some notifications may never be delivered. Combined with the fact that DDC makes identical reversal decisions as GB when presented with the same link state information, this implies that a trace of DDC over some time interval $[0, T]$ is a *prefix* of a valid trace for GB with arbitrary delay, in which the unsent notifications are delayed until after T. Since by Lemma 2.3 any trace of GB with delay is a valid trace of GB without delay, this proves the lemma. □

While it looks promising, the lemma above speaks only of (data plane) control events, leaving open the possibility that data packets might loop forever. Moreover, since the DDC trace is only a prefix of a GB trace, the network might never converge. Indeed, if no data packets are ever sent, *nothing* happens, and even if some are sent, some parts of the network might never converge. What we need to show is that from the perspective of any data packets, the network operates correctly. We now prove the main theorems stated previously in §2.4.

THEOREM 2.1. *DDC guides every packet to the destination, assuming the graph remains connected during an arbitrary sequence of failures.*

Proof. Consider a packet p which is not dropped due to physical layer loss or congestion. At each node p is forwarded to some next node by DDC, so it must either eventually reach the destination, or travel an infinite number of hops. We will suppose it travels an infinite number of hops, and arrive at a contradiction.

If p travels an infinite number of hops then there is some node v which p visits an infinite number of times. Between each visit, p travels in a loop. We want to show that during each loop, at least one control event—either a reversal or a reversal notification delivery—happens somewhere in the network.

We show this by contradiction. If there are no control events, the global abstract graph representing the network (§A.1) is constant; call this graph G_t. By Lemma A.3, G_t must match some graph produced by GB (in the instantaneous reliable notification setting), which never has loops. Thus G_t does not have loops, so there must exist

some edge (w,u) in the packet's loop which w believes is outgoing, but which is incoming according to G_t. If this occurs, then DDC specifies that u will bounce it back to w. Therefore, by the time the packet returns to w, by the in-order delivery assumption, w will have received a reversal notification. Therefore the assumption that there are no control events must be false: some control event must happen during each loop.

Therefore, there are an infinite number of control events. Therefore, DDC has an infinitely long trace of control events, which by Lemma A.3 means it is also an infinitely-long valid prefix of a trace for GB, which contradicts the fact [10] that GB converges in a finite number of steps. Thus, the assumption that the packet is not delivered must be false, and all packets not dropped due to physical layer loss or congestion are delivered. □

THEOREM 2.2. *If after time t, the network has no failures and is connected, then regardless of the (possibly infinite) sequence of packets transmitted after t, DDC incurs $O(n^2)$ reversal operations for a network with n nodes.*

Proof. Following the same argument as above, since any DDC-trace is a prefix of a GB-trace, and the GB algorithm incurs $\Theta(n^2)$ reversals [33], DDC can not incur more reversals. □

B Control Plane Correctness

In this appendix, we prove Theorem 3.1.

Lemma B.1. *If the AEO algorithm at node v terminates with the addition of a virtual node vn, then at the time of addition of the last edge between vn and a neighbor, all other edges at vn are directed outward.*

Proof. If `thread_watch_for_packets` did not delete *vn*, it saw no incoming packets for the destination at any of *vn*'s links until each neighbor had acknowledged the direction of the link as outward from *vn*. Such packets may be in flight and some link may have been reversed *after* dispatching the ack. However, these acks are requested only after each neighbor has acknowledged that its link with *vn* is functional. Thus, between the addition of the last edge between *vn* and a neighbor's dispatching the ack for the link, all edges were directed outward. □

Lemma B.2. *Given a DAG G, an AEO operation at node v leaves it a DAG with the same physical connectivity.*

Proof. First, we note that AEO operations include obtaining a lock from all neighbors, so no two neighbors can perform an AEO operation concurrently.

AEO either terminates with the addition of vnode *vn* and deletion of the old vnode at *v*, or it terminates leaving the old vnode's *physical* connectivity unchanged. In the latter case, *vn* has not sent any packets on its links, nor has it received any packets until the first reversal that reaches it – thus, its behavior so far is identical to absence. Further, *vn* is immediately killed on receipt of a data packet, in behavior identical to a link failing after a reversal was dispatched on it. Neither scenario can leave the rest of the graph with a loop. On the other hand, if *vn* was added to the graph successfully, then by Lemma B.1, it is added as a node with all edges directed outward, again resulting in no loops. Further, in either scenario, the graph's physical connectivity remains the same – in one case, *vn* copies the old vnode's physical connectivity, while in the other, the old vnode retains connectivity as is. Thus the graph remains a DAG with the same physical connectivity. □

Given that the algorithm only performs a sequence of AEO operations, Lemma B.2 suffices to prove safety. We next show routing efficiency, which completes the proof of Theorem 3.1.

Lemma B.3. *Assume that after some point in time there are no further failures, stable distances are assigned to the nodes such that they induce a destination-oriented target DAG, and control plane messages are eventually delivered. Then the data plane DAG eventually matches the target DAG induced by the distances.*

Proof. Call a node *v compliant* when every edge outgoing from *v* in the target DAG is also outgoing from *v* in the data plane DAG. We will construct a set D of nodes which are (1) compliant, and (2) form a destination-oriented subgraph of the data plane DAG. Specifically, we will prove by induction that eventually D expands to include all nodes. Note that the definition of D immediately implies that these nodes will never reverse their links in the data plane, or execute an AEO in the control plane: they are *done*.

