May 2008

Always Acyclic Distributed Path Computation

Saikat Ray
University of Bridgeport

Roch A. Guérin
University of Pennsylvania, guerin@acm.org

Kin-Wah (Eric) Kwong
University of Pennsylvania, kkw@seas.upenn.edu

Rute Sofia
INESC Porto

Always Acyclic Distributed Path Computation

Abstract
Distributed routing algorithms may give rise to transient loops during path recomputation, which can pose significant stability problems in high-speed networks. We present a new algorithm, Distributed Path Computation with Intermediate Variables (DIV), which can be combined with any distributed routing algorithm to guarantee that the directed graph induced by the routing decisions remains acyclic at all times. The key contribution of DIV, besides its ability to operate with any routing algorithm, is an update mechanism using simple message exchanges between neighboring nodes that guarantees loop-freedom at all times. DIV provably outperforms existing loop-prevention algorithms in several key metrics such as frequency of synchronous updates and the ability to maintain paths during transitions. Simulation results quantifying these gains in the context of shortest path routing are presented. In addition, DIV's universal applicability is illustrated by studying its use with a routing that operates according to a non-shortest path objective. Specifically, the routing seeks robustness against failures by maximizing the number of next-hops available at each node for each destination.

Keywords
Routing, Network, Loop-Free

Comments
University of Pennsylvania Department of Electrical and Systems Engineering Technical Report, May 2008.

Always Acyclic Distributed Path Computation

Saikat Ray, Roch Guérin, Kin-Wah Kwong, and Rute Sofia

***Abstract*—Distributed routing algorithms may give rise to transient loops during path recomputation, which can pose significant stability problems in high-speed networks. We present a new algorithm, D*istributed Path Computation with* I*ntermediate* V*ariables* (DIV), which can be combined with *any* distributed routing algorithm to guarantee that the directed graph induced by the routing decisions remains acyclic at all times. The key contribution of DIV, besides its ability to operate with any routing algorithm, is an update mechanism using simple message exchanges between neighboring nodes that guarantees loop-freedom at all times. DIV provably outperforms existing loop-prevention algorithms in several key metrics such as frequency of synchronous updates and the ability to maintain paths during transitions. Simulation results quantifying these gains in the context of shortest path routing are presented. In addition, DIV's universal applicability is illustrated by studying its use with a routing that operates according to a non-shortest path objective. Specifically, the routing seeks robustness against failures by maximizing the number of next-hops available at each node for each destination.**

***Index Terms*—Loop-free routing, Distance-vector routing.**

I. Introduction

Distributed path computation is a core functionality of modern communication networks and is expected to remain so, even though some recent proposals contemplate the use of more centralized solutions [1]. Depending on the mode of information dissemination, and subsequent computation using the disseminated information, there are two broad classes of algorithms: (i) link-state algorithms (also known as topology broadcast) and (ii) distance-vector algorithms [2]. In both approaches, nodes choose successor (next-hop) nodes for each destination based only on local information, with the objective that the chosen paths to the destination be *efficient* in an appropriate sense—*e.g.*, having the minimum cost. Because end-to-end paths are formed by concatenating computational results at individual nodes, achieving a global objective implies *consistency* across nodes both in computation and in the information on which those computations are based.

Inconsistent information at different nodes can have dire consequences that extend beyond not achieving the desired efficiency. Of particular significance is the possible formation of transient routing loops[1], which can severely impact network performance, especially in networks with no or limited loop mitigation mechanisms, *e.g.*, no Time-to-Live (TTL) field in packet headers or a TTL set to a large value. In the presence of a routing loop, a packet caught in the loop comes back to the same nodes repeatedly, thereby artificially increasing the traffic load many folds on the affected links and nodes. The problem is aggravated by broadcast packets, which not only are always caught in any loop present in the network, but also generate replicated packets on all network links. The emergence of a routing loop then often triggers network-wide congestion, which can lead to the dropping or delaying of the very same control (update) packets that are needed to terminate the loop; thereby creating a situation where a transient problem has a lasting effect. Avoiding transient routing loops remains a key requirement for path computation in both existing and emerging network technologies, *e.g.*, see [3–5] for recent discussions.

Link-state algorithms, of which the OSPF [6] protocol is a well-known embodiment, disseminate the state of each node's local links (their status and the node(s) they connect to) to all other nodes in the network by means of reliable flooding. After receiving link-state updates from the rest of the nodes, each node independently computes a path to every destination. The period of potential information inconsistency across nodes is small (a few 10's of milli-seconds per node for typical present day networks [7]), so that routing loops, if any, are very short-lived. On the flip side, link-state algorithms can have quite high overhead in terms of communication (broadcasting updates), storage (maintaining a full network map), and computation (a change anywhere in the network triggers computations at all nodes). These are some of the reasons for investigating alternatives as embodied in distance-vector algorithms, which are the focus of this paper.

Distance-vector algorithms couple information dissemination and computation. Information disseminated by a node now consists of the results of its own partial path computations (*e.g.*, its current estimate of its cost to a given destination) that it distributes to its neighbors, which in turn perform their own computations before further propagating any updated results to their own neighbors. The Distributed Bellman-Ford (DBF) algorithm is a well-known example of a widely used distance-vector algorithm (cf. RIP [8], EIGRP [9]) that computes a shortest path tree from a given node to all other nodes. Coupling information dissemination and computation can reduce storage requirements (only routing information is stored), communication overhead (no relaying of flooded packets), and computations (a local change needs not propagate beyond the affected neighborhood). Thus, distance-vector algorithms avoid several of the disadvantages of link-state algorithms, which can make them attractive, especially in situations of frequent local topology changes and/or when high control overhead is undesirable.

The down side of coupling information dissemination and computation is that information dissemination is gated by

Saikat Ray (saikatr@bridgeport.edu) is with the Department of Electrical & Computer Engineering, University of Bridgeport.

Roch Guérin and Kin-Wah Kwong are with the Department of Electrical and Systems Engineering, University of Pennsylvania.

Rute Sofia is with INESC Porto.

Research supported in part by a gift to the University of Pennsylvania by Siemens AG, Corporate Technology, Munich, and by NSF grant CNS-0627004. Part of this work was first presented at ITC'20, Ottawa, Canada, June 2007.

[1]In this paper, the term "routing" refers to the path computation process that creates the tables used in forwarding packets, irrespective of the layer (layer 2 or layer 3) at which this forwarding takes place. Hence, routing loops are simply loops in the resulting forwarding graph, regardless of the layer where they reside.

computation speed since a node cannot send updates before finishing its current computations. This can in turn extend periods when nodes have inconsistent information, which, as discussed earlier and illustrated in Section V, can lead to more frequent and longer lasting routing loops. In addition, coupling information dissemination and computation can also result in slower convergence. This is because each node depends on the (partial) computation results of its neighbors, which can introduce cyclic dependencies that increase the number of steps needed to reach a final, correct result. Indeed, when destinations become unreachable, a distance-vector algorithm may not even converge in a finite number of steps. This is known as the *counting-to-infinity* problem, which is absent from link-state algorithms where nodes compute paths independently. (In practice, when the cost-to-destination reaches a maximum value, the destination is declared unreachable and the computation is terminated.)

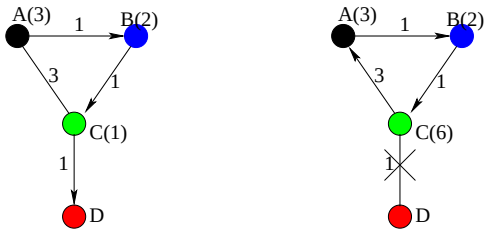
Thus, we see that realizing the benefits of distance-vector based solutions, even in environments where they might be a natural fit, calls for developing approaches to overcome these problems. Such a realization is not new. Since the 70's, several works [10–15] have targeted this goal in the context of shortest path computations. We review and contrast the most significant of these prior works in Section II, but the problem remains timely. Our research was triggered by a renewed interest in devising light-weight, loop-free path computation solutions for large-scale Ethernet networks. Specifically, we were considering extending the scalability of Ethernet networks through the introduction of distributed shortest path algorithms in lieu of the existing distributed spanning tree algorithm (see [3, 16] for similarly motivated efforts). Link-state based solutions have been proposed [17, 18] to improve Ethernet networks, although not necessarily for the sake of scalability, and a distance-vector solution seemed an attractive alternative.

