IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661 Volume 4, Issue 3 (Sep.-Oct. 2012), PP 13-20
www.iosrjournals.org

Distributed Path Computation Using DIV Algorithm

[1]P.Vinay Bhushan, [2]R.Sumathi

Abstract: *Distributed routing algorithms, 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.*

Index Terms: *Distance-vector routing, Loop-free routing, DIV Algorithm, Dual Algorithm.*

I. INTRODUCTION

NETWORK Links That To Provide The 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. 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 [1]. 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, a significant issue even with unicast packets, is further 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 –[2] for recent discussions.

Link-state algorithms, of which the OSPF [3] 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 tens of milliseconds per node for typical present day networks), 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 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 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, EIGRP [4]) 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 research was triggered by a renewed interest in devising light-weight, loop-free path computation solutions for large-scale Ethernet networks.

Distributed path computation with intermediate variables (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, [6] and outperforms previous solutions including known LFI and Diffusing Computation based algorithms, such as the *Diffusing Update Algorithm*. The rules and update mechanism of DIV and their correctness proofs are rather simple, which hopefully will also facilitate correct and efficient implementations.

II. RELATED WORK

2.1. 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 assume that nodes use *poison reverse*; i.e., each node reports an infinite cost-to-destination to its successor node. 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.

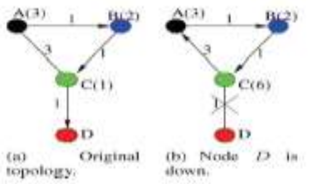

Fig .1. A simple example of counting-to-infinity problem

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 6 in RIP) and the destination declared unreachable once the cost reaches that value, the computation never ends.

2.2. 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 [5] 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 depends 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.

2.3. 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) is introduced in [7] and it is shown that if nodes maintain these invariances, then no transient loops can form (cf. Section III). Update mechanisms are required to maintain the LFI conditions: [7] introduces *Multiple-path Partial-topology Dissemination Algorithm* (MPDA) that uses a link-state type approach whereas [6] 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.

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

3.1. 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 $t = 0$;

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 neighbors 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.

DIV 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 its 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

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. where we explore a routing algorithm that attempts to increase the *robustness* of the network in terms of being able to reroute 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 where synchronous updates are time and resource consuming updates that might need to propagate to all upstream2 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); alternate mode.

3) *Maintaining a path*: A node can potentially switch to a new successor more quickly, while provably still guaranteeing loop prevention (alternate mode). 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 updates3 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

5) *Robustness*: DIV can tolerate arbitrary packet reordering and losses without sacrificing correctness.

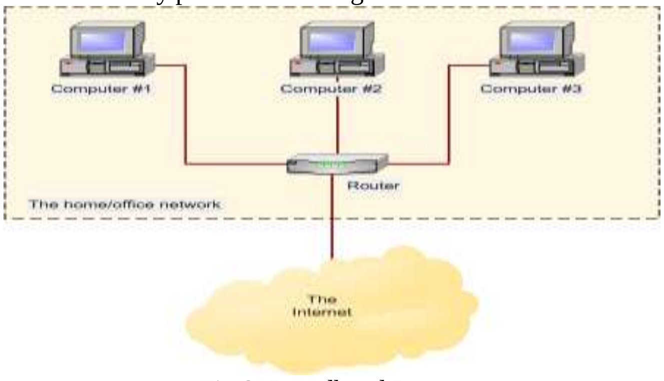

Fig 2. Broadband Router

3.2. 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 nonideal 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 is a neighbor of node. 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 neighbors {*y1,y2,…y}*) , it stores, for each destination:

1) its own value, V(*x;x\x*);

2) the values of its neighbors as known to itself,

V (*yi; x\x*) [*yi €{y1,y2,…,yn*}];

3) and the value of itself as known to its neighbors V(*x; yi \x*) [*yi €{y1,y2,…,yn*}].

That is, 2*n*+1values for each destination. This is *O*(N . d) storage complexity in a network with *N* destination nodes and average node degree *d* . The variables V(*yi; x\x*) and V(*yi; x\x*) are called intermediate variables since they endeavor to reflect the values V(*yi; yi \ yi*)and V(*x; x\x*) , respectively. In steady state, DIV ensures that V(*x; x\x*) = V(*x; yi \x*) = V(*x; yi\ yi*).

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.

