Data-Driven Network Connectivity

Junda Liu
U.C. Berkeley / Google Inc.
junda@google.com

Baohua Yang
Tsinghua University
baohuayang@gmail.com

Scott Shenker
U.C. Berkeley / ICSI
shenker@icsi.berkeley.edu

Michael Schapira
Hebrew University of Jerusalem / Google Inc.
michael.schapira@yale.edu

ABSTRACT

Routing on the Internet combines *data plane* mechanisms for forwarding traffic with *control plane* protocols for guaranteeing connectivity and optimizing routes (*e.g.,* shortest-paths and load distribution). We propose *data-driven connectivity* (DDC), a new routing approach that achieves the fundamental connectivity guarantees in the data plane rather than the control plane, while keeping the more complex requirements of route optimization in the control plane. DDC enables faster recovery from failures and easier implementation of control plane optimization.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Reliability

1. INTRODUCTION

Networking researchers often refer to two "planes" in networks. The *data plane* forwards packets based on the packet header and local forwarding state in the router (such as a FIB). The *control plane* is responsible for providing that forwarding state. By setting the forwarding state appropriately, the control plane enables networks to achieve connectivity (forwarding tables provide end-to-end paths), route optimization (choosing shortest or otherwise desirable paths), load distribution (links are not overloaded), and other network control goals.

The data plane is typically implemented in hardware and the control plane is typically implemented in software. However, the fundamental distinction between the two planes is not how they are implemented but instead lies in what state they use and what state they change. The data plane only uses forwarding state local to the router in making its decisions, and does not change this state.[1] The control plane typically uses external state — obtained from a distributed algorithm such as a routing protocol — to set the forwarding state. As a result, the two planes operate at very different speeds: for instance, on a 10gbps link, a 1500byte packet is sent in 1.2μsec while exchanging control plane messages takes on the order of 50msec or more (according to vendors we've spoken with) and global convergence of the dataplane can take many seconds.

In the naive version of this two-plane approach, the network can recover from failure (*i.e.,* restore connectivity) 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 and control plane convergence means that failures often lead to unacceptably long outages. To alleviate this, the control plane is now often assigned the task of precomputing failover paths; when a failure occurs, the dataplane utilizes this additional state to guide the forwarding of the packet. This approach works for a given failure scenario as long as an appropriate backup path has been established, but the degree of resilience achievable with a reasonable number of precomputed backup paths is quite limited (though perhaps enough for most network requirements). More recently, researchers have been developing methods for computing multiple paths between each source and destination; the end host chooses an alternate path when the primary goes down. This approach suffers from the same limited (but perhaps sufficient) degree of resilience, and also requires a downtime of roughly the round-trip-time in the network, which (if the path has

Hotnets '11, November 14–15, 2011, Cambridge, MA, USA.

[1]We consider state that tells the router which of its connected links are up to be local configuration state; it is not derived from either the control plane or data plane, but can be changed by local detection of the link.

any significant speed-of-light latency) is quite long compared to data plane timescales.

This raises the question of whether there is any way to extend the data plane so that one can achieve ideal connectivity, where by ideal connectivity we mean that packets are delivered as long as the network remains connected. Through some simple counterexamples, we have proven that the answer is no: if the forwarding state remains constant (*i.e.,* the control plane has not yet had a chance to recompute the forwarding state) one cannot always achieve ideal connectivity. We omit the formal statement and proof for lack of space, but the intuition is obvious: if you allow no state changes in the router (other than knowing which local links are up), and no rewriting of packet headers (as in [10, 9]), then there is no way that local state could compensate for any arbitrary set of connectivity-preserving failures. Given that this impossibility result precludes achieving ideal connectivity solely using the data plane, we must then ask: can we find a way to achieve ideal connectivity on a timescale much less than that of the control plane?

The timescale of the control plane is so large because it must deal with situations where changes far from a router will have an impact on its state (*e.g.,* it must check remote state to see if the shortest path has changed). However, one can ask if there is a much smaller class of forwarding state changes, ones that depend only on nearby information, that could support ideal connectivity. To this end, we propose the idea of *data-driven connectivity* (DDC), which maintains connectivity by allowing simple changes in forwarding state predicated only on the destination address and incoming port of an incoming packet. The DDC state changes we allow are those that are simple enough to be easily done at packet rates with revised hardware (and, in current routers, can be done quickly in software).