In this paper, we introduce the D*istributed Path Computation with* I*ntermediate* V*ariables* (DIV) algorithm that enables our goals of distributed, light-weight, loop-free path computation. DIV is *not* by itself a routing algorithm; rather, it can run on top of any routing algorithm to provide loop-freedom. DIV generalizes the *Loop Free Invariant* (LFI) based algorithms [14, 15] and outperforms previous solutions including known LFI and *Diffusing Computation* based algorithms, such as the *Diffusing Update Algorithm* [13]. The main advantages of DIV are as follows:

1) *Separation of Routing and Loop prevention*: DIV separates routing algorithms from the task of transient loop prevention. Emancipating routing decisions from the task of loop-prevention simplifies routing algorithms. In addition, DIV is not restricted to shortest path computations; it can be integrated with other distributed path computation algorithms. We illustrate this in Section IV, where we explore a routing algorithm that attempts to increase the *robustness* of the network in terms of being able to re-route packets *immediately* (*i.e.*, without the need for any route update) without causing a loop after a link or node failure.
2) *Reduced Overhead*: When applied to shortest path computations, DIV triggers synchronous updates less frequently as well as reduces the propagation radius of synchronous updates (cf. Theorem III.5), where synchronous updates are time and resource consuming updates that might need to propagate to all upstream[2] nodes before the originator is in a position to update its path. In fact, synchronous updates may altogether be removed if counting-to-infinity is not a significant issue (*e.g.*, mitigated using a TTL); cf. Section III-B3c, alternate mode.
3) *Maintaining a path*: A node can potentially switch to a new successor without forming a loop provably more quickly (cf. Section III-B3c, alternate mode) in DIV. This is particularly useful in situations where the original path is lost due to a link failure.
4) *Convergence Time*: When a node receives multiple overlapping cost updates[3] from its neighbor, DIV allows the node to process and respond to the updates in an arbitrary manner, thus enabling an additional dimension for optimization (cf. Theorem III.2).
5) *Robustness*: DIV can tolerate arbitrary packet reordering and losses without sacrificing correctness. (cf. Theorem III.3).

The rules and update mechanism of DIV and their correctness proofs are rather simple, which hopefully will also facilitate correct and efficient implementations.

The rest of this paper is organized as follows. We survey important related works in Section II. DIV is described in Section III along with its properties. The applicability of DIV to robust routing is investigated in Section IV. Section V presents comparative simulation results to quantify various measures of performance, and conclusions are drawn in Section VI.



(a) Original topology. (b) Node D is down.

Fig. 1. A simple example of counting-to-infinity problem.

II. Background

A. Routing Loops and Counting-to-Infinity

We begin our discussion with a simple classical example of a routing loop and counting-to-infinity which illustrates that these problems can occur quite frequently as they neither require complex topologies nor an unlikely sequence of events.

Consider the network shown in Fig. 1(a). In this figure, the nodes compute a shortest path to the destination D. The cost of each link is shown next to the link and the cost-to-destination of the nodes are shown in parenthesis next to the node. We

[2]Upstream nodes of a node x for a given destination z are the nodes whose path to z includes x.

[3]Two updates are overlapping if the latter appears before the algorithm has converged in response to the first.

assume that nodes use *poison reverse*; *i.e.,* each node reports an infinite cost-to-destination to its successor node [19]. Thus, node C believes that node A can reach the destination at a cost of 3 whereas node B cannot reach the destination since node B reported a distance of infinity to node C.

Now suppose that the link between nodes C and D goes down, as shown in Fig. 1(b). Node C detects this change and attempts to find a new successor. According to the information node C has at that moment, node A is its best successor. So node C chooses node A as its successor, reports a distance of infinity to node A and distance of 6 to node B. As Fig. 1(b) shows, a routing loop has been created due to node C's choice of successor.

To see how counting-to-infinity takes place in this example, note that due to poison reverse, node B believes that the destination is unreachable through node A. Thus when it receives the update from C containing C's new cost-to-destination as 6, node B simply changes its own cost-to-destination to 7 keeping node C as its successor, reports unreachability to node C and its new cost, 7, to node A. This way, each node increases its cost to D by a finite amount each time. So, unless a maximum diameter of the graph is assumed (*e.g.,* it is 16 in RIP) and the destination declared unreachable once the cost reaches that value, the computation never ends.

This simple example illustrates how easily a routing loop and counting-to-infinity can occur unless special care is taken (cf. Section V and [20]). Note that simple solutions such as *Split Horizon* or *Poison Reverse* do not help in this example [19]. Thus, several previous works have proposed more comprehensive solutions to the routing loop problem; next we survey some of the most salient ones among them.

B. Previous Works

1) The Common Structure: Most previous distance-vector type algorithms free from transient loops follow a common structure: Nodes exchange update-messages to notify their neighbors of any change in their own cost-to-destination (for any destination). If the cost-to-destination decreases at a node, the algorithms allow updating its neighbors in an arbitrary manner; these updates are called *local* (asynchronous) updates. However, after an increase in the cost-to-destination of a node, these algorithms require that the node potentially update all its upstream nodes *before* changing its current successor; these are *synchronous* updates. The algorithms differ in handling the situations where during the propagation of a node's cost-to-destination update to its upstream nodes, its cost-to-destination changes.

Note that the primary challenge in avoiding transient loops lies in handling inconsistencies in the information stored across different nodes. Otherwise, simple approaches can guarantee loop-free operations at each step [10]. In this context, approaches that are "in-between" link-state and distance vector and avoid counting-to-infinity are also possible; *e.g.,* [21] achieves this by having nodes learn the penultimate nodes in the shortest paths to each destination from its neighbors.

The algorithm proposed in [11] follows the above broad structure and is one of the earliest work in guaranteeing loop-free operations with inconsistent information. For handling multiple overlapping updates, it relies on unbounded sequence numbers that mark update epochs. An improvement to this algorithm is presented in [12]. For handling multiple overlapping updates, [12] maintains *bit vectors* at each node.

2) Diffusing Update Algorithm (DUAL): DUAL, a part of CISCO's widely used EIGRP protocol, is perhaps the best known algorithm. In DUAL, each node maintains, for each destination, a set of neighbors called the *feasible successor set*. The feasible successor set is computed using a *feasibility condition* involving *feasible distances* at a node. Several feasibility conditions are proposed in [13] that are all tightly coupled to the computation of a shortest path. For example, *Source Node Condition* (SNC) uses the feasible successor set to be the set of all neighbors whose current cost-to-destination is *less* than the minimum cost-to-destination seen so far by the node. Note that the definition of a feasible successor set depend on an origin of time, which is defined as the time when the node freshly computes the feasible successor set after it contains no preferred successor.

A node can choose any neighbor in the feasible successor set to be the successor (next-hop) without causing a routing loop regardless of how other nodes in the network choose their successors, as long as they also comply with this rule. Thus, the importance of the notion of feasible successor set lies in the fact that it allows nodes to switch neighbors immediately in response to a cost-increase without creating transient loops, and without the need for notifying any of its neighbors.

If the neighbor through which the cost-to-destination of the node is minimum is in the feasible successor set, then that neighbor is chosen as the successor. If the current feasible successor set does not include the best successor, the node initiates a synchronous update procedure, known as a *diffusing computation* (cf. [22]). The node sends queries to all its neighbors with its cost-to-destination through the current successor. From this point onwards the node does not change its successor until the diffusing computation terminates. Each neighbor replies to the query by sending their own cost-to-destination if they themselves have a feasible successor after they update the set following the new information received from the initiator node. Otherwise, they themselves send out queries and wait for the replies before replying to the original query. One easily sees that the queries propagate upstream in a recursive manner and are equivalent to notifying all upstream nodes about the new status of the originator node. Finally if there are multiple overlapping updates—*i.e.,* if a new link-cost change occurs when a node is waiting for replies to a previous query—the node uses a *finite state machine* to process these multiple updates sequentially.

3) Loop Free Invariance *(LFI) Algorithms:* A pair of invariances, based on the cost-to-destination of a node and its neighbors, called *Loop Free Invariances* (LFI) are introduced in [14] and it is shown that if nodes maintain these invariances, then no transient loops can form (cf. Section III-B2). Update mechanisms are required to maintain the LFI conditions: [14] introduces *Multiple-path Partial-topology Dissemination Algorithm* (MPDA) that uses a link-state type approach whereas [15] introduces *Multipath Distance Vector Algorithm* (MDVA) that uses a distance vector type approach. Similar to DUAL,

MDVA uses a diffusing update approach to increase its cost-to-destination, thus it also handles multiple overlapping cost-changes sequentially.

4) Comparative Merits of Previous Algorithms: The Jaffe-Moss algorithm [12] improves upon the earlier Merlin-Segall algorithm [11], however, with one potential problem that the bit-vectors it maintains to handle multiple overlapping updates can be exceedingly large for large dynamic networks. DUAL avoids large bit-vectors by not processing multiple overlapping updates simultaneously, rather processing them in a sequential manner by maintaining a finite state machine at each node. In terms of performance, DUAL supersedes the other two. The primary contribution of LFI based algorithms such as MDVA or MPDA is a unified framework applicable to both link-state and distance-vector type approaches and multipath routing. However, in DUAL with, say, SNC, many link-cost changes do not violate the feasibility condition, and therefore do not trigger synchronized updates—an important advantage over MDVA or MPDA. Because of the importance of this metric, we consider DUAL the benchmark against which to compare new solutions, and compare DIV with DUAL in Section V.

DIV combines advantages of both DUAL and LFI. DIV generalizes the LFI conditions, is not restricted to shortest path computations and, as LFI-based algorithms, allows for multipath routing. In addition, DIV allows for using a feasibility condition that is strictly more relaxed than that of DUAL, hence triggering synchronous updates less frequently than DUAL (and consequently, than MPDA or MDVA) as well as limiting the propagation of any triggered synchronous updates. The update mechanism of DIV is simple and substantially different from that of previous algorithms, and allows arbitrary packet reordering/losses. Moreover, unlike DUAL or LFI algorithms, DIV handles multiple overlapping cost-changes *simultaneously* without additional efforts resulting in potentially faster convergence. Finally, DIV allows an alternate synchronous update mode (in distance vector computations) where a synchronous update goes only one hop, similar to MPDA (note though that MPDA is link-state based), which allows nodes to switch to a new successor faster without creating loops.