3) Invariance 1: The value of a node is not allowed to be more than the value the node thinks is known to its neighbors.

V(*x; x\x*) <= V(*x; yi \x*) for each neighbor *yi* (1)

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

V(*x; x\x*) > V(*y; x\x*) (2)

Thus, due to Invariance 2, a node can choose a successor only from its *feasible successor set{ yi* \V(*x; x\x*) > V(*yi; 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.

5) 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. The ACKs contain the sequence number and the value (and the destination) of the corresponding Update::Inc message. DIV lays down precise rules for exchanging and handling these messages which we now describe.

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 V0 :

• Node *x* first simultaneously decreases the variables V(*x; x\x*) and the values V(*x; yi \x*) ¥*I* = 1,2,…,n ,to V0;

• Node *x* then sends an Update::Dec message to all its neighbors that contains the new value V0.

• Each neighbor y*i* of *x* that receives an Update::Dec message containing V0 as the new value updates V(*x; yi\yi*) to V0.

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 V1:

• Node *x* first sends an Update::Inc message to all its neighbors.

• Each neighbor *yi* 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). When *yi* is ready to send the ACK, it first modifies V(*x; yi \ yi*), 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; yi \ yi*). Note that in this case it is essential that node *yi* changes successor, if necessary, *before* sending the ACK.

• When node *x* receives an ACK from its neighbor *yi*, it modifies V(*x; yi \x*) to V1. At any time, node *x* can choose any value V(*x; x \x*) <= V(*x; yi \x*) ¥i=1, 2,…, n.

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 *yi* received an Update::Inc message from node x. Recall that node *yi* must increase V(*x; yi \ yi*) before sending an ACK. However, increasing V(*x; yi \ yi*) may remove node x from the feasible successor set at node *yi*. If node x is the only preferred node in the feasible successor set of node *yi*, then node *yi* may lose its path if V(*x; yi \ yi*) is increased without first increasing V(*yi; yi \ yi*) . In such a case node *yi* has two options: 1) first increase V(*yi; yi \ yi)* and then increase V(*x; yi \ yi)* and send the ACK to node x; or 2) increase V(x*i; yi \ yi*), send ACK to node x and then increase V(*yi; yi \ yi* . If a node uses option 1), we say that DIV is operating in its *normal mode*; if a node uses option 2), we say that DIV is operating in *alternate mode*. In the normal mode [i.e., using option 1)], 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 *yi*, then node *yi* will immediately respond to node x's Update::Inc message. Otherwise, node *yi* wants to increase its own value before sending an ACK. Node *yi* issues its own Update::Inc message to *all* its neighbors, including node x . The set of neighbors of node *yi* that do not depend on node *yi* for reaching the destination *based on their current values* (which includes node x), would immediately respond to node *yi* with an ACK. When node *yi* receives ACKs (in response to node *yi*'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 *yi* 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 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 2)], we trade simpler and faster processing of ACKs for the risk of having node *yi* 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. *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 nonzero delays between origination of a message at the sender and its reception at the receiver and possible reordering of messages and ACKs.

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

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

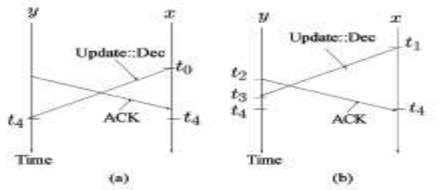

Fig. 2. Two cases of possible message exchanges between two neighboring nodes which would violate (3). Both cases are shown to be contradictory.

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).

3.2.1 PROPERTIES OF DIV

The two main properties of DIV are:

1) Loop-Free Operation at Every Instant
2) Multiple Overlapping Updates and Packet Losses
3) Counting-to-Infinity
4) Frequency of Synchronous Updates

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 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. Multiple available next-hops also improve load balancing by distributing packet transmissions across the multiple associated links.

V. Results And Performance

5.1. PERFORMANCE OF DIV IN SHORTEST PATH ROUTING

DBF, DUAL (using SNC as its feasibility condition), and DIV (using DBF to compute value updates), and compare their performance in terms of loop avoidance9 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 bimodal 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 bimodal distribution. All three algorithms are run on the same graphs and sequences of changes. Processing time of each message is random: it is 2 s with probability 0.0001, 200 ms with probability 0.05, and 10 ms otherwise.

