Abstract. We present a protocol for maintaining a routing tree that is maximal with respect to any given (bounded and monotonic) routing metric. This protocol has three 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 assumes no upper bound on the length $L$ of the longest network path; nonetheless, its stabilization time is $O(L \cdot \text{deg})$, where $\text{deg}$ is the node degree in the network. Third, the spanning tree remains connected while adapting to a change in the edge weights in the network. This last property makes the protocol suitable for a routing policy that ensures each message is delivered to its destination, even while the routing tree is adapting to the new edge weights.

keywords: stabilization, maximizable metrics, routing, computer networks

1 Introduction

Optimal-routing in a computer network consists of building a spanning-tree such that two conditions hold: a) the root of the tree is a distinguished node, and b) weights are assigned to the network links, and each path along the tree to the root is optimal with respect to these weights [1]. This differs from leader election protocols [2–4], where any node can be the root, and the tree need not be optimal with respect to link weights.

Maximizable routing metrics [1] are an abstraction of optimal-routing in computer networks. They encompass hop-count routing, minimum-cost routing, bottleneck-bandwidth routing, and many others metrics used in practice.

We are interested in stabilizing protocols for optimal-routing with respect to maximizable metrics. A protocol is said to be stabilizing if started from an arbitrary initial state, it eventually reaches a normal operating state [5][6].

Stabilizing protocols for this problem have been presented in the literature. A distance-vector-type protocol to build a maximal routing tree was given in [7], and a loop-free protocol was later presented in [8]. Both protocols assume there exists an upper bound $L_{\text{max}}$ on the length of simple network paths, and this bound is known by all nodes. This allows a hop-count to be used to detect and break routing loops before the system converges to a routing tree. The
convergence time is related to $L_{\max}$, even if the loops consists of a much smaller number of nodes.

The problem with this approach is that, under some conditions, the network size may not be known. Network nodes can be added or removed, changing the topology significantly. For example, in a sensor network, new sensors may be deployed, or a significant number of existing sensors exhaust their power.

In [9], we presented a routing protocol for maximizable metrics without assuming an upper bound $L_{\max}$. The convergence time is bounded by $L$, where $L$ is actual length of the longest simple path in the network. If $L_{\max} \gg L$, significant improvements can be achieved in convergence time.

To achieve fast converge, the protocol in [9] causes nodes to become temporarily disconnected from the routing tree, until convergence is reached. Although fast convergence is achieved, disconnected nodes prevent route preservation. Route-preservation, as defined in [10], is the ability to ensure that data messages reach the root irrespective of changes in the routing tree. That is, as the routing tree adapts to continuous changes in edge weights, data messages are routed in a manner that they are guaranteed to reach the root.

In [10], a stabilizing route-preserving protocol is presented, based on $r$-operators [11][12] without nodes assuming an upper bound $L_{\max}$. The protocol has two limitations: a) similar to distance-vector routing, it is prone to long-lived loops, and thus, convergence time is related to the maximum difference between metric values, and b) it is restricted to strictly-monotonic metrics, i.e., the metric value of a node must be worse than the metric value of its parent.

In this paper, we present a stabilizing routing protocol for maximizable metrics that combines the best of all the above approaches: a) it is suitable for all maximizable metrics, not just strictly-bounded metrics, b) nodes have no knowledge of $L_{\max}$, c) convergence time is proportional to $L$ (no long-lived loops), and d) it is loop-free and suitable for route-preservation.

We present our protocol in two stages. First, we present a protocol that is loop-free and route-preserving. We then strengthen the protocol to become stabilizing.

2 Problem Statement

A network is an undirected graph, with node set $N$ and edge set $E$. Nodes $u$ and $v$ are said to be neighbors iff edge $(u, v)$ is in the network. Each network has a distinguished node called root. A protocol is a set of actions to be executed at each node to accomplish a common task.