III. DIV

A. Overview

DIV lays down a set of rules on existing routing algorithms to ensure their loop-free operation at each instant. This rule-set is not predicated on shortest path computation, so DIV can be used with other path computation algorithms as well.

For each destination, DIV assigns a *value* to each node in the network. To simplify our discussion and notation, we fix a particular destination and from now on, speak of *the* value of a node. The values can be arbitrary—hence the independence of DIV from any underlying path computation algorithm. However, usually the value of a node will be related to the underlying objective function that the routing algorithm attempts to optimize and the network topology. Some typical value assignments are as follows: (i) in shortest path computations, the value of a node could be its cost-to-destination; (ii) as done in DUAL, the value could be the minimum cost-to-destination seen by the node from time $t = 0$; (iii) as done in TORA [23], the value could be the *height* of this node; (iv) as illustrated in Section IV, the value could be related to the number of next-hop neighbors for the destination, etc. We, however, impose one restriction on the value assignment: a node that does not have a path to a destination must assign a value of "infinity" (the maximum possible value) to itself. Intuitively, this restriction prevents other nodes from using it as a successor which is sensible since it does not have a path to the destination in the first place. This restriction turns out to be crucial for avoiding counting-to-infinity problems in shortest path environments.

The basic idea of DIV is that it allows a node to choose one of its neighbor as a successor only if the value of that neighbor is less than its own value: this is called the *decreasing value property* of DIV. This ensures that no routing loop can ever form.

The hard part is enforcing the decreasing value property. In particular, as the network topology changes, if the values of the nodes are held fixed, then the routing protocol may not be able to choose appropriate successors; *e.g.*, if a neighbor happens to be the only path to a destination, but with a higher value, then the node will not be able to reach that destination. Thus, node values must be updated in accordance to topological changes. However, how does a node then know the *current* value of one of its neighbors to ensure the decreasing value property? Clearly, each node must update its neighbors about its own current value by means of update messages. Since update messages are asynchronous, information at various nodes may be inconsistent and may lead to the formation of loops. This is where the non-triviality of DIV lies: it lays down specific update rules that guarantee that loops are not formed at any time even if the information at different nodes is inconsistent. DIV accomplishes this task by maintaining several intermediate variables that hold a replica of the value of a node at its neighbors and vice versa, and exchanging messages between neighboring nodes. Similar to (but not identical with) DUAL, the update mechanism sends update messages and for some of them, requires an acknowledgment from the neighbor. Depending on the rules for sending acknowledgments, DIV can be operated in one of the following two modes: (i) the *normal mode*, and (ii) the *alternate mode*. In the normal mode, a neighbor can hold on to sending an acknowledgment until it's own value is adjusted appropriately. In the alternate mode, on the other hand, the neighbor immediately sends the acknowledgment, but could temporarily lose all paths (to that particular destination). As we discuss later on, each mode embodies a different trade-off.

B. Description of DIV

There are four aspects to DIV: (i) the variables stored at the nodes, (ii) two ordering invariances that each node maintains, (iii) the rules for updating the variables, and (iv) two semantics for handling non-ideal message deliveries (such as packet loss or reordering). A separate instance of DIV is run for each destination, and we focus on a particular destination, which at a node is, therefore, associated with a given value.

1) The Intermediate Variables: Suppose that a node x is a neighbor of node y. These two nodes maintain intermediate variables to track each other's values. There are three aspects to each of these variables: whose value is this? who believes in that value? and where is it stored? Accordingly, we define $V(x;y|x)$ to be the value of node x as known (believed) by node y stored in node x; similarly $V(y;x|x)$ denotes value of node y as known by node x stored in node x.

Thus, assuming node x has n neighbors, $\{y_1, y_2, \ldots, y_n\}$, it stores, for each destination:

1) its own value, $V(x;x|x)$;
2) the values of its neighbors as known to itself, $V(y_i;x|x)$ $[y_i \in \{y_1, y_2, \ldots, y_n\}]$,
3) and the value of itself as known to its neighbors $V(x;y_i|x)$ $[y_i \in \{y_1, y_2, \ldots, y_n\}]$.

That is, $2n+1$ values for each destination. The variables $V(y_i;x|x)$ and $V(x;y_i|x)$ are called intermediate variables since they endeavor to reflect the values $V(y_i;y_i|y_i)$ and $V(x;x|x)$, respectively. In steady state, DIV ensures that $V(x;x|x) = V(x;y_i|x) = V(x;y_i|y_i)$.

2) The Invariances: As stated in the overview of DIV (cf. Section III-A), the fundamental idea of DIV is to ensure that a successor node always has a smaller value. However, a node may not know what the most recent value of one of its neighbors is due to inconsistency in information. Thus, DIV requires each node to maintain at all times the following two invariances based on its set of *locally stored variables.*

Invariance 1 *The value of a node is not allowed to be more than the value the node thinks is known to its neighbors. That is,*

$$V(x;x|x) \leq V(x;y_i|x) \quad \textit{for each neighbor } y_i. \tag{1}$$

Invariance 2 *A node x can choose one of its neighbors y as a successor only if the value of y is less than the value of x* as known by node x*;* i.e., *if node y is the successor of node x, then*

$$V(x;x|x) > V(y;x|x). \tag{2}$$

Thus, due to Invariance 2, a node x can choose a successor only from its *feasible successor set* $\{y_i|V(x;x|x) > V(y_i;x|x)\}$. The two invariances reduce to the LFI conditions if the value of a node is chosen to be its current cost-to-destination.

3) Update Messages and Corresponding Rules: There are two operations that a node needs to perform in response to network changes: (i) decreasing its value and (ii) increasing its value. Both operations need notifying neighboring nodes about the new value of the node. DIV uses two corresponding update messages, Update::Dec and Update::Inc, and acknowledgment (ACK) messages in response to Update::Inc (no ACKs are needed for Update::Dec). Both Update::Dec and Update::Inc contain the new value, (the destination), and a sequence number[4]. The ACKs contain the sequence number and the value (and the destination) of the corresponding Update::Inc message.

[4]For simplicity, sequence numbers are assumed to be large enough so that sequence number rollover is not an issue.

DIV lays down precise rules for exchanging and handling these messages which we now describe.

a) Decreasing Value: Decreasing value is the simpler operation among the two. The following rules are used to decrease the value of a node x to a new value V_0:

- Node x first simultaneously decreases the variables $V(x;x|x)$ and the values $V(x;y_i|x)$ $\forall i = 1, 2, \ldots, n$, to V_0,
- Node x then sends an Update::Dec message to all its neighbors that contains the new value V_0.
- Each neighbor y_i of x that receives an Update::Dec message containing V_0 as the new value updates $V(x;y_i|y_i)$ to V_0.

b) Increasing Value: Increasing value is potentially a more complex operation, however, conceptually it is simply an inverse operation: in the decrease operation a node first decreases its value and then notifies its neighbors; in the increase operation, a node first notifies its neighbors (and waits for their acknowledgments) and then increases its value. In particular, a node x uses the following rules to increase its value to V_1:

- Node x first sends an Update::Inc message to all its neighbors.
- Each neighbor y_i of x that receives an Update::Inc message sends an acknowledgment message (ACK) when it is able to do so according to the rules explained in details below (Section III-B3c). When y_i is ready to send the ACK, it first modifies $V(x;y_i|y_i)$, changes successor if necessary (since the feasible successor set may change), and then sends the ACK to x; the ACK contains the sequence number of the corresponding Update::Inc message and the new value of $V(x;y_i|y_i)$. Note that in this case it is essential that node y_i changes successor, if necessary, *before* sending the ACK.
- When node x receives an ACK from its neighbor y_i, it modifies $V(x;y_i|x)$ to V_1. At any time, node x can choose any value $V(x;x|x) \leq V(x;y_i|x), \forall i = 1, 2, \ldots, n$.

c) Rules for Sending Acknowledgment: The Two Modes: We now describe how a node decides whether it can send an ACK in response to an Update::Inc message. There are two possibilities: each possibility leads to a distinct behavior of the algorithm, which we refer to as modes.

Suppose that node y_i received an Update::Inc message from node x. Recall that node y_i must increase $V(x;y_i|y_i)$ before sending an ACK. However, increasing $V(x;y_i|y_i)$ may remove node x from the feasible successor set at node y_i. If node x is the only preferred node in the feasible successor set of node y_i, then node y_i may lose its path if $V(x;y_i|y_i)$ is increased without first increasing $V(y_i;y_i|y_i)$. In such a case node y_i has two options: (i) first increase $V(y_i;y_i|y_i)$ and then increase $V(x;y_i|y_i)$ and send the ACK to node x, or (ii) increase $V(x;y_i|y_i)$, send ACK to node x and then increase $V(y_i;y_i|y_i)$. If a node uses option (i), we say that DIV is operating in its *normal mode*; if a node uses option (ii), we say that DIV is operating in *alternate mode*.