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

In this paper we apply this approach to the general problem of intradomain routing (whether it be datacenter, WAN, or enterprise) at either layer 2 or layer 3. The only assumption we make is that all forwarding decisions are made on exact-match lookups (whether it be over MAC addresses or IP addresses/prefixes).[2] In what follows, we will refer to addresses as the unit of exact match and nodes as the forwarding element (whether it be a switch or a router). Because of the exact match requirement, one can consider the forwarding state for each address independently, so we typically consider only how DDC updates the forwarding state for a single address (because the state change is driven by the arrival of a single packet, and it is that packet's destination address that determines which address' forwarding state will be updated).

In the next section (Section 2) we present the DDC algorithm for maintaining connectivity and state the theorem about its correctness. In Section 3 we describe a control plane algorithm for achieving optimal routes, load distribution, and disconnection detection. DDC does not depend on this particular control plane algorithm; to the contrary, the control plane algorithm leverages properties of the DDC algorithm. We then evaluate the performance of this combined system in Section 4.

Before beginning the description of our algorithm, we first say a few words about related work. We are not aware of any work that adopts the DDC paradigm we describe here (though we would be surprised if there were not some earlier attempts at something similar), but the algorithmic details of DDC borrow heavily from the algorithm developed by Gafni and Bertsekas [5], whose ideas influenced a series of subsequent designs (*e.g.,* [3, 12]) categorized as "link reversal routing". The control plane ideas we discuss are similar to those considered in [8, 13, 7, 11] in that they (and we) use Directed Acyclic Graphs (DAGs) as a way to avoid loops while increasing resilience. Also, as mentioned earlier, there are proposals [10, 9] for achieving ideal connectivity by having data packets carry control plane information, but here the state update operations are not simple, and cannot be done at data plane speeds.

2. DATA-DRIVEN CONNECTIVITY

DDC maintains connectivity via simple changes to local forwarding state, predicated only on the destination address and incoming port of an incoming packet. We first give a brief overview of how DDC works, followed by a more detailed description and a brief sketch of its correctness.

2.1 Overview

Consider a network modeled as an undirected graph $G = (N, E)$, where N is the set of nodes, and E is the set of links. In what follows, we only consider the forwarding state associated with a unique destination node v, and packets addressed to that destination. The forwarding state at each node other than v (which is initialized by the control plane) specifies which links

[2]This assumption is not essential to the correctness of our algorithm, but it removes the possibility that a change in one address' route causes a much larger rewriting of the forwarding state (as can happen in LPM), which is a slow process.

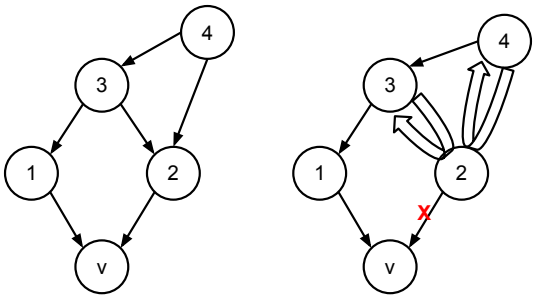

Figure 1: Illustration of DDC. (a) normal forwarding. (b) DDC bounce back when failure happens.

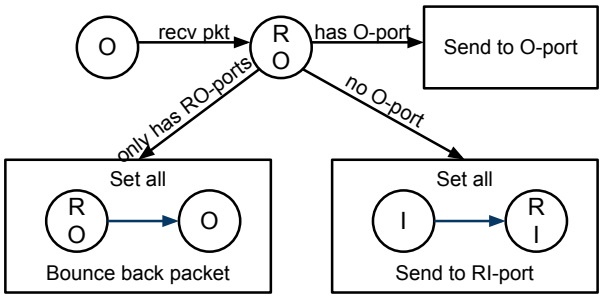

Figure 2: State transition when an O-port receives packet.

