A Distributed Algorithm for Delay-Constrained Unicast Routing *

H.F. Salama D.S. Reeves Y. Viniotis

Center for Advanced Computing and Communication

North Carolina State University

Box 7911, Raleigh, NC 27695

Phone: (919) 515-5348 Fax: (919) 515-2285

{hfsalama,reeves,candice}@eos.ncsu.edu

Abstract

We define the problem of unicast routing subject to delay constraints in point-to-point connection-oriented networks as a delay-constrained least-cost path problem. This problem is *NP*-complete, and therefore we propose a simple, efficient, distributed heuristic solution: the delay-constrained unicast routing (DCUR) algorithm. DCUR requires limited information about the network state to be kept at each node. This information is stored in a cost vector and a delay vector which resemble the distance vectors of some existing routing protocols. We prove the correctness of DCUR by showing that it is always capable of constructing a loop-free delay-constrained path within finite time, if such a path exists. The number of computations DCUR performs at each node participating in the path construction process is fixed, irrespective of the size of the network. The message complexity of DCUR is $O(|V|^3)$ messages in the worst case, where $|V|$ is the number of nodes in the network. However, simulation results show that, on the average, much fewer messages are required. Therefore, DCUR scales well to large networks. We also use simulation to compare DCUR to the optimal delay-constrained least-cost path algorithm, and to the least-delay path algorithm. Our results show that DCUR yields satisfactory performance with respect to both path cost and path delay.

Keywords: Routing, Delay Constraints, Quality of Service, Distributed Algorithms

*This work was supported in part by the Center for Advanced Computing and Communication at North Carolina State University, and by AFOSR grants F49620-92-J-0441DEF and F49620-96-1-0061.

1 Introduction

New distributed applications are emerging at a fast rate. These applications typically involve real-time traffic that requires quality of service (QoS) guarantees. Traffic streams carrying voice, video, or critical real-time control signals have particularly stringent end-to-end delay requirements. In addition, real-time traffic usually utilizes a significant amount of resources while traversing the network. Hence the need for routing algorithms which are able to satisfy the delay requirements of real-time traffic and to manage the network resources efficiently.

Routing problems can be divided into unicast (point-to-point) problems and multicast (point-to-multipoint or multipoint-to-multipoint) problems. In this paper, we only consider the unicast routing problem. Unicast routing protocols can be classified into two categories: distance-vector protocols, e.g., the routing information protocol (RIP) [1, 2], and link-state protocols, e.g., the open shortest path first protocol (OSPF) [3]. Distance-vector routing protocols are based on a distributed version of Bellman-Ford shortest path (SP) algorithm [4]. Considering the message complexity, distance-vector routing protocols scale well to large network sizes, because each node (router) sends periodical topology update messages only to its direct neighbors. No flooding or broadcasting operations are involved. Each node maintains only limited information about the shortest paths to all other nodes in the network. Due to their distributed nature, distance-vector protocols may suffer from looping problems when the network is not in steady state. In link-state routing protocols, on the other hand, each node maintains complete information about the network topology, and uses this information to compute the shortest path to a given destination centrally using Dijkstra's algorithm [4]. Link-state protocols have limited scalability, because flooding is used to update the nodes' topology information. They do not suffer from looping problems, however, because of their centralized nature. Recently, Garcia-Luna-Aceves and Behrens [5] proposed a distributed protocol, based on link vectors, that avoids looping problems and scales well to large networks.

Both Bellman-Ford and Dijkstra SP algorithms are exact and run in polynomial time. As the name indicates, an SP algorithm minimizes the the sum of the lengths of the individual links on the path from source to destination. The properties of the SP depend on the metric the link length represents. If unit link lengths are used, the resulting SP is a minimum-hop (MH) path. If the length of a link is a measure of the delay on that link, then an SP algorithm computes the least-delay (LD) path, and if the link length is set equal to the link cost, then an SP algorithm computes the least-cost (LC) path. Many variations of the SP problem have been studied over the years. For example, Simha and Narahari [6] studied the case where queueing delay is the dominant component of a link's delay, and Aida et. al [7] proposed an optimal SP algorithm that takes into account both the mean and the variance of the link delays. Rampal and Reeves [8] investigated the interaction between routing and call admission control for multimedia traffic. Plotkin [9] proposed a strategy that unifies the routing and call admission decisions. Chen and Liu [10] showed that the problem of routing a multimedia connection subject to a cell loss constraint is *NP*-complete and proposed heuristic solutions for that problem.

We study the problem of unicast routing of real-time traffic with end-to-end delay requirements (delay constraints) in connection-oriented networks. We formulate the problem as a delay-constrained LC (DCLC) path problem. This problem has been shown to be *NP*-complete [11]. The only prior work on this problem was reported by Widyono [12]. He proposed an optimal centralized delay-constrained algorithm to solve the DCLC problem. His algorithm, called the constrained Bellman-Ford (CBF) algorithm, performs a breadth-first search to find the DCLC path. Unfortunately, due to its optimality, CBF's worst case running times grow exponentially with the size of the network. Delay-constrained unicast routing is a special case of the delay-constrained multicast routing problem which has received a lot of attention in recent years [13]. Thus, delay-constrained multicast routing heuristics can be used to solve the DCLC problem. However, these delay-constrained multicast heuristics require complete information about the network topology to be available at every node, and their running times grow at fast rates with the network size [13]. Therefore, neither CBF nor the delay-constrained multicast routing heuristics can be applied to large networks.

We propose a distributed heuristic to solve the DCLC problem: the delay-constrained unicast routing (DCUR) algorithm. DCUR requires a limited amount of computation at any node, and its communication complexity is $O(|V|^3)$ in the worst case. On the average, however, DCUR requires much fewer messages, and therefore, it scales well to large network sizes. DCUR requires only a limited amount of information at each node. This information is stored in a delay vector and a cost vector that are similar to the distance vectors of some existing routing protocols [1]. The basic idea of DCUR is that when a node receives a request to construct a delay-constrained path to a given destination, that node is given the choice between two alternatives only. The node can either follow the direction of the LC path or the direction of the LD path. After deciding which direction to follow, the node sends a request to the next hop node in that direction to take over responsibility for the rest of the path construction operation. When the next hop node receives a request to construct a delay-constrained path, it follows the same procedure that has been explained for the previous node. Each node hands over the responsibility for path construction to the next hop node in the direction of the destination until the destination itself is reached. Limiting the number of paths to choose from at any node to only two restricts the amount of computation DCUR requires considerably.

Establishing a connection that provides guaranteed service involves routing, signaling, call admission control, and resource reservation. In this paper, we only consider the routing aspect of the problem and leave the other aspects for future investigation. The remainder of this paper is organized as follows. In section 2, we formulate the DCLC problem. In section 3, we describe the routing information needed at each node for successful execution of DCUR. Then, in section 4, we present the distributed DCUR algorithm, prove its correctness, and study its complexity. In section 5, we evaluate DCUR's performance using simulation. Section 6 concludes the paper. work in section 6.

2 Problem Formulation

A point-to-point communication network is represented as a directed simple connected network $N = (V, E)$, where V is a set of nodes and E is a set of directed links. A link $(u, v) \in E$ is an outgoing link for node $u \in V$ and an incoming link for $v \in V$. Any link $e = (u, v) \in E$ has a cost $C(e)$ (same as $C(u, v)$) and a delay $D(e)$ (same as $D(u, v)$) associated with it. $C(e)$ and $D(e)$ may take any nonnegative real values. The link delay $D(e)$ is a measure of the delay a packet experiences when traversing the link e. The link cost $C(e)$ may be either a monetary cost or some measure of the link's utilization.

We define a path as an alternating sequence of nodes and links $P(v_0, v_k) = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k$, such that every $e_i = (v_{i-1}, v_i) \in E$, $1 \leq i \leq k$. A path contains loops if not all its nodes are distinct. If all nodes are distinct, then the path is loop-free. In the remainder of this paper, it will be explicitly mentioned if a path contains loops. Otherwise a "path" always denotes a loop-free path. We will use the following notation to represent a path: $P(v_0, v_k) = \{v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_{k-1} \rightarrow v_k\}$. For a given source node $s \in V$ and destination node $d \in V$, $\mathcal{P}(s, d) = \{P_1, \ldots, P_m\}$ is the set of all possible alternative paths from s to d. The cost of a path P_i is defined as the sum of the costs of the links constituting P_i.

$$Cost(P_i) = \sum_{e \in P_i} C(e) \tag{1}$$

Similarly, the end-to-end delay along the path P_i is defined as the sum of the delays on the links constituting P_i.

$$Delay(P_i) = \sum_{e \in P_i} D(e) \tag{2}$$

The DCLC problem finds the LC path from a source node s to a destination node d such that the delay along that path does not exceed a delay constraint Δ. It is constrained minimization problem that can be formulated as follows.

