Convergent Multi-Path Routing

Jorge A. Cobb
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083-0688
jcobb@utdallas.edu

Abstract

We present a protocol for maintaining multiple paths to each destination in a network of processes. For each destination, each process in the network maintains a set of neighbors which are used as next-hops to reach the destination. This set is known as the successor set. Collectively, the successor sets from all processes in the network with respect to a given destination form a spanning, directed, and acyclic graph, whose only sink is the given destination. The protocol we present has two interesting properties. First, the graph is maintained acyclic at all times, even though the successor set is dynamic. Second, the protocol tolerates all types of transient faults, even those which may not be detected. Therefore, if the protocol is started from an arbitrary initial state, it will converge to a normal operating state in which a spanning, directed, and acyclic graph is obtained and subsequently maintained.

1. Introduction

Consider a network of processes, consisting of a set of processes and communication channels between these processes. To maintain a route from every process to a given destination process, a routing spanning tree is maintained in the network, where the root of the tree is the given destination process. This routing tree is maintained by making every process (other than the root process) store the identity of its "parent" in this routing tree.

In general, there are two approaches to build the above routing tree. One approach is link-state routing, also known as broadcast routing. Here, each process broadcasts the status of its incident channels to all other processes in the network. Each process receives these broadcasts, and recreates in its memory the topology of the network. Then, each process builds in its memory a spanning tree of the stored topology, and chooses as its parent in the routing tree the same parent it chose in the spanning tree it constructed. Examples of link-state routing protocols include [15] and [23].

Although link-state routing quickly obtains a spanning tree, it incurs significant message and storage overhead. To remedy this, another approach is distributed routing, also known as distance vector routing. In this approach, each process forwards to each neighbor a copy of its metric to the given destination. Based on this information, each process chooses its parent in the routing tree and updates its metric accordingly. Examples of distributed routing protocols include [9][12][13]. Distributed routing protocols have the advantage of requiring less memory and message overhead. However, they suffer from long-lived loops and the counting to infinity problem [5], which can deteriorate performance.

To eliminate the disadvantage of long-lived loops in distributed routing protocols, while maintaining their low message and memory overhead, loop-less distributed routing protocols were developed [5][6][14][16] (note that link-state routing does introduce loops, albeit of shorter term). These protocols achieve a loop-less state by maintaining at all times a relationship between the metric of each process and the metric of all the descendants of this process in the routing tree. This relationship is maintained through diffusing computations. Thus, loop-less distributed routing protocols have a quick convergence to the desired routing tree while maintaining low memory and message overhead, and furthermore, have the desirable property of preventing routing loops at all times.

The loop-less routing protocols presented in [5][6][14][16] tolerate link failures and fail-safe node failures. However, they have not been shown to tolerate a broader class of failures, some of which are hard to detect, for example, improper initialization of a node, undetected corrupted messages, hardware/software bugs in lower layers that manifest themselves on rare occasions, and temporary disruptions from a network intruder. Since loop-less routing is based on diffusing computations, these faults could lead to deadlocks and race conditions from which the protocol may not recover.

To overcome this weakness, self-stabilizing protocols for loop-less routing were developed. A protocol is said to be self-stabilizing iff, starting from any arbitrary state (such as the state after an undetected fault), the protocol converges to a normal operating state within finite time. Self-stabilizing protocols are desirable due to their high

0-7695-0921-5/00 $10.00 © 2000 IEEE

degree of fault-tolerance [18]. They have the advantage of not requiring a global initialization and they tolerate all types of transient faults. The first loop-less and self-stabilizing distributed routing protocol was presented in [1], followed by the protocols presented in [10][17][3][2].

Recently [21][22][23][26], protocols that maintain multiple loop-free routes to each destination have been presented. In [23], it was shown that network performance might be increased considerably if multiple routes to the destination are maintained. Therefore, rather than maintaining a spanning tree rooted at the destination, these protocols maintain a spanning, directed, and acyclic graph, in which there is only a single sink node, namely, the desired destination. This graph is maintained by making every process, other than the destination process, maintain a set of neighbors, called the successor set. An edge (v, u) is contained in the graph if process u is contained in the successor set of process v.

In this paper, we present the first protocol that is multi-path, loop-free, and stabilizing. Therefore, the protocol maintains, at all times, a spanning, directed, and acyclic graph whose only sink node is the destination process. Also, the protocol tolerates all types of transient faults, even those which may not be detected. Thus, if the protocol is started from an arbitrary initial state, it will converge to a normal operating state in which a spanning, directed, and acyclic graph is obtained and subsequently maintained.

The objective of most routing protocols is to find a path to the destination which is optimal with respect to a given metric (e.g., path length [4], bandwidth [24], etc.). In this paper, we make no assumptions about the optimality of the paths chosen from each process to the destination. Each process is free to add or remove any neighbor to or from its successor set. Our only restriction is that a loop is not formed when a neighbor is added to the successor set of a process, and that each process remains connected to the graph, i.e., its successor set never becomes empty. However, if it is desired to find an optimal path to the destination, the protocol can easily be extended to do so. We discuss how to perform this extension in the concluding remarks.