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

Multiple metrics are used in practice, such as bottleneck bandwidth, minimum cost, minimum hop, etc. To ensure our protocols are as general as possible, we adopt the general model of maximizable routing metrics [13, 1].
The four desirable properties of our protocol were listed in the introduction. To achieve them, we assume that there already exists a routing protocol for maximizable metrics that is stabilizing, has no knowledge of $L_{\text{max}}$, and converges in time proportional to $L$. We presented such a protocol in [9]. We will refer to this protocol as the base routing protocol. Our routing protocol is composed with the base protocol by reading some of the variables in the base protocol. The base protocol, however, is totally independent of our routing protocol.

3 Notation and Semantics

The behavior of each node in a protocol is specified by a set of inputs, a set of variables, a parameter, and a set of actions. The inputs declared in a node can be read, but not written, by the actions of that node. The variables declared in a node can be read and written by the actions of that node. The parameter is discussed further below.

Every action in a node is of the form:

\[ \text{<guard>} \rightarrow \text{<statement>}. \]

We assume a shared memory model, i.e., each node is able to read the variables of its neighbors. To provide a low level of atomicity, we assume each action can read only the variables of a single neighbor.

The \text{<guard>} is a boolean expression over the inputs, variables, and parameter declared in the node, and also over variables declared in a single neighboring node. The \text{<statement>} is a sequence of assignment statements of the form

\[ \text{<variable>} := \text{<expression>}. \]

The parameter declared in a node is used to expand one action into a set of actions, with one action for each possible value of the parameter. E.g., if we have the following parameter definition,

\[ \text{par } g : 1 .. 2 \]

then the action \( x = g \rightarrow x := x + g \) is a shorthand notation for the following two actions.

\[ x = 1 \rightarrow x := x + 1 \]
\[ x = 2 \rightarrow x := x + 2 \]

An execution step of a protocol consists in evaluating the guards of all actions of all nodes, 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, if at any point in an execution an action has a guard that evaluates to true, then, at some later step in the execution, either the guard of the action is no longer true, or the action is executed.
We say a protocol stabilizes to a predicate $P$ iff, for every execution (regardless of its initial state), there is a suffix in the execution where $P$ is true at every state in the suffix \cite{4,6}.

To distinguish between variables of different nodes, we prefix the variable names with the node name. E.g., $u.x$ corresponds to variable $x$ in node $u$.

We assume there is a distinguished node, root, and denote by $L$ the length (i.e., number of hops) of the longest simple path from any node to root. Nodes are unaware of the value of $L$; it is used, however, in proving the convergence time of our protocol. Finally, we denote by $deg$ the largest degree of any node.

4 Loop-Free Adaptive Routing

In this section, we present our loop-free adaptive routing (LFAR) protocol. We first present how, if there is a change in the weights of the edges, our protocol changes the structure of the routing tree, and transforms itself into an optimal routing tree. Throughout this change, there are neither loops nor disconnected nodes. Then, based on this loop-freedom, we present a routing policy that ensures each message reaches the root node, even while the routing tree is changing, i.e., the protocol is route-preserving as defined in \cite{10}.

4.1 Loop-Free Adaptive Routing

As mentioned in Section 2, we assume there exists a base protocol that builds a routing tree that is optimal with respect to some maximizing routing metric. Our only assumptions about this protocol are that, once the weight assignment to each edge is fixed, the protocol converges to an optimal routing tree.

At each node $u$, let variable $u.pr$ store $u$’s parent in the routing tree. We say that $v$ is a descendant of $u$ (and $u$ an ancestor of $v$) if there is a sequence of nodes $x_0, x_1, \ldots, x_j$ such that $u = x_0$, $v = x_j$, and for all $i$, $0 \leq i < j$, $x_i.pr = x_{i+1}$.

Let input $u.\hat{pr}$ contain the parent chosen for node $u$ by the base routing protocol. The value of $u.\hat{pr}$ varies until the base protocol converges to its final state. Our LFAR protocol is unaware of this convergence; it simply takes the current value of $u.\hat{pr}$ and assigns it to $u.pr$.

