Journal of Parallel and Distributed Computing **62**, 922–944 (2002)
doi:10.1006/jpdc.2001.1830

Stabilization of General Loop-Free Routing

Jorge A. Cobb[1]

Department of Computer Science (MS EC-31), The University of Texas at Dallas, Richardson, Texas 75083-0688
E-mail: cobb@utdallas.edu

and

Mohamed G. Gouda

Department of Computer Science, The University of Texas at Austin, Austin, Texas 78712-1188
E-mail: gouda@cs.utexas.edu

Received February 14, 2000; accepted January 11, 2002

We present a protocol for maintaining a spanning tree that is maximal with respect to any given (bounded and monotonic) routing metric. This protocol has two interesting adaptive properties. First, the protocol is stabilizing: starting from any state, the protocol stabilizes to a state where a maximal tree is present. Second, the protocol is loop-free: starting from any state where a spanning tree is present, the protocol stabilizes, without forming any loops, to a state where a maximal tree is present. The stabilization time of this protocol is $O(n\,deg)$, where n is the number of nodes, and deg is the node degree in the network. © 2002 Elsevier Science (USA)

Key Words: formal method; loop-free routing; network protocol; stabilization.

1. INTRODUCTION

Consider a network, where processes are connected via communication channels. To identify a route from every process to a given destination process, a rooted spanning tree needs to be maintained in this network: the destination process is the root of the tree, and every other process maintains the identity of its parent in the tree. There are two basic approaches to maintain such a tree.

The first approach is called link-state routing (or broadcast routing). Here, each process broadcasts the status of its incident channels to all processes in the network. From these broadcasts, each process recreates in its memory the network topology. Then, each process builds in its memory a spanning tree of this topology, and

[1] To whom correspondence should be addressed.

0743-7315/02 $35.00

identifies its parent accordingly. Examples of link-state routing protocols include [17, 21].

The second approach is called distance-vector routing (or distributed routing), and it incurs less overhead than link-state routing. Here, each process forwards to each neighbor a copy of its distance to the destination. Based on the distances received from its neighbors, each process chooses its parent in the spanning tree, and updates its distance accordingly. Examples of distributed routing protocols include [9, 14, 15].

Unfortunately, distributed routing protocols suffer from long-lived loops and the counting-to-infinity problem, which deteriorate performance. This motivated the introduction of loop-free distributed routing protocols [9, 10, 16, 20]. These protocols achieve the property of loop-freedom by maintaining a relationship between the distance of each process and the distance of all descendants of that process. This relationship is maintained via diffusing computations. Thus, these protocols converge quickly to the desired spanning tree, require low memory and message overhead, and have the desirable property of avoiding routing loops.

The loop-free routing protocols presented in [9, 10, 16, 20] tolerate fail-safe failures of processes and channels. However, they have not been shown to tolerate a broader class of failures, some of which are hard to detect. Examples include: improper initialization of processes, undetected corrupted messages, and hardware/software errors in lower layers that manifest themselves on rare occasions. These faults can lead to synchronization problems between processes, from which the protocol may never recover.

To overcome this weakness, stabilizing loop-free routing protocols were developed. A protocol is said to be stabilizing iff, starting from any arbitrary state (such as the state after an undetected fault), the protocol converges to a legitimate operating state within finite time. Stabilizing protocols are desirable due to their high degree of fault-tolerance [19], since they tolerate all types of transient faults.

The first stabilizing, loop-free, distributed routing protocol was presented in [11, 18]. In this protocol, the metric of each path is the path's bottleneck bandwidth. That is, the metric is the minimum bandwidth of the channels along the path.

The protocol operates in rounds. When no further updating is possible, i.e., when no action is enabled, the next round begins. Each round is implemented by propagating a sequence number along the entire network. In an odd round, the bandwidth of each channel is updated, and each process updates its bandwidth from the bandwidths of its parent and the channel to its parent. In an even round, processes may change parents to improve their bandwidth. This protocol has been shown to stabilize to a maximal bandwidth tree in $O(L\,n)$ time [18], where n is the number of processes, and L is the maximum length of a simple path in the network. Since $L = n - 1$ in most networks, the stabilization time of this protocol is $O(n^2)$.

In [5], we presented a stabilizing, loop-free, distributed routing protocol, also based on bottleneck bandwidth. This protocol more closely resembles the operation of the first loop-free routing protocols [9, 16, 20]. Diffusing computations to propagate changes in bandwidth are restricted to the subtree where the change occurred. Thus, a computation involving all processes in the system is often not necessary after a bandwidth change occurs. This protocol stabilizes in $O(L\,deg)$ time, where deg is the node degree in the network. Thus, it stabilizes in $O(n\,deg)$ time.

The protocol in [5] requires several rounds of propagating sequence numbers, even during periods when the bandwidth of each channel remains constant. The purpose of these rounds is to break existing loops after a fault occurs. Unfortunately, these rounds increase message and processing overhead. Also, the sequence numbers are unbounded. To bound the sequence numbers, an additional minimum-hop spanning tree is required. However, changes in the network topology, such as a channel going down, may temporarily affect the minimum-hop tree. This in turn affects the routing spanning tree, even though the downed channel is currently not involved in the routing spanning tree.

In this paper, we present a stabilizing, loop-free, distributed routing protocol, which requires no global propagation of sequence numbers. Consider the bottleneck bandwidth metric. While the channel bandwidths remain constant, and while no faults occur, all variables in our protocol remain constant. If a change in bandwidth needs to be propagated using a diffusing computation, then this diffusing computation begins immediately, without waiting for a sequence number to be propagated from the root. Also, the diffusing computation is restricted to the subtree where the change occurs. Finally, the protocol converges in $O(L\,deg)$ time, and therefore, in $O(n\,deg)$ time.

To make our protocol as general as possible, we adopt the model of general metrics introduced in [12, 18]. The general metric model captures most of the popular metrics, such as bandwidth, delay, distance, etc. A stabilizing, distributed routing protocol for general metrics was introduced in [13]. Also, the loop-free and stabilizing protocol presented in [11, 18] supports general metrics after a minor modification. Note that if a protocol is correct for one metric, it does not automatically become correct for general metrics. Thus, to ensure our protocol supports the general metric model, we take general metrics into consideration from the onset of our protocol.

The paper is organized as follows. In Section 2, we review the model of general routing metrics. In Section 3, we introduce our process notation. In Section 4, we present a simple protocol that is neither stabilizing nor loop-free. In Section 5, we strengthen this protocol to obtain a protocol that is loop-free but not stabilizing. In Section 6, we further strengthen the protocol to obtain a stabilizing and loop-free protocol. In Section 7, the stabilizing and loop-free protocol of Section 6 is proven correct. Concluding remarks are given in Section 8.

2. NETWORKS AND ROUTING METRICS

A *network* is an undirected graph, where each node is called a process and each edge is called a channel. Throughout the paper, we use the terms node and process interchangeably, and use the terms edge and channel interchangeably. Processes u and v are said to be *neighbors* iff edge (u, v) is in the network. Each network has a distinguished process.

A *routing tree* in a network is a spanning tree rooted at the distinguished process of the network. The objective of a routing protocol is to obtain a routing tree T, where the path along T from any process to the root is optimal with respect to some metric.

The question of how to define such a metric admits many answers. In one case [8], the metric is distance, where each edge is given a distance value, and an optimal path

is the shortest path. In another case [22], the metric is bottleneck bandwidth, where each edge is given a bandwidth value, and an optimal path is the path with largest bottleneck bandwidth.

To ensure our protocols are as general as possible, we adopt the general model of routing metrics [12, 18], which is defined as follows.

A *routing metric* is a five-tuple $(M, W, f, m_r, \prec)$, where:

M is a set of metric values
W is a set of edge weights
f is a metric function whose domain is $W \times M$ and whose range is M
m_r is a metric value in M called the *root value*
$\prec$ is a binary relation over M satisfying the following three conditions, for every m, m', and m'' in M:
Irreflexivity: $\neg(m \prec m)$,
Transitivity: $(m \prec m' \wedge m' \prec m'') \Rightarrow m \prec m''$,
Totality: $(m \prec m') \vee (m' \prec m) \vee (m = m')$.