The paper is organized as follows. In Section 2, we present the loop-avoidance technique used in our protocol. In Section 3, we discuss the problems encountered while trying to obtain a stabilizing loop-free protocol. Sections 4 and 5 present two components of the overall solution to the problem. In Section 6, we combine these two components into a single self-sufficient protocol. In Section 7, we discuss some implementation details of the protocol. A comparison of our protocol with existing protocols is given in Section 8.

Due to lack of space, the proofs have been deferred to the full paper.

2. Loop Avoidance

Consider a network of processes, which consists of a set of processes and a set of channels. Each channel allows the exchange of messages between two processes. We say that processes u and v are neighbors iff there are joined by a channel.

One process in the network is a distinguished process, called the *destination*. We consider the problem of maintaining multiple paths from each process to the destination process. These paths are represented by each process having a set of *successors*. The successor set of a process contains those neighbors that may be used as the next hop in the route to the destination process. Therefore, if we combine all edges of the form (v, u), where u is in the successor set of v, then these edges form an acyclic graph which spans all the processes in the network, and whose only sink node is the destination process. We refer to this graph as the *routing graph*.

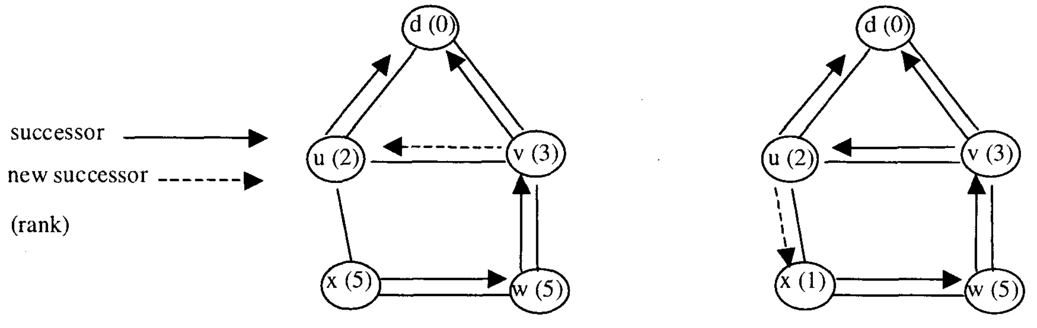
If process u is in the successor set of process v, we say that v is a *predecessor* of u, and u is a *successor* of v. If, starting from process w, process v can be reached by following a path in the routing graph, then we say that w is a *descendant* of v and v is an *ancestor* of w.

Each process has the freedom to arbitrarily choose which neighbors to add to its successor set, and which neighbors to remove from its successor set. However, to maintain a consistent routing graph, we impose two restrictions. First, the successor set of a process cannot become empty, since this would disconnect the node from the graph. Second, a neighbor is not allowed to be placed in the successor set of a process if by doing so a loop is formed. Thus, the requirement that the routing graph constitute an acyclic graph must be satisfied at all times, and not only at a steady sate of the network.

The routing graph is maintained free of loops as follows. Each process v maintains a non-negative integer variable, rk.v, known as the *rank* of process v. This rank will be used to maintain an order on the processes. When a process adds a neighbor to its successor set, it must preserve this order, and in doing so it avoids loops. We explain this in detail below.

We say that the *ordered rank property* holds at process v, iff, for each process w, where w is a descendant of v, $rk.w \geq rk.v$.

If the ordered rank property holds at v, then a simple technique to avoid the formation of loops is as follows. Process v adds neighbor u to its successor set only if $rk.v > rk.u$. That is, if $rk.v > rk.u$, then u cannot be a descendant of v, because all descendants of v have a rank at least rk.v. Thus, v may add u to its successor set without creating a loop. This is illustrated in Figure 1(a), where d is the destination process.



(a) v may add u as a successor, since rk.u < rk.v

(b) x decreases rk.x below rk.w, allowing u to choose x as its successor, forming a loop

Figure 1

The above restriction on choosing successors is sufficient to prevent loops, whenever the ordered rank property holds at a node. However, it is necessary to allow each process the freedom to change its rank value, which in turn may temporarily violate the ordered rank property. For example, if v wants to choose u as a new successor, and rk.v ≤ rk.u, then v must increase rk.v to a value greater than rk.u before making u its successor. However, this increase may violate the ordered rank property at v. Therefore, we must carefully consider the effects of an increase or decrease of the rank of a process, which we do next.

First, assume that process v wants to decrease its rank. If the ordered rank property holds at v, then decreasing rk.v preserves this property at v. However, for an ancestor u of v, this decrease in the rank of v may violate the ordered rank property at u (i.e., rk.v < rk.u could occur). Note that this occurrence is unknown to u. Thus, u may decide to choose v as a successor, and a loop is formed. This scenario is illustrated in Figure 1(b).

To prevent the above scenario, although we do allow each process v to decrease its rank, its new value should be at least the maximum of the ranks of its current successors. Therefore, the scenario depicted in Figure 1(b) is avoided.

Consider now when process v increases its rank. For any ancestor u of v, if the ordered rank property holds at u, then it continues to hold as v increases its rank. However, the ordered rank property may no longer hold at v, since its rank may be greater than that of one of its descendants. Therefore, v must initiate a diffusing computation along its descendants, forcing them to increase their rank to at least the rank of v. During this diffusing computation, v may not choose a new successor, since the ordered rank property does not hold at v. When this diffusing computation terminates, then the ordered rank property is restored at v, and v may choose new successors. This is illustrated in Figure 2.

