

Fault-Tolerant Metric Vector Routing

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

Abstract

We present a distributed routing protocol for obtaining the best path between any pair of nodes in a computer network. The metric to determine the best path between the two nodes is an n -dimensional vector. Each element of the vector is a simple metric, such as hop-count, link cost, link bandwidth, etc.. Given that no path may provide the best value for each of the simple metrics in the vector, the metrics are given distinct priorities. Ties between two paths with equal high priority metrics are broken in favor of the path with the best lower priority metrics. The protocol has the nice property that the routing tables remain loop-less at all times. In addition, the protocol is very robust. In particular, it is self-stabilizing. That is, the protocol tolerates any transient fault.

1. Introduction

The routing problem consists of finding a path from any node in the computer network to any other node in the network. Nodes in the network have a routing table which, for each destination node in the network, contains the identity of the neighbor corresponding to the next-hop to the destination.

If an algorithm exists to obtain a path to a single destination, the algorithm may be replicated to obtain a path to any destination. Thus, when presenting a distributed routing protocol, it is common to only consider a single destination. We take this approach in this paper.

Let the destination be called *root*. If we consider all network edges (p, q) , where q is the next hop neighbor of p to reach the root, we obtain a spanning tree of the network. Thus, the routing problem may be reduced to finding a spanning tree rooted at the destination. This tree is known as the *routing tree*.

The path from each node p to the root along the routing tree must satisfy some metric criteria. For example, the path along the routing tree from p to the root could be the shortest path between p and the root.

Most routing algorithms deal with either one or two metrics. For example, a single metric could be the total hop count of the path. A double metric could be finding the path whose bottleneck bandwidth is the greatest, and among those with the greatest bottleneck bandwidth, choose the path with the smallest hop count [e.g. 8, 12].

In this paper, we present a routing algorithm where the metric is an n -dimensional vector V . Each element of a the

vector is a simple metric, such as hop count, bottleneck bandwidth, etc.. The metrics are arranged in order of priority; that is, metric $V[1]$ is more important than metric $V[2]$. In the previous example, the bottleneck bandwidth would be $V[1]$ and the hop count would be $V[2]$.

The need for a metric vector arises from the complex policy and quality of service routing protocols of the future. For example, one may envision a metric with three components. The first is security; the path should be as secure as possible to avoid theft of information. Second, from the most secure paths, choose the path with greatest bandwidth. Third, from these paths, choose the one with the smallest hop count.

We also address the issue of loop-less routing. Loop-less routing protocols for a single metric has been studied in the past [3] [9] [11]. In this paper, we present a loop-less routing protocol for metric vector routing.

Finally, the protocol we present is also self-stabilizing. A system is *self-stabilizing* [6] [10] iff, when started from an arbitrary initial state, it eventually converges to a desired operating state. Thus, regardless of the initial value of any variable in the system, the system converges to its desired operating state. This allows the system to recover from memory read errors, undetected corrupted messages, etc.. Hence, a self-stabilizing system is extremely robust.

Due to the desirability of self-stabilization, the routing algorithm we develop in this paper is also self-stabilizing. During its normal execution, its routing tables will remain loop-less, and a spanning routing tree is always maintained. However, if a fault occurs, causing a node from being disconnected from the tree or introducing a loop, then the protocol will converge and reconstruct the routing spanning tree, and remain loop-less thereafter.

To our knowledge, the only other study of a self-stabilizing loop-less routing protocol is [7] [8]. In this protocol, the metric is the pair of bottleneck bandwidth and hop-count. The protocol requires a global termination detection algorithm, and only the root may propagate metric updates through the network. The protocol we present may use a metric vector of any dimension, requires no global termination detection algorithm, and any node may propagate metric updates at any time.

To simplify the presentation, we assume each computer can read the variables of its neighboring computers. We will relax this assumption to a message passing model in the full paper. Due to space restrictions, the correctness proof is also deferred to the full paper.

Throughout, we use quantifications of the form:

$$(\otimes x : R(x) : B(x))$$

Above, \otimes is any commutative and associative operator, such as +, -, max, min, \forall (conjunction), or \exists (disjunction). $R(x)$ is a Boolean function defining the range of values for the dummy variable x , and $B(x)$ is a function defining the value given as an operand to \otimes . For example,

$$(\min x : 1 \leq x \leq 3 : x^2)$$

denotes the minimum of 1^2 , 2^2 , and 3^2 .

2. Metric Vector Routing

The metric of a path from a node of the routing tree to the root of the tree is an integer value expressing the desirability of the path. For example, if the metric of a path is its number of hops, a path with smaller number of hops has a better metric (is more desirable) than a path with a greater number of hops. If the metric of the path is its bottleneck bandwidth (i.e., the minimum of the bandwidths of the links along the path), then a path with greater bottleneck bandwidth is more desirable than a path with smaller bottleneck bandwidth.