In the normal mode (*i.e.*, using option (i)), update requests

propagate to upstream nodes in the same manner as in DUAL and other previous works. If node x is not the sole desirable successor of node y_i, then node y_i will immediately respond to node x's Update::Inc message. Otherwise, node y_i wants to increase its own value before sending an ACK. Node y_i issues its own Update::Inc message to *all* its neighbors, including node x. The set of neighbors of node y_i that do not depend on node y_i for reaching the destination *based on their current values* (which includes node x), would immediately respond to node y_i with an ACK. When node y_i receives ACKs (in response to node y_i's Update::Inc message) from all its neighbors, it will send ACK to node x (as a response to node x's Update::Inc message). This process terminates due to acyclicity of the successor graph; when the node y_i is a "leaf" node (i.e., a node that is not a downstream node for any node), all its neighbors will immediately respond with an ACK.

At this point, we pause to briefly discuss a few basic aspects of "protocol machinery" associated with waiting for ACKs at a node. We assume the existence of a separate "liveness" protocol operating between neighbors and used to detect link/node failures. When waiting for ACKs from neighbors, node x maintains a list of pending ACKs for each Update::Inc message it is keeping track of. Nodes are removed from the list either upon receipt of the intended ACK or upon notification of the failure of the liveness protocol to the node. A timer is associated with pending ACKs and retransmission of the Update::Inc message is performed when the timer expires. After a given number of unsuccessful retransmission attempts, the node declares its session to the neighbor failed irrespective of the current status of the liveness protocol, and proceeds to re-initialize it. By following these simple rules, we can ensure that the transmission of ACKs proceeds unimpeded even in the presence of losses and node/link failures.

Turning next to the alternate mode (*i.e.,* using option (ii)), we trade simpler and faster processing of ACKs for the risk of having node y_i without a successor for a period of time (until it is allowed to increase its value). At a first glance, this may seem unwise. However, if node x originated the value-increase request in the first place because the link to its successor was *down* (as opposed to only a finite cost change), then the old path does not exist and the normal mode has no advantage over the alternate mode in terms of maintaining *a* path. In fact, in the alternate mode, the downstream nodes get ACKs from their neighbors more quickly and thus can switch earlier to a new successor (which hopefully has a valid path) than in the normal mode.

It is not necessary for all nodes to use the same mode (either normal or alternate); each node can make an independent selection. In particular, the following scheme can be used to choose the best mode: we include a bit in the Update::Inc message that indicates whether the request is in response to a loss of old path. Then an intuitive strategy is that nodes use the normal mode if the old path exists, and use the alternate mode if the old path does not exist. However, the normal mode does have one significant advantage over the alternate mode: the counting-to-infinity problem cannot arise in the normal mode whereas there is no such guarantee in the alternate mode. Thus the previous strategy is perhaps useful only in the cases where counting-to-infinity is not a significant problem or a mitigation mechanism is in place.

d) Semantics for Handling Message Reordering: So far we have been working under the implicit assumption that all update messages and ACKs come to a node in order and without any loss. In practice both loss and reordering are possible. Thus it is important to ensure the correctness of DIV under possible packet reordering or packet losses.

Towards this goal, we maintain the following two semantics that account for non-zero delays between origination of a message at the sender and its reception at the receiver and possible reordering of messages and ACKs.

Semantic 1 *A node ignores an update message that comes out-of-order (*i.e., *after a message that was sent later).*

Semantic 2 *A node ignores outstanding ACKs after issuing an Update::Dec message.*

These semantics are enforced using the embedded sequence numbers in the update messages (recall that an ACK includes the sequence number of the Update::Inc that triggered it).

C. Properties of DIV

The two main properties of DIV are: (i) it prevents loops at every instant, and (ii) it prevents counting-to-infinity in the normal mode. In this section, we prove these properties. Note that even in the specific case where the value of a node is set to its current cost-to-destination, the proofs of DIV's properties cannot be obtained from those for the LFI conditions since DIV operates without any assumption on packet reordering, delay or losses.

1) Loop-free Operation at Every Instant: The following is the key proposition based on which our result follows.

Proposition 1 *For any two neighboring nodes x and y, we always have*

$$V(x; y|x) \leq V(x; y|y) \tag{3}$$

Proof: The proof is by contradiction. Suppose at time $t = 0$ condition (3) is satisfied and at time $t = t_4$ condition (3) is violated for the first time. *I.e.,* at time $t = t_4$, we have $V(x; y|x) = V_1$ and $V(x; y|y) = V_0$ with $V_1 > V_0$. Thus, at time t_4 either $V(x; y|y)$ decreases or $V(x; y|x)$ increases. We consider these two cases separately.

Case (i): $V(x; y|y)$ decreases at time t_4 to V_0. Thus node y receives an Update::Dec message from node x at time t_4. As shown in Fig. 2(a), suppose that this message originated at node x at time t_0. Therefore, at time t_0, we have $V(x; y|x) = V_0$. But as per our assumption, $V(x; y|x) = V_1 > V_0$ at time t_4. Thus, node x must receive an ACK from node y that increases $V(x; y|x)$ during the period (t_0, t_4) (cf. Fig. 2(a)). Suppose t_2 denotes the time when node x sent the update message that triggered this ACK. We then have two cases:

- $t_2 < t_0 < t_4$: In this case, the Update::Inc message that triggered the ACK was outstanding at t_0; the time when node x sent an Update::Dec message. Thus node x would

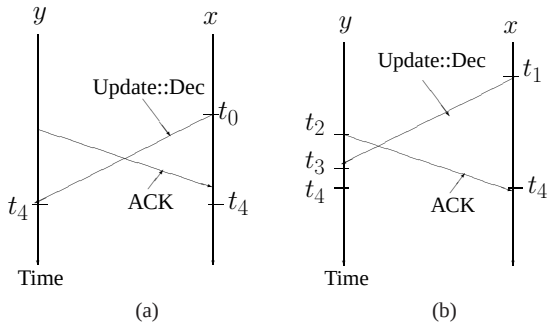


Fig. 2. Two cases of possible message exchanges between two neighboring nodes which would violate Eq. (3). Both cases are shown to be contradictory.

disregard this ACK due to Semantic 2, and therefore not increase $V(x; y|x)$.

- $t_0 < t_2 < t_4$: In this case, the Update::Inc message that triggered the ACK was sent by node x after the Update::Dec message, but node y received the Update::Inc message before the Update::Dec message; *i.e.,* the Update::Dec message arrived node y out of order and thus node y would disregard the Update::Dec message due to Semantic 1, and therefore not decrease $V(x; y|y)$.

We therefore have a contradiction in both the cases.

Case (ii): $V(x; y|x)$ increases at time t_4 to V_1. Thus node x receives an ACK from node y at time t_4. As shown in Fig. 2(b), suppose that this ACK originated at node y at time t_2. Thus, we have $V(x; y|y) = V_1$ at time t_2. But by assumption, $V(x; y|y) = V_0$ at time t_4. Thus, node y must receive an Update::Dec message during the period (t_2, t_4), say at time t_3. Suppose that node x originated this Update::Dec message at time t_1 (cf. Fig. 2(b)). Moreover, suppose node x originated at time t_0 the Update::Inc message that triggered the ACK it receives from node y at time t_4. As before, we then have two possibilities:

- $t_0 < t_1 < t_4$: In this case, the Update::Inc message that triggered the ACK was outstanding at t_1; the time when node x sent an Update::Dec message. Thus node x would disregard this ACK due to Semantic 2, and not increase $V(x; y|x)$ to V_1 at time t_4.
- $t_1 < t_0 < t_4$: In this case, the Update::Inc message that triggered the ACK was sent by node x after the Update::Dec message, but node y received the Update::Inc message before the Update::Dec message; *i.e.,* the Update::Dec message arrived node y out of order and thus node y would disregard the Update::Dec message due to Semantic 1, and not decrease $V(x; y|y)$ to V_0 at time t_3.

We therefore again have a contradiction in both the cases.

Thus we have shown that both case (i) and case (ii) lead to contradictions. Hence, we conclude that it is not possible to violate Eq. (3). ■

Using Proposition 1, we now prove that DIV guarantees that at every instant, the successor graph (*i.e.,* the graph formed by connecting each node to its successor by a directed edge) is acyclic.

Theorem III.1 *The successor graph created following DIV's update algorithm is an acyclic graph at each instant.*

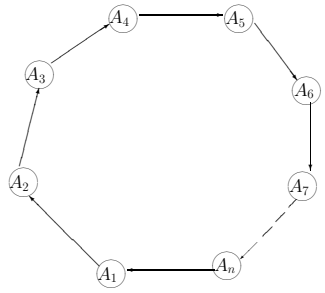


Fig. 3. A possible loop in the successor graph.

Proof: The proof is again by contradiction. Suppose at some instant of time there is a loop in the successor graph, as shown in Fig. 3. Since the number of nodes in this loop is finite, there is a node in this loop whose value is smaller than or equal to the value of its successor. Without any loss of generality, let A_n be this node and let A_1 be its successor. Thus,

$$V(A_1; A_1|A_1) \geq V(A_n; A_n|A_n). \tag{4}$$

But since node A_1 maintains the first invariance, we have

$$V(A_1; A_1|A_1) \leq V(A_1; A_n|A_1). \tag{5}$$

Also since node A_n maintains the second invariance, we have

$$V(A_n; A_n|A_n) > V(A_1; A_n|A_n). \tag{6}$$

But equations (4), (5) and (6) together imply that $V(A_1; A_n|A_1) > V(A_1; A_n|A_n)$, which contradicts Proposition 1. ■

2) Multiple Overlapping Updates and Packet Losses: This is an attractive feature of DIV, which unlike earlier algorithms [11–15] can handle multiple updates without additional efforts. A node can send multiple Update::Inc or Update::Dec messages in any order; a neighbor can postpone sending an ACK for an arbitrary time—*e.g.,* it can use a hold-down time—and when replying with an ACK, it can chooses to respond to only a subset of pending ACKs—even just one: none of these actions would cause a routing loop. We summarize this important property in the following theorem.