In the base case, D simply includes the destination. In the general case, suppose not all nodes are in D. Since the subgraph D complies with the target DAG, it also forms a destination-oriented subgraph of the target DAG. Combined with the fact that the target DAG is itself destination-oriented, this means D forms a "sink region" into which all paths in the target DAG must flow. Then if not all nodes are in D, there must exist some node $v \notin D$ which is non-compliant, but whose neighbors with lower distances are all in D. Let L be that subset of v's neighbors. Eventually, every node in L will send a heartbeat to v, and v will execute an AEO (if its edges are not already pointing outward to L). At this point, (1) v is compliant, and (2) all v's links to nodes in the destination-oriented subgraph D are outgoing, so that $D \cup \{v\}$ is itself a destination-oriented subgraph of the data plane DAG. Hence, v satisfies the two conditions for inclusion into D and the set D expands. Iterating this argument, D eventually includes all nodes, which implies the lemma. □

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review*, 40(4):63–74, 2010.

[3] M. Arregoces and M. Portolani. *Data center fundamentals*. Cisco Press, 2003.

[4] B. Charron-Bost, J. Welch, and J. Widder. Link reversal: How to play better to work less. In S. Dolev, editor, *Algorithmic Aspects of Wireless Sensor Networks*, volume 5804 of *Lecture Notes in Computer Science*, pages 88–101. Springer Berlin / Heidelberg, 2009.

[5] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82. ACM, 2009.

[6] Cisco. Mpls traffic engineering fast reroute – link protection. `http://www.cisco.com/en/US/docs/ios/12_0st/12_0st10/feature/guide/fastrout.html`.

[7] I. Cisco Systems. Cisco Nexus 3064-X and 3064-T Switches. Datasheet: `http://goo.gl/E3Wqm`, 2012.

[8] M. S. Corson and A. Ephremides. A distributed routing algorithm for mobile wireless networks. *Wireless Networks*, 1(1):61–81, 1995.

[9] J. Feigenbaum, P. B. Godfrey, A. Panda, M. Schapira, S. Shenker, and A. Singla. On the resilience of routing tables. In *Brief announcement, 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2012.

[10] E. M. Gafni and D. P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 1981.

[11] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, pages 350–361. ACM, 2011.

[12] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[13] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container based emulation. *Proc. CoNEXT (to appear)*, 2012.

[14] A. Kvalbein, A. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP network recovery using multiple routing configurations. In *INFOCOM 2006*, pages 1–11. IEEE, 2007.

[15] K. Kwong, L. Gao, R. Guérin, and Z. Zhang. On the feasibility and efficacy of protection routing in IP networks. In *INFOCOM, 2010*, pages 1–9. IEEE, 2010.

[16] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.

[17] J. Liu, B. Yang, S. Shenker, and M. Schapira. Data-driven network connectivity. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

[18] S. Lor, R. Landa, and M. Rio. Packet re-cycling: eliminating packet losses due to network failures. In *HotNets IX*, page 2. ACM, 2010.

[19] N. Lynch. Link-reversal algorithms. In *MIT 6.895 lecture notes*, April 2006. `http://courses.csail.mit.edu/6.885/spring06/notes/lect14a.pdf`.

[20] D. Madory. The 10 Most Bizarre and Annoying Causes of Fiber Cuts. Retrieved September 18, 2012: `http://goo.gl/tIATg`, 2011.

[21] D. Madory. Renesys blog: Large Outage in Pakistan. Retrieved September 18, 2012: `http://www.renesys.com/blog/2011/10/large-outage-in-pakistan.shtml`, 2011.

[22] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *Proc. Networked Systems Design and Implementation*, Apr. 2010.

[23] ns-3. `http://www.nsnam.org/`.

[24] Y. Ohara, S. Imahori, and R. V. Meter. Mara: Maximum alternative routing algorithm. In *INFOCOM*, pages 298–306. IEEE, 2009.

[25] P. Pan, G. Swallow, and A. Atlas. RFC 4090 Fast Reroute Extensions to RSVP-TE for LSP Tunnels, May 2005.

[26] V. Park and M. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *INFOCOM*, 1997.

[27] S. Ray, R. Guérin, K. Kwong, and R. Sofia. Always acyclic distributed path computation. *IEEE/ACM Transactions on Networking (ToN)*, 18(1):307–319, 2010.

[28] C. Reichert, Y. Glickmann, and T. Magedanz. Two routing algorithms for failure protection in IP networks. In *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, pages 97–102. IEEE, 2005.

[29] R. Singel. Threat Level: Fiber Optic Cable Cuts Isolate Millions From Internet. Retrieved September 18, 2012: `http://www.wired.com/threatlevel/2008/01/fiber-optic-cab/`, 2008.

[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *In Proc. ACM SIGCOMM*, pages 133–145, 2002.

[31] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proc. ACM SIGMETRICS*, June 2011.

[32] J. Welch and J. Walter. Link reversal algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2(3):1–103, 2011.

[33] J. L. Welch and J. E. Walter. Link reversal algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2(3):1–103, 2012/01/27 2011.

[34] Wikitech: Site issue Aug 6 2012. Retrieved September 18, 2012: `http://wikitech.wikimedia.org/view/Site_issue_Aug_6_2012`, 2012.

[35] C. Wilson. 'Dual' fiber cut causes Sprint outage. Retrieved September 18, 2012: `http://connectedplanetonline.com/access/news/Sprint_service_outage_011006/`, 2006.

[36] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *SIGCOMM-Computer Communication Review*, 41(4):50, 2011.