Delay-Constrained Least-Cost (DCLC) Path Problem: *Given a directed network $N = (V, E)$, a nonnegative cost $C(e)$ for each $e \in E$, a nonnegative delay $D(e)$ for each $e \in E$, a source node $s \in V$, a destination node $d \in V$, and a* *positive delay constraint* Δ*, the constrained minimization problem is:*

$$\min_{P_i \in \mathcal{P}'(s,d)} Cost(P_i) \tag{3}$$

where $\mathcal{P}'(s,d)$ *is the set of paths from* s *to* d *for which the end-to-end delay is bounded by* Δ*. Therefore* $\mathcal{P}'(s,d) \subseteq \mathcal{P}(s,d)$*. If* $P_i \in \mathcal{P}(s,d)$ *then* $P_i \in \mathcal{P}'(s,d)$ *if and only if*

$$Delay(P_i) \leq \Delta. \tag{4}$$

The DCLC problem is *NP*-complete [11][1]. It is also *NP*-complete in the case of undirected networks. However, it is solvable in polynomial time if all link costs are equal or all link delays are equal.

3 Routing Information

In this section, we discuss the routing information which needs to be present at any node in the network to assure successful execution of DCUR. Then, in section 4, we describe the operation of DCUR. Every node $v \in V$ must have the following information available during the computation of the delay-constrained path: the costs of all outgoing links, the delays of all outgoing links, a cost vector, a delay vector, and a routing table.

The cost vector at node v consists of $|V|$ entries, one entry for each node w in the network. Each entry in the cost vector holds the following information:

- the destination node ID, w,
- the cost of the LC path from v to w, $least\_cost\_value(v, w)$, and
- the ID of the next hop node on the LC path from v to w, $least\_cost\_nhop(v, w)$.

Similarly, the delay vector at node v has one entry for each node w in the network. However, each entry in the delay vector holds:

- the destination node ID, w,
- the total end-to-end delay of the LD path from v to w, $least\_delay\_value(v, w)$, and
- the ID of the next hop node on the LD path from v to w, $least\_delay\_nhop(v, w)$.

The cost vectors and delay vectors are similar to the distance vectors of some existing routing protocols [1]. Distance-vector based protocols discuss in detail how to update the distance vectors in response to topology changes, and how

[1] In [11] the same problem is called the shortest constrained-weight path problem.

to prevent instability. These procedures are simple and require the contents of the distance vector at each node to be periodically transmitted to direct neighbors of that node only. They do not involve any flooding or broadcasting operations. The same procedures used for maintaining the distance vectors can be used for maintaining the cost vectors and delay vectors. We will not discuss these procedures in this paper, because our focus is on a routing algorithm that uses the cost vectors and delay vectors as input information. Thus, we assume, in the remainder of this paper, that the cost vectors and delay vectors at all nodes are up-to-date. We also assume that the link costs, the link delays, the contents of the cost vectors, and the contents of the delay vectors do not change during the execution of the routing algorithm.

In addition to the cost vector and delay vector, each node v maintains a routing table. Each entry in the routing table corresponds to an established path from a source node s to a destination d that passes through v. A routing table entry will be described in the next section. Routing table entries are created during the connection establishment phase for a session involving real-time traffic that flows from a source s to a destination d. In addition to the routing information, a routing table entry could also be used to hold information about the resources reserved for the connection from s to d. When that a real-time session terminates, the corresponding path is torn down, and all routing table entries corresponding to that path are deleted.

4 The Delay-Constrained Unicast Routing (DCUR) Algorithm

We start by presenting a simple version of DCUR, and explaining how to implement it in a distributed fashion. Then we discuss how loops may be created, and how DCUR detects them and eliminates them. After completing the description of DCUR we prove its correctness and study its complexity.

DCUR is a source-initiated algorithm that constructs a delay-constrained path connecting source node s to destination node d. The path is constructed one node at a time, from the source to the destination. Any node v at the end of the partially-constructed path can choose to add one of only two alternative outgoing links. One link is on the LC path from v to the destination, while the other link is on the LD path from v to the destination. This limitation restricts DCUR's ability to construct the optimal path, but it considerably reduces the amount of computation required at any node.

In the following, we describe a simple version of DCUR which assumes that no routing loops can occur. The source node s initiates path construction by looking up the $least\_delay\_value(s,d)$ from its delay vector. If this value is greater than the delay constraint Δ, then no delay-constrained paths exist between s and d, and DCUR fails and stops. If, however, delay-constrained paths do exist, i.e.,

$$least\_delay\_value(s,d) \leq \Delta, \tag{5}$$

the algorithm proceeds. The source s becomes the current active node, denoted $active\_node$. At all times there is only one active node, at the end of the partially-constructed path. The variable $delay\_so\_far$ is set to 0, and the variable $previous\_active\_node$ is set to $null$.

The $active\_node$ reads the ID of the next hop node on the LC path towards d, $least\_cost\_nhop(active\_node, d)$, from its cost vector. We denote $least\_cost\_nhop(active\_node, d)$ as lc_nhop for convenience. Then $active\_node$ sends a $Query$ message to lc_nhop, requesting about the LD value from lc_nhop to d. lc_nhop looks up the requested value $least\_delay\_value(lc\_nhop, d)$ from its delay vector, and sends a $Response$ message back to $active\_node$ with this information. After $active\_node$ receives the $Response$ message, it checks if

$$delay\_so\_far + D(active\_node, lc\_nhop) + least\_delay\_value(lc\_nhop, d) \leq \Delta. \tag{6}$$

If the inequality is satisfied, then there exist delay-constrained paths from $active\_node$ to d which use the link $(active\_node, lc\_nhop)$, and $active\_node$ selects the direction of the LC path towards d. If the inequality is not satisfied, then $active\_node$ selects the direction of the LD path towards d. The LD path from $active\_node$ to d is guaranteed to be part of at least one delay-constrained path from s to d; otherwise, $active\_node$ could not have been selected in a previous step (a proof is provided in subsection 4.2). After deciding which direction to follow, $active\_node$ creates a routing table entry with the following information:

- the ID of s,
- the ID of d,
- $previous\_node$ = ID of the $previous\_active\_node$,
- $next\_node = \begin{cases} lc\_nhop, & \text{if the LC path direction is chosen,} \\ least\_delay\_nhop(active\_node, d), & \text{if the LD path direction is chosen,} \end{cases}$
- $previous\_delay = delay\_so\_far$, and
- $flag = \begin{cases} LCPATH & \text{if the LC path direction is chosen,} \\ LDPATH & \text{if the LD path direction is chosen.} \end{cases}$

Then $active\_node$ adds $D(active\_node, next\_node)$ to the variable $delay\_so\_far$. Finally the $active\_node$ sends a $Construct\_Path$ message to $next\_node$ that contains: the ID of the source s, the ID of the destination d, the value of the delay constraint Δ, and the updated value of $delay\_so\_far$ which represents the delay along the already constructed path from s to $next\_node$. After sending out the $Construct\_Path$ message, $active\_node$ becomes inactive.

When a node $v \neq d$ receives a $Construct\_Path$ message, it becomes the new $active\_node$. The new $active\_node$ sets $previous\_active\_node$ to be the ID of the node which sent it a $Construct\_Path$ message. Then the new $active\_node$ executes the same procedure just described.

When the destination node d receives a $Construct\_Path$ message, it records the ID of the node which sent the message. d creates a routing table entry, with the following values: ID of the source s, ID of the destination d, $previous\_node = previous\_active\_node$, $next\_node = null$, and $previous\_delay = delay\_so\_far$. Then the destination sends an acknowledgment back to the source. When the source receives the acknowledgment message [2], it signals to the application that the path construction has been successfully completed, and traffic can be transmitted along that path.