In presenting the above restrictions on the methods in which a successor is chosen and the rank is updated, the least possible restrictions were presented, in order to ensure the generality of the protocol. In particular, no semantic meaning was assigned to a rank. We simply presented some general restrictions for the sole purpose of avoiding the formation of loops. Thus, a rank may be implemented in various ways.

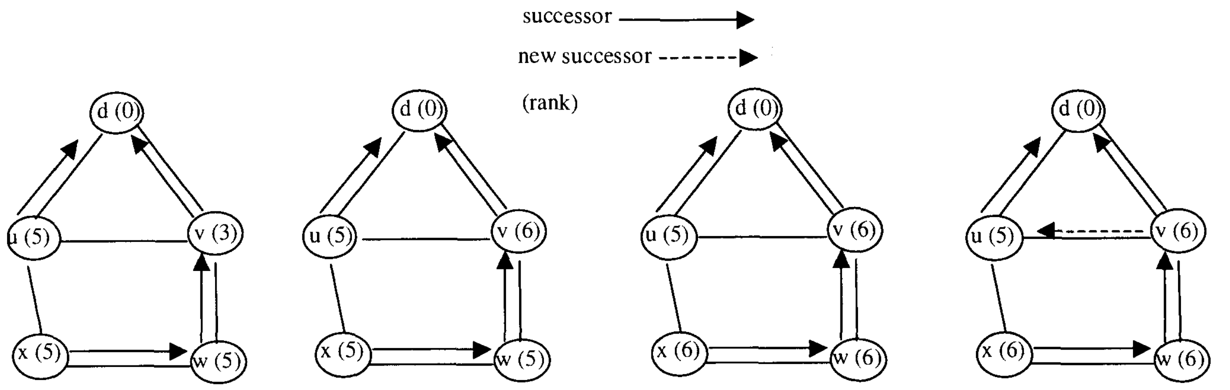
One implementation choice is to allow a higher level protocol to make a recommendation on which neighbors should be added to and removed from the successor set, and let our protocol follow these indications. Our protocol could ensure through the use of ranks that loops are always avoided.

Another implementation choice is to set the rank to a routing metric. For example, assume the rank is the cost to the destination. The cost of a process would be set to the maximum, over all successors, of the cost of the successor plus the cost of the link between the process and the successor. Notice that this satisfies the requirement above that a process must set its rank (i.e., cost) to at least the maximum rank (costs) of all its successors.[1] To minimize its cost, a process would remove from its successor set neighbors with higher costs and add to it neighbors with lesser costs. Other routing metrics such as bottleneck bandwidth, packet loss probability, etc., may also be used to implement a rank.

3. Fault Recovery

In this section, we discuss how the loop avoidance mechanism is affected by faults, and how it may recover from them.

[1] Assuming no negative edge costs.



successor
new successor
(rank)
d (0)
u (5)
v (3)
x (5)
w (5)
d (0)
u (5)
v (6)
x (5)
w (5)
d (0)
u (5)
v (6)
x (6)
w (6)
d (0)
u (5)
v (6)
x (6)
w (6)
(a) Initial state
b) v increases its rank to add u to its successor set
c) v forces all its descendants to increase their ranks to at least v's rank
d) v now may add u to its successor set

Figure 2

If faults occur in the network, they may alter the variables of a process in such a way that a loop is formed, even though loops do not occur during normal execution. If a loop exists due to a fault, it may cause a diffusing computation along the loop, which may never terminate. Since a process cannot change successors until the diffusing computation of its rank update has finished, the processes involved in the loop may remain in it forever.

Therefore, a technique to accurately detect the presence of loops is required. If loops exist, the processes involved in the loop must be able to detect this condition within finite time. This allows the processes to change successors in an attempt to break the loop. Furthermore, the technique should not incorrectly indicate that a loop exists if none is present. This would cause the processes to change successors unnecessarily.

A simple technique to detect if a loop exists is for each process to keep an estimate of the length of the path to the destination along the routing tree. We refer to this as the hop count of the process, which is periodically updated to the maximum of the hop counts of the successors of the process plus one. We assume that all simple paths in the network have a length less than D, for some constant D known to all processes. Thus, a process only adds a neighbor to its successor set if the neighbor's hop count is less than D - 1. Furthermore, if a process detects that one of its successors has a hop count of at least D - 1, the process removes this successor from its successor set in an attempt to break the loop.

This simple technique, however, has its drawbacks. Since successors are based on rank and not on hop count, it is possible to obtain a race condition in which no loop is ever created, yet the hop count of the processes increases without bound (an example of this race condition may be found in [25]). Thus, a process may incorrectly believe it is involved in a loop. Although there are techniques for loop detection that do not rely on hop count [3][25], these are not amenable for the case of multiple successors.

To solve the loop detection problem, we must prevent the hop count from reaching a value of D during normal operation. We accomplish this by introducing a sequence number which originates at the destination process and propagates downward throughout the entire routing graph. That is, when all the successors of a process v have the same sequence number, and this sequence number is different from v's sequence number, then v adopts the sequence number of its successors.

