

# Forward-Only Uni-Directional Routing

Jorge A. Cobb

Department of Computer Science  
The University of Texas at Dallas  
Richardson, TX 75083-0688  
{cobb}@utdallas.edu

**Abstract**—Wireless computer networks have properties that differ significantly from traditional wireline networks. One prominent difference is the presence of unidirectional channels. That is, it may be possible for a computer  $p$  to send a message to a neighboring computer  $q$ , but it may not be possible for computer  $q$  to send a message to computer  $p$ . This introduces significant changes in routing protocols. In particular, routing protocols that tolerate unidirectional links incur an  $O(N^2)$  message overhead between neighboring nodes, where  $N$  is the number of nodes in the network. This is significantly higher than the typical  $O(N)$  message overhead in some traditional routing protocols. We present a protocol with an  $O(N)$  message overhead even in the presence of unidirectional links. This low overhead comes at the price of not always finding the optimum path. However, the protocol is guaranteed to always find a path between each pair of nodes. Furthermore, the cost of this path is guaranteed not to exceed the cost of the optimal bidirectional path between the nodes.

## I. INTRODUCTION

A collection of computing nodes that communicate via a wireless medium without an additional wired infrastructure is called an ad-hoc network. The nodes in an ad-hoc network usually have multiple constraints, such as limits on battery life, memory, and computing power. Battery life is critical, since nodes are typically battery operated, and have a short battery life. To maximize battery life, an efficient utilization of the wireless transmitter/receiver is important. Two techniques are commonly used: setting the transmitter/receiver to standby mode if no traffic is present, and reducing the transmission power.

Reducing the transmission power has the disadvantage of increasing the likelihood of unidirectional channels. That is, if the transmission power of node  $p$  is greater than the transmission power of node  $q$ , it may be possible for  $q$  to receive a message from  $p$ , but not for  $p$  to receive a message from  $q$ . Other factors foster unidirectional channels, such as different signal-to-noise ratios at each node [14]. It has been shown that the number of unidirectional channels may become significant [16]. In this paper, we present a routing protocol that tolerates unidirectional channels while maintaining a low message complexity.

Routing protocols developed for ad-hoc networks are divided into two approaches. In the proactive approach, each node maintains at all times a path to each node in the network. Examples of this approach may be found in [1], [5], [9], [10]. In the reactive approach, a path to the destination node is found only in response to a request from a source node, and typically requires

flooding the network in a search for the destination node. Examples of this approach may be found in [8], [15]. Both approaches have advantages, and each has a set of network conditions under which it is the best.

In this paper, we focus on the proactive approach. Furthermore, we desire only an  $O(N)$  message overhead between neighbors. In traditional routing protocols, there are two methods that obtain this efficiency: distance vectors [2] and source-trees [3]. With distance vectors, each node forwards to all its neighbors a vector with the estimated cost to each node in the network. With source-trees, each node forwards to all its neighbors the set of network paths from itself to all other nodes in the network. The union of all these paths form a tree. Source-trees outperform distance vectors, since their overhead is approximately the same, but they avoid long-term loops.

Unfortunately, source-trees and distance vectors cannot be directly applied to networks with unidirectional channels. Consider Figure 1, and let  $p$  be the source and  $q$  be the destination. Consider the case of distance vectors. Node  $s$  determines that its distance to  $q$  is one, but it cannot give this information to  $r$  since there is no channel from  $s$  to  $r$ . Node  $u$  determines that its distance to  $q$  is one, and it forwards this distance to  $t$ . This distance is useless to  $t$  (and hence also to  $p$ ) since there is no channel from  $t$  to  $u$ , and thus,  $t$  cannot choose  $u$  as its next hop to  $q$ . The case of source-trees is similar.

A solution to the above problem using distance vectors was proposed in [4], [5], [9], and a solution using source-trees was proposed in [1]. As we will see in Section III, it is easy for nodes to compute their sink-tree, i.e., the paths from all other nodes to itself. Thus, in Figure 1,  $s$  can use this information to source-route its distance vector or its source-tree towards its backward neighbor  $r$ . In this way,  $r$  (and hence also  $p$ ) learns there is a path to  $q$  via  $s$ . The problem with these solutions is that the message overhead between neighbors increases to  $O(N^2)$ . Consider the case of a ring of nodes,  $p_0, p_1, \dots, p_{N-1}$ , where each node  $p_i$  has a unidirectional channel to  $p_{(i+1) \bmod N}$ . Since each node  $p_i$  must send its distance vector or source-tree around the ring to reach node  $p_{(i-1) \bmod N}$ , then each channel will carry  $N$  messages, each of size  $O(N)$ . Another solution was proposed in [13], but it requires each node to transmit an  $O(N^2)$  sized matrix to each of its neighbors.