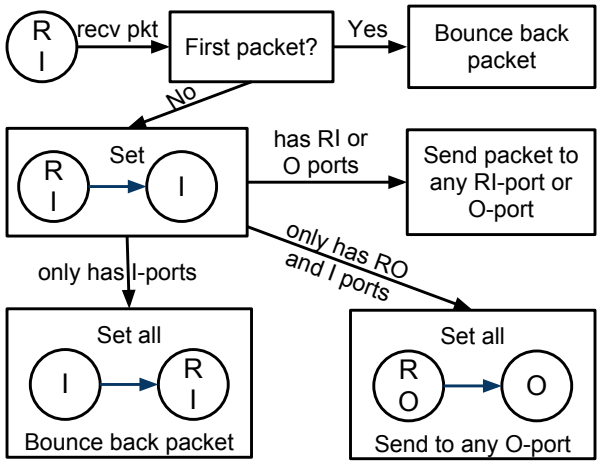

Figure 3: State transition when an RI-port receives packet.

it should use to reach v. Directed edges in Figure 1(a) illustrate the forwarding state in a simple example, *e.g.,* node 3 forwards packets (destined to node v) to either node 1 or node 2. If node i forwards packets through next-hop node j, we call link (i, j) an *outgoing link* for i, and an *incoming link* for j.[3] Let I_n and O_n be the sets of incoming and outgoing links (respectively) at node n. When node n's outgoing link fails, that link is immediately removed from n's set of outgoing links O_n. In Figure 1(a), $I_2 = \{3, 4\}$ and $O_2 = \{v\}$.

Intuitively, in DDC, a router should send out a received packet along an outgoing link as long as such a link exists, and "bounce back" the packet to the sending neighbor otherwise. To wit, when a packet destined to v arrives at node n, the following two steps are executed:

Update: If the packet arrived on an incoming link, no updates are needed. If it arrived on an outgoing link, remove that link from O_n and place it in I_n.
Forward: If O_n is not empty, forward the packet along one of the available outgoing links. If O_n is empty, forward packet back through the incoming link from which it arrived.

Consider Figure 1(b). When node 2 loses its outgoing link $(2, v)$ due to link failure it immediately removes the link from O_2, which then becomes empty. When a packet destined to v arrives at 2 from nodes 3 or 4, it sends it back through the same incoming link ("bouncing it back"). Once node 3 receives a packet from 2, it removes the outgoing link $(3, 2)$ from O_3 and send the packet through its other outgoing link $(3, 1)$. Similarly, once node 4 receives a packet from 2 it removes its outgoing link to 2 from O_4 and send that packet through its other outgoing link.

Note how node 2's "bounce back mechanism" prevents on-the-fly packets from being dropped, and that after 3 and 4 remove $(3, 2)$ and $(4, 2)$ from their outgoing link sets this bounce back is no longer needed. It is also obvious DDC works without global communication. Instead, nodes' actions under DDC are fast and simple, and are based on local information only. However, what if multiple nodes need to bounce back? We next extend the basic idea to a complete design.

[3]For convenience we assume that all links are either incoming or outgoing (as opposed to unused), as it makes our later state transition diagrams easier.

2.2 Design Details

Because forwarding state is local, we choose to refer to ports (which are local) rather than links (which are shared between the two endpoints). We define four types of ports: incoming ports (I-ports), outgoing ports (O-ports), reversed incoming ports (RI-ports), and reversed outgoing ports (RO-ports).

Initially, all ports are either incoming ports (I-ports), through which packets are received, or outgoing ports (O-ports) through which packets are sent out. In the example described in Figure 1, the port connecting node 3 to node 4 is an I-port whereas the port connecting node 3 to node 1 is an O-port. Thus, I-ports and O-ports capture "normal" behavior, and the network is initialized in a state where all ports are in one of these categories.

In the presence of failures, packets in DDC must occasionally be "bounced back"; to do so, the corresponding ports must be "reversed" so that packets can travel in the opposite direction than dictated by their original

I-port or O-port designation. To handle such "reversal" situations, we introduce the categories of reversed incoming ports (RI-ports), and reversed outgoing ports (RO-ports). At the DDC level, the following transitions are allowed: I to RI, RI to I, O to RO, and RO to O. The only way an I-port can become an O-port, or vice-versa, is through actions of the control plane (which we discuss in the next section). However, the DDC transitions are sufficient to guarantee connectivity. We now describe these transitions in greater depth, but forewarn the reader that space limitations preclude a full explanation of why they are sufficient.