Theorem III.2 *The correctness of DIV remains valid under arbitrary policies for handling multiple overlapping updates.*

This gives tremendous flexibility to a node in choosing various policies for replying with ACKs to optimize different criteria.

Proof: Only Semantics 1 and 2 are used to prove Proposition 1, and the proof of Theorem III.1 relies only on Proposition 1 and Invariances 1 and 2. Thus, it is sufficient to ensure that the Semantics and the Invariances remain valid under multiple overlapping updates.

However, it is easily seen that the Semantics are satisfied by each node by using the sequence numbers of the messages, and the invariances depend only on the locally stored variables. Thus they are never violated. ■

Finally, DIV can also handle an arbitrary sequence of lost packets in the sense that its correctness is never jeopardized. If an Update::Dec message sent by node x to neighbor y is lost, then $V(x; y|x)$ is lowered (by x), but not $V(x; y|y)$;

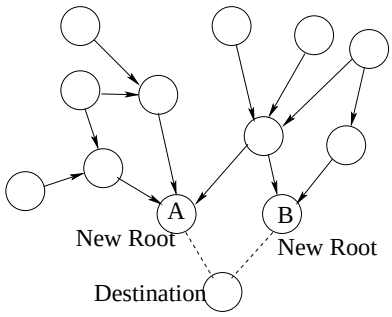


Fig. 4. Proof of avoidance of counting-to-infinity.

i.e., we have $V(x;y|x) < V(x;y|y)$. But this still satisfies Proposition 1, hence does not jeopardize DIV's correctness.

If an Update::Inc message sent by node x to neighbor y is lost, then node x cannot increase its value, but the invariances remains valid. Finally, if an ACK is lost, then $V(x;y|y)$ is increased (by y), but not $V(x;y|x)$; *i.e.,* we have $V(x;y|x) < V(x;y|y)$. Again, this satisfies Proposition 1 and DIV remains correct. We have already shown that Semantics 1 and 2 handle arbitrary reordering and delay of messages. Thus, we summarize these features in the following important property of DIV:

Theorem III.3 *The correctness of DIV remains valid under arbitrary sequence of loss, reordering or delay of messages.*

3) Counting-to-Infinity: Shortest path algorithms represent a very important class of routing algorithms. When used with distance-vector style shortest path computation, DIV, in the normal mode, prevents *counting-to-infinity*.

Suppose a given destination becomes unreachable. With any shortest-path computation algorithm, *eventually* the cost-to-destination estimate of all nodes separated from the destination would reach ∞. However, if nodes increment their cost-to-destination estimates by a finite amount at each step, then the true cost-to-destination values are not reached in a finite number of steps. This is the counting-to-infinity problem which slows down routing convergence. We show that counting-to-infinity cannot happen when DIV is used in the normal mode.

We assume that before the destination becomes unreachable, the network was in a steady state and link costs do not change. DIV is used in normal mode. The value of each node is its estimate of minimum cost-to-destination.

With DIV, the successor graph of a given destination induced by the forwarding decisions is a *Directed Acyclic Graph* (DAG) (or a collection of DAGs if a network partition occurs) at every instant. If a node is allowed to have only one successor that offers the minimum cost-to-destination, then the successor graph will in fact be a tree (or a forest). However, there may be more than one neighboring nodes that offer the minimum cost-to-destination, and we allow the node to treat all of them as successors.

Let a *root* node in the DAG be a node with *no* successor (for a destination). In the steady state, only the destination is a root. Furthermore, in the normal mode of DIV, a non-root node becomes a root node only after a network partition, *e.g.,* a link or a node failure, disconnects if from its last successor. In particular, when the value of a node's only successor's becomes ∞, although the node increases then its own value to ∞, it still maintains the old successor, and hence does *not* become a root.

Assume now that after a network partition, k nodes become roots, as shown in Fig. 4 ($k = 2$; nodes A and B are the roots). Following the network partition, nodes proceed to change (increase) their values in response to this event and subsequent updates.

Claim 1 *An increment in the value of a node corresponds to an increase in the value of a* root *node.*

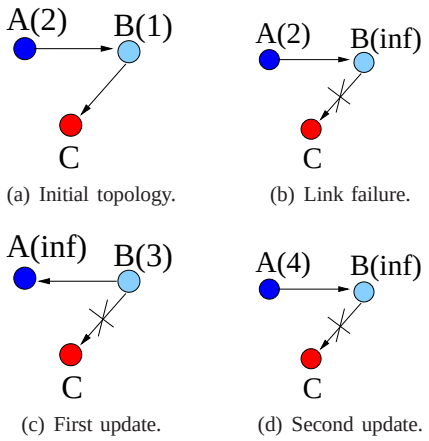
Proof: Suppose that node x_m increases its value. This can happen only when the last successor of x_m, x_{m-1} increased its value, which in turn must mean that x_{m-2}, the last successor of x_{m-1} increased its value, and so on. Since there is no loop in the successor graph, the chain of implications must end at a root node, x_0, which must increase its value. ■

Lemma 1 *If after a network partition, k nodes become roots; nodes in the network can increment their value at most k times.*

Proof: Let a *root-increment* event be the event that a root node increments its value. Claim 1 implies that the number of occurrences of value-increase events at any node cannot exceed the total number of root increment events. We bound the number of root-increment events.

Consider a root node z. Since in DIV, a node with no feasible successors (a root) increments its value (cost-to-destination estimate) to ∞, a root node starts by issuing to all of its neighbors Update::Inc messages carrying its new value of ∞. Neighbors that do not rely on z as their successor or have another feasible successor will immediately send ACKs, while others will originate their own Update::Inc messages. As discussed in Section III-B3c, this eventually results in node z receiving ACKs from all its neighbors, and therefore increasing its own value to ∞, *i.e.,* experiencing a root-increment event. There are two possible situations for node z at this point: (i) all its neighbors have an infinite value; (ii) one or more of its neighbors have a finite value. Case (i) is the simpler and more favorable one in that all nodes in the connected component to which node z belongs, converged to realizing that the destination is unreachable after receiving the Update::Inc message from node z. In other words, all nodes in the connected component of node z increased their value only once.

Case (ii) is more complex and the one we focus on next. In case (ii), because of the existence of one or more neighbors with a finite value, root node z can then choose any of these neighbors as its successor, and decrease its value to some finite number (since the shortest path estimate will become finite). However, at that point, z is *no longer* a root node; by choosing a successor, z merges with another DAG. Further, recall that in the absence of additional failures a non-root node never becomes a root node in the normal mode of DIV. As a result, in case (ii) the root-increment event that root node z experienced (when it was able to update its value to ∞), was followed by a decrease by one in the number of root nodes. Thus, in the



(a) Initial topology. (b) Link failure. (c) First update. (d) Second update.

Fig. 5. Counting-to-infinity without loops.

absence of additional failures, each root-increment event either ensures that nodes go through only one value increment (case (i)), or it reduces the number of root nodes in the successor graph by one (case (ii)). Since there were k root nodes at the beginning, there can at most be k root-increment events. The case of additional failures (following the first one) can be handled based again on the discussion of Section III-B3c, *i.e.,* nodes expecting ACKs from a failed/unreachable neighbor will remove the node from the list of pending ACKs, and if the failure creates another root node, the value of k is simply incremented to reflect the change. Hence, the lemma follows. ■

Finally, since a node can increment its value at most a finite number of times, we have

Theorem III.4 *Counting to infinity does not occur in DIV in the normal mode.*

Remark: Guaranteeing an acyclic successor graph at every instant is not sufficient for preventing counting-to-infinity[5]. Fig. 5 illustrates an example where DIV in alternate mode is used[6]. Recall that in the alternate mode, upon receiving an Update::Inc message a node immediately replies with an ACK even though it may not have any other path to switch to. Fig. 5(a) shows the initial configuration; the shortest-path costs to the destination node C are indicated in the parentheses, where ∞ indicates no path. When the link to the destination goes down, node B detects it, and issues an Update::Inc message to A to set its value to 3. Node A, upon receiving this Update::Inc, sends ACK immediately. This ACK increments the value of node B to 3, but leaves node A without any path. Then node A issues an Update::Inc message to node B to increase its (A's) value to 4. Upon receiving this Update::Inc message, node B immediately sends an ACK, that leaves node B without path, but increments node A's value to 4. And the cycle continues. Note that the successor graph at every time is acyclic, a fact that did not prevent counting-to-infinity. The crucial feature of the normal mode is that it does not allow a node to become pathless unless there is a failure. It is this property that was used in proving Lemma 1.

Recall that a node can choose to use the normal mode or the alternate mode at will on a *per update* basis (i.e., for one update it might choose the normal mode and for the next, it might use the alternate mode) without causing transient loops. A simple generalization of Lemma 1 shows that if nodes use the alternate mode at most M times (for some finite M), then counting-to-infinity does not occur.