However, the above assignment cannot be performed without constraints. This is because, to provide route-preservation, its advantageous for a routing protocol to be loop-free \cite{10}. The base routing protocol, however, is not assumed to be loop-free. Hence, node $u$ cannot assign $u.\hat{pr}$ to $u.pr$ if in doing so a routing loop is formed.

To ensure loop-freedom, each node can be in one of two states: high or low. During steady-state operation, i.e., when both the LFAR and base routing protocols have the same tree ($u.pr = u.\hat{pr}$ for all $u$), all nodes are in the high state. When $u.pr \neq u.\hat{pr}$, node $u$ must change its parent. To avoid a loop, this new
parent must not be a descendant of $u$. Therefore, node $u$ must distinguish which of its neighbors are descendants and which are not. E.g., consider Fig. 1(a). Node $u$ must change its parent from $a$ to $v$. Before doing so, it must ensure that $v$ is not its descendant.

The above distinction is achieved via a request that begins at $u$, and causes all of its descendants to change their state to low. This is shown in Fig. 1(b), where both $u$ and $b$ have asked their descendants to become low.

To implement this diffusing request, each node $u$ has a boolean bit, $u\.req$, indicating the need for its descendants to become low. When $u$’s child, $x$, notices that $u$ is requesting a change, $x$ sets $x\.req$ to true. Similarly, $y$ sets $y\.req$ to true. Since $y$ has no children, $y$ changes its state to low, which in turn causes $x$ to do the same, and finally $u$. Thus, changing state from high to low starts from the leaf nodes, and it converges upward toward the node originating the request.

Once $u$’s state is low, all its descendants are also low. This allows $u$ to freely choose $v$ as its new parent, because $v$’s state is high, and thus, it cannot be a descendant. This is shown in Fig. 1(c). Since now $u$’s parent is the same on both trees, it changes its state to high. This in turn causes its children, and in turn its descendants, to change their state to high, as shown in Fig. 1(d).

We next present the specification for our route-preserving protocol. The specification for the root node is below. It simply consists of a set of constants that are read by its neighbors. It has no actions, since there are no variables to update.

```
node root
const
root\.st : element of {high}
root\.pr : element of {root}
```
\text{root.req} : \quad \text{false} \\
\text{root.hc} : \quad \text{0}
end

We next present a non-root node \(u\). In addition to \(u.pr\) and \(u.req\), \(u\) maintains three variables: \(u.st\) is the state of \(u\) (high or low), \(u.end\) is the set of neighbors for which the diffusing request has ended, and \(u.hc\) is the hop count to root\(^2\).

For simplicity, \(u.high\) is a shorthand for \(u.st = \text{high}\), similarly for \(u.low\).

\textbf{node} \(u\)

\textbf{inp}
\begin{itemize}
    \item \(u.N\) : set of node id’s \{neighbors of \(u\)\}
    \item \(u.\hat{pr}\) : element of \(u.N\) \{next parent of \(u\)\}
\end{itemize}

\textbf{var}
\begin{itemize}
    \item \(u.pr\) : element of \(u.N\) \{parent of \(u\)\}
    \item \(u.st\) : element of \{high, low\} \{state of \(u\)\}
    \item \(u.req\) : boolean \{request lowering of descendants\}
    \item \(u.end\) : subset of \(u.N\) \{neighbors where diffusion ended\}
    \item \(u.hc\) : integer \{hop count to root\}
\end{itemize}

\textbf{par}
\begin{itemize}
    \item \(g\) : element of \(u.N\) \{any neighbor of \(u\)\}
\end{itemize}