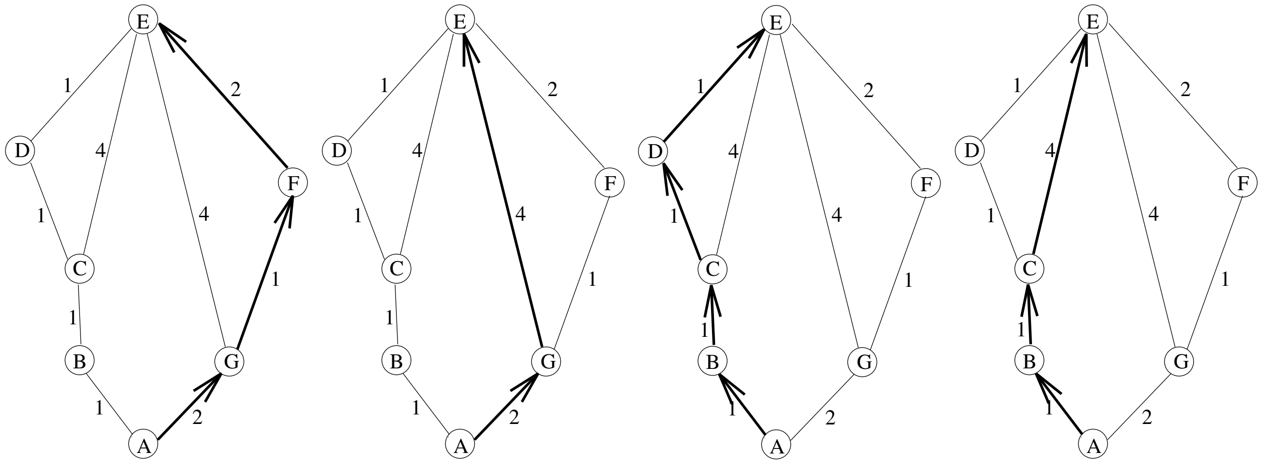
An $active\_node$, does not send a $Query$ message if the next hop node is the same on both the LC path and the LD path from $active\_node$ to the destination, i.e., $least\_cost\_nhop(active\_node, d) = least\_delay\_nhop(active\_node, d)$. It is known in advance that the LD direction satisfies the delay constraint, so there is no need for the $Query$ message. In this case, $active\_node$ sets the $flag$ in the routing table entry to $LDPATH$. The reason for that particular setting will be explained later in this section, when routing loops are discussed.

The need for the $Query$ and $Response$ messages can be completely eliminated by making use of the fact that a node transmits the contents of its cost vector and delay vector periodically to all its neighbors. Thus if a node saves a copy of the cost vector and delay vector from each of its neighbor nodes, then there is no need for the $Query$ and $Response$ messages. However, this increases the storage requirements at each node.

Figure 1 shows an example[3] of the paths obtained by different routing algorithms to connect source node A to destination node E, with a delay constraint of 3. Subfigure 1(d) shows the path DCUR constructs. DCUR proceeds as follows. The source A adds the first link on the LC path towards E, link (A, B), after checking that there exist delay-constrained paths from A to E that utilize (A, B). Then node B adds the first link on its LC path towards E, link (B, C), after checking that there exist delay-constrained paths from A to E that utilize (A, B) and (B, C). Node C next determines that the first link on its LC path towards E, link (C, D), cannot be used. This is because the subpath $\{A \rightarrow B \rightarrow C \rightarrow D\}$ is not part of any delay constrained path from A to E. Thus C decides to continue via the LD path direction. It adds the first link in that direction, link (C, E), which leads directly to the destination.

[2] The acknowledgment message can either travel the constructed path backwards or it can be sent over either the LC path or the LD path from the destination to the source. The choice is straight-forward and will not be described in this paper. Tearing down an existing path is also a simple operation that will not be discussed in this paper.

[3] Figures 1 and 2 show examples of undirected networks for simplicity. DCUR can be applied to both directed and undirected networks.



(a) Optimal DCLC path, cost = 5, delay = 3

(b) LD path, cost = 6, delay = 2

(c) Unconstrained LC path, cost = 4, delay = 4

(d) DCUR path, cost = 6, delay = 3

Figure 1: Paths constructed by different algorithms from source node A to destination node E. All link delays are equal to 1, and are not shown in the figure. Link costs are shown next to each link. The delay constraint, Δ, is equal to 3.

The paths constructed by existing distance-vector protocols are guaranteed to be loop-free if the contents of the distance vectors at all nodes are up-to-date and the network is in stable condition. However, up-to-date cost vectors and delay vectors contents and stable network condition are not sufficient to guarantee loop-free operation for DCUR. In DCUR, each node involved in the path construction operation selects either the LC path direction or the LD path direction as has been explained before. If all nodes choose the LC path direction, or all nodes choose the LD path direction, then no loops can occur, because the resulting paths are the LC path or LD path respectively. However, if some nodes choose the LC path direction while others choose the LD path direction, loops may occur. In the following subsection, we discuss how DCUR detects and eliminates loops.

4.1 Loop Removal

Figure 2 shows a scenario that results in a loop. The source node A initiates the construction of a path towards the destination node D with an imposed delay constraint value of 8. Subfigures 2(a), 2(b), and 2(c) show successive stages of path construction until a loop is created. The source A follows the LD path direction towards the destination D and starts the path construction by adding link (A, B). Node B follows the LC path direction towards D and adds link (B, C) to the path. Node C also attempts to follow the LC path direction towards D by using link (C, E). This is not possible however, because C's calculations reveal that if (C, E) is added, DCUR will not be able to construct a delay-constrained

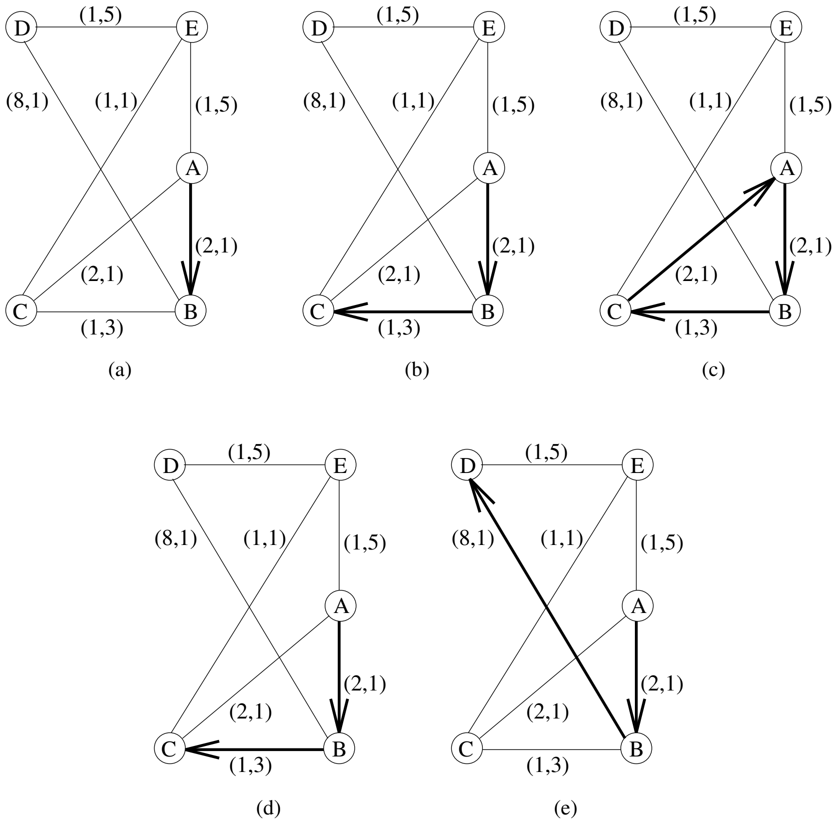


Figure 2: Example of a loop scenario. A is the source and D the destination. Link costs and link delays are shown next to each link as (cost,delay). The delay constraint, Δ, is equal to 8.

path to D (the least total delay that can be achieved in this case is 10). Thus C follows the LD path direction and adds link (C, A) to the path being constructed. This creates the loop $\{A \rightarrow B \rightarrow C \rightarrow A\}$, as shown in subfigure 2(c).

DCUR detects loops as follows. When a node receives a $Construct\_Path$ message, it searches its routing table. A loop is detected if a routing table entry already exists for the source-destination pair specified in the $Construct\_Path$ message.