The O-RO and I-RI transitions are simple: RI-ports are (originally) incoming ports along which the node "bounces back" an incoming packet, and RO-ports are (originally) outgoing ports along which the node receives an incoming packet. The transitions from RO to O, and RI to I, are more complicated and the associated state-diagrams are shown in Figures 2 and 3.

To illustrate these scenarios, consider a node i that has a single port which is an RO-port. When a packet arrives along this RO-port, i has no choice but to bounce it back through the same port, with the hope that the receiving neighbor j is able to forward the packet towards the destination. According to our rules (see Figure 2), i must make its single port an O-port again, hence the term "re-reverse". Observe that as i received the packet from j through an RO-port it must be that i's packet will reach j through j's RI-port. At this point j should, if possible, send to another RI-port or O-port. The transition is illustrated in Figure 3.

2.3 Ideal Connectivity

DDC achieves ideal connectivity, in the sense that so long as a set of network failures does not disconnect a node n from the destination v, packets from n are delivered to v. More specifically, we can prove the following:

THEOREM 1. *Let $\bar{G}$ be a network graph obtained from G via the removal of a subset of the links in E. For every node n that has some route to the destination v in $\bar{G}$ it holds that under DDC every packet sent from i to v is guaranteed to reach v.*

Observe that the fact that packets are never dropped follows immediately from our DDC forwarding rules (a packet is always either bounced back or forwarded through another port). To establish Theorem 1, we are left with proving that packets never enter endless loops, and thus always reach v eventually. Our proof of this claim is similar in spirit to the correctness proof for the algorithm in [5], and is omitted due to space constraints. The key idea is showing inductively that the claim holds for a gradually expanding set of nodes in v's connectivity component in $\bar{G}$.

3. CONTROL PLANE

DDC adjusts the forwarding state to maintain connectivity, but it does not ensure that the resulting paths are shortest paths, nor does it try to distribute the load or detect network disconnections. For these tasks we use a control plane algorithm called "Routing Along DAGs" (RAD). What we present here is just an example of how the control plane can be simplified once connectivity has been handled by DDC. The goal of RAD is to compute shortest path routes that are resilient in the sense that failures can often be repaired by merely taking an alternate outgoing port.

Optimizing routes, detecting disconnection:

Our RAD design can be viewed as an enhanced Distance Vector protocol which, instead of computing a routing tree, computes a Directed Acyclic Graph (DAG), so that a node may have multiple outgoing links to the destination. For every destination v, RAD simply computes the shortest path distance between each node n and v (given this metric) and sets n's set of outgoing links (along which traffic to v is forwarded) to be all links that point from n to a neighbor node whose distance is smaller. There is some arbitrary ordering $<$ of the nodes in N. So when two neighboring nodes, i and j, have the same distance from v, then i forwards traffic to j if $i < j$ (and vice versa). Observe that the resulting forwarding configuration is indeed loop-free and that every link in the network is used for forwarding (in a single direction). Once RAD is run, all ports are either I-ports or O-ports. RAD can be executed in a more incremental fashion, where each node periodically updates its distance estimate and adjusts the state of its ports (in cooperation with the peer nodes).

To detect nodes that are disconnected from a destination, each destination v periodically sends out a message with a monotonically increasing timestamp. All other nodes keep track of the time the last message they received from v was sent. In the event that the time elapsed since the last received message exceeds a certain threshold, the node will consider destination v unreachable.

Load Distribution:[4] A link is considered congested once its utilization level exceeds a certain threshold. When a node n's outgoing link becomes congested, n diverts traffic to uncongested outgoing links, if such outgoing links exist (this applies to all flows traversing that link and so the node has the freedom to decide for which specific flows to divert traffic).

When all of a node n's outgoing links for a certain destination are congested, n sends a congestion signal to one or more of its neighbors. Upon receipt of this

[4]Note that we do not attempt to minimize maximum utilization (which is what we call *load balancing*) Our aim is merely to avoid congestion (which is what we call *load distribution*).