Once the sequence number has reached all processes in the routing graph, then the destination process can change its sequence number. Since a new sequence number is not introduced until the previous one has finished propagating, a sequence number in the range 0 .. 1 suffices.

When a process changes its sequence number, the process may increase or decrease its rank. *However, while the process maintains the same sequence number, it does not increase its rank*, it may only decrease it. If the process wishes to increase its rank, it must do so the next time its sequence number changes.

Therefore, the approach can be considered as a periodic global diffusing computation that begins at the destination process, and expands throughout the entire routing graph.

As an example, consider Figure 3(a). Here, all processes have the same sequence number, namely one, and the rank of each process is at least the rank of its successors. Since all processes have the same sequence number, the destination process changes its sequence number to zero, and this sequence number propagates down the routing graph, as shown in Figure 3(b). As it propagates, each process updates its rank, which may now

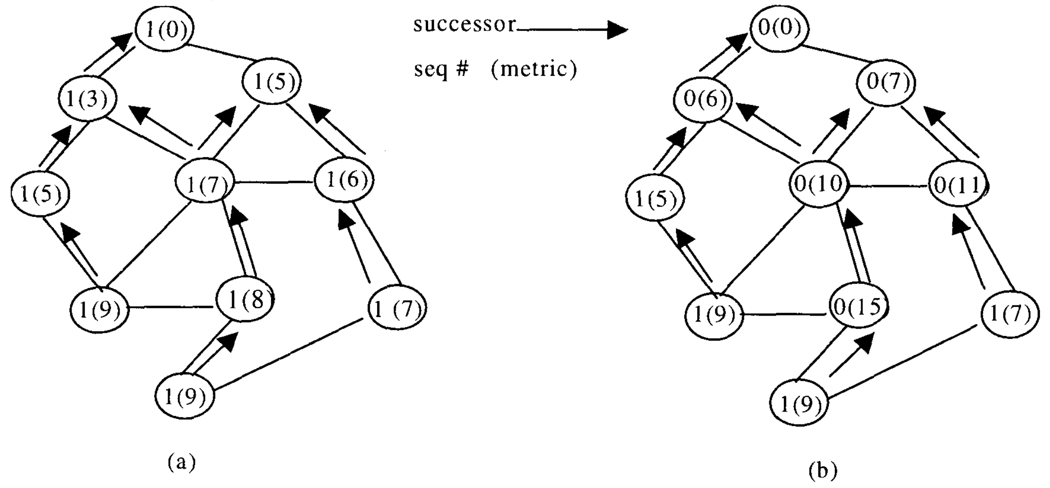


Figure 3

be greater than before. Notice, however, that the ranks of any path from a process to the destination are decreasing, provided there is no change in the sequence number along the path. *Thus, the ordered rank property holds for processes sharing the same sequence number.* This allows a process to choose as its successor any neighboring process with the same sequence number and smaller rank, without forming a loop.

It is easy to show that the hop count of each process under the above scheme will never reach the value D under a fault-free execution. Therefore, a process will not falsely detect the presence of a loop. However, in an execution with faults, a loop may occur, causing the hop count of a process to reach D. Thus, to break loops, a process should remove from its successor set any neighbor whose hop count is at least D - 1.

We assume each process has a time-delayed knowledge of the sequence number of the destination. That is, each process eventually learns the correct sequence number of the destination, even though its successors have a different sequence number than the destination. Processes use this knowledge in the selection of successors: a successor is added to the successor set only if it has the same sequence number as the destination. This is necessary for the protocol to converge from an arbitrary state to a normal operating state.

Below, we present two protocols. The first protocol is the ordered rank protocol, and is based on the above assumption of knowledge of the destination's sequence number. The second is a core tree protocol, which is used to implement the above assumption. Finally, we combine both protocols into a single self-sufficient protocol.

4. Ordered Rank Protocol

For clarity and simplicity, the processes in the ordered rank protocol are specified using a shared memory notation. A message-passing version of the protocol will be presented in detail in the full version of the paper. In this notation, each process is specified by a set of constants, inputs, a set of variables, an optional parameter, and a set of actions as follows (similar notations for defining network protocols are discussed in [7] and [8]).

```
process <process name>
const   <constant name>    :   <type>
inp     <input name>       :   <type>
var     <variable name>    :   <type>
par     <parameter name>   :   <type>
begin
    <action>
[]
    <action>
end
```

The inputs declared in a process may only by read by the process, and the values of inputs may change over time. The variables declared in a process can be read and written by the process. Also, a process may read, but not write, the variables of its neighboring processes.

Every action in a process is of the form <guard> → <statement>. The <guard> is a Boolean expression over the variables, inputs, and the parameter declared in the process and the variables declared in the neighbors of that process. The <statement> is a sequence of **skip**, assignment, and conditional (**if fi**) statements that update only the variables declared in that process. We assume that the execution of actions is fair, that is, an action whose guard continuously evaluates to true will be eventually executed.

The parameter declared in a process is used to write a set of actions as one action, provided these actions differ only by the parameter value. For example, let j be a parameter whose type is the range 0 . . 1. The action

```
x < j →   y[j] := true
```

is a shorthand notation for the following two actions.

```
    x < 0 →   y[0] := true
[]
    x < 1 →   y[1] := true
```

When referring to the variables of a process, we use the process name as a suffix. Thus, x.v is variable x in process v. When it is clear from the context to which process we are referring, we omit the suffix. In particular, in the code of each process v, variables without suffix correspond to variables of v.