In this paper, we present a protocol which has  $O(N)$  message overhead even in the presence of unidirectional links. This low overhead comes at the price of not always being able to find the optimum path between nodes. However, the protocol is guaran-

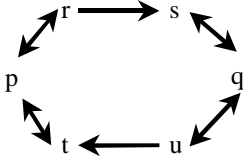


Fig. 1. Unidirectional network example.

teed to always find a path between each pair of nodes, even if all channels are unidirectional<sup>1</sup>. In addition, the cost of the chosen path is guaranteed not to exceed the cost of the optimal bidirectional path between the pair of nodes. Therefore, the cost of the chosen path is sensible, and in the absence of unidirectional links, it is optimal.

Due to length restrictions, the proofs have been omitted.

## II. NOTATION

Due to lack of space, we do not give a detailed description of our process notation. For more details on the notation, the reader is referred to [6], [7].

A network is a set of processes and a set of unidirectional channels between these processes. A network may also be viewed as a graph  $(V, E)$ , where  $V$  is the set of processes, and each edge  $(p, q)$  in  $E$  is a channel from  $p$  to  $q$ . Thus, we use the terms node and process interchangeably, and the terms edge and channel interchangeably. We assume  $N = |V|$ , and  $L$  is an upper bound on the number of hops of a simple path in the network.

A channel from process  $p$  to process  $q$  is denoted by  $ch(p, q)$ . If channel  $ch(p, q)$  exists, then we say process  $p$  is a *backward neighbor* of process  $q$  and process  $q$  is a *forward neighbor* of process  $p$ . We make the common assumption that for every pair of processes  $p$  and  $q$ , there is a path in the network from  $p$  to  $q$  and a path in the network from  $q$  to  $p$ . Without this assumption,  $p$  is not able to learn of the path to  $q$ .

Channels may lose messages, but if a message is sent repeatedly across  $ch(p, q)$ , then eventually  $q$  receives the message. A channel that is down is simply removed from the network.

A process learns of its neighboring processes through a hello protocol [12]. If there is a channel from  $p$  to  $q$  and a channel from  $q$  to  $p$ , then both  $p$  and  $q$  will learn of the bidirectional edge between them. However, if only edge  $(p, q)$  exists, then  $q$  learns that  $p$  is its backward neighbor, but  $p$  does *not* learn that  $q$  is its forward neighbor. We assume processes execute a typical hello protocol. To simplify our presentation, we do not incorporate the hello protocol into our process definitions. Instead, we simply assume each process has as an input the set of backward neighbors that were obtained from the hello protocol.

## III. SINK-TREES

Consider the network of Figure 2(a). In this figure, dark arrows indicate channels between processes. Notice that there is

<sup>1</sup>We make the necessary assumption that a path exists from the source to the destination, and a path also exists from the destination to the source. This assumption is also assumed in the literature [1], [4], [5], [9], [13].

a path from  $p$  to  $q$  and a path from  $q$  to  $p$ . Therefore,  $p$  must be able to learn the path from itself to  $q$ . However, unidirectional edges in the network complicates this task. Consider processes  $r$  and  $s$ . Process  $s$  determines that  $q$  is a forward neighbor, because it can exchange messages with  $q$ . However,  $s$  is not able to notify  $r$  of edge  $(s, q)$ , because there is no channel from  $s$  to  $r$ . In addition, consider processes  $p$  and  $t$ . Process  $t$  determines that there is a path  $(t, u, q)$  from itself to  $q$ , and it informs  $p$  of this path. However, this information is of no use for  $p$  to reach  $q$ , since there is no channel from  $p$  to  $t$ .

Although there is no simple way for  $p$  to learn the path from itself to  $q$ , note that there is a simple way for  $p$  to learn the path *from*  $q$  to  $p$ . This is because every node is aware of its backward neighbors. Process  $u$  is aware of edge  $(q, u)$ , and informs  $t$  of this edge. Process  $t$  is aware of edge  $(u, t)$ . Thus, it combines this edge with the received edge  $(q, u)$ , and forwards path  $(q, u, t)$  to  $p$ . Process  $p$  is aware of edge  $(t, p)$ , and thus, it learns the entire path  $(q, u, t, p)$ .

From the above reasoning, each process can learn its entire *sink-tree*, i.e., the path from all other nodes to itself. Each process receives the sink-trees of each backward neighbor, and combines the edges in these sink-trees to obtain its own sink-tree. For example, Figure 2(b) shows the sink-trees of  $r$  and  $t$ . The sink-tree of  $p$  is derived from these trees.