4) Frequency of Synchronous Updates: A Comparison with DUAL: A synchronous update occurs when a node notifies all its upstream neighbors about its impending change of status. In the case of DUAL, the change of status is an increase in its feasible distance; in DIV in normal mode, the change of status is an increase in its value. This similarity calls for asking which algorithm produces fewer synchronous updates under identical situations? This is an important question since synchronous updates are time and resource consuming and nodes are left with a non-optimal path—in the worst case, no path at all—for some period. We show that DIV improves on DUAL in this regard by issuing fewer synchronous updates.

For comparison, we need to fix a feasibility condition for DUAL and an equivalent value assignment for DIV. Due to space constraint, we prove the claim only for SNC in DUAL and the equivalent value assignment in DIV as the minimum cost-to-destination seen by a node from time $t = 0$ (we suppress the phrase "from time $t = 0$" from now on).

Claim 2 *Suppose x and y are neighbors. If SNC is true at x through y, then with DIV x can choose y as a successor.*

Proof: We need to show that SNC being true at x through y implies $V(x;x|x) > V(y;y|y)$. From the definition of SNC (cf. Section II-B2), since SNC is satisfied, we have the minimum cost-to-destination of x, $V(x;x|x)$, is more than the *current* cost-to-destination of y. However, the current cost-to-destination of y is clearly as large as the minimum cost-to-destination of y, $V(y;y|y)$; *i.e.,* $V(x;x|x) > V(y;y|y)$. ■

However, the other direction is clearly not true. For example, suppose that $V(x;x|x) = 2$, $V(y;y|y) = 1$ and the current cost-to-destination of y is 3. Then SNC is not satisfied, but with DIV, x can still choose y as its successor. Since the condition of DIV is strictly more relaxed than SNC, and a synchronous update is issued only when the condition of DIV (or SNC for DUAL) is not satisfied, we have

Theorem III.5 *DIV issues synchronous updates less frequently than DUAL under SNC.*

Note that this cannot be remedied simply by replacing SNC in DUAL with the conditions of DIV since without DIV's update mechanisms, these are not sufficient to guarantee loop-free operation.

IV. Routing Under General Cost Functions

One of the important advantages of DIV is that it is not tied to a particular cost function when computing a routing. We illustrate the benefits of this decoupling using a cost function

[5]This remark applies to all loop-freedom algorithms, not only DIV.

[6]Counting-to-infinity can be avoided in this very simple example by using poison reverse, but the conclusion drawn from the example remains true.

that instead of the standard shortest path distance function, seeks to maximize the number of next-hops available at all nodes for each destination. The availability of multiple next-hops ensures that the failure of any one link or neighbor does not impede a node's ability to continue forwarding traffic to a destination. A failure results in the loss of at most one next hop to a destination, so that the node can continue forwarding packets on the remaining ones without waiting for new paths to be computed. In other words, the routing is *robust* to local failures. This may be an appropriate objective in settings where end-to-end latency is small and bandwidth plentiful, *e.g.,* as in the previously mentioned large-scale Ethernet networks spanning entire metropolitan areas, which provided some of the early motivations for developing DIV.

We note that mechanisms to increase routing robustness, *i.e.,* the ability to continue forwarding packets in the presence of failures while avoiding transient loops, is a topic that has and continues to receive significant attention in the academic community and in the industry through various standardization efforts, *e.g.,* [24]. For example, a variety of schemes have been proposed, *e.g.,* [28–32], that offer "fast local re-routing" through pre-configured, backup forwarding strategies that are to be used when a failure is detected. This wealth of proposals targeting routing robustness to failures demonstrates the importance of the topic, and further motivates exploring how such an objective can be realized using DIV. In the rest of this section, we detail one possible approach and highlight the benefits of using DIV in that context.

A. Robust Routing with DIV

Recall that with DIV a value is assigned (per destination) to each node and a node is allowed to choose a neighbor as a successor only if the value of the neighbor is less than the node's own value. DIV is designed to allow updates to these values, while preserving loop-free routing. However, the values themselves are not specified by DIV; as far as DIV is concerned, these values can be arbitrary. Therefore, values could be chosen to realize some measure of robustness in the resulting routing scheme. Indeed, computing a routing is equivalent to assigning the values: Each value assignment across nodes induces a DAG by defining an edge from neighboring node x to node y if and only if $V(x) > V(y)$. Conversely, each routing decision, upon convergence, naturally induces a value assignment: The successor graph induced by the routing decisions, which is a DAG, has a topological sort, *i.e.,* using the order of a node in the topological sort as its value.

Routing by means of value assignment naturally lends to local robustness. For a given node x with a set of neighboring nodes $\mathcal{N}(x)$, *any* node from the *feasible successor set*

$$\mathcal{S}(x) = \{y \in \mathcal{N}(x) | V(x) > V(y)\}$$

can be chosen as a successor, and the choice can be made on a *packet by packet basis* without creating loops. Thus, when $\mathcal{S}(x)$ has multiple members, the failure of one of them does not disrupt packet forwarding which can proceed using the remaining ones[7]. This is realized without the need for any routing update. Hence, for robustness purposes, it is desirable to increase the feasible successor sets of nodes. However, these sets are not independent; for every neighboring nodes x and y, either $x \in \mathcal{S}(y)$ or $y \in \mathcal{S}(x)$. Increasing $\mathcal{S}(x)$ by incorporating y would reduce $\mathcal{S}(y)$; thus a trade-off exists. We explore one possible global metric that captures this trade-off, and develop an algorithm that can be used with DIV to optimize this metric in a distributed manner.

B. Robustness Oriented Value Assignment Algorithm: DIV-R

We model the network as a graph $G = (\overline{N}, E)$ with node set $\overline{N}$ and edge set E. For convenience, we define N by removing the destination node from $\overline{N}$. The essential idea is to increase the size of the feasible successor (next-hop) set at each node. The obvious choice of a global objective function: $\sum_{x \in N} \mathcal{S}(x)$, fails since the sum, which is the sum of all directed edges, is always equal to $|E|$. (Note that this implies that under a value-based routing, the average number of next-hops per node is always $|E|/N$). Therefore, we propose the following function:

$$\textbf{Obj 1: } F = \prod_{x \in N} |\mathcal{S}(x)| \tag{7}$$

Obj 1 is the product of the number of 1-hop paths (the size of the feasible successor set) at the nodes, and we seek to maximize **Obj 1** by choosing appropriate node values. We conjecture that this optimization problem is NP-hard, although we do not have a proof yet. Note that **Obj 1** only requires nodes to exchange the size of their feasible successor sets with 1-hop neighbors. Moreover, as we explain later, an individual node can determine the effect of its own action (in the sense of increasing or decreasing a value) without any additional knowledge. Thus **Obj 1** is well suited for distributed implementation, as allowed by DIV.

Intuitively, maximizing **Obj 1** attempts to "equalize" the feasible successor set sizes. The arithmetic mean-geometric mean inequality gives

$$F \leq \left(\tfrac{1}{|N|} \sum\nolimits_{x \in N} |\mathcal{S}(x)|\right)^{|N|} = (|E|/|N|)^{|N|}$$

where the maximization occurs at $\mathcal{S}(x) = |E|/|N|$ for all x, although topological restrictions usually will not allow achieving this optimum.

Other objective functions are clearly possible. For example, **Obj 2** below is another reasonable alternative: It considers the sum of next-hops available to all the neighbors of a node and seeks to maximize their own sum.

$$\textbf{Obj 2: } G = \sum_{x \in N} \sum_{y \in \mathcal{S}(x)} |\mathcal{S}(y)|$$

Obj 2 has the disadvantage of requiring nodes to know the values of all their 2-hop neighbors, but its more "global" metric could possibly result in better solutions. However, our

[7]Clearly, failures do affect available resources, but the impact is lessened by the availability of multiple alternatives. Note also that there is considerable flexibility in which and how many of these alternatives are used — from all of them being continuously used to using only one at a time.

focus here is not on providing a comprehensive investigation of robust routing solutions, but instead on demonstrating the benefits afforded by DIV's ability to operate with a broad range of cost or objective functions. Hence, we focus on **Obj 1**, and describe next how a node can determine if an action on its part would increase or decrease **Obj 1**. Such a determination will then drive the updates that nodes send to each other using DIV to ultimately converge to an optimized robust routing solution (in the sense of **Obj 1**).