Henceforth, we adopt the notation $(m \preceq m')$ to denote $(m \prec m' \vee m = m')$.

Several steps are needed before we can use routing metrics to identify optimal paths. First, the routing metric is assigned to the network, as follows.

An *assigned routing metric* to a network is a six-tuple $(M, W, f, m_r, \prec, wt)$ where $(M, W, f, m_r, \prec)$ is a routing metric, and wt is a function that assigns to each edge (u, v) in the network a weight $wt.u.v$ from set W.

Next, a metric value needs to be assigned to every process in the network. The metric value of a process is a measure of the quality of its path to the root. This is defined as follows.

Let T be a routing tree of a network, and $(M, W, f, m_r, \prec, wt)$ be an assigned routing metric to that network. Each process u in the network has a metric value $m.u$ that is computed over T as follows:

$m.u = m_r$ if u is the network root,
$m.u = f(wt.u.v, m.v)$ if v is the parent of u in T, $m.v$ is the metric value of v in T, and $wt.u.v$ is the weight assigned to (u, v).

Every metric value m in M is assumed to satisfy the following *utility requirement*. There is a non-empty sequence of edge weights $w_0, w_1, \ldots, w_{k-1}$ in W and a sequence of metric values $m_0, m_1, \ldots, m_k$ in M such that the following conditions hold:

$$
\begin{aligned}
m_0 &= m_r,\\
m_1 &= f(w_0, m_0),\\
&\vdots\\
m_k &= f(w_{k-1}, m_{k-1}),\\
m &= m_k.
\end{aligned}
$$

Note that if M has an element m that does not satisfy this utility requirement, then m can be removed from M without any effect on the metric.

Finally, given that a routing tree defines a metric value for every process, the objective of a routing protocol is to identify a routing tree that provides the

"maximum" metric value to each process. This routing tree is known as a maximal routing tree, and is defined as follows.

Let $(M, W, f, m_r, \prec, wt)$ be an assigned routing metric to a network. A routing tree T of this network is called *maximal* with respect to this assigned routing metric iff for every routing tree T' and every process u in the network, $m'.u \preceq m.u$, where $m'.u$ is the metric value of process u over tree T', and $m.u$ is the metric value of process u over tree T.

A routing protocol should be capable of operating on any network. Thus, a metric used by a routing protocol needs to identify a maximal routing tree in every network. In other words, a metric needs to be maximizable as defined next.

A routing metric is *maximizable* iff for every network and every assignment of this metric to this network, the network has a maximal routing tree with respect to the assigned metric.

It is shown in [12, 18] that a necessary and sufficient condition for a metric $(M, W, f, m_r, \prec)$ to be maximizable is for this metric to be both bounded and monotonic, as defined below:

Boundedness: $(\forall\ m, w : m \in M\ \wedge\ w \in W : f(w, m) \preceq m)$.
Monotonicity: $(\forall\ m, m', w : m \in M\ \wedge\ m' \in M\ \wedge\ w \in W :$
$(m \prec m') \Rightarrow f(w, m) \preceq f(w, m'))$.

As an example, consider the network in Fig. 1(i). This network has four processes (named root, a, b, c) and four edges. Assume that the metric assigned to this network is the bottleneck bandwidth metric $(M, W, f, m_r, \prec)$. In this case, both M and W are sets of positive integers. The metric function is defined as $f(w, m) = \min(w, m)$. The root value m_r is infinity. Relation $\prec$ is defined as the less-than relation over integers. In Fig. 1(i), the weight of each edge is indicated to the side of the edge. Also, edges in

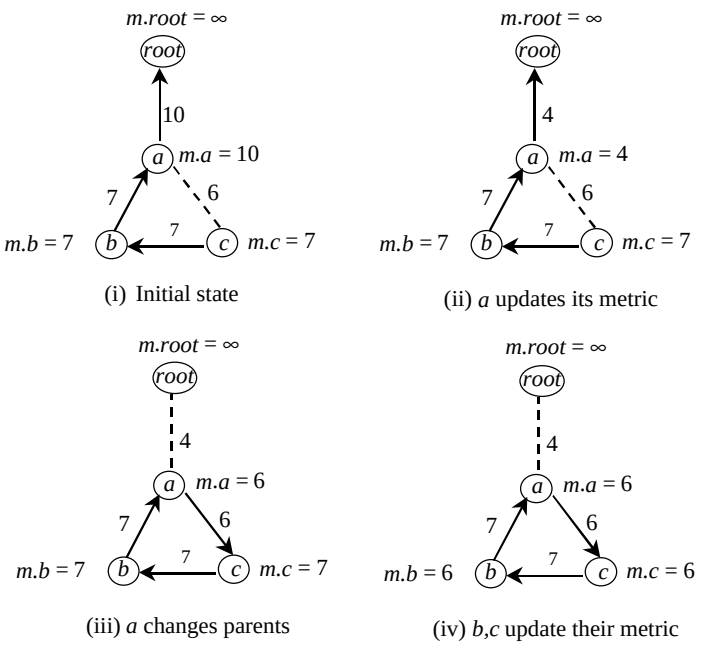

FIG. 1. Forming a permanent loop.

the routing tree are indicated with arrows. Therefore, $m.a = \min(wt.a.root, m.root) = \min(10, \infty) = 10$. Similarly, $m.b = \min(wt.b.a, m.a) = \min(7, 10) = 7$, and finally, $m.c = \min(wt.c.b, m.b) = \min(7, 7) = 7$.

In this paper, we present a protocol for obtaining and maintaining a maximal routing tree in any network that has been assigned a maximizable (i.e., bounded and monotonic) routing metric. The protocol is loop-free, and thus, no loops are formed when the edge weights change during any execution of the protocol. The protocol is also stabilizing, and thus, it converges to a maximal routing tree starting from an arbitrary state.

3. NOTATION FOR PROTOCOL SPECIFICATION

Before presenting our protocols, we first give a short overview of the notation that we use in specifying network protocols. For simplicity, the processes in our protocols are specified using a shared memory notation. In particular, each process is specified by a set of inputs, a set of variables, a parameter, and a set of actions. A process is specified as follows:

process ⟨process name⟩
inp
  ⟨input name⟩ : ⟨type⟩,
  ...
  ⟨input name⟩ : ⟨type⟩
var
  ⟨variable name⟩ : ⟨type⟩,
  ...
  ⟨variable name⟩ : ⟨type⟩
par
  ⟨parameter name⟩ : ⟨type⟩
begin
  ⟨action⟩
[]
  ...
[]
  ⟨action⟩
end

The inputs declared in a process can be read, but not written, by the actions of that process. The variables declared in a process can be read and written by the actions of that process. The parameter is discussed below.

Every action in a process is of the form: ⟨guard⟩ → ⟨statement⟩. The ⟨guard⟩ is a boolean expression over the inputs, variables, and parameter declared in the process, and also over the variables declared in the neighboring processes of that process. The ⟨statement⟩ is a sequence of conditional statements of the following form:

⟨variable⟩ := ⟨expression⟩ **if** ⟨boolean expression⟩.

If the ⟨boolean expression⟩ is true before the conditional statement is executed, then the ⟨variable⟩ is assigned the current value of the ⟨expression⟩. If the ⟨boolean expression⟩ is false, then the ⟨variable⟩ remains unchanged. If the phrase **if** ⟨boolean expression⟩ is not present in a conditional statement, then the value of the ⟨expression⟩ is assigned to the ⟨variable⟩ unconditionally.