The advantage of receiving the sink-trees from each backward neighbor is that each process learns the path to reach its backward neighbor. For example, in Figure 2(b), process  $p$  learns from the sink-tree of  $t$  that in order to reach  $t$  its messages must follow the path  $(p, r, s, q, u, t)$ . We will take advantage of this in later sections. Thus, as a first step towards our final network, we present a network where each process computes its sink-tree.

Each process  $p$  in the network maintains a variable  $T_{sk}$ , where it stores its sink-tree.  $T_{sk}$  is a set of undirected edges whose union forms a tree. The direction of the edges is known since process  $p$  is assumed to be the root. Edge  $(q, r)$  on the tree is denoted  $T_{sk}(q, r)$ . Each edge  $(q, r)$  carries a cost, which is denoted  $T_{sk}(q, r).c$ . We denote by  $Cost(T_{sk}, q)$  the sum of the cost of the edges along the path in  $T_{sk}$  from  $q$  to  $p$ . If  $q$  is not in  $T_{sk}$ , then  $Cost(T_{sk}, q)$  is defined to be infinity.

Each process  $p$  has an input an array  $c_{in}$ , which is obtained from the hello protocol. This array contains, for each backward neighbor  $g$ , the cost of edge  $(g, p)$ .

The specification of a process  $p$  in the network is given in Figure 3.

Process  $p$  contains two actions. In the first action, it receives the sink-tree from a backward neighbor  $g$ . Then, it updates its own sink-tree based on the sink-tree of its neighbor. This is done through a greedy procedure *graft*, which inserts edges from the sink-tree of  $g$  into the sink-tree of  $p$  provided a shorter path is found to  $p$ . Procedure *graft* is defined in Figure 4.

In the second action, process  $p$  checks whether enough time has passed since the last broadcast of its *sink* message. If enough time has passed for its forward neighbors to process the previous *sink* message, process  $p$  broadcasts another *sink* message with the latest version of its sink-tree. A copy of this message is placed in each of the outgoing channels of the process.

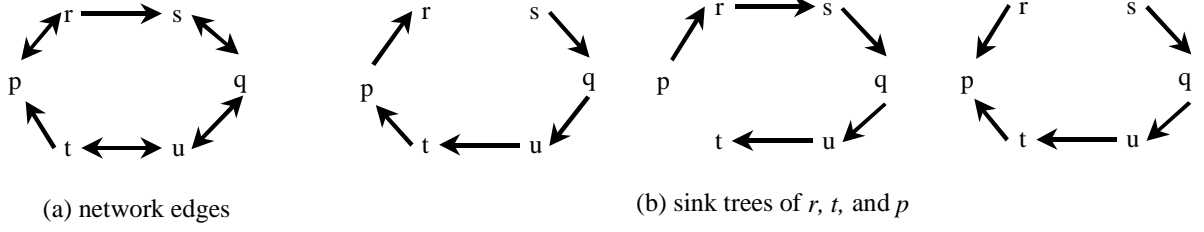


Fig. 2. Sink-Tree Example

```

process  $p$ 
inputs
   $B$  : set of pid's           {set of backward neighbors}
   $c_{in}$  : array[B] of integer {cost of incoming edges}
variables
   $T_{sk}$  : set of edges       {sink-tree}
parameters
   $g$  : any element of  $B$      {any backward neighbor}
begin
  rcv  $sink(T_g)$  from any  $g$  →  $graft(T_g, T_{sk}, c_{in}[g])$ 
  timeout  $sink$  → bcast  $sink(T_{sk})$ 
end

```

Fig. 3. Process  $p$  in a network computing sink-trees.

Processes acquire a correct value of their sink-trees in a short time, as stated below.

Given that some processes may execute faster than others, a typical measure of convergence time is the number of *rounds* before convergence. In each round, each process has had the opportunity to execute all its actions. That is, a round is an interval of time during which each action of each process is either executed, or the action is disabled at some point in the interval.

*Lemma 1:* Starting from an arbitrary state of the network, within  $O(L)$  rounds, the network reaches a steady-state where for each process  $p$ , the sink-tree  $T_{sk}$  of  $p$  contains the minimum cost path from every process to  $p$ . ■

#### IV. SOURCE-TREES

In our approach, we desire to exchange only  $O(N)$  information between neighboring process. Two common techniques in traditional routing accomplish this. The first is to exchange a distance vector between neighbors [2]. However, this suffers from the count-to-infinity problem and introduces long-lived loops. A technique which avoids these drawbacks while still ensuring  $O(N)$  information exchange between neighbors is the exchange of *source-trees*. That is, each process  $p$  keeps track of a tree  $T_{sr}$ , where  $p$  is the root of the tree, and a path from  $p$  to any node  $q$  in  $T_{sr}$  is also a network path from  $p$  to  $q$ . In this section, we enhance the processes of the previous section to enable them to compute a source-tree even though edges may be unidirectional.