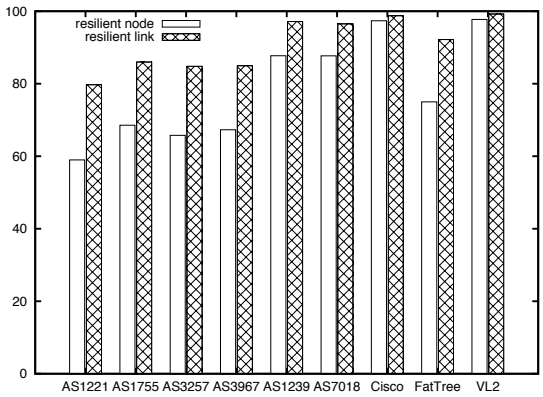

Figure 4: Percentages of resilient nodes and redundant links in DAGs over various topologies

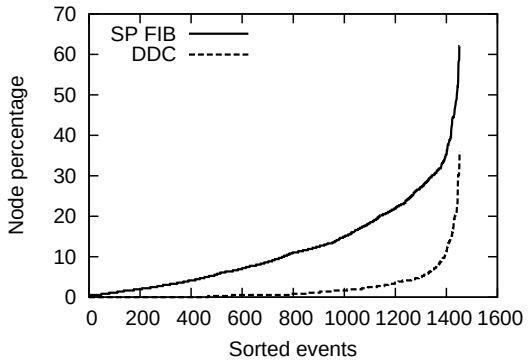

Figure 5: Scope of Node Reversal and FIB changes on AS1239

message, the neighbor will decrease its transmission rate along the link to n by a certain predetermined amount, and treat this link to n as congested.

4. EVALUATION

We now evaluate our design via simulation. We feel that the DDC paradigm, and its rigorously established connectivity properties, represent the bulk of our contribution. The material presented here merely illustrates some of the performance properties of DDC and the control plane algorithm RAD.

For our simulations we used 6 ISP topologies and 3 datacenter topologies. The size of ISP topology varies from small (AS1221, 83 nodes and 131 links) to large (AS1239, 361 nodes and 1479 links). The three datacenter topologies —*Cisco* (a 3-tier hierarchical topology recommended by Cisco [2]) FatTree [1] and VL2 [6] — are highly symmetric. Due to our limited space, most of our results are shown for only a subset of the topologies, but the results on the other topologies are qualitatively similar.

4.1 Evaluation of DDC

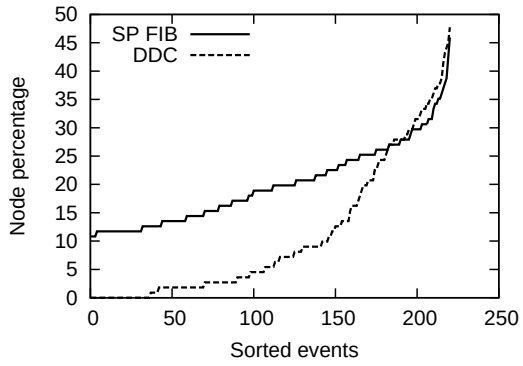

Figure 6: Scope of Node Reversal and FIB changes on AS1755

Outdegree and local resilience The first question we ask is: how often can a node recover locally from a single link failure by simply using another outgoing link?[5] Figure 4 shows, for each topology and averaged over all DAGs (*i.e.,* a separate DAG per destination), the percentage of resilient nodes (outdegree bigger than one), and "redundant" links (whose failure does not cause a link reversal). In the two larger ISP topologies (AS1239 and AS7018), over 90% of the nodes are resilient and over 95% of the links are redundant. In the other four ISP topologies, the resilient node percentage is between 60% and 65%, while the redundant link percentage is roughly 80%. The datacenter topologies Cisco and VL2 have over 97% resilient nodes and redundant links. In contrast, FatTree has only about 70% resilient nodes, mainly because in any particular FatTree pod all aggregation layer switches have outdegree one to each destination edge switch.