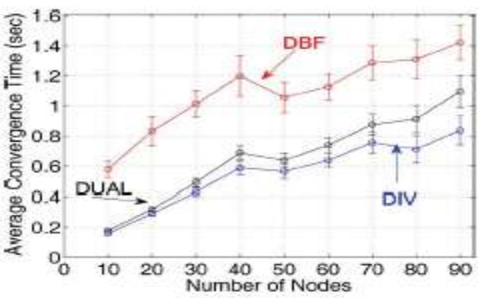

Fig. 7 shows average convergence time

The time from a link cost change until 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

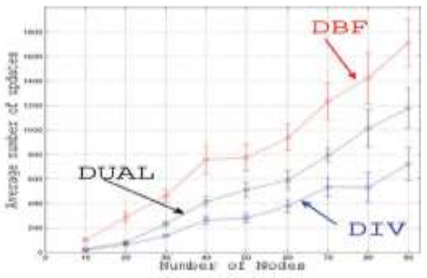

Fig. 8. Number of update messages required for convergence.

Fig. 8 shows the number of control messages used by each algorithm. The trend is very similar to that of the convergence time. DIV issues the least number of control messages before convergence, followed by DUAL and DBF. 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 "black box optimization" approach as in, where link-weights are randomly perturbed to explore the space of possible solutions and optimize the out-degrees of nodes produced by the resulting SPF calculations. The essential difficulty is that the SPF computations for different destinations are coupled through the link-weights since each link uses only one weight for all destinations. The SPF results across different destinations must, therefore, be balanced.

VI. CONCLUSION

Distributed Path Computation with Intermediate Variables (DIV), that achieves this by laying down a rule-set over existing routing algorithms (eg.RIP, IGRP, EIGRP)and defining an efficient update mechanism for enforcing those rules; both are easy to implement. 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 are intuitive, which should facilitate efficient (and correct) implementations. The benefit of an operation decoupled from shortest path computations was illustrated through the DIV-R algorithm. DIV-R assigns node values with the view of optimizing the network's "local repair" ability in the event of node (or link) failures. These is mainly depends on the simulation results of DIV. We believe this flexibility of DIV to have applicability in other environments.

VII. Acknowledgement

We would to thank the anonymous referee for helpful Comments

REFERENCES

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

[2] P. Francois and O. Bonaventure, "Avoiding transient loops during IGP convergence in IP networks," in *Proc. IEEE INFOCOM*, Miami, FL, Mar. 2005, vol. 1, pp. 237–247.

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

[4] R. Albrightson, J. J. Garcia-Luna-Aceves, and J. Boyle, "EIGRP – A fast routing protocol based on distance vectors," presented at the Netw./ Interop. 1994.

[5] J. J. Garcia-Lunes-Aceves, "Loop-free routing using diffusing computations," *IEEE/ACMTrans. Netw.* vol. 1, no. 1, pp. 130–141, Feb. 1993.

[6] S. Vutukury and J. J. Garcia-Luna-Aceves, "MDVA: A distance-vector multipath routing protocol," in *Proc. IEEE INFOCOM*, Anchorage, AK, Apr. 2001, vol. 1, pp. 557–564.

[7] S. Vutukury and J. J. Garcia-Luna-Aceves, "A simple approximation to minimum-delay routing," in *Proc. ACM SIGCOMM*, 1999, pp.227–238.

[8] K. Elmeleegy, A. L. Cox, and T. S. E. Ng, "On count-to-infinity induced forwarding loops in Ethernet networks," in *Proc. IEEE INFOCOM*, 2006, pp. 1–13.

[9] W. T. Zaumen and J. J. Garcia-Luna-Aceves, "Dynamics of distributed shortest path routing algorithms," in *Proc. ACM SIGCOMM*, Zurich, Switzerland, Sep. 1991, pp. 31–42.

Author Bibliography

Mr. P. vinay Bhushan received his B.Tech in Computer science and Engineering from J.B. Institute Of Engineering and Technology, JNTU, Hyderabad and Pursuing M.Tech in Computer science Engineering from Aurora's Technological And Research Institute, JNTU, Hyderabad.

Mrs. R. Sumathi working as Sr.Assistant Professor in the Department of Computer Science and Engineering, in Aurora's Technological And Research Institute with a teaching experience of 5 years. She has received her Master Degree in Technology in computer science From VNR VJIET JNTU, Hyderabad.