Observe that $\mathcal{S}(x)$ depends only on the ordering of the values in the set $\{V(y) : y \in \mathcal{N}(x) \cup x\}$. Let $V(y_1) < V(y_2) < \cdots < V(x) < \cdots < V(y_{|\mathcal{N}(x)|})$ ($y_i \in \mathcal{N}(x)$) and let $rank(x)$ be the rank of $V(x)$ in this set. Suppose $rank(x) = a$ and after node x changes $V(x)$, new $rank(x) = b$. Then, under the assumption that no changes in other nodes are made, we have the following: For $b > a$

$$\begin{aligned} & \frac{F_{\text{new}}}{F_{\text{old}}}(a,b) \\ = \; & (|\mathcal{S}(x)| + b - a)\,(|\mathcal{S}(y_{a+1})| - 1)\cdots(|\mathcal{S}(y_b)| - 1)\,/ \\ & (|\mathcal{S}(x)||\mathcal{S}(y_{a+1})|\cdots|\mathcal{S}(y_b)|)\,. \end{aligned} \tag{8}$$

Similarly, for $a > b$,

$$\begin{aligned} & \frac{F_{\text{new}}}{F_{\text{old}}}(a,b) \\ = \; & (|\mathcal{S}(x)| + b - a)\,(|\mathcal{S}(y_{a-1})| + 1)\cdots(|\mathcal{S}(y_b)| + 1)\,/ \\ & (|\mathcal{S}(x)||\mathcal{S}(y_{a-1})|\cdots|\mathcal{S}(y_b)|)\,, \end{aligned} \tag{9}$$

where in both Eqs. (8) and (9), $\mathcal{S}(.)$ denotes $\mathcal{S}_{\text{old}}(.)$, *i.e.*, prior to the change of value at node x. Thus, a node determines the change in the global objective function utilizing only local information (assuming its neighbors provide node x with their own $\mathcal{S}(.)$ values).

Based on this, we propose a simple distributed heuristic, DIV-R, that greedily seeks to maximize **Obj 1**. DIV-R proceeds as follows: If node x decides to update its value, it starts by sending messages to "lock" its 2-hop neighbors[8]. Then all nodes $y \in \mathcal{N}(x)$, send $|\mathcal{S}(y)|$ to node x, which determines how to change its value so as to maximize $F_{\text{new}}/F_{\text{old}}$ using Eqs. (8) and (9). Next, node x notifies its neighbors of its new value as per the operation of DIV. Note that in order for this to successfully conclude and avoid possible deadlocks, this requires that DIV be used in the alternate mode. Finally, node x sends out messages to "unlock" the locked nodes. We remark that there is no restriction on the order of value updates among the nodes and the update frequency because the underlying DIV mechanism guarantees that no loop can ever form irrespective of the sequences of updates.

In Section V-B, we evaluate the performance of this algorithm and compare it to that of a "shortest path" algorithm, where link weights have been selected (off-line) so as to optimize **Obj 1**. The comparison to a shortest path algorithm is aimed at illustrating the benefits afforded by DIV's ability to accommodate more general objective functions.

[8]Locking the two-hop neighbors is necessary so as to ensure that $|\mathcal{S}(y)|$, $y \in \mathcal{N}(x)$, is kept constant when the heuristic is being executed. Note that this also assumes that node x is itself not locked when it decides to update its value.

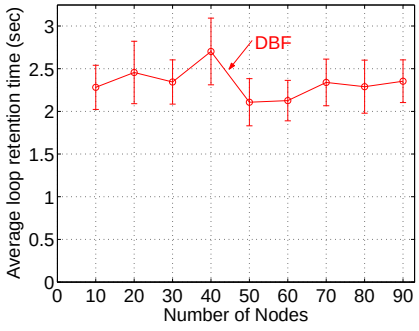


Fig. 6. Mean loop-retention time. No loops are found with DUAL or DIV.

V. Performance Evaluation

A. Performance of DIV in Shortest Path Routing

In this section, we consider three shortest paths algorithms, DBF, DUAL (using SNC as its feasibility condition), and DIV (using DBF to compute value updates), and compare their performance in terms of loop avoidance[9] and convergence time. The simulations are performed on random graphs with fixed average degree of 5, but in order to generate a reasonable range of configurations, a number of other parameters are varied. Networks with sizes ranging from 10 to 90 nodes are explored in increments of 10 nodes. For each network-size, 100 random graphs are generated. Link costs are drawn from a bi-modal distribution: with probability 0.5 a link cost is uniformly distributed in [0,1]; and with probability 0.5 it is uniformly distributed in [0,100]. For each graph, 100 random link-cost changes are introduced, again drawn from the same bi-modal distribution. All three algorithms are run on the same graphs and sequences of changes. Processing time of each message is random: it is 2 secs with probability 0.0001, 200 ms with probability 0.05, and 10 ms otherwise.

Fig. 6 shows the average loop-retention time in seconds, T_{loop}—the time from when a routing loop is detected to the time when again no routing loop exists—given that a loop is formed with DBF, as the size of the graphs are varied. As expected, no loops were found with DUAL or DIV. As seen in the figure, loops can be retained for a significant time. The figure supports the need for loop-prevention algorithms such as DUAL and DIV, by demonstrating that even in relatively small networks, transient loops can last for non-negligible amounts of time.

Fig. 7 shows average convergence time—the time from when a link cost changes to the time when no more update messages are exchanged—of all three algorithms as the size of the graphs are varied. The vertical bars show the standard deviations. Both DIV and DUAL converge faster than the vanilla DBF; however, DIV performs better, especially for larger graphs. This is because DIV's conditions are satisfied more easily, and hence a synchronous update can be performed faster (recall that a node with a feasible neighbor will reply immediately). This observation is supported by Table I where we show the fraction of times the condition of DIV is satisfied

[9]Obviously, neither DUAL nor DIV should give rise to transient loops.

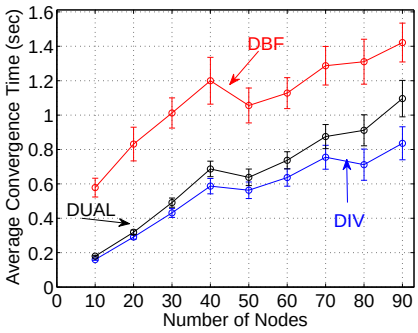


Fig. 7. Mean convergence time.

Nodes	10	20	30	40	50	60	70	80	90
Fraction	0.717	0.784	0.823	0.843	0.846	0.832	0.843	0.846	0.840

TABLE I
FRACTION OF TIMES DIV IS SATISFIED GIVEN THAT SNC IS NOT.

given that SNC is not satisfied. As Table I shows, the fraction becomes more than 80% for larger graphs.

B. Performance of DIV-R

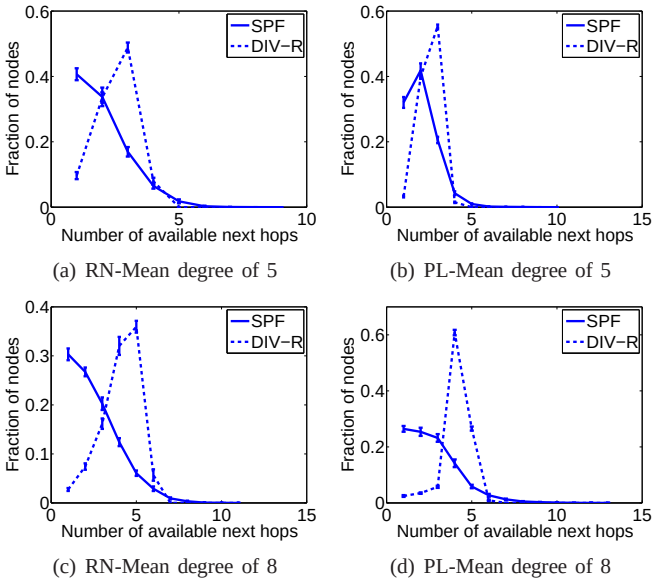
In this section, we illustrate the benefits of using a value-assignment based routing over a shortest path routing in terms of optimizing a robustness-oriented metric, namely, **Obj 1** (Eq. (7)). To ensure fairness, in shortest path computations, link weights are selected so as to optimize **Obj 1**. However, note that in addition to optimizing **Obj 1**, the routing must also satisfy the added constraint of being a shortest path based on the computed link weights[10].

We simulate two types of topologies in our comparison: (i) random topologies and (ii) power-law topologies; both with varying sizes and numbers of links. Random topologies are generated by randomly connecting links to nodes with equal probability. Power-law topologies are generated using the preferential attachment model [33]. Connectivity is ensured in all cases. For each combination of topology type and network size, we generate 10 graph samples for which we evaluate the performance of each algorithm.

In DIV-R, node values are initialized to the minimum hop counts to the destinations. At each step, a random node executes DIV-R, that emulates the distributed operation. Iterations are stopped when no improvement is detected for any node.

The Shortest-Path-First (SPF) solution is computed using a "blackbox optimization" approach as in [34]. Each set of link weights, W, induces shortest paths and in turn evaluates to a vector $\vec{F}(W)$ of F-values (according to Eq. (7)), one per destination. We consider a link weight setting W_1 better than another link weight setting W_2 if $\vec{F}(W_1) > \vec{F}(W_2)$ *lexicographically*. In other words, the optimization searches for link weights that maximize the smallest F-value across all destinations. The search for an "optimal" set of link weights is carried out for a total of 5000 iterations, where if $\vec{F}$ fails to improve for 100 consecutive iterations, 10% of the link weights are randomly perturbed in order to let the search escape from local maxima. Link weights are selected from the set $\{1, 2, \ldots, 20\}$ to balance computational complexity and the quality of a solution.

[10]It is precisely the need to use common *link weights* across destinations that introduces dependencies that severely limits the ability of a solution to yield good results for *all* destinations.



(a) RN-Mean degree of 5 (b) PL-Mean degree of 5

(c) RN-Mean degree of 8 (d) PL-Mean degree of 8

Fig. 8. Distribution of number of next-hops across nodes and destinations (50-node networks).