Scope of failure recovery. We now investigate the impact of single failures in a different way; when a link fails, how many nodes need to respond? For DDC we look at the "reversals" (*i.e.,* cases where an incoming link had to become an outgoing link based on the data-driven state changes described in Section 2); in particular, we count how many nodes had to reverse a link (and call it a "node reversal"). To benchmark these results, we compare against shortest-path routing, where we measure the number of nodes whose outgoing port changed (we call this a change in the FIB).[6]

Distributions of failure recovery scope is shown in Figure 5-6, where the link events are ordered in terms of the percentage of responding nodes. On the large ISP 1239, most events created very few node reversals,

[5]In answering this question, we ignore the nodes that are physically connected by a single link, because there is no way routing can help them. We know that there must be some occasions when a single failure disconnects a node (and forces it to initiate a reversal) because for each DAG there is at least one node with outdegree one.

[6]We also tested up to 10 concurrent link failures, in which DDC retained its smaller scope.

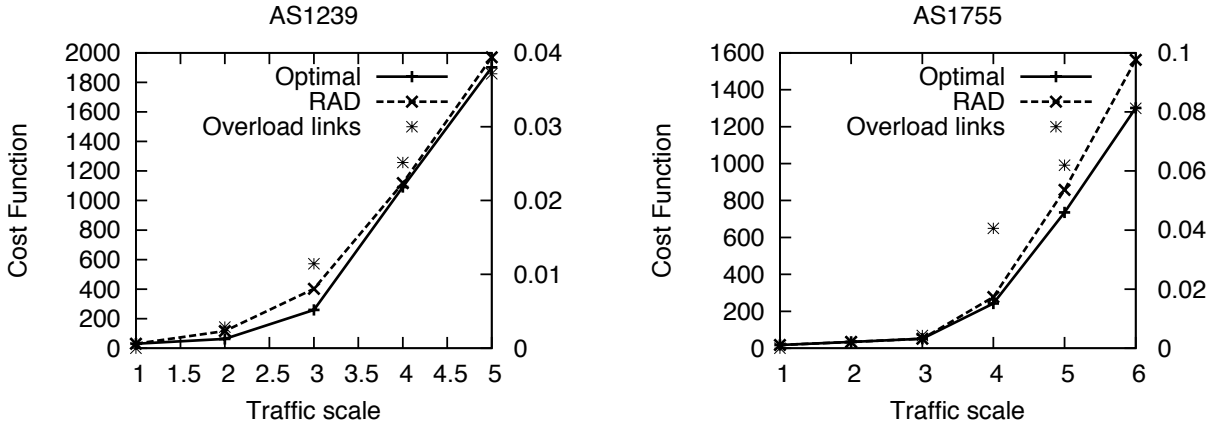

Figure 7: Comparison of RAD to the optimal linear programming solution. The dots (using the right hand scale) indicate the fraction of links with over 90% utilization in the optimal solution (which indicate how heavily loaded the network is).

but the tail of the distribution had over 30% of the nodes responding. In contrast, shortest path routing had a high fraction of FIB changes for all events. For the smaller ISP 1755, the distribution of node reversals increases more gradually, and is smaller than shortest path in 75% of failures.

We also investigated the stretch of the path that results after DDC has restored connectivity but before the control plane has optimized the path. For AS1755 the stretch is distributed between 1 and 1.4, with an average of 1.15. Note that the path stretch of DDC is only temporary, because when the control plane updates forwarding entries, the new shortest path will be made available.

4.2 Load Distribution.

We evaluate RAD's ability to distribute load by comparing it to the optimal load balancing algorithm in a linear programming model from the traffic engineering literature (*e.g.,* [4]). This model assigns a penalty for various usage levels on each link, with the penalties rising steeply as the link becomes overloaded; links are allowed to carry more than 100% of their capacity (but at a heavy penalty). The goal is to minimize the total penalty.

Note that RAD was designed merely to distribute load when links get overloaded and is not trying to optimize any given utility function; for instance, RAD does not try to balance the load on two outgoing links as long as neither is overloaded. Nonetheless, we ran several experiments to measure how close to optimal RAD's load distributions are. Figure 7 shows the results on two ISP topologies. On the AS1239 topology, RAD tracks the optimal solution quite well. It similarly does well on the AS1755 topology, although under high loads (when over 8% of the links are over 90% utilized) the gap becomes significant.

5. REFERENCES

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

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

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

[4] B. Fortz and M. Thorup. Increasing internet capacity using local search. *Computational Optimization and Applications*, 29(1):13–48, 2004.

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

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] 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.