Consider Figure 5(a), where we have the same network as in Figure 2, except that all edges are bidirectional. Assume for

```

 $graft(T_g, T_p, c)$ 
if  $(p, g) \in T_p$  then /* remove old information from  $g$  */
   $T_p := T_p - subtree(T_p, g)$ 
end if
 $V := T_g$ ;
while  $V \neq \emptyset$  /* iterate over all nodes in  $T_g$  */
  Let  $v$  be the process in  $V$  with minimum  $Cost(T_g, v)$ 
  if  $Cost(T_p, v) \leq c + Cost(T_g, v)$  then
     $V := V - subtree(T_g, v)$ ;
  else
     $T_p := T_p - subtree(T_p, v)$ ;
    if  $v = g$  then
       $w := p$ ;
       $d := c$ 
    else
       $w := parent(T_g, v)$ ;
       $d := T_g(v, w).c$ ;
    end if
     $T_p := T_p \cup (v, w)$ ;
     $T_p(v, w).c := d$ ;
     $V := V - \{v\}$ 
  end if
end while

```

Fig. 4.  $graft$  procedure

simplicity that the cost of each edge is one. Each process learns from the hello protocol which other processes are its neighbors, and thus it can complete the first level of its source-tree from this information. E.g., in Figure 5(b), process  $p$  learns that  $r$  and  $t$  are its forward neighbors, so it adds edges  $(p, r)$  and  $(p, t)$  to its source-tree. Each process forwards a copy of its source-tree to all its neighbors. This allows all processes to complete the second level of their source-tree. E.g., process  $p$  learns of the edge  $(r, s)$  from the source-tree it receives from  $r$ , and it learns of the edge  $(t, u)$  from the source-tree it receives from  $t$ . Thus,  $p$  adds these two edges to its source-tree. In this manner, within  $O(L)$  rounds, each process completes its source-tree.

Consider computing source-trees in the network of Figure 2(a). Consider processes  $p$  and  $t$ . Process  $t$  will add the path  $(t, u, q)$  to its source-tree, and forward its tree to  $p$ . Process  $p$  is unable to use this path to  $q$ , because there is no channel from  $p$  to  $t$ . Consider instead processes  $r$  and  $s$ . Process  $s$  is aware of edge  $(s, q)$ , but is unable to communicate this to  $r$ , because there is no channel from  $s$  to  $r$ . Thus,  $r$  does not become aware of the path  $(r, s, q)$ , and  $p$  does not become aware of path  $(p, r, s, q)$ . From this we conclude that  $p$  is unable to learn a path to  $q$ .

Although there is no edge from  $p$  to  $t$ , we saw in the previous section that process  $p$  will learn the shortest path to  $t$  from

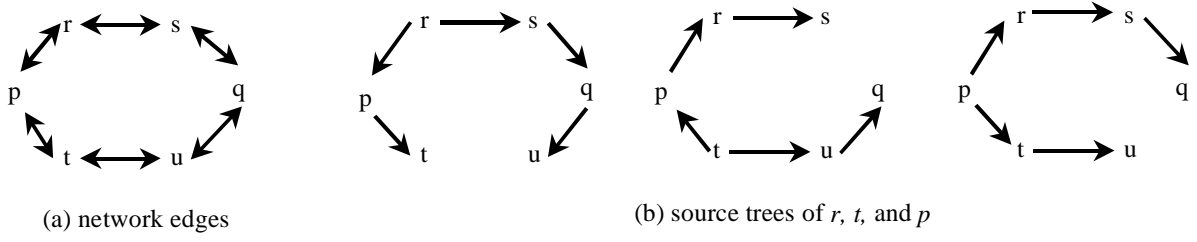


Fig. 5. Source-Tree example

the sink-tree it receives from  $t$ . We take advantage of this information, and we add a *virtual edge* from process  $p$  to process  $t$ . Virtual edge  $(p, t)$  represents the shortest path from  $p$  to  $t$ , namely  $(p, r, s, q, u, t)$ . Thus, the cost of  $(p, t)$  is the cost of  $(p, r, s, q, u, t)$ . Note that  $p$  is aware of this path to its backward neighbor  $t$ , as shown in the previous section.