Each non-destination process v in the ordered rank protocol has three inputs: D, G, and ds. Input D is an upper bound on the length of any simple path in the network. We assume all processes are given the same value for D. Input G is the set of neighboring processes of v. Input ds is the sequence number of the destination process. Although v cannot read directly the sequence number of the destination if the destination is not a neighbor, in the next section we show how v can obtain a time-delayed estimate of this value.

Process v has four variables, S, rk, sn, and hc, which are, respectively, the set of successors of v, the rank of v, the sequence number of process v, and an estimated hop count of the longest path to the destination via the successor set.

Finally, process v has a parameter g, which is instantiated with the identity of any neighbor of v.

The specification of a non-destination process v is as follows.

```
process v
const  D  :  integer             {network diameter}
inp    G  :  set {u | u is a neighbor of v},
       ds :  0 . . 1             {destination seq. #}
var    S  :  subset of N,        {set of successors}
       rk :  integer,            {rank}
       sn :  0 . . 1,            {sequence #}
       hc :  0 . . D             {hop count to dest.}
par    g  :  element of N        {any neighbor}
begin
  $sn \neq ds \wedge (\forall u, u \in S, sn.u = ds) \wedge S \neq \varnothing$  →
            sn := ds;
            hc := min(D, max{hc.u | u ∈ S} + 1);
            rk := atleast(max{rk.u | u ∈ S})
[]
  $(\exists u, u \in S, sn.u = sn)$  →
        hc := min(D, max{hc.u | u ∈ S ∧ sn.u = sn} + 1);
        rk := min(rk, atleast(max{rk.u| u∈S ∧ sn.u=sn}))
[]
  g ∈ S  →  if hc.g ≥ D-1 ∨ |S| > 1 →       S := S - {g}
            [] hc.g < D-1           →       skip
            fi
[]
  sn.g = ds ∧ ds = sn ∧ hc.g < D-1 ∧ rk.g < rk
        → S := S ∪ {g}
[]
  S = ∅ →  if sn.g = ds ∧ sn ≠ ds  →
                      S := S ∪ {g};
                      sn := ds;
                      rk := atleast(rk.g);
                      hc := min(D, hc.g + 1)
           [] sn.g ≠ ds ∨ sn = ds →   hc := D
           fi
end
```

Process v has five actions. In the first action, v checks if it should change its sequence number. Process v should change its sequence number if it is different from the destination's sequence number. Before doing so, process v must wait for all of its successors to have the same sequence number as the destination. This is required to avoid the following scenario. Assume ds = 0, sn.v = 1, and assume v sets sn.v to 0. Assume v still has a successor x with sn.x = 1, and x changes sn.x to 0. Since x changed its sequence number, it is allowed to increase its rank to any value. If x increases its rank to a value larger than v's rank, then the ordered rank property is violated for sequence number 0.

If the sequence number is changed, the hop count and rank are updated. The hop count is one greater than the maximum of the hop counts of all successors. The rank is set to any value at least as large as the maximum rank over all successors, as required in Section 2.

In the second action, process v updates its rank and hop count from its successors, but without changing its sequence number. However, since v is not changing its sequence number, it must update its rank relative only to the ranks of its successors with the same sequence number as v. Notice that the rank of v is not allowed to increase in this action. This is necessary to maintain the ordered rank property and avoid the formation of loops, as described in Section 3.

In the third action, process v removes a neighbor from its successor set. If the neighbor's hop count is at least D - 1, it implies that the neighbor may be involved in a loop, and must be removed. To be flexible, we leave it as an option whether the neighbor should be removed or not when the neighbor's hop count is less than D - 1. Hence, if hc.g < D - 1, either branch of the **if** statement may be taken, as desired. However, the successor set should not be allowed to be empty, since this would leave v without a path to the destination.

In the fourth action, a neighbor is added to the successor set. This neighbor is added only if it has the same sequence number as v and as the destination, if it will not cause v's hop count to become D (i.e., $hc.g < D-1$), and if no loop will be formed (i.e., $rk.g < rk.v$).

Note that in the third action we remove successors whose hop count is at least D - 1. Therefore, if all the successors of v have a hop count of at least D - 1, it is possible for v's successor set to become empty. If this is the case, the last action has two choices. If a neighbor is found whose sequence number is the same as the destination's, and v's sequence number is different, then this neighbor is chosen as a successor, and the sequence number, rank, and hop count are updated. A process with the same sequence number as v is not chosen since it may cause a new loop to be formed. If no such neighbor is found, the hop count is set to D to indicate all predecessors that the destination is not reachable via this process.

We next present the specification of the destination process d.

```
process d
const
    h  :  0,
    rk :  0
var
    sn :  0 .. 1
begin
  (∀ v, v ≠ d,  ds.v = sn ∧ sn.v = sn)  →  sn :=(sn+1) mod 2
end
```

The hop count and rank of the destination are constants whose values are always zero. The destination sequence number is variable sn. When the sequence number has propagated throughout the entire network, the destination changes its sequence number. Although the destination cannot read the variables of non-neighboring processes, we will see in the next section how the destination can implement the guard of its action.