The parameter declared in a process is used to write a set of actions as one action, with one action for each possible value of the parameter. For example, if we have the following parameter definition

par g : 1 .. 3
then the following action

$$x = g \rightarrow x := x + g$$

is a shorthand notation for the following three actions:

$$x = 1 \rightarrow x := x + 1$$

[]

$$x = 2 \rightarrow x := x + 2$$

[]

$$x = 3 \rightarrow x := x + 3$$

An execution step of a protocol consists in evaluating the guards of all the actions of all processes, choosing an action whose guard evaluates to true, and executing the statement of this action. An execution of a protocol consists of a sequence of execution steps, which either never ends, or ends in a state where the guards of all the actions evaluate to false. We assume all executions of a protocol are weakly fair, that is, an action whose guard is continuously true must be eventually executed.

To distinguish between the variables of different processes, we suffix the variable names with the process name. For example, variable $p.u$ corresponds to variable p in process u. In our protocols, each process u maintains a variable, $p.u$, where it stores the name of its parent in the current routing tree.

Since throughout the paper we deal primarily with spanning trees, we introduce the following notation.

A process u is an *ancestor* of a process v if u can be reached by following the parent variables starting from the parent variable of v. Process v is *descendant* of process u if process u is an ancestor of process v. More formally:

$$\begin{aligned}
L &= \text{maximum number of processes in a simple network path,} \\
parent(k,u) &= u && \text{if } k = 0, \\
parent(k,u) &= parent(k-1, p.u) && \text{if } k > 0, \\
hops(v,u) &= \min\{\, k \mid parent(k,v) = u\} && \text{if } (\exists k :: parent(k,v) = u), \\
hops(v,u) &= L && \text{otherwise,} \\
ancestor(u,v) &= (0 \leqslant hops(v,u) \ \wedge \ hops(v,u) < L), \\
descendant(v,u) &= ancestor(u,v), \\
height(u) &= \max_{v} \{\, hops(v,u) \mid descendant(v,u) = \text{true}\}.
\end{aligned}$$

4. THE UNSTABLE TREE PROTOCOL

In this section, we present a simple protocol that attempts to build a maximal routing tree. We refer to this protocol as the *unstable tree protocol*. Note that the protocol is called unstable because, as shown below, starting from a state where the parent variables define a spanning tree, the protocol may reach a state where the parent variables define a permanent loop.

The protocol simply consists of two actions, that are described next.

In the first action, each process u periodically updates its metric value, $m.u$, to the value $f(wt.u.(p.u), m.(p.u))$. In this way, the process ensures its metric value is consistent with the metric value of its parent on the tree.

In the second action, each process u chooses a neighbor g as its new parent, provided the metric value of u increases. The metric of u increases if $m.u \prec f(wt.u.g, m.g)$. Thus, in this case, u chooses g as its parent, and updates $m.u$ and $p.u$ accordingly.

The specification of a process u, other than the root, is given below:

process u
inp
$G.u$: set of neighbors of u
$wt.u.g$: weight of edge (u, g) for every g in $G.u$
var
$p.u$: element of $G.u$ { parent of u }
$m.u$: element of M { metric value of u }
par
g : element of $G.u$ { g is any neighbor }
begin
$p.u = g \rightarrow m.u := f(wt.u.g, m.g)$
[]
$m.u \prec f(wt.u.g, m.g) \rightarrow m.u := f(wt.u.g, m.g);$
$p.u := g$
end

The root process is specified as follows.

process *root*
var
$m.root$: element of M { metric value of *root* }
begin
$m.root \neq m_r \rightarrow m.root := m_r$
end

Note that *root* has no parent, and its metric value is set to m_r.

The unstable tree protocol allows processes to adapt to changes in edge weights and to changes in the metric values of processes. It also allows processes to improve their metric values by changing parents.

Unfortunately, if edge weights change, this protocol can cause loops to form. Consider the network example in Fig. 1. Fig. 1(i) shows the initial state of the network. Assume the weight of edge $(a, root)$ decreases to four. Thus, $m.a$ is decreased to four, as shown in Fig. 1(ii). Now, c offers a better parent choice to a, because $m.a = 4 < 6 = \min(wt.a.c, m.c)$. Hence, a chooses c as its new parent, and sets

$m.a$ to six, as shown in Fig. 1(iii). However, c is actually a descendant of a, and a loop is formed. Furthermore, b and c will update their metric values to six, as shown in Fig. 1(iv). Note that since the weight of edge $(a, root)$ is only four, a will not choose *root* as its parent, and the loop is never broken.

5. THE STABLE TREE PROTOCOL

In this section, we strengthen the unstable tree protocol to ensure that, under a fault-free execution, loops are avoided, and a maximal routing tree is obtained. We refer to this stronger protocol as the *stable tree protocol.*

Consider again the network example in Fig. 1. Note that the loop in this example could be avoided if process a did not change parents until all its descendants have a metric value at most that of a. To see this, assume that a is temporarily prevented from changing parents after its metric value is decreased to four. Then, b reduces its metric value to four, since the metric value of its parent a is four, and finally, c also reduces its metric value to four, since the metric value of its parent b is four. Thus, the network reaches a state where $m.a = 4 \ \wedge \ m.b = 4 \ \wedge \ m.c = 4$. In this state, process a does not need to change parents, since its metric value does not improve by doing so. Thus, the loop is avoided.

The above technique, with some variations, makes up the foundation of the first loop-free routing protocols [9, 16, 20], and also of recent loop-free routing protocols [10]. We use this technique in our stable tree protocol. In particular, each process u maintains a set of children of u, named $mwait.u$. The meaning of $mwait.u$ is as follows. If $g \in mwait.u$, then u is waiting for its metric value to propagate along the subtree of g. Thus, if $g \notin mwait.u$, then propagation of the metric value of u along the subtree of g has ended, and the subtree of g has only metric values which are at most that of u. More formally, the protocol should preserve the following predicate *mordered*:

$$mordered = (\forall\ u, g, v\ ::\ (p.g = u \ \wedge \ g \notin mwait.u \ \wedge \ descendant(v, g)) \ \Rightarrow \ m.v \preceq m.u).$$

Assuming mordered holds, let us examine when a process u should change its parent to g. Process u changes its parent to g only if its metric improves. This and metric monotonicity imply

$$m.u \prec f(wt.u.g, m.g) \preceq m.g.$$

If $mwait.u = \emptyset$ then, from *mordered*, any descendant v of u satisfies $m.v \preceq m.u$. Since we showed $m.u \prec m.g$, the new parent g cannot be a descendant of u, and no loop is formed. In summary, a process u may choose a neighbor g as its new parent without forming a loop when

$$m.u \prec f(wt.u.g, m.g) \ \wedge \ mwait.u = \emptyset.$$

Next, we examine how to preserve mordered. Consider when a neighbor g should be added to $mwait.u$. Assume $p.g = u \ \wedge \ g \notin mwait.u \ \wedge \ descendant(v, g)$. In this

case, $m.v \preceq m.u$ holds. The neighbor g should be added to $mwait.u$ when $m.v \preceq m.u$ ceases to hold. This may occur when $m.v$ increases or $m.u$ decreases:

- If $m.v$ increases, then, from boundedness, $m.v \preceq m.(p.v)$ after the increase. Furthermore, because $p.v$ is a descendant of g, $m.(p.v) \preceq m.u$. Thus, increasing $m.v$ preserves $m.v \preceq m.u$, and it is not necessary to add g to $mwait.u$.
- If $m.u$ decreases, then $m.v \preceq m.u$ may no longer hold, and g should be added to $mwait.u$. Note that process u does not know for which descendants v, $m.v \preceq m.u$ does not hold. Thus, if $m.u$ decreases, u must add all its neighbors to $mwait.u$.

Let us now examine when a neighbor g may be removed from $mwait.u$. If g is not a child of u, then obviously g may be removed. Assume g is a child of u. If $mwait.g = \emptyset$, then g is not involved in a metric value propagation. If $m.g \preceq f(wt.u.g, m.u)$, then g will not begin a new metric value propagation. Hence, if $m.g \preceq f(wt.u.g, m.u) \wedge mwait.g = \emptyset$, then the propagation of $m.u$ along the subtree of g has ended, and g may be removed from $mwait.u$.