Each process maintains a source-tree that is computed in the same manner as the traditional way. The difference, however, is that each edge in the source-tree will be a virtual edge from a node to its backward neighbor. E.g., the source-tree of process  $p$  will contain the path  $(p, t, u, q)$ . The cost of the virtual edges in the path are as follows. The cost of virtual edge  $(p, t)$  is the cost of the path  $(p, r, s, q, u, t)$ . The cost of virtual edge  $(t, u)$  is the cost of edge  $(t, u)$ , because the shortest path from  $t$  to  $u$  is the edge  $(t, u)$ . Finally, the cost of virtual edge  $(u, q)$  is the cost of edge  $(u, q)$ , also because the shortest path from  $u$  to  $q$  is  $(u, q)$ .

Although not all paths in the source-tree are network paths, the source-tree provides a first step in finding a path to each destination. In particular, it is easily shown that under the assumptions of Section II, when the network reaches a steady state, the source-tree of each process will contain all other processes in the network.

We next present a network of processes where each process computes its source-tree,  $T_{sr}$ . Similar to  $T_{sk}$ ,  $T_{sr}$  is a set of undirected edges whose union forms a tree. The direction of the edges is known since process  $p$  is assumed to be the root. In addition, each process maintains a variable  $c_{out}$ . Variable  $c_{out}$  is an array that maintains, for each backward neighbor, the cost of the path from  $p$  to the backward neighbor.

The specification of a process  $p$  in the network is given in Figure 6.

The first action of process  $p$  is similar to the first action of  $p$  in Section III, except that  $c_{out}[g]$  is assigned the cost of virtual edge  $(p, g)$ , i.e., the cost of the path from  $p$  to  $g$ . The second action remains the same as before.

The third and fourth actions are new. In the third action, the source-tree is updated using the source-tree received from  $g$ . Since  $T_{sr}$  is undirected with  $p$  as its root, we may also use procedure *graft* to update  $T_{sr}$  from  $T_g$ . The cost of the edge  $(p, g)$ , however, is obtained from  $c_{out}$ , since  $(p, g)$  is a virtual edge. The fourth action broadcasts a new *source* message with a copy of the source-tree of  $p$ .

Similar to sink-trees, processes quickly acquire a correct value of source-trees, as stated below.

```

process  $p$ 
inp
   $B$  : set of pid's           {set of backward neighbors}
   $c_{in}$  : array[B] of integer {cost of incoming edges}
var
   $T_{sk}$  : set of edges       {sink-tree}
   $T_{sr}$  : set of edges       {source-tree}
   $c_{out}$  : array[B] of integer {cost of virtual outgoing edges}
par
   $g$  : element of  $B$          {any backward neighbor}
begin
  rcv  $sknk(T_g)$  from any  $g$  →  $graft(T_g, T_{sk}, c_{in}[g]);$ 
   $c_{out}[g] := Cost(T_g, p)$ 
  [] timeout  $sknk$  → bcst  $sknk(T_{sk})$ 
  [] rcv  $source(T_g)$  from any  $g$  →  $graft(T_g, T_{sr}, c_{out}[g]);$ 
  [] timeout  $source$  → bcst  $source(T_{sr})$ 
end

```

Fig. 6. Process  $p$  in a network computing source-trees.

*Lemma 2:* Starting from an arbitrary state of the network, within  $O(L)$  rounds, the network reaches a steady-state where for each process  $p$ , the source-tree  $T_{sr}$  of  $p$  contains the minimum cost virtual path from  $p$  to every process. ■

## V. SHORTCUTS OVER SOURCE-TREES

Source-trees, as defined in the previous section, suffer from an inconsistency problem. Consider process  $p$  in Figure 7. The source-tree of  $p$  contains the path  $(p, t, s, q)$ . Since edge  $(p, t)$  is virtual, it represents the path  $(p, r, s, t)$ . Thus,  $p$  forwards messages destined to  $q$  to its forward neighbor  $r$ . However, the source-tree of  $r$  is equal to the path  $(r, p, t, s, q)$ . Hence,  $r$  forwards to  $p$  those messages that are destined to  $q$ , which is inconsistent.

The above conflict would be eliminated if  $r$  learns that there is a path to a node higher on its source-tree than  $p$ . In particular,  $r$  should learn that there is a path from  $r$  to  $s$ . In this case,  $r$  would forward to  $s$  those messages that are destined to  $q$ , and thus “shortcut” the lower part of its source-tree.

Notice that  $r$  does not learn about edge  $(r, s)$  from the sink-tree it receives from  $p$ . However,  $p$  is aware of the path  $(p, r, s, t)$  from the sink-tree it receives from  $t$ . Thus,  $p$  can inform  $r$  of this path, which resolves the conflict.