\textbf{begin}
\begin{itemize}
    \item \{start request if new parent\}
    \(u.high \land \neg u.req \land u.pr \neq u.\hat{pr} \quad \rightarrow \quad u.req := \text{true}; \quad u.end := \emptyset\)
    \item \{propagate request\}
    \((u.high \land \neg u.req) \land ((u.pr).high \land (u.pr).req) \quad \rightarrow \quad u.req := \text{true}; \quad u.end := \emptyset\)
    \item \{add neighbor to end set\}
    \(g.pr \neq \emptyset \lor g.low \quad \rightarrow \quad u.end := u.end \cup \{g\}\)
    \item \{terminate request and lower state\}
    \(u.high \land u.req \land u.end = u.N \quad \rightarrow \quad u.st := \text{low}\)
    \item \{change to new parent\}
    \(u.low \land (u.\hat{pr}).high \land \neg (u.\hat{pr}).req \quad \rightarrow \quad u.pr := u.\hat{pr}; \quad u.hc := (u.pr).hc + 1; \quad u.st := \text{high}; \quad u.req := \text{false}\)
\end{itemize}
end

Node \(u\) has five actions. In the first action, if \(u\)’s parent in the routing tree is not the same as in the base routing tree, \(u\) begins a request for all its descendants

\(^2\) This value is not necessary for the protocol to operate, but it will come into play in the stabilizing version of the protocol.
to change their state to low. This is done by setting \( u\.req \) to true, and assigning the empty set to \( u\.end \). Later, as neighbors change their state to low, they are added to \( u\.end \). In the second action, if \( u \)'s parent requests a change of state, \( u \) propagates this request.

In the third action, a neighbor \( g \) is added to \( u\.end \), indicating that the diffusing request has ended via \( g \). This is the case if \( g \) is not a child of \( u \), or if the state of the \( g \) is low.

In the fourth action, \( u \) ends the diffusing request if all its neighbors have ended it. In this case, \( u \) sets its state to low, and is free to change parents. In the fifth action, \( u \) changes parents, provided \( u \)'s state is low, the new parent’s state is high, and the new parent is not performing a diffusing request.

### 4.2 Correctness

The LFAR protocol is not stabilizing; we strengthen it to become stabilizing in Section 5. Nonetheless, it is correct in the sense that, if started in a loop-free state, it remains loop-free.

**Definition 1. (loop-freedom properties)** Let \( P_1 \) be the following predicate. For every node \( u \), and for every neighbor \( g \) of \( u \), the conjunction of:

\[
\begin{align*}
\text{\( u\.high \Rightarrow (u.\text{pr}).\text{high} \)} & \quad (1) \\
\text{\( u\.high \land u\.req \Rightarrow g.\text{pr} \neq u \lor g \notin u.\text{end} \lor g\.low \)} & \quad (2) \\
\text{\( (u.\text{pr}).\text{low} \Rightarrow u\.low \)} & \quad (3)
\end{align*}
\]

It is straightforward from the actions to show the following.

**Lemma 1.** If \( P_1 \) holds before an execution step of the LFAR protocol, then it also holds after the execution step.

From the above lemma, the normal behavior of the protocol is exemplified by Fig. 1. Any path to the root consists of two sub-paths: one of nodes in the low state, followed by another of nodes in the high state. Also from the above lemma, the following follows from the guard of the fifth action.

**Theorem 1. (global loop-freedom property)** Assume the LFAR protocol is started from a state in which \( P_1 \) holds, and no node is disconnected from root (the parent variables form a spanning tree). Then, after executing any action in the protocol, all nodes remain connected to root.

It is also necessary to argue that the protocol adapts to the optimal tree. That is, once the base routing protocol reaches a steady state, the LFAR protocol adapts its parent variables to reflect the new routing tree. We defer this argument until Section 6, where we discuss the proof of the stabilizing version of the protocol.
4.3 Route-Preservation Policy

Similar to [10], we define a routing policy that is route-preserving. I.e., we define a set of guidelines to route data messages toward the root, such that, even though edge weights are currently changing, and hence, the routing tree is changing, data messages are guaranteed to reach the root.

We assume each message has a one-bit flag that is reserved for the routing policy. The flag is set whenever the message arrives at a node whose state is low, or if the message is queued at a node whose state transitioned from high to low. Once the flag is set in a message, it may not be cleared.

Once a message is flagged and has reached a node in the high state, we require the message to continue only along nodes with a high state. This ensures no parent changes along its path, and thus, it eventually arrives at the root.