The above protocol can be proven correct, provided the following assumptions hold.

a) sn.d remains constant until, for all v, where $v \neq d$, $sn.d = ds.v = sn.v$.
b) if $ds.v \neq sn.d$, eventually $ds.v = sn.d$.
c) if $ds.v = sn.d$, then ds.v remains constant until sn.d changes.
d) if for all v, where $v \neq d$, $sn.d = ds.v = sn.v$, then eventually sn.v changes.

In the next section, we show one method to implement the above assumptions.

5. Core Tree Protocol

In this section, we present an abstract protocol to allow each non-destination process v a time-delayed estimate of the destination sequence number, and to allow the destination to know that all processes have received and adopted the destination's sequence number. The protocol is abstract in the sense that it deals only with sequence numbers, and not with ranks and successor sets. However, in the next section we combine both protocols to obtain a complete version of the multi-path protocol.

To allow each process v to learn about the destination's sequence number, we build a spanning tree in the network, whose root is the destination d. We propagate the destination's sequence number along this tree from the root (i.e. destination) towards the leaves. We refer to this tree as the core tree.

The core tree should be stable, and should not fluctuate with changes in the ranks of processes or other network conditions. Therefore, we choose to build the core tree as a minimum hop tree, i.e., the path from any process v to the root d along the tree is a minimum hop path from v to d.

To build this tree, each non-destination process v needs to maintain a parent variable p, with its parent on the tree, and a hop count i to the destination[2]. We implement a simple greedy approach based on the Bellman-Ford technique to find the minimum hop path to the destination.

Process v has a variable ds, which contains its estimate of the sequence number of the destination. Also, in variable sn it keeps its own sequence number. Process v adopts the root's sequence number when it assigns ds to sn.

Process v also needs an additional bit, called end, which indicates if all processes below v on the core tree have received the new sequence number from the destination and adopted the sequence number. Thus, once the end bit is true at the destination process, the destination may change its sequence number.

The specification of a non-destination process v follows.

```
process v
const   D     :  integer
inp     G     :  set {u | u is a neighbor of v}
var     p     :  N,          {parent}
        i     :  0 .. D,     {hop count to dest.}
        ds    :  0 .. 1,     {dest. seq. number}
        sn    :  0 .. 1,     {local sequence #}
        end   :  Boolean {end of seq. # propagation}
par     g     :  element of G
begin
  i ≠ min(D, i.p + 1)  →      i := min(D, i.p + 1)
[]
  i.g + 1 < i         →    p := g;  i := min(D, i.g + 1)
```

[2] Note that his hop count is different from the hop count in the ordered rank protocol. The former is the hop count along the core tree, and the latter the hop count along any path to the destination via the routing graph.

```
[]
  ds ≠ ds.p         →       ds := ds.p;  end := **false**
[]
  sn ≠ ds           →       sn := ds
[]
  end ≠ (∀ w, p.w = v,  sn.w = ds.w ∧ ds.w = ds ∧ end.w)
                    →       end := ¬end
end
```

In the first action, the hop count is updated from the hop count of the parent of v. In the second action, if a neighbor provides a shorter hop count to the destination, this process is chosen as the new parent. In the third action, if the parent's destination sequence number ds.p is different from v's, v updates ds.v to ds.p. It also sets end to false since this new sequence number needs to be propagated down the tree.

In the fourth action, process v adopts ds as its own sequence number sn. This represents the behavior of the ordered rank protocol, which will eventually choose ds as v's local sequence number sn. Finally, in the last action, end is updated to reflect if the sequence number has propagated down the tree. This will be true if all children of v on the tree have learned and adopted the new sequence number, and their end bits are equal to true.

The specification of the destination process follows.

```
process d
const   i       :   0
var     ds      :   0 . . 1,
        end     :   Boolean
begin
  end ≠ (∀ w, p.w = d,  sn.w = ds.w ∧ ds.w = ds ∧ end.w)
        →  end := ¬end
[]
  end   →  ds := (ds + 1) mod 2;  end := false
end
```

In the first action, the root updates its end bit. In the second action, if end is true, it changes its sequence number and sets end to false, waiting for the sequence number to propagate down the tree.

6. Complete Protocol

Now that we have both the ordered rank protocol and the core tree protocol, we can combine both protocols into a single self-sufficient protocol. The protocol consists of merging the actions of both protocols, with the exception that the fourth action of process v in the core tree protocol is removed, since it is an abstraction of how the sequence number is changed by the decreasing metric protocol, and an additional action is added to the destination process.