In general, the technique is as follows. Each process  $p$  maintains a shortcut-tree,  $T_{sc}$ . The root of the tree is  $p$ , and its edges

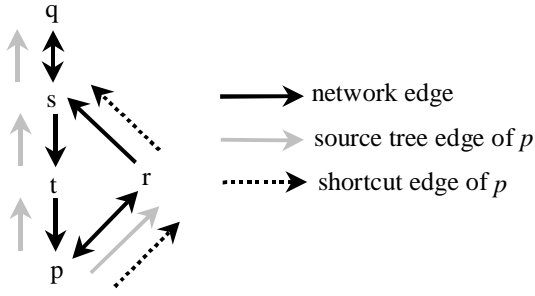


Fig. 7. Shortcuts over Source-Trees

originate from two sources. The first source are the paths from  $p$  to each of its backward neighbors. These paths are learned from the sink-trees received from the backward neighbors. The second source is the shortcut-trees which  $p$  receives from its backward neighbors. If the shortcut-tree of a backward neighbor has a path through  $p$ , then the path, starting at  $p$ , is added to the shortcut-tree of  $p$ .

Note that the paths propagated through the use of shortcut-trees originate from sink-trees, and sink-trees are optimal. This implies that the paths in a shortcut-tree will also be optimal (although probably not spanning the entire network).

In a sink-tree or a source-tree, it is easy to determine which backward neighbor provided the information about the edge. The backward neighbor is simply the next-to-last node along the path to the root on the tree. However, in a shortcut-tree, this is no longer the case. The paths on the tree give no indication about the backward neighbor that provided each edge. Given that edges in the shortcut-tree may no longer exist due to channel failure, these stale edges must be removed. To identify stale edges, we add two additional fields to each edge  $T_{sc}(q, r)$ . The first is  $T_{sc}(q, r).n$ , which contains the neighbor which propagated the edge. The second is  $T_{sc}(q, r).t$ , and it indicates the type of the source of the edge. A value of 'sk' indicates that the edge was obtained from the sink-tree of neighbor  $T_{sc}(q, r).n$ . A value of 'sc' indicates that the edge was obtained from the shortcut-tree of neighbor  $T_{sc}(q, r).n$ . If edge  $T_{sc}(q, r)$  is not refreshed by neighbor  $T_{sc}(q, r).n$ , it is removed from  $T_{sc}$ .

The specification of a process  $p$  is given in Figure 8.

The procedure *replace* used in process  $p$  is given in Figure 9. It replaces a portion of the tree  $T_{sc}$  with another tree. Notice that it does not use cost as a guide to replace nodes. This is simply because paths are derived from sink-trees, and thus are already (or will soon be) optimal.

In the first action, a sink-tree is received from neighbor  $g$ . The path from  $p$  to  $g$  along the sink-tree of  $g$  is used as the tree which updates  $T_{sc}$ . In the fifth action, a shortcut-tree is received from  $g$ . The subtree rooted at  $p$  in this tree is used as the tree which updates  $T_{sc}$ .

Given that sink-trees converge quickly to their optimal value, shortcut-trees also converge quickly, as stated below.

*Lemma 3:* Assume we have reached a network state where the sink-trees of all processes are stable and have reached their optimal value. Let  $p_0, p_1, \dots, p_k$  be a path in the sink-tree of

```

process  $p$ 
inp
   $B$  : set of pid's           {set of backward neighbors}
   $c_{in}$  : array[B] of integer {cost of incoming edges}
var
   $T_{sk}$  : set of edges       {sink-tree}
   $T_{sr}$  : set of edges       {source-tree}
   $T_{sc}$  : set of edges       {shortcut-tree}
   $c_{out}$  : array[B] of integer {cost of virtual outgoing edges}
par
   $g$  : element of  $B$          {any backward neighbor}
begin
  rcv  $sk(T_g)$  from any  $g$   $\rightarrow$ 
     $graft(T_g, T_{sk}, c_{in}[g]);$ 
     $c_{out}[g] := Cost(T_g, p);$ 
     $replace(T_{sc}, path(T_g, p), g, 'sk')$ 
  timeout  $sk$   $\rightarrow$  bcast  $sk(T_{sk})$ 
  rcv  $source(T_g)$  from any  $g$   $\rightarrow$   $graft(T_g, T_{sr}, c_{out}[g]);$ 
  timeout  $source$   $\rightarrow$  bcast  $source(T_{sr})$ 
  rcv  $shortcut(T_g)$  from any  $g$   $\rightarrow$ 
     $replace(T_{sc}, subtree(T_g, p), g, 'sc')$ 
  timeout  $shortcut$   $\rightarrow$  bcast  $shortcut(T_{sc})$ 
end

```

Fig. 8. Process  $p$  in a network computing shortcut-trees

$p_k$ , and let  $p_k$  be a backward neighbor of  $p_0$ . Then, for every  $i$ ,  $0 \leq i < k$ , the path  $p_i, \dots, p_k$  is included in the shortcut-tree of  $p_i$ . ■

Given that each process will obtain a sink-tree  $T_{sk}$ , a source-tree  $T_{sr}$ , and a shortcut-tree  $T_{sc}$ , the final step is to populate the routing table. The routing table  $rtb$  is an array with an element for each process in the network. In process  $p$ ,  $rtb[q]$  indicates the next hop neighbor to reach destination  $q$ . The routing table is completed as follows.