The active node, $active\_node$, that detects a loop initiates the loop removal operation. The contents of $active\_node$'s routing table entry are left unchanged. $active\_node$ sends a $Remove\_Loop$ message to the previous node on the loop, $previous\_active\_node$ (the node from which $active\_node$ received the last $Construct\_Path$ message), and then $active\_node$ becomes inactive. The IDs of the source and destination nodes are all that needs to be included in the $Remove\_Loop$ message. The $Remove\_Loop$ message traverses the loop backwards, removing routing table entries, until it finds a node w whose routing table entry's $flag$ is set to $LCPATH$ indicating that this node is following the LC path direction towards the destination. There must be at least one node on the loop that follows the LC path direction, because, as we mentioned before, loops can not be created if all nodes follow the LD path direction. The $Remove\_Loop$ message is not sent any further backwards along the loop, after it arrives at w. Node w then decides to follow the LD path direction, instead of the LC path direction, in order to avoid the conditions that caused the loop. This decision can never lead to any delay constraint violations. Thus w adjusts the contents of its routing table entry so that $next\_node = least\_delay\_nhop(w, d)$ and $flag = LDPATH$. The variables $previous\_node$, $previous\_delay$, and $delay\_so\_far$ remain unchanged. Then w sends a $Construct\_Path$ message to $next\_node$, and path construction continues.

For the example of figure 2, node A detects the existence of a loop. A reacts by sending a $Remove\_Loop$ message that traverses the loop backwards. Node C receives the $Remove\_Loop$ message from A, but C is already following the LD path direction towards the destination, so all it does is to send the $Remove\_Loop$ message further backwards to B, and to delete its routing table entry, thereby removing link (C, A) from the path (subfigure 2(d)). Node B receives the $Remove\_Loop$ message. It is following the LC path direction towards the destination, so it decides to follow the LD path direction instead, and modifies its routing table entry accordingly. Thus removing link (B, C) from the path and adding link (B, D) instead. Then B continues constructing the path by sending a $Construct\_Path$ message to D, which is the destination. The final delay-constrained path from A to D is the one shown in subfigure 2(e).

It was mentioned above that, at a node w, the routing table entry's $flag$ is set to $LDPATH$ when both the LC path direction and the LD path direction share the same link to the next hop. The reason is that if the $flag$ was set to $LCPATH$, and then w received a $Remove\_Loop$ message, it would have removed the link leading to the next node in the LC path direction, and then it would have added the same link to the path again, because that link leads also to the next node in the LD path direction. The result would have been the same loop occurring twice.

The description of DCUR is now complete. Pseudo code for the algorithm is given in the appendix. In the remainder of this section, we prove the correctness of DCUR and study its complexity.

4.2 Correctness of DCUR

We verify the correctness of DCUR by proving that it can always construct a loop-free delay-constrained path within finite time, if such a path exists. If no feasible path exists for a given source-destination pair, DCUR fails immediately at the source node after it determines that inequality 5 is not satisfied. Thus there is no unnecessary overhead if no solution exists.

Theorem 1 *DCUR always constructs a delay-constrained path for a given source s and destination d, if such a path exists.*

Proof. If no feasible path exists for a given source-destination pair, DCUR fails immediately at the source node after checking that the delay along the LD path exceeds the delay constraint, i.e., inequality 5 is not satisfied. If the LD path can not satisfy the delay constraint, there can be no other path that satisfies the given constraint. If at least one delay-constrained path from s to d exists, then inequality 5 will be satisfied, and path construction can start. Initially, the source s is the only member in the path. The rest of this proof is done by induction on j, where P_j denotes the subpath constructed starting at the source s and j denotes the length of the path in hops. The basis for induction is $P_0 = \{v_0\}$ where $v_0 = s$. Since inequality 5 is satisfied and $Delay(P_0) = 0$, it follows that $Delay(P_0) + least\_delay\_value(v_0, d) \leq \Delta$. Assume that

$$Delay(P_j) + least\_delay\_value(v_j, d) \leq \Delta \tag{7}$$

where $P_j = \{v_0 \rightarrow \ldots \rightarrow v_j\}$. Inequality 7 guarantees that the subpath P_j is part of a delay-constrained path from s to d. When v_j is the $active\_node$, DCUR proceeds by adding either the first link along the LC path from v_j to d or the first link along the LD path from v_j to d. If DCUR adds the first link along the LC path, then inequality 6 must be satisfied. We restate that Inequality for the sake of clarity.

$$delay\_so\_far + D(active\_node, lc\_nhop) + least\_delay\_value(lc\_nhop, d) \leq \Delta$$

Note that $delay\_so\_far = Delay(P_j)$ and $active\_node = v_j$ and $lc\_nhop = v_{j+1}$. Thus the inequality can be rephrased as

$$\begin{aligned} Delay(P_j) + D(v_j, v_{j+1}) + least\_delay\_value(v_{j+1}, d) &= \\ Delay(P_{j+1}) + least\_delay\_value(v_{j+1}, d) &\leq \Delta \end{aligned} \tag{8}$$

The other alternative for DCUR is to proceed from $active\_node = v_j$ by adding the first link along the LD path, link (v_j, v_{j+1}). Using $least\_delay\_value(v_j, d) = D(v_j, v_{j+1}) + least\_delay\_value(v_{j+1}, d)$, the inequality 7 can be rewritten as

$$\begin{aligned} Delay(P_j) + D(v_j, v_{j+1}) + least\_delay\_value(v_{j+1}, d) &= \\ Delay(P_{j+1}) + least\_delay\_value(v_{j+1}, d) &\leq \Delta \end{aligned} \tag{9}$$

In both cases, v_{j+1} becomes the next $active\_node$. Therefore DCUR guarantees that the subpath from s to the $active\_node$ is delay-constrained and that it is part of a delay-constrained path from s to d. DCUR stops only when d

becomes the $active\_node$. □

Theorem 2 *The final path constructed by DCUR for a given source s and destination d does not contain any loops.*

Proof. We denote s as v_0 and d as v_k for convenience. The final path constructed by DCUR, $P(v_0, v_k) = \{v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_{k-1} \rightarrow v_k\}$, contains a loop if it includes the same node twice, i.e., $v_i = v_j$ where $i \neq j$ and $0 \leq i, j \leq k$. DCUR starts at the source, and adds one link at a time to the path being constructed. When DCUR adds a link (v_y, v_x), then, if the node v_x is added to the path for the first time, DCUR creates a routing table entry for the given source-destination pair at v_x. The resulting path after v_x is added for the first time is $\{v_0 \rightarrow \ldots \rightarrow v_x\}$. If, at a subsequent point in time, DCUR adds a link (v_z, v_x), the result would be $\{v_0 \rightarrow \ldots \rightarrow v_x \rightarrow \ldots \rightarrow v_z \rightarrow v_x\}$. This is a path that contains a loop. Node v_x becomes the $active\_node$ immediately after link (v_z, v_x) is added. v_x may be the source v_0 or any node in the network other than the destination v_k. v_x checks its routing table and finds that a routing table entry already exists for the given source-destination pair, thus detecting a loop. As a result v_x initiates the loop removal operation. The loop is broken using the technique described in subsection 4.1. Then, the path construction resumes, and DCUR attempts to use an alternate path towards the destination v_k. Thus the final path constructed by DCUR, $P(v_0, v_k)$, can not include the source v_0 or any intermediate node more than one time, because each time such a node is added to the path for the second time, it immediately detects a loop and removes it. The destination v_k is also included only once in the path, because DCUR stops as soon as it reaches the v_k for the first time. □

Theorem 3 *The execution time of DCUR for a given source s and destination d is always finite.*