In summary, a neighbor g may be removed from $mwait.u$ when the following holds:

$$p.g \neq u \vee (mwait.g = \emptyset \wedge m.g \preceq f(wt.u.g, m.u)).$$

We incorporate the above observations into the unstable tree protocol and obtain the following *stable tree protocol*:

process u
inp
  $G.u$: set of neighbors of u
  $wt.u.g$: weight of edge (u, g) for every g in $G.u$
var
  $p.u$: element of $G.u$ { parent of u }
  $m.u$: element of M { metric value of u }
  $mwait.u$: subset of $G.u$
par
  g : element of $G.u$ { g is any neighbor }
begin
  $p.u = g \rightarrow mwait.u := G.u$ **if** $f(wt.u.g, m.g) \prec m.u$;
    $m.u := f(wt.u.g, m.g)$
[]
  $m.u \prec f(wt.u.g, m.g) \wedge mwait.u = \emptyset \rightarrow$
    $m.u := f(wt.u.g, m.g)$;
    $p.u := g$
[]
  $p.g \neq u \vee (m.g \preceq f(wt.u.g, m.u) \wedge mwait.g = \emptyset) \rightarrow$
    $mwait.u := mwait.u - \{g\}$
end

Process u has three actions. In the first action, the metric value is updated to be consistent with the metric value of the parent. Note that if $m.u$ decreases, then $mwait.u$ is assigned the entire set of neighbors $G.u$ to ensure *mordered* is preserved. In the second action, process u changes parents. The guard of the action has been strengthened to

include $mwait.u = \emptyset$ to prevent the formation of loops. In the third action, a neighbor g is removed from set $mwait.u$ according to the criteria described above.

The specification of process *root* remains the same as in the unstable tree protocol.

The stable tree protocol satisfies the following two properties. First, if the protocol begins executing in a state where *mordered* holds, then *mordered* is preserved throughout the execution. Second, if the parent variables define a routing tree in the initial state of this execution, then a routing tree is always preserved, and furthermore, when the edge weights cease to change, a maximal routing tree is obtained.

The stable tree protocol, even though loop-free, is not stabilizing. For example, assume the protocol begins executing in the state depicted in Fig. 1(iv). As long as the edge weights remain constant, the existing loop will remain. In the next section, we present the stabilizing tree protocol, which remedies this problem.

6. THE STABILIZING TREE PROTOCOL

The obstacle for stabilization of the stable tree protocol is the possible presence of loops in the initial state of the protocol. If a loop exists in a state of the protocol, then this state is illegitimate, and the protocol should be reset to a legitimate state. However, a reset should not be performed if the current state is legitimate, since this may interrupt any underway convergence to a maximal tree. Therefore, the protocol needs to accurately detect the existence of loops.

To detect loops, each process u maintains a distance variable, $d.u$. This variable contains an estimate of the number of edges in the routing tree from process u to the root process. Variable $d.u$ is assigned the distance of u's parent, $d.(p.u)$, plus one.

Given that the maximum number of edges in any simple path is $L - 1$, it is intuitive to assume that $d.u \geqslant L$ indicates that u is involved in a loop. Hence, a reset should be performed. However, this is not necessarily the case, as illustrated by the next example.

Consider the network in Fig. 2. The metric chosen for this network is the bottleneck bandwidth metric. In this network, $L = 7$. Also, dashed edges are assumed to have a small weight, namely, one. Alongside each process are its metric value and its distance, in that order. The weight of each edge is also indicated alongside the edge. The initial state of the network is given in Fig. 2(i).

Assume edge (e, f) increases its weight to 20. This causes e to point to f and update its metric value and distance. For the moment, assume d is slow, so it does not update its values from those of e for some time. After e changes parents, the weight of edge (c, e) drops to one, and the weight of $(a, root)$ drops to five, which is then propagated to b and c. Still, d has not updated its values from those of e. This is shown in Fig. 2(ii).

Next, assume the weight of edge (b, d) rises to 20. Since the propagation on the left subtree has finished, b is free to change parents, and chooses d as its parent. The new metric value and distance are propagated to c. Still, d has not updated its values from those of e. Also, edge (b, a) drops to one. This is shown in Fig. 2(iii).

Note that in Fig. 2(iii), $d.c = 7 = L$. Thus, $d.c$ would indicate the presence of a loop, even though none exists. (The scenario in Fig. 2 can be extended further to show that $d.c$ grows without bound.)

From this discussion, we need to strengthen the protocol to accurately detect loops. To do so, the guard of the action where process u changes its parent to g is

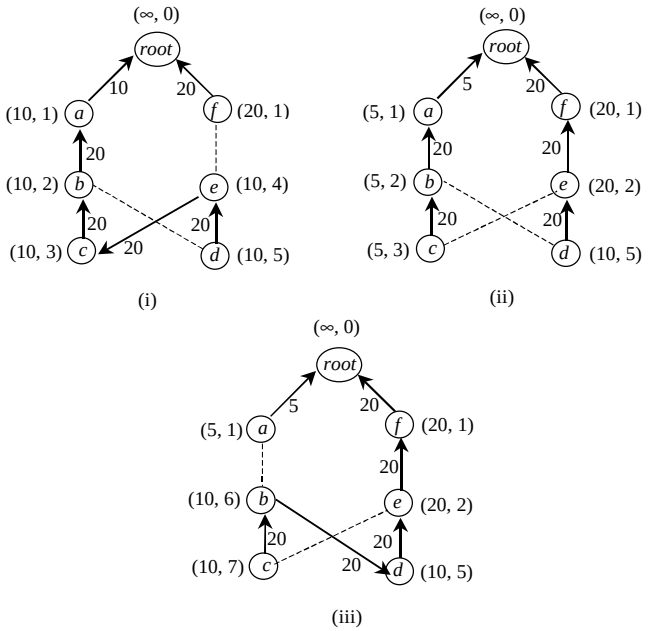

FIG. 2. Erroneous loop detection.

strengthened with the following conjunct:

$$d.u < L \ \wedge \ d.g < L - 1.$$

Note that $d.g < L - 1$ is required to ensure that the new distance of u, namely $d.g + 1$, is sensible, i.e., less than L. The reason for $d.u < L$ is a little more involved, and is explained as follows. On one hand, if process u is not in a loop, and $d.u$ becomes L, then a distance of at least L will propagate to the descendants of u, and so these descendants can neither change parents nor gain children. In this case, the subtree of u becomes fixed, and since $height(u) < L$, the maximum distance of any descendant of u is at most $(2 \cdot L) - 1$. On the other hand, if process u is indeed in a loop, and $d.u$ becomes L, then the distances will increase throughout the loop, and eventually the distance of every process in the loop is at least $2 \cdot L$. In summary, if the distance of a process becomes $2 \cdot L$, then this process detects the existence of a loop and requests a network reset to restore the network to a legitimate state.

One problem remains: it is possible for the protocol to reach a state where the distance of some process is $2 \cdot L$ even though no loop exists in this state. This problem is illustrated by the example in Fig. 3.

In this figure, the distance of a process is indicated to its side. In addition, $hops(b, a) = L/2$. The path from b to a is indicated by a dashed edge. Assume $d.a$ becomes L, and this distance has propagated down to process d, but has not yet reached process e. This is indicated in Fig. 3(i). Next, assume the distance of the parent of a decreases below $L - 1$, causing the distance of a to drop below L. This allows a to change parents to increase its metric value. Assume a chooses *root* as its parent, resulting in $d.a = 1$. This distance is then propagated down to c, resulting in

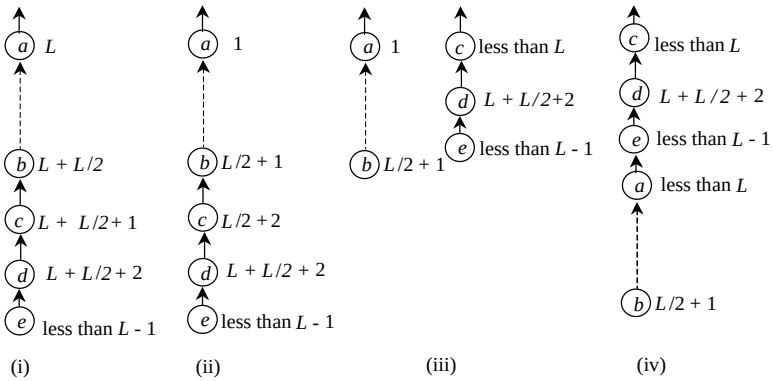

FIG. 3. Reaching a distance of $2 \cdot L$.