Let  $(p, q_1, q_2, \dots, q_k)$  be a path in the source-tree  $T_{sr}$  of  $p$ . Let  $i$  be the largest index,  $1 \leq i \leq k$ , such that the shortcut-tree  $T_{sc}$  has a path from  $p$  to  $q_i$ . Note that  $i \geq 1$ , since  $T_{sc}$  has a path to each backward neighbor of  $p$ . Then,  $rtb[q_k]$  is set to the first node along the path from  $p$  to  $q_i$  in  $T_{sc}$ .

## VI. ROUTE OPTIMALITY

We have seen in the previous sections that the network will reach a stable state within  $O(L)$  rounds. In this section, we focus on the optimality of the resulting routing path between each pair of processes. We begin by stating our main result.

*Theorem 1:* From an arbitrary state, the network will reach a steady state within  $O(L)$  rounds. In the steady state, the following holds for every pair of nodes  $p$  and  $q$  in the network:

- 1) The routing tables in the network define a network path from  $p$  to  $q$  (i.e., no routing conflicts).
- 2) Let  $OBP(p, q)$  (optimal bidirectional path) be the network path from  $p$  to  $q$  with minimum cost, where all edges in this path are bidirectional. Let  $R(p, q)$  be the path from  $p$  to  $q$  defined by the routing tables. Then,

$$Cost(R(p, q)) \leq Cost(OBP(p, q))$$

```

replace( $T_{sc}, T', n, t$ )
/* remove stale edges */
for all ( $v, w$ ) where
  ( $v, w$ )  $\in T_{sc} \wedge T_{sc}(v, w).n = n \wedge$ 
   $T_{sc}(v, w).t = t \wedge (v, w) \notin T'$ 
do
   $T_{sc} := T_{sc} - subtree(T_{sc}, v)$ 
end for
/* iterate over all nodes in  $T'$  */
 $V := T'$ ;
while  $V \neq \emptyset$ 
  Let  $v$  be the process in  $V$  with minimum  $Cost(T', v)$ 
  if  $parent(T_{sc}, v) \neq parent(T', v)$  then
     $T_{sc} := T_{sc} - subtree(T_{sc}, v)$ ;
     $w := parent(T', v)$ ;
     $T_{sc} := T_{sc} \cup (v, w)$ 
     $T_{sc}(v, w).c := T'(v, w).c$ ;
     $T_{sc}(v, w).n := n$ ;
     $T_{sc}(v, w).t := t$ ;
  end if
   $V := V - \{v\}$ 
end while

```

Fig. 9. *replace* procedure

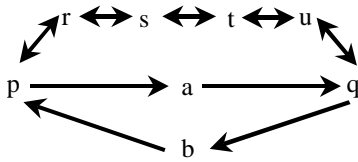


Fig. 10. Routing path of lesser cost than optimum bidirectional path.

From the theorem above, the routing tables define a path between any pair of nodes. Thus, the routing protocol is sound. Also, the routing path has a cost at most that of the optimum bidirectional path in the network. Therefore, if all edges are bidirectional, then the optimum path is obtained.

The  $O(N)$  complexity in the communication between neighboring nodes comes at a price. Consider a network that has unidirectional edges between processes  $p$  and  $q$ . As stated above, the network does find a path from  $p$  to  $q$ . However, the path may not be optimal. Although the routing path is not optimal, this does not imply that introducing unidirectional edges into an existing network reduces the optimality of existing routing paths. This is because the cost of the routing path is at most the cost of the optimum bidirectional path.

To illustrate the above point, consider Figure 10, where the cost of each edge is one. The minimum bidirectional path from  $p$  to  $q$  is  $(p, r, s, t, u, q)$ , and the minimum path from  $p$  to  $q$  is  $(p, a, q)$ . From the sink-tree of  $b$ ,  $p$  learns of the path  $(p, a, q, b)$ . Hence, the shortcut-tree of  $p$  includes this path. Furthermore, this shortcut-tree is propagated to  $a$ , and  $a$  learns of edge  $(a, q)$ . Hence, the routing path becomes the minimum path, even though it contains unidirectional edges.