Proof. If no delay-constrained paths exist, then DCUR fails immediately at the source after determining the inequality 5 is not satisfied. If inequality 5 is satisfied, then DCUR proceeds. If no loops occur, then, after adding at most $(|V| - 1)$ links, DCUR reaches the destination d. It is sufficient to show that, even if loops occur, DCUR will still reach d within finite time. Assume that DCUR constructs a subpath that contains a loop, $P_{loop} = \{s \rightarrow \ldots \rightarrow v_q \xrightarrow{e_{LC}} \ldots \rightarrow v_x^{(1)} \rightarrow \ldots \rightarrow v_r \xrightarrow{e'_{LC}} \ldots \rightarrow v_x^{(2)}\}$. $v_x^{(1)}$ denotes the first occurrence of v_x and $v_x^{(2)}$ denotes the second occurrence of the same node v_x. After a loop is detected at $v_x^{(2)}$, the loop removal operation described in subsection 4.1 traverses P_{loop} backwards, removing links, until it reaches a node v_r whose routing table entry's $flag$ is set to $LCPATH$. There must be at least one such node between $v_x^{(1)}$ and $v_x^{(2)}$ (v_r may be $v_x^{(1)}$), because loops can not be created if all nodes follow the LD path direction towards the destination. Node v_r is then forced to follow the LD path direction, instead of the LC path direction, and it removes the first link on the LC path from v_r to the destination, link e'_{LC}, and adds the first link on the LD path instead. This link switching procedure prevents the same subpath P_{loop} from occurring another time. If, at a subsequent point in time, another loop occurs, and DCUR removes the node v_r completely from the path as part of the loop removal operation. Then if, when path construction resumes after that, DCUR adds the node v_r and the link e'_{LC} to the path once again. Still the the same subpath P_{loop} can not be reconstructed another time, because the second loop removal operation (which removed node v_r) must have selected a node v_q, further upstream from v_r (v_q may be s), that followed the LC path direction, and forced it to follow the LD path direction instead, thus removing the link e_{LC} and replacing it with another outgoing link from v_q. Therefore the same subpath containing a loop, P_{loop}, can not be constructed twice during the execution of DCUR. Each loop removal operation removes a number of nodes (≥ 0) from the path being constructed, and forces one more of the remaining nodes to follow the LD path direction instead of following the LC path direction. Therefore if loops keep occurring, the solution paths, DCUR attempts to construct, will converge towards the LD path from the s to d. Since the size of the network is finite, the number of possible subpaths starting at s and containing a loop is also finite. Hence DCUR will always converge towards the LD path solution from s to d within a finite time, if its attempts to reach the destination d via other paths keep failing because of loops. The LD path is guaranteed to reach the destination and to satisfy the delay constraint or else DCUR would have failed and stopped immediately at the source. $\square$

4.3 Complexity of DCUR

Each time a node receives a $Construct\_Path$ message or a $Remove\_Loop$ message, it performs a fixed amount of computations, irrespective of the size of the network. Therefore, the computational complexity of the proposed distributed algorithm at any node is $O(1)$ time[4].

We now consider the worst case message complexity of DCUR, i.e., the number of messages needed in the worst case, in order to construct a path for a given source-destination pair. If no loops occur, then the number of messages needed to construct a path is proportional to the number of links in the path. This is because a node running DCUR exchanges three messages (one $Query$ message, one $Response$ message, and one $Construct\_Path$ message) to add one link. For a network size of $|V|$ nodes, the longest possible loop-free path from source to destination consists of $|V|$ nodes and $(|V| - 1)$ links. Therefore the number of messages needed in the worst case is $O(|V|)$, if it is guaranteed that no loops will occur. Unfortunately, the occurrence of loops complicates the analysis.

The tree of the LC paths from any node in the network to the destination node d, denoted *LCTREE*, consists of $(|V|-1)$ links. Similarly, the tree of the LD paths from any node in the network to the destination d, denoted *LDTREE*, also consists

[4]In this analysis we consider the complexity of DCUR only. We do not consider the complexity of the underlying mechanisms for maintaining and updating the cost vectors and delay vectors.

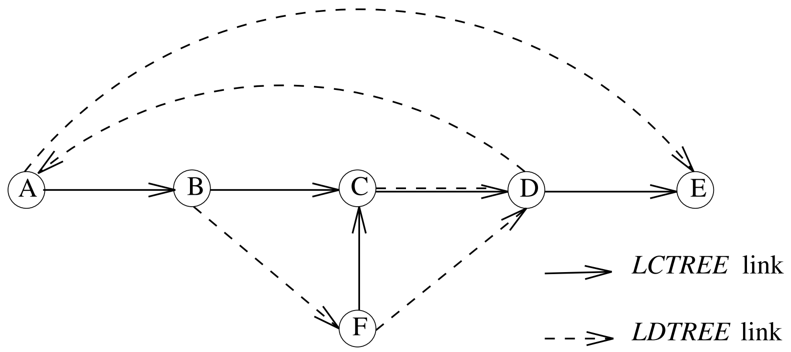


Figure 3: An example of a subnetwork constructed by taking the union of the *LCTREE* and the *LDTREE*. The destination is node E.

of $(|V|-1)$ links. The union of these two trees is a subnetwork $N' = (V, E')$, where $(|V|-1) \leq |E'| \leq 2*(|V|-1)$, because some links may be members of both trees. Figure 3 shows an example of the union of an *LCTREE* and an *LDTREE*. In this example, the link (C, D) is a member of both trees. The $|E'|$ links are the only links considered by DCUR when constructing a path from a source s to the destination d, because, as has been explained before, at any node DCUR considers only the LC path direction and the LD path direction towards the destination.

Let the links of *LCTREE* be called tree links. We add the links of the *LDTREE* to the *LCTREE* to obtain the subnetwork N'. The links of the *LDTREE* which are not already in the *LCTREE* will be classified into one of the following three link types:

- a back link which may result in a loop,
- a descendent link which may provide one or more nodes with two alternate paths towards the destination, or
- a cross link which may also provide one or more nodes with two alternate paths towards the destination[5].

In the example of figure 3, links (A, B), (B, C), (C, D), (D, E), and (F, C) are tree links. The link (D, A) is a back link. Links (A, E) and (F, D) are descendent links, and the link (B, F) is a cross link.

A subnetwork N' has X back links, Y descendent links, and Z cross links where $0 \leq X, Y, Z \leq (|V|-1)$ and $X + Y + Z \leq (|V|-1)$. Adding a back link to a path under construction may or may not result in loop. Since we are studying the worst case, we assume that adding a back link to a path always results in a loop. Consider only one back link, e. Link e may be added and removed from the path being constructed several times, if it is reachable via multiple alternate paths from the source node. A loop results each time e is added. The back link e is reachable via $(Y + Z)$ alternate paths in the worst case. This happens when the $(Y + Z)$ descendent links and cross links are upstream (closer to

[5] See, for example [14], for definitions of the terms: back link, descendent link, and cross link.

the source node) from the back link e. In this case each time DCUR attempts to use one of the $(Y + Z)$ resulting alternate paths, it may continue downstream (towards the destination) and add the link e, thus creating a loop. If DCUR attempts to use all $(Y + Z)$ alternate paths while constructing the delay-constrained path, the link e will be added and removed $(Y + Z)$ times, which means that $(Y + Z)$ loops will be created and removed during the path construction. The example of figure 3 is not a worst case scenario. However, it shows how the back link (D, A) can be reached via three alternate paths when node A is the source. The first alternative is the original path along the *LCTREE*: $\{A \rightarrow B \rightarrow C \rightarrow D\}$. The second alternative was created due to the addition of the cross link (B, F), and it is $\{A \rightarrow B \rightarrow F \rightarrow C \rightarrow D\}$. The final alternative is $\{A \rightarrow B \rightarrow F \rightarrow D\}$. This path was brought to existence by the descendent link (F, D).

So far we considered only one back link. However, the subnetwork N' contains X back links. In the worst case, each of the X back links is reachable via $(Y + Z)$ alternate paths. In this case we may end up with $X * (Y + Z)$ loops. Since $X + Y + Z \leq (|V| - 1)$, it follows that, in the worst case, DCUR may create and remove $O(|V|^2)$ loops before completing the construction of the delay-constrained path.

The largest possible loop consists of $(|V| - 1)$ nodes and $(|V| - 1)$ links (the destination can not be part of a loop in DCUR). A maximum of three messages are needed to add one loop link. Thus it takes $O(|V|)$ messages to create the largest loop. One message is needed for removing one loop link, which means that at most $O(|V|)$ messages are needed if all loop links have to be removed before path construction resumes. Therefore $O(|V|)$ messages are needed, to create and remove the largest loop. In the worst case $O(|V|^2)$ loops may be created and removed. This means that, in the worst case, DCUR needs $O(|V|^3)$ messages to handle loops. Compared to the $O(|V|)$ messages required to add the permanent links that constitute the final loop-free path, it is obvious that, in the worst case, loop handling dominates the operation of DCUR, and that the overall worst case message complexity of DCUR is $O(|V|^3)$. Fortunately, our simulation results show that DCUR's average performance is much better than the worst case just studied. These results will be presented in the next section.