$d.c = L/2 + 2$. This is indicated in Fig. 3(ii). Next, since c has distance less than L, c changes parents. This is shown in Fig. 3(iii). Then, the metric value of a decreases, and this metric value is propagated along its subtree. Afterwards, a changes parents to increase its metric value. Let the new parent of a be e. This is shown in Fig. 3(iv). At this point, if the distance of process d propagates down its subtree, then the distance of b will be greater than $2 \cdot L$.

To avoid this problem, we add the following *wait* restriction:

> If $d.u \geqslant L$, then process u prevents its distance from decreasing below L, until all of its descendants have a distance at least L.

Note that the scenario in Fig. 3 is avoided under this restriction.

To implement this wait restriction, each process u maintains a set of children of u, named $dwait.u$. The meaning of $dwait.u$ is as follows. If a child g is in $dwait.u$, then $d.u \geqslant L$, and u is waiting for all the descendants of g to have distances of at least L. Thus, if a child g is not in $dwait.u$, then $d.u < L$ or all descendants of g have distance at least L. More formally, the protocol should preserve the *dordered* predicate, which is defined as follows:

$$dordered = (\forall\, u, g, v \;::\; (p.g = u \;\wedge\; g \notin dwait.u \;\wedge\; descendant(v, g)) \;\Rightarrow\; (d.v \geqslant L \;\vee\; d.u < L)).$$

To enforce the wait restriction, assuming *dordered* holds, $d.u$ is assigned $d.(p.u) + 1$ only if $d.(p.u) + 1 \geqslant L \;\vee\; dwait.u = \emptyset$.

Let us examine when to add processes to $dwait.u$. When $d.u$ increases from less than L to L or more, then *dordered* may no longer hold. In this case, to preserve *dordered*, all neighbors of u are added to $dwait.u$.

Next, we examine when to remove processes from $dwait.u$. If a neighbor g of u is not a child of u, obviously g may be removed from $dwait.u$. Also, if $d.u < L$, then u has no restrictions on the distance of its descendants, and any child g may be removed from $dwait.u$. Moreover, if a child g is such that $dwait.g = \emptyset \;\wedge\; d.g \geqslant L$, then all descendants of g have distance at least L, and hence, g may be removed from $dwait.u$. In summary, a neighbor g may be removed from $dwait.u$ if the following holds:

$$p.g \neq u \;\vee\; d.u < L \;\vee\; (d.g \geqslant L \;\wedge\; dwait.g = \emptyset).$$

We incorporate this loop-detection technique into our stable tree protocol, and obtain the following *stabilizing tree protocol*:

process u
inp
  $G.u$: set of neighbors of u
  $wt.u.g$: weight of edge (u, g) for every g in $G.u$
var
  $p.u$: element of $G.u$ { parent of u }
  $m.u$: element of M { metric value of u }
  $d.u$: integer { distance of u }
  $mwait.u$: subset of $G.u$
  $dwait.u$: subset of $G.u$
par
  g : element of $G.u$ { g is any neighbor }
begin
  $p.u = g \rightarrow mwait.u := G.u$ **if** $f(wt.u.g, m.g) \prec m.u$;
    $m.u := f(wt.u.g, m.g)$;
    $dwait.u := G.u$ **if** $d.u < L \wedge d.g + 1 \geqslant L$;
    $d.u := d.g + 1$ **if** $d.g + 1 \geqslant L \vee dwait.u = \emptyset$
[]
  $m.u \prec f(wt.u.g, m.g) \wedge mwait.u = \emptyset \wedge d.u < L \wedge d.g < L - 1 \rightarrow$
    $p.u := g$;
    $m.u := f(wt.u.g, m.g)$;
    $d.u := d.g + 1$
[]
  $p.g \neq u \vee (m.g \preceq f(wt.u.g, m.u) \wedge mwait.g = \emptyset) \rightarrow$
    $mwait.u := mwait.u - \{g\}$
[]
  $p.g \neq u \vee d.u < L \vee (d.g \geqslant L \wedge dwait.g = \emptyset) \rightarrow$
    $dwait.u := dwait.u - \{g\}$
end

Process u has four actions. In the first action, variables $m.u$ and $mwait.u$ are updated as before. Then, $dwait.u$ is assigned the entire set of neighbors to preserve *dordered* under the condition described earlier. Also, $d.u$ is assigned $d.(p.u) + 1$ provided the wait restriction is not violated. In the second action, process u changes parents. The guard has been strengthened to include $d.u < L \wedge d.g < L - 1$ as discussed above. The third action remains the same as the third action in the stable tree protocol. In the fourth action, a neighbor g is removed from $dwait.u$ according to the criteria discussed above.

The specification of process *root* is as follows:

process *root*
var
  $m.root$: element of M { metric value of *root* }
  $d.root$: integer { distance of *root* }
begin
  true $\rightarrow$ $m.root := m_r$;
    $d.root := 0$
end

Remaining to be specified is what to do when a loop is detected. If $d.u$ reaches at least $2 \cdot L$, then a reset is performed, using a reset protocol similar to that in [1]. In [1],

a spanning tree is constructed by electing a leader process, then a minimum-hop spanning tree rooted at this process is built. For our purposes, we require a slightly simpler version, where the spanning tree constructed by the reset protocol has the same root as the routing tree, i.e., the reset spanning tree is also rooted at process *root*. Let $rp.u$ be the parent of u in this reset spanning tree. The desired reset state, which we denote by *restart*, is as follows:

$$\begin{aligned} restart = (d.root = 0) \ \wedge \ (m.root = m_r) \ \wedge \\ (\forall u : u \neq root : p.u = rp.u \ \wedge \ d.u = 0 \ \wedge \ m.u = m_r \ \wedge \\ mwait.u = \emptyset \ \wedge \ dwait.u = \emptyset). \end{aligned}$$

Notice that $restart \Rightarrow (mordered \ \wedge \ dordered)$, as desired.

In the next section, we show that the stabilizing tree protocol, in combination with a reset protocol, is stabilizing. That is, starting from an arbitrary state, a maximal routing tree is obtained. Furthermore, under a fault-free execution, no loops are formed.

7. CORRECTNESS OF THE STABILIZING TREE PROTOCOL

In this section, we present a sketch for the correctness proof of the stabilizing tree protocol. Detailed proofs for all the theorems in this section can be found in [4].

The properties that we need to verify for this protocol are of the following two forms, where P and Q are predicates over states of the protocol.

1. P stable: This property is true iff, for all executions of the protocol,

$$(\forall i :: P(state_i) \Rightarrow (\forall j : j \geqslant i : P(state_j))).$$

2. $P \mapsto Q$: This property is true iff, for all executions of the protocol,

$$(\forall i :: P(state_i) \Rightarrow (\exists j : j \geqslant i : Q(state_j))).$$

We say that a protocol *stabilizes* to predicate P iff the following two properties hold:

1. P stable.
2. **true** $\mapsto P$.

Finally, define a round to be a subsequence of an execution, such that, for each action in the network, either the action is disabled at some state in the round, or the action is executed in the round.

We first show that the protocol stabilizes to *mordered*. We begin by giving an equivalent definition of *mordered*:

$$\begin{aligned} mordered_k = (\forall u, v, g :: (p.g = u \ \wedge \ g \notin mwait.u \ \wedge \ descendant(v, g) \ \wedge \\ hops(v, u) \leqslant k) \Rightarrow m.v \preceq m.u); \end{aligned}$$

$$mordered = mordered_{L-1}.$$

LEMMA 1. *For any* k, $0 < k < L$,

$$mordered_k \text{ stable}.$$

Proof. For a given process u, $mordered_k$ may be falsified when

1. $m.u$ decreases, or
2. $m.v$ increases for a descendant v of u, where $hops(v, u) \leqslant k$, or
3. u gains a new descendant v, where $hops(v, u) \leqslant k$, or
4. a neighbor g is removed from $mwait.u$.