The complete protocol is as follows.

```
process v
const   D   :   integer             {network diameter}
inp     G   :   set {u | u is a neighbor of v},
var     S   :   subset of N,        {set of successors}
        sn  :   0 . . 1,            {sequence #}
        rk  :   integer,            {metric}
        hc  :   0 . . D             {distance to root}
        ds  :   0 . . 1             {root sequence #}
        p   :   N,                  {parent}
        i   :   0 . . D,            {hop count to root}
        end :   Boolean {end of seq. # propagation}
par     g   :   element of N        {any neighbor}
begin
  sn ≠ ds ∧ (∀ u, u ∈ S, sn.u = ds) ∧ S ≠ ∅  →
            sn := ds;
            hc := min(D, max{hc.u | u ∈ S} + 1);
            rk := atleast(max{rk.u | u ∈ S})
[]
  (∃ u, u ∈ S, sn.u = sn)    →
          hc := min(D, max{hc.u | u ∈ S ∧ sn.u = sn} + 1);
          rk := min(rk, atleast(max{rk.u| u∈S ∧ sn.u=sn}))
[]
  g ∈ S  →  if hc.g ≥ D-1 ∨ |S| > 1  →  S := S - {g}
            [] hc.g < D-1            →       skip
            fi
[]
  sn.g = ds ∧ ds = sn ∧ hc.g < D-1 ∧ rk.g < rk
        →  S := S ∪ {g}
[]
  S = ∅  →  if sn.g = ds ∧ sn ≠ ds  →
                      S := S ∪ {g};
                      sn := ds;
                      rk := atleast(rk.g);
                      hc := min(D, hc.g + 1)
            [] sn.g ≠ ds ∨ sn = ds   →  hc := D
            fi
[]
  i ≠ min(D, i.p + 1)   →    i := min(D, i.p + 1)
[]
  i.g + 1 < i       →       p := g;  i := min(D, i.g + 1)
[]
  ds ≠ ds.p         →       ds := ds.p;  end := false
[]
  end ≠ (∀ w, p.w = v,  sn.w = ds.w ∧ ds.w = ds ∧ end.w)
                    →       end := ¬end
end

process r
const   i       :   0
        hc      :   0,
        rk      :   0
var     sn, ds  :   0 . . 1,
        end     :   Boolean
begin
  ds ≠ sn   →  ds := sn
```

```
[]
  end ≠ (∀ w, p.w = r,   sn.w = ds.w ∧ ds.w = ds ∧ end.w)
          → end := ¬end
[]
  end     →  ds := (ds + 1) mod 2;   sn := ds; end := false
end
```

7. Protocol Implementation

As mentioned earlier, ranks may be implemented in many possible ways. One of these ways is to set the rank to a routing metric, such as the sum of the cost of the edges to the destination. If routing metrics are used as ranks, then finding an optimum path to the destination would be desired. To obtain this optimum path, the routing metric must satisfy the properties of monotonicity and boundedness, as described in [11].

To ensure that each process has in its successor set the neighbor along the optimum path to the destination, we only need to restrict the behavior of the protocol a little. When process v is about to change its sequence number, v keeps in its successor set the neighbor g which, if used as next hop, would give v the best metric, and v removes all others neighbors from its successor set. Process v then updates its sequence number and metric from the values in g. Finally, after updating its metric, v adds to its successor set those neighbors which have a better metric than v's metric (even though they don't offer the best path, since the weight of their channel with v may be very high). It is straightforward to show that with this modification, the optimum path to the destination is always found, while at the same time maintaining multiple successors to the destination.

Finally, the protocol can be implemented using message passing in a similar way as the usual distance vector routing protocol. Periodically, each process collects the values of its variables and forwards them in a message to all its neighboring processes. In particular, the process must include the following information in the message it forwards to all its neighbors. For each destination, the process includes: a) its rank, b) the sequence number sn, c) the sequence number ds of the destination, d) the end bit, e) the hop count h, and f) the hop count i. Thus, the overhead would be four additional bits and two small integers per destination.

8. Related Work

In our protocol above, it is possible for all processes to choose having a single successor at all times. In this case, the routing graph would simply be a routing tree. Furthermore, if the rank of a process is implemented via a routing metric, then this protocol can be used as a loop-less routing protocol. Below, we compare other loop-less routing protocols with our protocol under the restriction that only a single successor is chosen at all times.

In our protocol, we use sequence numbers. The use of sequence numbers to propagate information in a network of processes and to aid in their self-stabilization was introduced in [20]. The use of sequence numbers in a loop-less routing protocol was first presented in [1]. Here, although the routing tables were loop-less at all times, the protocol did not necessarily converge to the best route to the destination. This is because the protocol was designed for networks with unstable links, and the protocol follows any route as soon as it is found, even though it may not necessarily be the best route.

In [10][17], another self-stabilizing loop-less routing protocol was presented which also propagates sequence numbers. In this protocol, sequence numbers propagate between *any* pair of nodes. Therefore, in [10][17], the protocol takes two rounds of sequence number propagation to find the best route. The first round is used to enforce an ordering on the metrics similar to the ordered rank property. During this first round, processes may not change parents in the routing tree. During the second round, processes are allowed to change parents and converge to the best route to the destination.

In our approach, sequence numbers propagate only from parent to child along the routing tree. Also, a node may begin to change parents immediately after changing its sequence number, without having to wait for the sequence number to propagate throughout the entire tree.

It is difficult to estimate which of the two protocols above will reach the best route sooner. On one hand, the protocols in [10][17] allow for a faster propagation of the sequence number, since it propagates between any pair of nodes, and on the other hand, our protocol allows a node to immediately begin to change parents in search for the best path, rather than having to wait for the next round of sequence number propagation.