5 Simulation Results

We used simulation for our evaluation of the average performance of DCUR. Full duplex, directed, simple, strongly connected networks of different sizes with homogeneous link capacities of 155 Mbps (OC3) were used in the experiments. The positions of the nodes were fixed in a rectangle of size 3000 $*$ 2400 Km^2, roughly the area of the USA. A random generator (based on Waxman's generator [15] with some modifications) was used to create links interconnecting the

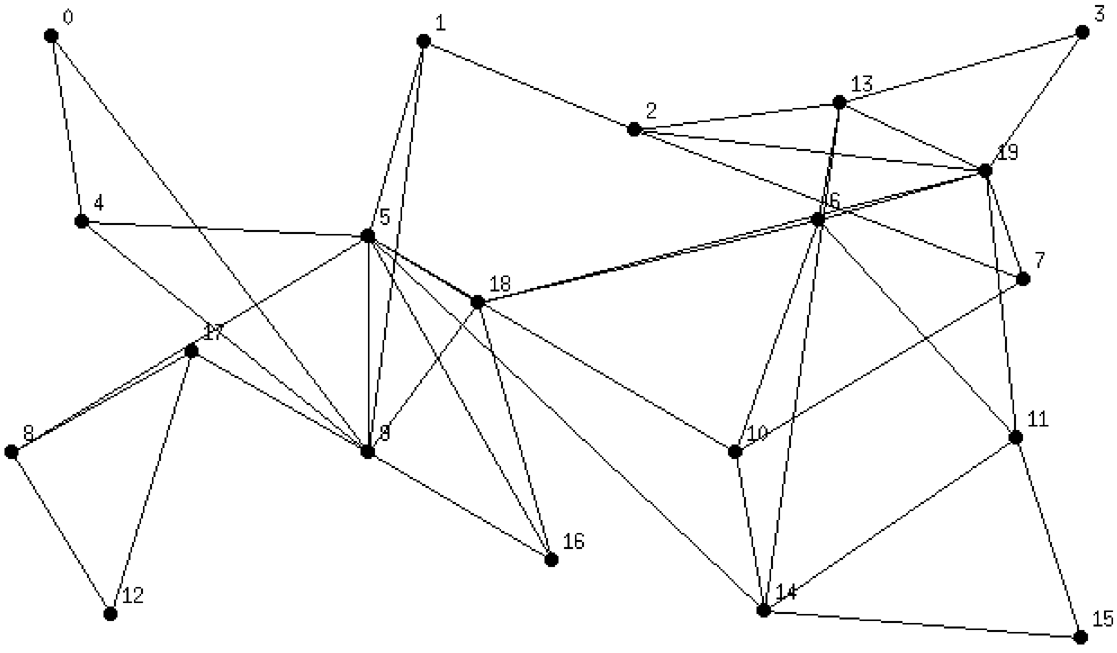


Figure 4: A randomly generated network, 20 nodes, average degree 4.

nodes. The output of this random generator is always a connected network in which each node's degree is at least 2. The probability a link exists between any two nodes u and v is given by:

$$P(u,v) = \beta \exp \frac{-l(u,v)}{L\alpha} \tag{10}$$

where $l(u,v)$ is the distance between u and v, and L is the maximum distance between any two nodes. The parameter α controls the ratio of short links to long links, while β controls the average node degree of the network. We obtained realistic network topologies with an average node degree of 4 by carefully adjusting the values of α and β[6]. Figure 4 shows an example of a randomly generated network.

The propagation speed through the links was taken to be two thirds the speed of light. Under this assumption, the size of the rectangle enclosing our network is $15 * 12$ msec^2. In addition, we assumed a high-speed networking environment with small packet (cell) sizes and limited buffer space at each node. The link propagation delay was dominant under these assumptions, and the queueing component of the link delay was neglected. The link delays were thus symmetric, $D(u,v) = D(v,u)$, because the link lengths were symmetric.

We defined the cost, $C(e)$, of link e, as a function of its utilization[7]. Link costs were asymmetric, because $C(u,v)$

[6] The average node degree of current internetworks ranges between 3 and 4. The denser the network connectivity the clearer the advantages and disadvantages of a routing algorithm appear in the simulation results. That's why we used an average node degree of 4.

[7] In our experiments we reserved bandwidth along the constructed path. The amount of bandwidth reserved on each link of a path connecting a given source-destination pair was taken equal to the equivalent capacity of the traffic streaming from the source to the destination. We set the cost of

and $C(v, u)$ were independent. Two experiments were conducted over the networks we have just described.

5.1 The Average Message Complexity of DCUR

In the first experiment, we measured the average number of messages DCUR requires to establish a delay-constrained path. For each run of the experiment, we generated a random set of links to interconnect the fixed nodes, we selected a random source node, and we generated random background traffic to utilize each link. The cost (utilization) of a link was a random variable uniformly distributed between 5 Mbps and 125 Mbps. The experiment was repeated with network sizes ranging from 20 nodes up to 200 nodes. We also varied the delay constraint value from 15 msec to 55 msec. The delay constraint represents only an upper bound on the end-to-end propagation time across the network. Relatively small values were chosen for the delay constraint in order to allow the higher level end-to-end protocols, at both the source and the destination, enough time to process the transmitted information without affecting the quality of the real-time session. In this experiment, we measured the average number of messages exchanged between the nodes which execute the distributed DCUR algorithm to construct a delay-constrained path. Note that any message generated by DCUR at some node is always destined to an immediate neighbor of that node. Therefore any DCUR generated message travel a distance of one only hop. Unless otherwise stated, DCUR was run repeatedly until confidence intervals of less than 5% of the mean value, using 95% confidence level, were achieved for all measured values presented in this subsection and in the next subsection. At least 500 different networks were simulated for each measured value.

Figure 5 shows the average number of messages versus the size of the network for three different values of the delay constraint: a strict delay constraint of 20 msec, a moderate delay constraint of 35 msec, and a lenient delay constraint of 50 msec. All three curves of figure 5 indicate clearly that the average number of messages grows very slowly with the size of the network. For any of the delay constraint values shown in the figure, doubling the size of the network increases the average number of DCUR's messages by roughly one message only. Thus the average growth rate of the number of messages is roughly logarithmic in the network size, for the experiments we ran.

The number of messages exchanged while constructing a path is smallest when the delay constraint value is small. This is due to the following:

- A path that satisfies the strict delay constraint consists on the average of fewer links than a path the satisfies a lenient delay constraint.
- DCUR is forced to follow the LD path direction most of the time in order to satisfy the strict delay constraint.

a link to be equal to the sum of the equivalent capacities of the traffic streams traversing that link.

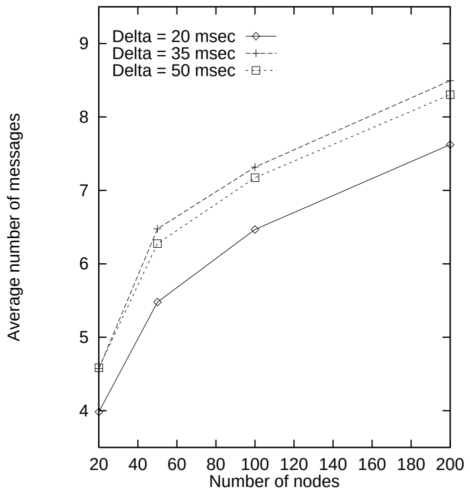


Figure 5: Average number of messages, variable network size, average node degree 4, three delay constraint settings: 20 msec, 35 msec, and 50 msec.

> Therefore, the probability of the occurrence of a loop is small. As has been discussed in the previous section, the occurrence of loops increases in the number of messages.

When the delay constraint is increased to 35 msec, the number of messages is largest. One reason is that the average number of links on the solution path is greater than it is when the delay constraint is only 20 msec[8]. The other reason is that 35 msec is a moderately strict delay constraint, and DCUR may be able to follow the LC path direction at some nodes and to follow the LD path direction at others. This toggling between LC path direction and LD path direction increases the probability loop occurrence, and hence increases the average number of messages exchanged.