Since low states propagate from descendants to ancestors, a wave of low-state-changes should not overcome a flagged message. To do so, we restrict when a node can transition to a low state. This is prevented if the node currently has flagged messages in its queue. Thus, if a node’s state is high, before it transitions to a low state, it must first forward all flagged messages to its parent.

The above restriction could permanently prevent a node from performing the transition to a low state. To ensure the restriction is only temporary, the node may communicate with its children (which from Lemma 1 are in a low state), to prevent them from forwarding any more flagged messages. Once the node’s queue is empty of flagged messages, it changes its state to low, and allows its children to resume the forwarding of flagged messages.

5 Stabilizing Protocol

To stabilize the LFAR protocol, two obstacles must be cleared: a) if routing loops exist, they must be broken, and all of their nodes must rejoin the routing tree, and b) the local loop-freedom properties (or similar properties) must re-establish themselves automatically from any faulty state. We discuss each of these obstacles in turn.

There are two traditional ways to detect routing loops. The first is simply not to do anything specific to break loops, provided the routing metric is strictly-bounded [1], i.e., the metric becomes worse from parent to child (e.g., distance or hop-count metrics). In this case, the metric of nodes along a loop becomes progressively worse, until the metric of the routing tree is better than that of the loop. This causes nodes in the loop to change parents and rejoin the routing tree. However, this method suffers from long-lived loops, i.e., the time to convergence depends on the difference between metric values. Also, it is not useful when the metric is not strictly-bounded, such as with bottleneck-bandwidth metric.

Another approach is to maintain a hop-count to the root [8][15][1]. Nodes are aware of an upper bound $L_{\text{max}}$ on $L$, where $L$ is the length of the longest simple network path. When the hop-count reaches $L_{\text{max}}$, then a loop is assumed, and nodes will choose a new parent with a hop count less than $L_{\text{max}}$, even though
the metric via the new parent is worse than via the old parent. This approach has the disadvantage that the time to converge is proportional to $L_{\text{max}}$, which could be significantly greater than $L$.

Our approach also uses a hop-count, but without assuming an upper bound $L_{\text{max}}$. Instead, loops are assumed to exist if the hop-count of a child is not equal to the hop-count of its parent plus one. This requires that during normal operation, the hop-count of a node is always consistent with its parent. However, due to the dynamic structure of the routing tree, this is not possible.

We relax the above by taking advantage of the two-state nature of our diffusing computation. Consider again the routing-tree in Fig. 1. Each path to the root consists of a sequence of nodes in the low state, followed by a sequence of nodes in the high state. The hop-count requirement is thus relaxed as follows. For all non-root nodes $u$

\[ \neg(u.\text{low} \land (u.\text{pr}).\text{high}) \Rightarrow u.\text{hc} = (u.\text{pr}).\text{hc} + 1 \]  

(4)

That is, the hop-count is allowed to be inconsistent at the crossover point between low nodes and high nodes.

Consider a routing loop. There must be at least one pair of nodes whose hop count does not match between parent and child, and thus, a loop is detected. However, if the mismatch occurs between a low node and its high parent, then the loop is not detected. Nonetheless, in this case, the loop must contain another pair of nodes where one is a parent in a low state and the child is in a high state. This violates our loop-free properties\(^3\), and thus, a loop is also detected.

Given that a loop can be detected, what remains is breaking the loop, and ensuring that new loops are not created.

A loop may be broken by a node $u$ simply by setting its parent variable $u.\text{pr}$ to itself ($u.\text{pr} := u$). Node $u$ would then have to rejoin the routing-tree. This can be done by choosing as a new parent a node $t$ such that $t.\text{pr} \neq t$.

To prevent forming new loops, $u$ must not choose a descendant as a new parent. This can easily be achieved by $u$ choosing a new parent only if $u$ has no children (and hence, no descendants). A child $v$ of $u$ can observe that $u$ has no parent (i.e., $u.\text{pr} = u$), and $v$ can also set its parent to itself ($v.\text{pr} := v$). Thus, $u$ eventually has no children, and is free to choose any node as its parent, without causing a loop.