In [2], we presented an additional loop-less and stabilizing routing protocol (with a single successor to the destination). This protocol uses a combination of the diffusing computations presented in [3] and the sequence number propagation presented in [10][17]. The protocol presented in [2] has the advantage that no temporary loops are introduced when links fail. Unfortunately, the complete protocol of Section 6 may introduce a temporary loop in the event of a link failure. However, the likelihood of a loop forming can be reduced significantly if we increase the range of the sequence number. Furthermore, the protocol of Section 6 has the advantage of allowing multiple successors to the destination.

In the loop-less protocols presented in [3][5], and the self-stabilizing loop-less protocol presented in [6], each process on its own may begin a diffusing computation whenever it detects a change in the routing metric, as opposed to having to wait for the next sequence number to be propagated from the destination. This approach has the potential advantage of a faster response time to changes in the routing metric. Unfortunately, we were unable to obtain a self-stabilizing protocol with similar diffusing computations and that maintains multiple successors to the destination. Thus, we chose to use the propagation of sequence numbers instead. Nonetheless, although a faster response time is lost, more stability may be gained. The routing metric is only updated after the destination issues a new sequence number, which might help in alleviating the route hoarding problem.

References

[1] A. Arora, M. G. Gouda, and T. Herman, "Composite Routing Protocols", Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990.

[2] Cobb, J.A., Gouda M. G., "Stabilization of General Loop-Free Routing", submitted for journal publication.

[3] Cobb, J. A. and M. Waris, "Propagated Timestamps: A Scheme for the Stabilization of Maximum-Flow Routing Protocols", Proceedings of the Third Workshop on Self-Stabilizing Systems, pp. 185-200, 1997.

[4] Dijkstra, E. W., "A Note on Two Problems on Connection with Graphs", Numerical Mathematics, Vol. 1, pp. 269-271, 1959.

[5] Garcia-Luna-Aceves, J. J., "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking, Vol. 1, No. 1, Feb. 1993.

[6] Garcia-Luna-Aceves, J. J., Murthy S., "A Path-Finding Algorithm for Loop-Free Routing", IEEE/ACM Transactions on Networking, Vol. 5, No. 1, Feb. 1997.

[7] M. Gouda, "Protocol Verification Made Simple", Computer Networks and ISDN Systems, Vol. 25, 1993, pp. 969-980.

[8] M. Gouda, The Elements of Network Protocols, Wyley publishers, 1997.

[9] M.Gouda and M. Schneider, "Stabilization of Maximum Flow Trees", Proceedings of the third Annual Joint Conference on Information Sciences, 1994, pp. 178-181.

[10] Gouda, M. G. and M. Schneider, "Maximum Flow Routing", Proceedings of the Second Workshop on Self-Stabilizing Systems, Technical Report, Department of Computer Science, University of Nevada, Las Vegas, May 1995.

[11] Gouda, M. G. and M Schneider, "Maximizable Routing Metrics", Proceedings of the IEEE International Conference on Network Protocols, pp. 71-78, 1998.

[12] Hedrick, C. "Routing Information Protocol", RFC 1058, June 1998.

[13] Hinden, R., Sheltzer, A., "DARPA Internet Gateway", RFC 823, September 1982.

[14] Merlin, P. M. and A. Segall, "A Failsafe Distributed Routing Protocol", IEEE Transactions on Communications, Vol. COM-27, No. 9, pp. 1280-1288, 1979.

[15] Moy J, "OSPF Version 2", RFC 1247, August 1991.

[16] Segall, A. , "Distributed Network Protocols", IEEE Transactions on Information Theory, Vol. IT-29, No. 1, pp. 23-35, Jan. 1983.

[17] Schneider, M., "Flow Routing in Computer Networks", Ph.D. dissertation, The University of Texas at Austin, December 1997.

[18] Schneider, M., "Self-Stabilization", ACM Computing Surveys, Vol. 25, No. 1, 1983.

[19] Sur, S. and P. K. Srimani, "A Self-Stabilizing Distributed Algorithm for BFS Spanning Tree of a Symmetric Graph", Parallel processing Letters, Vol. 2, pp. 171-179, 1992.

[20] Varghese, G., "Self-Stabilization by Counter Flushing", Proceedings of the ACM Principles of Distributed Computing (PODC) conference, 1994.

[21] Vutukury, S., Garcia-Luna-Aceves, J. J., "A Distributed Algorithm for Multi-Path Computation", Proceedings of the 1999 Global Telecommunications Conference.

[22] Vutukury, S., Garcia-Luna-Aceves, J. J., "An Algorithm for Multi-Path Computation using Distance-Vectors with Predecessor Information", Proceedings of the 1999 ICCCN conference.

[23] Vutukury, S., Garcia-Luna-Aceves, J. J., "A Simple Approximation to Minimum Delay Routing", Proceedings of the 1999 SIGCOMM conference.

[24] Wang, Z. and J. Crowcroft, "Bandwidth-Delay Based Routing Algorithm", Proceedings of the IEEE Global Telecommunications Conference, 1995.

[25] M. Waris, "Propagated Timestamps: A Scheme for the Stabilization of Maxmum-Flow Routing Protocols", Master's Thesis, The University of Houston, Fall 1997.

[26] Zaumen, W., Garcia-Luna-Aceves, J.J., "Loop-Free Multipath Routing Using Generalized Diffusing Computations", Proceedings of the 1998 INFOCOM conference.