Increasing the delay constraint further, from 35 msec to 50 msec, leads to a reduction in the average number of messages. 50 msec is a lenient delay constraint. Thus DCUR is able to follow the LC path direction most of the time without violating the delay constraint, and therefore it no longer toggles between the LC path direction and the LD path direction. The consequence is that loops occur rarely. Of course when the delay constraint is increased to 50 msec, the average number of links per path also increases, but figure 5 indicates that the lack of loop occurrence has the dominant effect in this case.

It has been mentioned in the previous section that when the LC path direction and the LD path direction from a node towards the destination coincide, then there is no need for sending the $Query$ message and waiting for the $Response$ message. Both messages can be eliminated, and only one message is needed to add the next link, the $Construct\_Path$

[8]For a 200-node network the average number of links per path is 4.28 for a 20 msec delay constraint, 4.72 for a 35 msec delay constraint, and 5.12 for a 50 msec delay constraint.

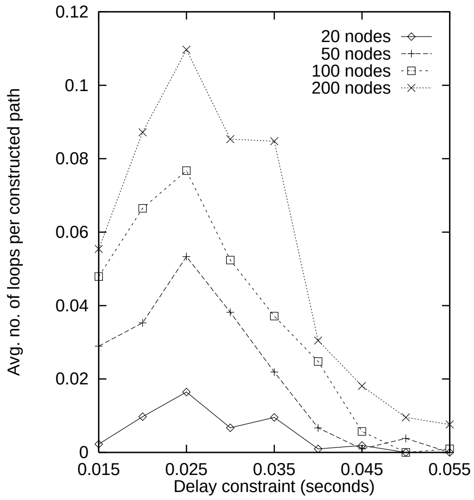


Figure 6: Average number of loops occurring while constructing a single delay-constrained path, network sizes of 20 nodes, 50 nodes, 100 nodes, and 200 nodes, average node degree 4, variable delay constraint.

message. Otherwise, three messages are needed to add one link. Our simulation results show that, at many nodes, the LC path direction and the LD path direction towards the destination are identical, and thus eliminating the $Query$ and $Response$ messages at such nodes, results in a significant reduction in the number of messages per added link. For example, figure 5 shows that for a 200-node network and a delay constraint of 50 msec, on the average 8.3 messages are needed to construct the path. We also measured the average number of links on the path, and it is 5.12 links. Thus (8.3/5.12) = 1.62 messages are needed per added link assuming that no loops occur. This is more than a 45% reduction when compared to the need for 3 messages per added link if the $Query$ and $Response$ messages were always exchanged.

In order to verify our assumption, that loops occur most frequently when the delay constraint is moderately strict, we measured the average number of loop occurrences during one successful run of DCUR, i.e., a run that successfully constructs a delay-constrained path for a given source-destination pair. We found that loops do not occur frequently (less than 12 loops every 100 successful runs of DCUR). Therefore, it was not possible (due to the excessive simulation times) to repeat the experiment until small enough confidence intervals were achieved for the measured values of the average number of loop occurrences. 1,000 successful runs of DCUR were simulated for each point in figure 6. Figure 6 shows the average number of loop occurrences per successful run of DCUR versus the delay constraint for different network sizes. Figure 6 shows that loops occur most frequently when the delay constraint value ranges from 20 msec to 45 msec. When the delay constraint is lenient (larger than 45 msec) loop occurrences are very infrequent, less than one loop every 100 successful runs of DCUR. The average number of loop occurrences also decreases when strict delay constraint values of less than 20 msec are used. It is obvious from figure 6 that loops do not occur frequently for all the network sizes we simulated. However, the figure also indicates that loops occur more frequently as the size of the network increases.

5.2 Comparison to Other Algorithms

In this subsection, we show the results of the second experiment which compares DCUR with two algorithms that are also suitable for delay-sensitive applications. The first algorithm is the LD path algorithm, or simply LDP. LDP is optimal with respect to the end-to-end delay, but it does not attempt to minimize the cost of the constructed path. Therefore, it may result in inefficient utilization of the link bandwidth. Simple, distributed implementations of LDP already exist over current networks. The other algorithm is CBF which was briefly described in section 1. CBF constructs the optimal DCLC path, but it is centralized and its execution time grows exponentially with the network size. Note that CBF and LDP are the only existing unicast routing algorithms capable of satisfying the delay requirements of real-time applications.

The structure of the second experiment is similar to that of the first experiment. The only difference is that for each randomly selected source-destination pair we applied DCUR, LDP, and CBF, one at a time, to construct the delay-constrained path from source to destination. For each algorithm, we measured the average path cost, the average inefficiency relative to CBF, the average end-to-end delay, and the success rate (how frequently the algorithm succeeds to find a delay-constrained path). The average inefficiency of an algorithm x is defined as:

$$inefficiency_x = \frac{(cost_x - cost_{CBF})}{cost_{CBF}} \tag{11}$$

We show the results of the experiment for 200-node networks and variable delay constraint. Figure 7 shows the average inefficiency of LDP and DCUR relative to CBF. When the delay constraint is small (15 msec) the number of alternate delay-constrained paths, available for the algorithms to choose from, is small, and therefore the differences between the algorithms are also small. For delay constraint values between 20 msec and 45 msec, DCUR is up to 10% worse than the optimal CBF. The reason is that, because of the tight delay constraint, DCUR can not always follow the unconstrained LC path direction. In some cases, it has to follow the LD path direction instead. The toggling between these two directions affects DCUR's ability to create low-cost paths. However, DCUR remains on the average more efficient than LDP. When the value of the delay constraint exceeds 45 msec, its effect on the constructed path is minimal. In that range, DCUR's inefficiency approaches zero, because it almost exclusively elects to follow the LC path direction. LDP does not attempt to minimize the path cost at all. That's why its inefficiency is up to 50% when the delay constraint

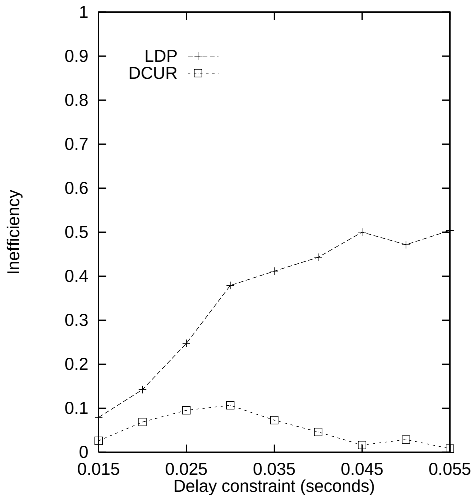


Figure 7: Inefficiency, 200-node networks, average node degree 4, variable delay constraint.

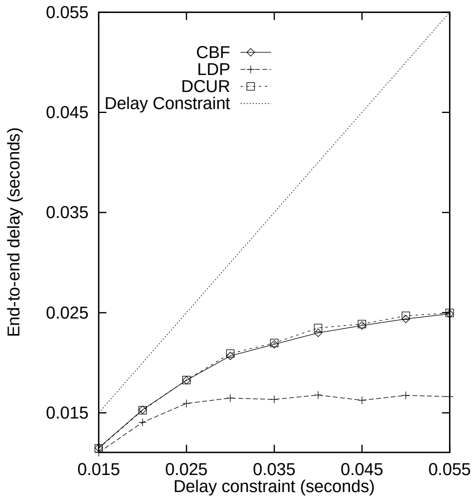


Figure 8: Average end-to-end delay, 200-node networks, average node degree 4, variable delay constraint.

value is large.

Figure 7 indicates that the path cost of DCUR is always within 10% from the path cost of the optimal CBF. Thus DCUR's cost performance is quite satisfactory, especially when considering that CBF is a centralized algorithm that requires global information about the network topology while DCUR is a distributed heuristic that requires only limited information to be maintained at each node (one cost vector and one delay vector).

The average end-to-end delays of DCUR and CBF are considerably larger than the minimal delays achieved by LDP as shown in figure 8. This is not a big advantage for LDP, though. More important is that all three algorithms have the same success rate in satisfying the imposed delay constraint. The success rates of the three different algorithms are identical, because all of them are always capable of constructing a delay-constrained path, if one exists. All three algorithms achieve a 100% success rate when the delay constraint value is greater than 30 msec. As the delay constraint value decreases below 30 msec the success rate decreases indicating that delay-constrained solutions do not always exist for networks spanning large areas when the delay constraint value is sufficiently small.