The problem with the above is that a race condition may occur, as shown in Fig. 2(a). The arrows in the figure denote the child-parent relationship.

Assume node $u$ detects that it is involved a loop, e.g., because its hop count does not match that of node $x$. Thus, $u$ breaks away from the loop, by setting its parent variable to itself. Then, $v$, after noticing the lack of a parent at $u$, also sets its parent variable to itself, as shown in Fig. 2(b). Since $u$ no longer has children, it may freely choose a new parent, in this case $x$, as shown in Fig. 2(c). Since $x$ is the new parent of $u$, $u$ would set its hop count to that of $x$ plus\(^3\)

\(^3\) Thus, the loop-free properties must restore themselves automatically, which we address in Section 6
Fig. 2. Race condition while breaking loops.

one, and thus be consistent with \( x \). This process can repeat itself when \( w \) is no longer a child of \( v \), and \( v \) chooses \( u \) as its new parent.

To avoid this, we introduce a new state for a node, \textit{dirty}, which implies that its tree is rooted at a node different from the distinguished node \textit{root}. When a node \( u \) breaks away from a loop, as in Fig. 2(i), it sets its state to \textit{dirty} (but leaves its hop count intact). This causes all of its descendants to change to the \textit{dirty} state, as shown in Fig. 2(ii). Nodes without children can break away from their parent, by setting their parent variable to themselves. This causes the entire tree to be dismantled, bottom up. Nodes without children can then choose a non-\textit{dirty} node as their new parent.

One final observation remains: when all nodes in a loop are \textit{dirty}, some node must be the first to break away from its parent. This is also implemented with a hop count: a \textit{dirty} node’s hop count must match that of its parent. Thus, in a loop, at least one node has an incorrect hop count, causing the node to break away from its parent, and break the loop.

We may now present the specification of the Stabilizing Loop-Free Adaptive Routing (SLFAR) protocol. Node \textit{root} remains as in the LFAR protocol. The variables of each node \( u \) only have two small changes: the state variable allows the additional value \textit{dirty}, and the parent variable allows \( u \) itself as a value.

The local loop-freedom properties defined in Lemma 1 are modified as follows. For every node \( u \), and for every neighbor \( g \) of \( u \),

\[
\begin{align*}
  u.\text{high} & \Rightarrow (u.pr).\text{high} \tag{5} \\
  u.\text{high} \land u.\text{req} & \Rightarrow g.pr \neq u \lor g \notin u.end \lor g.\text{low} \tag{6} \\
  (u.pr).\text{low} & \Rightarrow u.\text{low} \tag{7} \\
  u.\text{dirty} & \Rightarrow (u.pr).\text{dirty} \tag{8} \\
  u.\text{dirty} & \Rightarrow g.pr \neq u \lor g \notin u.end \tag{9} \\
  u.pr \neq u \land \neg(u.\text{low} \land (u.pr).\text{high}) & \Rightarrow u.hc = (u.pr).hc + 1 \tag{10}
\end{align*}
\]
The first three properties are as before. The last three properties deal with the propagation of dirty states and hop-counts. Property (8) requires the parent of a dirty node to be dirty. Property (9) requires $u.end$ to contain only those neighbors who are no longer children of $u$. Finally, property (10) requires a consistent hop-count between parent and child.

We define $\text{good-neighbor}(u, g)$ to be the conjunction of Properties (6) and (9). We define $\text{good-parent}(u)$ to be the conjunction of the remaining four properties.