We consider each of these in turn.

Consider 1 above. If $m.u$ decreases, then $mwait.u$ is assigned the entire set of neighbors of u. Thus, there is no longer a requirement of $m.v \preceq m.u$, and $mordered_k$ is preserved.

Consider 2 above. Let g be the ancestor of v which is a child of u (note that $g = v$ is possible). If $g \in mwait.u$, then increasing $m.v$ cannot falsify $mordered_k$. Assume instead that $g \notin mwait.u$. If $v = g$, then, from boundedness, $m.v \preceq m.u$ after v increases its metric, and $mordered_k$ is preserved. Assume that $v \neq g$. Then, from $mordered_k$, $m.(p.v) \preceq m.u$, and from boundedness, $m.v \preceq m.(p.v)$. From transitivity, $m.v \preceq m.u$, and $mordered_k$ is preserved.

Consider 3 above. Assume u gains a new descendant v. Let t be the ancestor of v which changed parents, and g the child of u which is an ancestor of both v and t. If after t changes parents, $g \in mwait.u \ \vee \ hops(v, u) > k$, then $mordered_k$ is not affected. Assume instead that after t changes parents, $g \notin mwait.u \ \wedge \ hops(v, u) \leqslant k$. If $v \neq t$, since t changes parents, $mwait.t = \emptyset$, and from $mordered_k$, $m.v \preceq m.t$ before t changes parents. Since $m.t$ increases, then $m.v \preceq m.t$ continues after t changes parents. If $v = t$, then $m.v \preceq m.t$ holds trivially after t changes parents. Let s be the new parent of t. Thus, $m.t \preceq m.s$ after t changes parents, and from transitivity, $mv \preceq m.s$. If $s = u$, then $m.v \preceq m.u$, as desired. If $s \neq u$, then $hops(s, u) \leqslant k$, and g is also an ancestor of s. Thus, from $mordered_k$, $m.s \preceq m.u$, and from transitivity, $m.v \preceq m.u$, as desired.

Consider 4 above. If $p.g \neq u$, then $mordered_k$ is unaffected. Otherwise, from the guard of the third action, $mwait.g = \emptyset$. From $mordered_k$, all descendants v of g up to k hops below g have $m.v \preceq m.g$. From the guard of the third action, $m.g \preceq m.u$, and from transitivity, $m.v \preceq m.u$. Hence, $mordered_k$ is preserved. ∎

We next address the progress property.

LEMMA 2. *For any* k, $0 < k < L - 1$,

$$mordered_k \mapsto mordered_{k+1} \text{ in one round}.$$

Proof. Since $mordered_k$ is stable (from Lemma 1), we assume $mordered_k$ holds throughout our execution. Define the *routing graph* to be the set of all edges $(u, p.u)$ for all processes u.

Let v be a descendant of u, and let g be the child of u which is an ancestor of v. Consider the following relation:

$$g \notin mwait.u \Rightarrow m.v \preceq m.u. \tag{1}$$

The theorem follows from the following three results:

1. If process u has a descendant v, where $hops(v,u) = k+1$, and relation (1) holds, then it continues to hold as long as the path from v to u in the routing graph remains constant.
2. If process u has a descendant v, where $hops(v,u) = k+1$, and relation (1) does not hold, then either relation (1) holds within one round, or the path from v to u in the routing graph changes.
3. If process u gains a new descendant v, where $hops(v,u) = k+1$, then relation (1) holds.

Consider first part one. In this case, assuming the path in the routing graph from v to u remains constant, we must consider the following events which may falsify relation (1):

(i) $m.u$ decreases. In this case, $mwait.u = G.u$ after the action, so relation (1) continues to hold.

(ii) $m.v$ increases. In this case, $m.v \preceq m.(p.v)$, and $hops(p.v,u) = k$. Thus, from $mordered_k$, $g \notin mwait.u \Rightarrow m.(p.v) \preceq m.u$, and from transitivity, $g \notin mwait.u \Rightarrow m.v \preceq m.u$, as desired.

(iii) g is removed from $mwait.u$. In this case, $hops(v,g) = k \wedge mwait.g = \emptyset \wedge m.g \preceq m.u$. From $mordered_k$, $m.v \preceq m.g$, and from transitivity, $m.v \preceq m.u$. Thus, relation (1) is preserved.

Consider now part two. If within a round, the path in the routing graph from v to u has changed, then we are done. Otherwise, within one round v will execute its first action. Thus, $m.v \preceq m.(p.v)$ holds. Note that $hops(p.v,u) = k$, and from $mordered_k$, $g \notin mwait.u \Rightarrow m.(p.v) \preceq m.u$, and from transitivity, $g \notin mwait.u \Rightarrow m.v \preceq m.u$, so relation (1) holds.

Consider now part three. Let t be the ancestor of v which changed parents. Assume $t \neq v$. Then, $hops(v,t) \leqslant k$. Since t changes parents, $mwait.t = \emptyset$, and from $mordered_k$, $m.v \preceq m.t$ before the action. Since $m.t$ increases when t changes parents, then $m.v \preceq m.t$ after t changes parents. If $t = v$, then $m.v \preceq m.t$ holds trivially after t changes parents. Thus, in either case, $m.v \preceq m.t$. Let s be the new parent of t. Thus, after t changes parents, $m.t \preceq m.s$, and from transitivity, $m.v \preceq m.s$. If $s = u$, then $m.v \preceq m.u$, and relation (1) holds. If $s \neq u$, then $hops(s,u) \leqslant k$, and g is an ancestor of s. Thus, from $mordered_k$, $g \notin mwait.u \Rightarrow m.s \preceq m.u$. From transitivity, $g \notin mwait.u \Rightarrow m.v \preceq m.u$, i.e., relation (1) holds. ∎

THEOREM 7.1.

(1) *mordered stable*.
(2) **true** $\mapsto$ *mordered within* $O(L)$ *rounds*.

Proof. Part (1) follows from Lemma 1 with $k = L - 1$. For part (2), we must first show that $mordered_1$ holds within a round. This reduces to, for all neighbors g of u,

$$p.g \neq u \ \vee \ g \in mwait.u \ \vee \ m.g \preceq m.u. \tag{2}$$

Assume relation (2) does not hold for a node u and its neighbor g. Thus, $p.g = u$. Within one round, either g changes parents and $p.g \neq u$, or g executes its first action, and $m.g \preceq m.u$. Hence, relation (2) holds for u and its neighbor g within one round.

What we must also show is that relation (2) continues to hold. Thus, eventually relation (2) holds for all u and g, and thus $mordered_1$ holds. Thus, assume relation (2) holds. We show that if any of the three disjuncts is falsified, then another one of the disjuncts becomes true.

First, assume g chooses u as its parent (first disjunct becomes false). In this case, $m.g \preceq m.u$, and thus relation (2) continues to hold. Next, assume g is removed from $mwait.u$ (second disjunct becomes false). In this case, the guard of the action ensures that $p.g \neq u \vee m.g \preceq m.u$ holds. Finally, (third disjunct) if $m.g$ increases, it cannot increase beyond $m.u$ unless $p.g \neq u$, and if $m.u$ decreases, g is added to $mwait.u$.

Hence, relation (2) cannot be falsified, and $moredered_1$ holds within a round.

Part two then follows from induction over k in Lemma 2. ∎

We next address the issue of loop freedom. We define predicate *loopfree* as follows:

$$(\forall\, u \ : \ u \neq root \ : \ ancestor(root, u)).$$

THEOREM 7.2.

(1) $mordered \ \wedge \ loopfree$ *stable*.
(2) **true** $\mapsto$ $(mordered \ \wedge \ loopfree) \ \vee \ (\exists\, u :: d.u = 2 \cdot L)$ *within* $O(L)$ *rounds*.