To illustrate a case where the optimal path is not found, consider Figure 11. In this network,  $p$ ,  $a$ , and  $b$  learn very little from the sink-trees of their predecessors. Namely, they only learn the

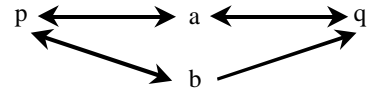


Fig. 11. Failure in finding the optimal path.

edge from themselves to their backward neighbor. Process  $q$ , on the other hand, learns of the path  $(q, a, p, b)$ , and propagates this path using its shortcut-tree. Notice that edge  $(b, q)$  is absent from the paths propagated by the shortcut-tree. Thus, the shortcut-tree of  $p$  will not contain edge  $(b, q)$ . For  $p$  to reach  $q$  via  $b$ , the source-tree of  $b$  would need to have edge  $(b, q)$ . However, since  $b$  does not receive messages directly from  $q$ , the source-tree of  $b$  will be the path  $(b, p, a, q)$ . Hence,  $p$  never learns about edge  $(b, q)$ , and never learns the optimal path to  $q$ .

Finally, we revisit the example of Figure 1 from Section I. If you recall, we showed that distance-vector and source-tree routing protocols are unable to find a path from  $p$  to  $q$ . However, the routing protocol we presented above is capable of finding the path from  $p$  to  $q$  as follows. Note that, from the sink-tree of  $u$ ,  $t$  learns of the path from  $t$  to  $u$ , namely,  $(t, p, r, s, q, u)$ . Process  $t$  propagates this path using its shortcut-tree, and in doing so,  $p$  learns of the path  $(p, r, s, q)$ , as desired.

## REFERENCES

- [1] Bao L, Garcia-Luna-Aceves J.J., "Link-State Routing in Networks with Unidirectional Links", *Proceedings of the ICCCN*, 1999.
- [2] Bertsekas D., Gallager R., *Data Networks*, Englewood Cliff NJ: Prentice-Hall, 1992.
- [3] Cheng C., Riley R., Kumar S.P.R., Garcia-Luna-Aceves J.J., "A Loop-Free Extended Bellman-Ford Routing Protocol without Bouncing Effect", *ACM Computer Comm. Review*, Vol. 19, No. 4., pp. 224-236, 1989.
- [4] Chen G., Lau F.C.M., Huang H., Li X., "Distance Vector Routing Protocols for Networks with Unidirectional Links", *Proceedings of the International Conference on Parallel Processing*, 2001.
- [5] Cobb J., Gouda M., "Stabilization of Routing in Directed Networks", *Fifth Workshop on Self-Stabilizing Systems (WSS)*, Springer-Verlag LNCS 2194, Datta A.K. and Herman T. Eds., October 2001.
- [6] Gouda M., "Protocol Verification Made Simple", *Computer Networks and ISDN Systems*, Vol. 25, 1993, pp. 969-980.
- [7] Gouda M., *The Elements of Network Protocols*, Wiley Publishers, 1998.
- [8] Johnson D. B., Maltz D. A., "Dynamic Source Routing in Ad-Hoc Wireless Networks", *Mobile Computing*, ed. T. Imielinski and H. Korth, Chapter 5, pp. 153-181, Kluwer 1986.
- [9] Lau F.C.M., Chen G., Du P., Xie L., "A Distance Vector Routing Protocol for Networks with Unidirectional Links", *Computer Communications*, Vol. 23 (2000) pp. 418-424.
- [10] Perkins C., Bhagwat P., "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers." *Proceedings of the ACM SIGCOMM Conference*, 24(4), Oct. 1994, pp. 234-244.
- [11] Pearlman M.R., Haas Z., "The Zone Routing Protocol (ZRP) for Ad-Hoc Networks", IETF MANET working group Internet draft, March 2001.
- [12] Pearlman M.R., Haas Z., Manvell B., "Using Multi-Hop Acknowledgments to Discover and Reliable Communicate over Unidirectional Links in Ad-Hoc Networks", *Proceedings of the Wireless Communication and Networking Conference*, 2000.
- [13] Prakash R., "A Routing Algorithm for Wireless Ad-Hoc Networks with Unidirectional Links", *Wireless Networks* Vol. 7, No. 6, 2001 pp. 617-625.
- [14] Ramasubramanian V., Chandra R., Mosse D., "Providing a Bidirectional Abstraction for Unidirectional AdHoc Networks", to appear in *Proceedings of the INFOCOM conference*, 2002.
- [15] Royer E., Perkins C., "Ad-Hoc On Demand Distance Vector Routing", *Proceedings of the second IEEE Workshop on Mobile Computing Systems and Applications*, 1999.
- [16] Sousa S. et. al., "Computer-Aided Modeling of Spread Spectrum Packet Radio Networks", *IEEE JSAC*, Vol 9., No. 1, January 1991.