The actions of node $u$ in the SLFAR protocol are a super-set of the actions in the LFAR protocol. The additional actions are as follows.

\[
\begin{align*}
\{\text{sanity}\} & \quad u.pr = u \land \neg u.d \quad \rightarrow \quad u.st := \text{dirty} \\
\{\text{break if mismatched parent}\} & \quad \neg \text{good-parent}(u) \quad \rightarrow \quad u.pr := u; \quad u.st := \text{dirty}; \quad u.end := \emptyset \\
\{\text{break if mismatched neighbor}\} & \quad \neg \text{good-neighbor}(u, g) \quad \rightarrow \quad u.pr := u; \quad u.st := \text{dirty}; \quad u.end := \emptyset \\
\{\text{propagate dirty}\} & \quad \neg u.d \land (u.pr).d \quad \rightarrow \quad u.st := \text{dirty}; \quad u.end := \emptyset \\
\{\text{add neighbor to end set}\} & \quad g.pr \neq u \quad \rightarrow \quad u.end := u.end \cup \{g\} \\
\{\text{if no children, break away from parent}\} & \quad u.d \land u.end = u.N \land u.pr \neq u \quad \rightarrow \quad u.pr := u \\
\{\text{rejoin the tree if childless }\} & \quad u.d \land u.end = u.N \land \neg g.d \quad \rightarrow \quad u.st := \text{low}; \quad u.hc := g.hc + 1
\end{align*}
\]

The first action is a sanity action; node $u$ should be dirty if it has no parent. The second and third actions break away from $u$’s parent if the parent’s values or the values of a neighbor $g$ are inconsistent with those of $u$. We require two actions instead of one because we assume each action can only read the variables of a single neighboring node.
The fourth action propagates the dirty state if the parent is dirty. Also, it sets \( u.end \) to the empty set, which will keep track of which neighbors are no longer a child of \( u \).

In the fifth action, a neighbor \( g \) is added to \( u.end \) if it is no longer a child of \( u \). In the sixth action, if all neighbors are in \( u.end \), i.e., if there are no more children, and if \( u \) is dirty, it breaks away from its parent.

Finally, in the seventh action, the dirty node rejoins the routing-tree by choosing a parent that is not dirty. The new state of \( u \) is low, since this is a safe state that will not violate any relationship with its parent. Furthermore, the hop-count is set to that of its parent plus one.

6 Convergence Properties

We next present an overview of the correctness proof of the SLFAR protocol. The interested reader can find the details in [16].

Our execution model in Section 3 relies on a weak form of fairness; we only required that should an action of a process remain continually enabled, then eventually it is chosen for execution. Furthermore, within each process, we placed no restrictions on the order of actions selected for execution. In order to demonstrate convergence complexity of the SLFAR protocol, we assume an execution model similar to that in [15].

We assume an arbitrary but fixed ordering on the actions of a process, and we require that actions of a process be attempted in this order. Thus, within \( O(|u.N|) \) rounds, every action of node \( u \) will be attempted at least once.

First, we consider the convergence time of the local loop-freedom properties of SLFAR.

**Lemma 2. (stabilizing loop-freedom properties)**

Let \( P_2 \) be the following predicate: for every pair of nodes \( u \) and \( g \),

\[
good-neighbor(u, g) \land good-parent(u)
\]

Within \( O(deg) \) rounds, where \( deg \) is the degree of the network, the SLFAR protocol stabilizes to \( P_2 \).

Note that, if \( good-neighbor(u, g) \land good-parent(u) \) holds, then the second and third actions in the SLFAR protocol are disabled. Hence, from this point onwards, no node breaks away from its parent due to an inconsistent state with its parent. This prevents the race condition that is pictured in Figure 2. This implies that our network forms a forest of trees. Only one of these trees is rooted at the designated node \( root \).

Next, from \( P_2 \), we have that if \( u.dirty \) is true, then all the nodes along the path from \( u \) to its “root”, i.e., to the nodeo whose parent is itself, are all dirty. Furthermore, the dirty state will propagate down to the leaves of \( u \)’s tree. Afterwards, any node who breaks away will do so only if it has no children. The network will then become a forest, where all roots other than the distinguished node \( root \) will consist of a single childless node.
Lemma 3. (single tree) Let $P_3$ be the following predicate:

$$(\forall u : u = (u.pr) \land u \neq \text{root} : \forall v : v \neq u : v.pr \neq u)$$

Then, within $O(L \cdot \deg)$ rounds, the SLFAR protocol stabilizes to $P_2 \land P_3$.