In addition to the 200-node networks, we simulated 20-node, 50-node, and 100-node networks as well. The results for the different network sizes are similar to results shown above for the 200 nodes case. This indicates that the performance of the different algorithms relative to each other does not depend significantly on the network size.

6 Conclusions

We studied the delay-constrained routing problem in point-to-point connection-oriented networks. Our work was motivated by the fast evolution of delay-sensitive distributed applications. We formulated the problem as a delay-constrained least-cost path problem, which is known to be *NP*-complete. Therefore we proposed a distributed, source-initiated heuristic solution, the delay-constrained unicast routing (DCUR) algorithm, to avoid the excessive complexity of the optimal solutions. DCUR is always capable of constructing a delay-constrained path within a finite time, if one exists, for a given source-destination pair. DCUR requires only a limited amount of information at each node. The information at each node is stored in a cost vector and a delay vector. These vectors are constructed and maintained in exactly the same manner as the distance vectors which are widely deployed over current networks. DCUR is capable of detecting and eliminating any loops that may occur while it constructs a delay-constrained path. We proved the correctness of DCUR by showing that it is always capable of constructing a loop-free delay-constrained path within finite time, if such a path exists. The number of computations at each node participating in the path construction process is fixed, irrespective of the network size. The worst case message complexity of DCUR is dominated by the occurrence and removal of loop. It requires $O(|V|^3)$ messages to construct a single path in the worst case. Fortunately, however, our simulation results show that DCUR requires much fewer messages on the average, because loop occurrence is rare in realistic networks. We compared the performance of DCUR to CBF, which is an optimal DCLC algorithm with running times that grow exponentially with the size of the network. We also compared DCUR to LDP, a shortest path algorithm that minimizes the end-to-end delay, but does not attempt to minimize the path cost. Our results indicated that DCUR yields satisfactory performance with respect to both path cost and path delay. Our evaluation of the cost performance of the algorithms showed that DCUR is always within 10% from the optimal CBF, while LDP is up to 50% worse than optimal in some cases. In summary, DCUR is a simple, efficient, distributed algorithm that scales well to large network sizes.

References

[1] C. Hedrick, “Routing Information Protocol.” Internet RFC 1058, June 1988.

[2] G. Malkin, “RIP Version 2, Carrying Additional Information.” Internet RFC 1723, November 1994.

[3] J. Moy, “OSPF Version 2.” Internet RFC 1583, March 1994.

[4] D. Bertsekas and R. Gallager, *Data Networks*. Prentice-Hall, second ed., 1992.

[5] J. Garcia-Luna-Aceves and J. Behrens, “Distributed, Scalable Routing Based on Vectors of Link States,” *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 1383–1395, October 1995.

[6] R. Simha and B. Narahari, “Single Path Routing with Delay Considerations,” *Computer Networks and ISDN Systems*, vol. 24, pp. 405–419, 1992.

[7] M. Aida, I. Nakamura, and T. Kubo, “Optimal Routing in Communication Networks with Delay Variations,” in *Proceedings of IEEE INFOCOM ’92*, pp. 153–159, 1992.

[8] S. Rampal and D. Reeves, “An Evaluation of Routing and Admission Control Algorithms for Multimedia Traffic,” *Computer Communications*, vol. 18, pp. 755–768, October 1995.

[9] S. Plotkin, “Competitive Routing of Virtual Circuits in ATM Networks,” *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 1128–1136, August 1995.

[10] W.-T. Chen and U.-J. Liu, “Routing Problem with Performance Requirement Translation for Multimedia Communications in an ATM Wide-Area Network,” in *Proceedings of IEEE ICC ’94*, pp. 1490–1494, 1994.

[11] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Co., 1979.

[12] R. Widyono, “The Design and Evaluation of Routing Algorithms for Real-Time Channels,” Tech. Rep. ICSI TR-94-024, University of California at Berkeley, International Computer Science Institute, June 1994.

[13] H. Salama, D. Reeves, Y. Viniotis, and T.-L. Sheu, “Evaluation of Multicast Routing Algorithms for Real-Time Communication on High-Speed Networks,” in *Proceedings of the Sixth IFIP Conference on High Performance Networking (HPN ’95)*, pp. 27–42, Chapman and Hall, September 1995.

[14] S. Baase, *Computer Algorithms, Introduction to Design and Analysis*. Addison-Wesley Publishing Company, 2nd ed., 1988.

[15] B. Waxman, “Routing of Multipoint Connections,” *IEEE Journal on Selected Areas in Communications*, vol. 6, pp. 1617–1622, December 1988.

Appendix Pseudo-Code of DCUR

First here is a list of all the control messages exchanged between nodes implementing DCUR:
$Construct_Path($source node, destination node, delay constraint value, value of the delay from the source to the downstream node receiving the message$)$
$Query($destination node$)$
$Response($destination node, least delay value from the responding node to the destination$)$
$Remove_Loop($source node, destination node$)$

The following function is executed by the source node s when it receives a request from an application to construct a delay-constrained path to a destination node d.

```
Initiate_Path_Construction(source node s, destination node d, delay constraint Δ) {
    if least_delay_value(s, d) > Δ send a failure indication to the application;
    else {
        active_node := s;
        previous_active_node := null;
        delay_so_far := 0;
        call Path_Construction(active_node, previous_active_node, s, d, Δ, delay_so_far);
    };
};
```

The following function is executed by node $active\_node$ when it receives a $Construct\_Path$ message from a node $previous\_active\_node$. It is also called a the source node when initiating the path construction process.

```
Path_Construction(current node active_node, previous node previous_active_node, source node s,
                        destination node d, delay constraint Δ, current delay delay_so_far) {
    if active_node = d {
        create a routing table entry with source := s, destination := d, previous_node := previous_active_node,
        next_node := null, and previous_delay := delay_so_far;
        send an acknowledge message (path construction is complete) back to s;
    }
    else {
        if a routing table entry corresponding to the source s and destination d already exists,
            send a Remove_Loop(s, d) message to previous_active_node;
        else {
            use_LDPATH := False;
            if least_cost_nhop(active_node, d) = least_delay_nhop(active_node, d)
                use_LDPATH := True;
            if use_LDPATH = False {
                lc_nhop := least_cost_nhop(active_node, d);
                send Query(d) message to lc_nhop;
                wait to receive a Response(d, delay) message from lc_nhop;
                if (delay_so_far + D(active_node, lc_nhop) + delay) ≤ Δ {
                    create a routing table entry with source := s, destination := d,
                        previous_node := previous_active_node, next_node := lc_nhop,
                            previous_delay := delay_so_far, and flag := LCPATH;
                    delay_so_far := delay_so_far + D(s, lc_nhop);
                    send a Construct_Path(s, d, Δ, delay_so_far) message to lc_nhop;
                }
                else use_LDPATH := True;
            };
            if use_LDPATH = True {
                ld_nhop := least_delay_nhop(n, d);
                create a routing table entry with source := s, destination := d,
                    previous_node := previous_active_node, next_node := ld_nhop,
                        previous_delay := delay_so_far, and flag := LDPATH;
                delay_so_far := delay_so_far + D(s, ld_nhop);
                send a Construct_Path(s, d, Δ, delay_so_far) message to ld_nhop;
            };
        };
    };
};
```

The following function is executed by node n when it receives a $Query$ message from node $active\_node$.

Process_Query(current node n, querying node $active\_node$, destination node d) {
  send a $Response(d, least\_delay\_value(n, d))$ message back to $active\_node$;
};

The following function is executed by node $active\_node$ when it receives a $Remove\_Loop$ message.

```
Loop_Removal(current node active_node, source s, destination d) {
    find the routing table entry corresponding to s and d;
    in that routing table entry, if flag = LCPATH {
        nhop := least_delay_nhop(active_node, d);
        in the routing table entry corresponding to s and d, set {
            flag := LDPATH;
            next_node := nhop;
        };
        delay_so_far := previous_delay + D(active_node, nhop);
                        (previous_delay is given in the routing table entry)
        send a Construct_Path(s, d, Δ, delay_so_far) message to nhop;
    }
    else {
        send a Remove_Loop(s, d) message to the previous_node given in that routing table entry;
        delete that routing table entry;
    };
};
```