Proof. Consider part (1). From Theorem 7.1, *mordered* is stable. Thus, it will continue to hold. Note that if a process u changes parents to neighbor g, then from its second action, $mwait.u = \emptyset \ \wedge \ m.u \prec f(wt.u.g, m.g)$. From *mordered*, any descendant v of u must have $m.v \preceq m.u$. From boundedness, $m.u \prec m.g$. Hence, g cannot be a descendant of u, and no new loops may be introduced. Thus, *loopfree* continues to hold.

Consider now part (2). From Theorem 7.1, within $O(L)$ rounds, *mordered* holds and continues to hold. If there are no loops at this state, we are done. Otherwise, assume loops exist. From the proof of part one above, once *mordered* holds, no new loops are formed. If all loops disappear within $O(L)$ rounds, we are done. Otherwise, the distance of each process in a loop increases by at least one in each round. Thus, within $2 \cdot L$ rounds, there is at least one process in each loop with distance $2 \cdot L$, as desired. ∎

We have shown that within $O(L)$ rounds, either the protocol achieves and maintains a routing tree, or a reset occurs. Although the network is in a legitimate state after a reset is performed, we must show that eventually resets cease to occur, so the protocol may then converge to a maximal routing tree. That is, we must show that the distance of a process does not reach $2 \cdot L$ starting from a legitimate state. We begin by showing that the protocol stabilizes to *dordered*.

THEOREM 7.3.

(1) *dordered stable*.
(2) **true** $\mapsto$ *dordered within* $O(L)$ *rounds*.

Proof. Consider the similarities in the way the stabilizing tree protocol deals with metrics and distances.

The first action reestablishes a relationship, namely $\preceq$, between the metric of the parent and the metric of the child. If the metric relationship between process u and its descendants no longer holds, $mwait.u$ is assigned $G.u$. Similarly, the first action reestablishes a relationship between the distance of the parent and the distance of the child, namely $d.(p.u) < L \ \vee \ d.u \geqslant L$. If the distance relationship between process u and its descendants no longer holds, $dwait.u$ is assigned $G.u$.

In the second action, process u changes parents only if both the metric and distance relationships between itself and all its descendants hold.

In the third and fourth actions, a neighbor is removed from $mwait.u$ and $dwait.u$ if either the neighbor is not a child of u, or the metric and distance relationships have been reestablished along the subtrees of the neighbors.

Thus, given the similarities in the way that metric and distance are treated in the protocol, a very similar proof to that of Theorem 7.1 applies to Theorem 7.3. We leave the details to the reader. ∎

COROLLARY 7.4.

(1) $mordered \ \wedge \ dordered \ \wedge \ loopfree$ *stable*.
(2) **true** $\mapsto$ $(mordered \ \wedge \ dordered \ \wedge \ loopfree) \ \vee \ (\exists u :: d.u = 2 \cdot L)$ *within* $O(L)$ *rounds*.

Proof. The corollary follows from Theorems 7.2 and 7.3. ∎

A process u is said to be *sound* if propagating its distance to its descendants cannot cause any of them to obtain a distance of at least $2 \cdot L$. More formally,

$$
\begin{array}{lcl}
sound(u) & = & d.u \leqslant L \\
 & & \vee \ (\exists i : 0 < i < L : d.u = L + i \ \wedge \ hops(u, root) \geqslant i \ \wedge \\
 & & \qquad (\forall t : hops(u,t) \leqslant i : d.t \geqslant L)) \\
 & & \vee \ (\exists i : 0 < i < L : d.u = 2 \cdot L - i \ \wedge \ height(u) < i \ \wedge \\
 & & \qquad (\forall v : descendant(v,u) : d.v \geqslant L)), \\
dsound & = & (\forall u :: sound(u)).
\end{array}
$$

We define the predicate *legitimate* as follows:

$$legitimate = (mordered \ \wedge \ dordered \ \wedge \ loopfree \ \wedge \ dsound).$$

If a fault-free execution starts in a state satisfying *legitimate*, then a spanning tree is always maintained, and no resets occur.

THEOREM 7.5.

(1) *legitimate is stable*.
(2) **true** $\mapsto$ *legitimate* $\vee$ $(\exists u :: d.u \geqslant 2 \cdot L)$ *within* $O(L)$ *rounds*.
(3) *restart* $\Rightarrow$ *legitimate*.

From the theorem, the state achieved after a reset implies *legitimate*, and from the definition of *legitimate*, $d.u < 2 \cdot L$ for all u. Thus, within $O(L)$ rounds, *legitimate* is achieved, and no reset is requested afterwards. Note that if a reset protocol is used which stabilizes in $O(L)$ rounds, such as [1], then the network reaches a legitimate state in $O(L)$ rounds.

We are now required to show that, starting from a legitimate state, the network reaches a maximal routing tree. To do so, we assume that the weights of the edges remain constant, since otherwise, the desired maximal routing tree keeps changing, and thus the protocol may not ever reach it.

We begin by showing that eventually the metric of all processes can only increase. This is expressed by predicate *magree*, which is defined as follows:

$$magree = (\forall u : u \neq root : m.u \preceq f(wt.u.(p.u), m.(p.u))).$$

THEOREM 7.6.

(1) *legitimate* $\wedge$ *magree stable*.
(2) *legitimate* $\mapsto$ *legitimate* $\wedge$ *magree*, *within* $O(L)$ *rounds*.

From the above theorem, within $O(L)$ rounds, the metrics of each process can only increase. To make sure that a process can change parents if desired, we must show that $mwait.u = \emptyset$ holds and continues to hold. This is expressed by predicate *mnowait*, which is defined as follows:

$$mnowait = (\forall u : u \neq root : mwait.u = \emptyset).$$

THEOREM 7.7.

(1) *legitimate* $\wedge$ *magree* $\wedge$ *mnowait stable*.
(2) *legitimate* $\mapsto$ *legitimate* $\wedge$ *magree* $\wedge$ *mnowait within* $O(L)$ *rounds*.

We have shown above that $m.u$ and $mwait.u$ no longer prevent a process from changing parents to improve its metric. We must then show that the same holds for the distance, that is, $d.u$ and $dwait.u$ do not prevent a process from changing parents. We begin with some definitions. Below, $|p|$ is the number of edges in path p:

$$\begin{array}{rcl} above(u) & = & \text{maximum length path such that:} \\ & & above(u) \text{ is a simple path from } u \text{ to } root, \text{ and} \\ & & (\forall x : x \in above(u) \wedge x \neq u : m.u \preceq m.x \wedge \neg descendant(x, u)), \\ dagree & = & (\forall u :: d.u \leqslant |above(u)|), \\ dnowait & = & (\forall u : u \neq root : dwait.u = \emptyset). \end{array}$$

The above definitions assume that *legitimate* and *magree* hold. In this case, we have a single tree, and the metrics are non-increasing from parent to child. Note that

dagree $\Rightarrow d.u < L$. If *dagree* holds, then all the processes in *above*(u) cannot become descendants of u, since their metrics are only allowed to increase. Thus, if *dagree* holds, propagating $d.u$ to the descendants of u will not cause their distance to reach L.

THEOREM 7.8.

(1) *legitimate* $\wedge$ *magree* $\wedge$ *dagree* $\wedge$ *dnowait stable*.
(2) *legitimate* $\mapsto$ *legitimate* $\wedge$ *magree* $\wedge$ *dagree* $\wedge$ *dnowait within* $O(L)$ *rounds*.

COROLLARY 7.9.

(1) *legitimate* $\wedge$ *magree* $\wedge$ *dagree* $\wedge$ *mnowait* $\wedge$ *dnowait stable*.
(2) *legitimate* $\mapsto$ *legitimate* $\wedge$ *magree* $\wedge$ *dagree* $\wedge$ *mnowait* $\wedge$ *dnowait within* $O(L)$ *rounds*

Proof. The corollary follows from Theorems 7.6–7.8. ∎

We thus conclude that the network will reach a state in which a process is free to change parents if a neighbor offers a better metric. Thus, the protocol will quickly obtain a maximal routing tree. We define *maxtree* as follows:

$$\{(u, v) \mid v = p.u\} \text{ is a maximal routing tree.}$$

THEOREM 7.10.