Note that, once a dirty node is childless, nothing stops it from joining the routing tree (rooted at $\text{root}$), provided it can find a neighbor that is on the routing tree, i.e., that is not dirty. Hence, all dirty nodes quickly rejoin the routing tree, and the routing tree becomes and remains a spanning tree.

Theorem 2. (restore routing tree) Let $P_4$ be the following predicate.

$$\left(\bigcup u, v : u.pr = v \land u \neq \text{root} : (u,v)\right)$$

forms a spanning tree rooted at $\text{root}$.

Then, within $O(L \cdot \deg)$ rounds, the SLFAR protocol stabilizes to $P_2 \land P_3 \land P_4$.

Finally, once a spanning tree is obtained, the spanning tree must change its structure to match the optimal routing tree. That is, once the base protocol reaches a fixed point (assuming no future edge-weight changes), each node in the network must make its parent equal to the parent in the base protocol.

Consider a node $u$ where $u.pr \neq u.\widehat{pr}$. Since the base protocol is at a fixed point, $u.\widehat{pr}$ remains constant. Let $T$ be the routing tree in the SLFAR protocol, and $\widehat{T}$ be the optimal tree of the base routing protocol. Let $w$ be the node closest to $\text{root}$ from $u$ to $\text{root}$ along $T$, such that $w.pr \neq w.\widehat{w}$. Then, $w$ will perform a request for all its descendants in $T$ be change their state to low (including $u$). The state of all these nodes will remain low until their parent in both trees are equal.

Consider again node $u$, and its path $u, x_1, x_2, \ldots, x_j\text{root}$ along the optimal tree $\widehat{T}$. Any $x_i$, $1 \leq i \leq j$, by the same the same argument as for $u$ above, will become low, and will remain low until its parents in both trees are the same.

Consider node $x_j$. Its neighbor is $\text{root}$, and the state of $x_j$ will be low. Thus, nothing prevents $x_j$ from choosing $\text{root}$ as its new parent, and changing to a high state. A similar argument can be made for $x_{j-1}$, down to $x_1$ and $u$. Hence, we have the following.

Theorem 3. (optimal tree) Let $P_5$ be the following predicate.

$$(\forall u : u \neq \text{root} : u.pr = u.\widehat{pr})$$

Within $O(L \cdot \deg)$ rounds after the base protocol reaches a fixed point, the SLFAR protocol stabilizes to $P_2 \land P_3 \land P_4 \land P_5$.

Corollary 1. If the base protocol is the routing protocol presented in [9], then, the SLFAR protocol stabilizes to $P_2 \land P_3 \land P_4 \land P_5$ within $O(L \cdot \deg)$ rounds.
7 Concluding Remarks

We have presented a stabilizing protocol for routing with maximizable metrics that converges in $O(L \cdot \text{deg})$ rounds, and nodes are unaware of the value of $L$. In addition, it is route-preserving, and thus ensures all data messages are delivered even in the presence of continuous edge weight changes. The $\text{deg}$ factor is needed due to each node having to check the status of each of its neighbors. Many networks have a constant degree, irrespective of the number of nodes, because nodes are neighbors if they are physically close to each other. If $\text{deg}$ is assumed to be constant, then the protocol converges in $O(L)$ rounds.

We have assumed a shared memory model for simplicity. Our level of atomicity is low; each action can read from only a single neighboring node. Thus, a message-passing version of the protocol should be relatively straightforward, and is postponed for future work.

Our method to perform a diffusing computation to mark all inconsistent nodes “dirty” is similar to earlier work on loop-free routing protocols, [17][18][19]. However, these protocols make the assumption that nodes are fail-safe, and that there exists a perfect channel in-between neighboring nodes.

Our protocol is a routing protocol, and thus it constructs a spanning tree of the network. Thus, it shares some features with leader election protocols that can construct a spanning tree without using knowledge of the network size [3][4]. However, the protocols in [3][4] are specialized for leader election, and transforming their approach into a routing protocol for maximizable metrics (especially if route-preserving) is not straightforward.

References