TABLE II
COMPARING AVERAGE NUMBER OF NEXT-HOPS ACROSS TOPOLOGIES.

Average degree	3		5		8		12	
Network type	RN	PL	RN	PL	RN	PL	RN	PL
DIV-R	1.53	1.53	2.55	2.55	4.08	4.08	6.12	6.12
SPF	1.39	1.41	1.96	2.01	2.51	2.66	3.68	3.60

Our main comparison metric is the distribution (normalized histogram) of the number of next-hops realized across all nodes and destinations. The results are presented for a representative 50-node network in Table II (average only) and Fig. 8 (full histogram) from which a number of basic conclusions emerge. First, Table II illustrates for both random (RN) and power-law (PL) topologies and varying average node degrees, that imposing a shortest-path constraint does indeed result in a lower average number of next hops available across nodes and destinations. Moreover, the difference between DIV-R and an SPF-based solution grows with the average node degree (other experiments showed that it also grows with the network size, although less rapidly). This is in part due to DIV-R's greater flexibility in exploiting all available neighbors unimpeded by the shortest-path constraint. Note that some of this difference could be recouped by relaxing the shortest-path constraint of the SPF-based solution and allowing a node to use any neighbor as its next-hop, provided that this does not result in the possible formation of loops. This is akin the "variance" concept of the EIGRP proposal that allows the selection of next-hops among *all* nodes in the set of feasible successors computed by the DUAL algorithm.

The more significant difference between the solutions produced by DIV-R and SPF lies in the underlying *distributions* as illustrated in Fig. 8, which reports the results for random and power-law topologies with average node degrees of 5 and 8.

The solution produced by DIV-R is much more concentrated around its mean, which reflects its success in ensuring that as few nodes as possible have below average connectivity for all destinations. Qualitatively similar results were obtained for different network sizes and node degrees. In addition, further investigation of the results revealed that in many instances where the DIV-R solution produced a number of next-hops below the average, it was due to the constraint imposed by actual degree of the node itself, *i.e.*, further improvements were not possible.

VI. Conclusion

Distance-vector routing algorithms offer a number of advantages over link-state algorithms, *e.g.*, lower resource requirements and often greater stability by keeping the impact of changes local. However, the dependencies across nodes that a distance-vector operation induces can also magnify the impact and duration of inconsistent decisions across nodes manifested through transient loops and the counting-to-infinity problem described earlier. It is, therefore, important to devise mechanisms to overcome these limitations without affecting the intrinsic benefits of a distance-vector operation. In this paper, we present a novel algorithm, D*istributed Path Computation with* I*ntermediate* V*ariables* (DIV), that achieves this by laying down a rule-set over existing routing algorithms and defining an efficient update mechanism for enforcing those rules; both are easy to implement. In addition, because DIV is not integrated with shortest-path computations, it can be used with any routing algorithm.

When used with shortest-path computation algorithms, DIV was shown to perform better than current alternatives, such as *Diffusing Update Algorithm* (DUAL) (and consequently the protocols based on DUAL), both analytically and by simulation along various metrics. Another significant advantage of DIV is that it handles message losses and out-of-sequence delivery, and allows nodes to adopt arbitrary policies for handling multiple overlapping updates opening up the possibility of various optimizations. Finally, the rule-set and proof of correctness of DIV is relatively intuitive, which will hopefully facilitate efficient (and correct) implementations. The fact that DIV is not coupled to a shortest path computation, and in particular that it uses a per-destination node potential ("value"), can be leveraged to enforce a broad range of routing objectives. We illustrated this by proposing DIV-R, an algorithm to assign the node values used by DIV, with the view of optimizing the network's "local repair" ability in the event of node (or link) failures. We demonstrated the effectiveness of DIV-R through simulations showing that it significantly outperformed SPF-based routing in robustness. Hence, highlighting the benefits and flexibility of DIV, which we believe has applicability in many other environments.

Acknowledgment

The authors gratefully acknowledge J.J. Garcia-Luna-Aceves for introducing them to the LFI-based algorithms.

References

[1] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe, "The case for separating routing from routers," in *Proceedings of ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, Portland, OR, September 2004.

[2] A. Shankar, C. Alaettinoglu, K. Dussa-Zieger, and I. Matta, "Transient and steady-state performance of routing protocols: Distance-vector versus link-state," *Internetworking: Research and Experience*, vol. 6, no. 2, pp. 59–87, June 1995.

[3] A. Myers, E. Ng, and H. Zhang, "Rethinking the service model: Scaling Ethernet to a million nodes," in *Proceedings of ACM SIGCOMM HotNets*, San Diego, CA, 2004.

[4] Y. Ohba, "Issues on loop prevention in MPLS networks," *IEEE Comm. Mag.*, vol. 37, no. 12, pp. 64–68, December 1999.

[5] P. Francois and O. Bonaventure, "Avoiding transient loops during IGP convergence in IP networks," in *Proceedings of IEEE INFOCOM*, Miami, FL, 2005.

[6] J. Moy, "OSPF version 2," Internet Engineering Task Force, RFC 2328, Apr. 1998. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2328.txt

[7] A. Shaikh and A. Greenberg, "Experience in black-box OSPF measurement," in *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW)*, San Francisco, CA, November 2001.

[8] G. Malkin, "RIP version 2," Internet Engineering Task Force, RFC 2453, Nov. 1998. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2453.txt

[9] R. Albrightson, J. J. Garcia-Luna-Aceves, and J. Boyle, "EIGRP – a fast routing protocol based on distance vectors," in *Proc. Network/Interop*, 1994.

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

[11] P. M. Merlin and A. Segall, "A failsafe distributed routing protocol," *IEEE Transactions on Communications*, vol. 27, no. 9, pp. 1280–1288, Sept 1979.

[12] J. M. Jaffe and F. M. Moss, "A responsive routing algorithm for computer networks," *IEEE Transactions on Communications*, vol. 30, no. 7, pp. 1768–1762, July 1982.

[13] J. J. Garcia-Lunes-Aceves, "Loop-free routing using diffusing computations," *IEEE/ACM Transactions on Networking*, vol. 1, no. 1, pp. 130–141, Feb 1993.

[14] S. Vutukury and J. J. Garcia-Luna-Aceves, "A simple approximation to minimum-delay routing," in *Proc. ACM SIGCOMM*, 1999.

[15] ——, "MDVA: A distance-vector multipath routing protocol," in *Proceedings of IEEE INFOCOM*, Anchorage, AK, April 2001.

[16] K. Elmeleegy, A. L. Cox, and T. S. E. Ng, "On count-to-infinity induced forwarding loops in Ethernet networks," in *IEEE INFOCOM*, 2006.

[17] "Transparent interconnection of lots of links (trill)." [Online]. Available: http://www.postel.org/rbridge/

[18] "802.1aq - shortest path bridging." [Online]. Available: http://www.ieee802.org/1/pages/802.1aq.html

[19] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Prentice Hall, 1991.

[20] W. T. Zaumen and J. J. Garcia-Luna-Aceves, "Dynamics of distributed shortest path routing algorithms," in *Proceedings of ACM SIGCOMM*, Zurich, Switzerland, September 1991.

[21] P. Humblet, "Another adaptive shortest-path algorithm," *IEEE Transactions on Communications*, June 1991.

[22] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, August 1980.

[23] V. D. Park and M. S. Corson, "A highly adaptive distributed routing algorithm for mobile wireless networks," in *Proceedings of IEEE INFOCOM*, 1997. [Online]. Available: citeseer.ifi.unizh.ch/park97highly.html

[24] M. Shand and S. Bryant, "IP fast reroute framework," Internet Draft (draft-ietf-rtgwg-ipfrr-framework-08.txt), February 2008.

[25] M. Shand, S. Bryant, and S. Previdi, "IP fast reroute framework using Not-via addresses," Internet Draft (draft-ietf-rtgwg-ipfrr-notvia-addresses-02.txt), February 2008.

[26] C. Alattinoglu, V. Jacobson, and H. Yu, "Towards milli-second igp convergence," IETF Internet Draft, Nov 2000, draft-alaettinoglu-ISISconvergence-00.txt.

[27] A. Basu and J. G. Riecke, "Stability issues in OSPF routing," in *Proc. ACM SIGCOMM*, 2001.

[28] S. Lee, Y. Yu, S. Nelakuditi, Z. L. Zhang, and C. N. Chuah, "Proactive vs. reactive approaches to failure resilient routing," in *Proc. IEEE INFOCOM*, 2004.

[29] Z. Zhong, S. Nelakuditi, Y. Yu, S. Lee, J. Wang, and C. N. Chuah, "Failure inferencing based fast rerouting for handling transient link and node failures," in *Proc. IEEE Global Internet*, 2005.
[30] G. Apostolopoulos, "Using multiple topologies for IP-only protection against network failures: A routing performance perspective," ICS-FORTH, Greece, Tech. Rep., 2006.
[31] S. Gjessing and O. Norway, "Implementation of two resilience mechanisms using multi topology routing and stub routers," in *Proc. AICT-ICIW*, 2006.
[32] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne, "Fast IP network recovery using multiple routing configurations," in *Proc. IEEE INFOCOM*, 2006.
[33] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, 1999.
[34] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *Proc. IEEE INFOCOM*, 2000.