(1) *legitimate* $\wedge$ *magree* $\wedge$ *dagree* $\wedge$ *mnowait* $\wedge$ *dnowait* $\wedge$ *maxtree stable*.
(2) *legitimate* $\mapsto$ *legitimate* $\wedge$ *magree* $\wedge$ *dagree* $\wedge$ *mnowait* $\wedge$ *dnowait* $\wedge$ *maxtree within* $O(L)$ *rounds*.

8. CONCLUDING REMARKS

In this paper, we presented three protocols to obtain a maximal routing tree in any network that has been assigned a maximizable (i.e., bounded and monotonic) routing metric. We first presented the unstable tree protocol. This protocol will either obtain a maximal routing tree or form a permanent loop. Next, we presented the stable tree protocol. This protocol is loop-free. That is, if in the initial state of the network all parent variables define a routing tree, then the protocol always maintains a routing tree, and furthermore, the protocol will reach a maximal routing tree. However, this protocol is not stabilizing. Finally, we presented the stabilizing tree protocol, which is both loop-free and stabilizing.

In the stabilizing tree protocol, a process u stops changing parents when $d.u \geqslant L$. This can be generalized to $d.u \geqslant C$, for some constant C, $C \geqslant L$. Then, a loop would be indicated when $d.u \geqslant C + L$. In our case, $C = L$. A greater C value allows greater flexibility, since it has a lesser probability of preventing a process from changing parents. However, a greater C value decreases the convergence time.

We have made the simplifying assumption that edges are undirected. However, the protocol is easily adapted to directed edges. In particular, each process u requires two inputs for every neighbor g, $wt.u.g$ and $wt.g.u$. These correspond, respectively, to the weights of edges (u, g) and (g, u). In the third action of the stable tree protocol, and also of the stabilizing tree protocol, $wt.u.g$ is replaced for $wt.g.u$.

The space overhead of a stabilizing reset protocol is not significant. It consists of one parent variable, one distance variable, and a few bits representing the state of the reset protocol [1]. The parent and distance variables are required to build a reset spanning tree. Notice that if the reset protocol has stabilized, then a channel failure will not cause a reset to occur. Therefore, if a channel fails, and if it is not being used currently in the routing tree, then this does not affect the behavior of the routing protocol.

Regarding the convergence time of the stabilizing tree protocol, we showed that convergence is achieved in $O(L)$ rounds, and thus, in $O(n)$ rounds. Since each process has $O(deg)$ actions, where deg is the process degree of the network, then the protocol will converge in $O(n\ deg)$ time. This is assuming an action takes constant time. Notice that if we assume that deg is constant, then the protocol will converge in $O(n)$ time.

The way in which the protocol reestablishes *mordered* and *dordered* is very similar. The protocol performs the following for both the metric and distance values. In the first action of u, the relationship between the values of the parent and the values of u is restored. If by doing so the relationship between the values of u and the values of the descendants of u is violated, then a set $wait.u$ is assigned all the neighbors of u. In the second action, a new parent is chosen, provided the values of u and its new parent are related. Finally, there is an action which removes a child g from $wait.u$ if the relationship between the values of u and its descendants in the subtree of g has been restored. An abstract version of this protocol is presented in [4].

We have considered thus far only monotonic metrics. In [18], non-monotonic metric vectors of the form $(m_0, m_2, \ldots, m_{r-1})$ are considered, where each $m_i, 0 \leqslant i \leqslant r-1$, is a monotonic metric. Here, the objective is to look for a path which maximizes m_0, and of those with maximum m_0, choose a path maximizing m_1, etc. Depending on the individual metrics, the vector $(m_0, m_2, \ldots, m_{r-1})$ may not be monotonic. In [18], metrics of this form are handled by introducing additional rounds for each additional metric. For our protocol, metrics of this form can be handled using the technique introduced in [2]. Here, we dealt with metric vectors, and the strategy to break loops involved the propagation of sequence numbers, as in [5]. The technique can be easily adapted to the protocol we presented in this paper.

Regarding a message passing implementation, note that each action reads the local variables of the process and the variables of only one neighboring process. Thus, a message passing implementation is possible using request-reply interactions between neighbors, as done in [9, 10].

ACKNOWLEDGMENT

We are thankful to Dr. Marco Schneider for his valuable comments which helped improve the quality of this paper.

REFERENCES

1. A. Arora and M. G. Gouda, Distributed reset, *IEEE Trans. Comput.* **43**(9) (September 1994) 1026–1039.
2. J. A. Cobb, Fault-tolerant multi-metric routing, *IASTED Principles of Distributed Computing and Networks*, PDCN 1998.
3. J. A. Cobb and M. G. Gouda, The request-reply family of group routing protocols, *IEEE Trans. Comput.* **46**(6) (June 1997), 659–672.
4. J. A. Cobb, and M. G. Gouda, "Stabilization of General Loop-Free Routing, Technical Report," UTDSC-02-02 Department of Computer Sciences, The University of Texas at Dallas, February, 2002.
5. J. A. Cobb and M. Waris, Propagated timestamps: A scheme for the stabilization of maximum-flow routing protocols, *in* "Proceedings of the Third Workshop on Self-Stabilizing Systems," pp. 185–200, 1997.
6. Y. K. Dalal and R. M. Metcalfe, Reverse path forwarding of broadcast packets, *Comm. Assoc. Comput. Mach.* **21**(12) (December 1978), 1040–1048.
7. S. Deering and D. Cheriton, Multicast routing in datagram networks and extended LANs, *ACM Trans. Comput. Syst.* **8**(2) (May 1990), 85–110.
8. E. W. Dijkstra, A note on two problems on connection with graphs, *Numer. Math.* **1** (1959), 269–271.
9. J. J. Garcia-Luna-Aceves, Loop-free routing using diffusing computations, *IEEE/ACM Trans. Networking.* **1**(1) (February 1993), 130–141.
10. J. J. Garcia-Luna-Aceves and S. Murthy, A path-finding algorithm for loop-free routing", *IEEE/ACM Trans. Networking* **5**(1) (February 1997), 148–160.
11. M. G. Gouda and M. Schneider, Maximum flow routing, *in* "Proceedings of the Second Workshop on Self-Stabilizing Systems," Technical Report, Department of Computer Science, University of Nevada, Las Vegas, May 1995.
12. M. G. Gouda and M. Schneider, Maximizable routing metrics, *in* "Proceedings of the IEEE International Conference on Network Protocols," pp. 71–78, 1998.
13. M. Gouda and M. Schneider, Stabilization of maximal metric trees, *in* "19th International Conference on Distributed Computing Systems, Workshop on Self- Stabilizing Systems," June 1999.
14. C. Hedrick, "Routing Information Protocol," RFC 1058, 1988.
15. R. Hinden and A. Sheltzer, "DARPA Internet Gateway Protocol," RFC 823, September 1982.
16. P. M. Merlin and A. Segall, A failsafe distributed routing protocol, *IEEE Trans. Comm.* **27**(9) (1979), 1280–1288.
17. J. Moy, "OSPF Version 2," RFC 1247, August 1991.
18. M. Schneider, "Flow Routing in Computer Networks," Ph.D. dissertation, The University of Texas at Austin, December 1997.
19. M. Schneider, Self-Stabilization, *ACM Comput. Surveys* **25**(1) (1983), 45–67.
20. A. Segall, Distributed network protocols, *IEEE Trans. Inform. Theory* **29**(1) (January 1983), 23–35.
21. S. Vutukury and J. J. Garcia-Luna-Aceves, A simple approximation to minimum delay routing, *in* "Proceedings of the 1999 SIGCOMM conference."
22. Z. Wang and J. Crowcroft, Bandwidth-delay based routing algorithm, *in* "Proceedings of the IEEE Global Telecommunications Conference," 1995.