We assume there exists a relation, \prec , on the values of a metric. Also, let $p.m$ denote the metric of the path from node p to the root node. If $p.m \prec q.m$, then the metric of p 's path to the root is more desirable than the metric of q 's path to the root.

Relation \prec must satisfy the following requirements.

- $(x \prec y \wedge y \prec z) \Rightarrow (x \prec z)$ for all x, y , and z .
- $\neg(x \prec x)$ for all x .
- $\neg(x \prec y \wedge y \prec x)$ for all x and y .
- $(x \prec y \vee y \prec x)$ for all x and y , $x \neq y$.
- $\perp \prec x$ for every x , $x \neq \perp$.

We define relation \preceq as follows: $x \preceq y$ iff $(x \prec y \vee x = y)$. Relation \preceq is total.

For example, if the metric is hop count, then \prec is the "less than" relation on integers, and \perp is zero. On the other hand, if the metric is bottleneck bandwidth, then \prec is the "greater than" relation on integers, and \perp is the largest bandwidth of any link in the network.

Let p be the parent of q on the routing tree. Let $p.m$ and $q.m$ be the metrics of these two nodes. We say that metric m is *increasing* if it must be the case that $p.m \prec q.m$. Metric m is *non-decreasing* if it must be the case that $p.m \preceq q.m$. For example, hop count is an increasing metric, since $p.m + 1 = q.m$. However, bottleneck bandwidth is a non-decreasing metric, since $p.m = q.m$ is possible.

A *metric vector* V of dimension n is a vector with n metric values, $V[1], V[2], \dots, V[n]$. Each one of the n metrics of V may be either an increasing or non-decreasing metric. The sub-vector $V[1], \dots, V[i]$ is denoted by $V|_i$. The metrics of the vector are arranged in order of priority. Thus, $V[i]$ has higher priority than $V[i+1]$, $1 \leq i < n$.

With a slight abuse of notation, we extend \prec to metric vectors V and W of the same dimension. We assume that the type of $V[i]$ is compatible with the type of $W[i]$, for all i , $1 \leq i \leq n$. We say that $V \prec W$ iff there exists an i , $1 \leq i \leq n$, such that $V[i] \prec W[i]$, and $V|(i-1) = W|(i-1)$.

Thus, for the first $i-1$ metrics, V and W are identical, but for metric i , V is a better choice than W . Therefore, overall, V is a better choice than W . It can be shown that if \prec satisfies the five requirements mentioned earlier for a single metric, then \prec also satisfies the extension of these requirements to metric vectors. Also, the following observation on the relation \prec is straightforward to prove.

Observation 1

- a) If $V|_j \prec W|_j$, then $V|_k \prec W|_k$, for $1 \leq j < k \leq n$.
- b) If $V|_j \preceq W|_j$, then $V|_k \preceq W|_k$, for $1 \leq k < j \leq n$.

♦

Let q be the parent of process p in the routing tree. Associated with the link between p and q is a metric vector $(p, q).V$. The metric vector of p , $p.V$, is defined as follows.

$$\langle \forall i : 1 \leq i \leq n : p.V[i] = (q.V[i] +_i (p, q).V[i]) \rangle (1)$$

Above, $+_i$ is the binary operator that combines the i th metric of the link and the i th metric of the parent to obtain the i th metric of the child. E.g., if the i th metric is hop count, then $(p, q).V[i] = 1$, and $+_i$ is integer addition. If the i th metric is bottleneck bandwidth, then $(p, q).V[i]$ is the bandwidth of link (p, q) , and $+_i$ is the minimum operator.

Below, we use the function **vupd** (vector update) to obtain the metric vector of a child given the metric vector of the parent and the metric vector of the link, as done in (1) above. Hence, the following assignment is performed when the metric vector of p needs to be updated.

$$p.V := \mathbf{vupd}(q.V, (p, q).V)$$

Let $P = p_0, p_1, \dots, p_{(k-1)}$ be a directed path of k nodes from p_0 to $p_{(k-1)}$, where $p_{(k-1)} = \text{root}$. Let $(p_i, p_{(i+1)}) . V$ be the metric vector of link $(p_i, p_{(i+1)})$. The metric vector $P.V$ of path P is defined recursively as follows.

$$\text{root}.V = \perp$$

$$P.V = p_0.V$$

$$p_i.V = \mathbf{vupd}(p_{(i+1)}.V, (p_i, p_{(i+1)}) . V), \text{ for } 0 \leq i < k-1$$

Above, \perp is the n -dimensional vector of \perp values.

The routing problem may be considered as finding a minimum metric tree. A path from p to the root is a *minimum metric path* iff there is no path from p to the root whose metric is better than that of this path. A *minimum metric tree* is a directed spanning tree rooted at the root node, such that, for any node p in the tree, the path from p to the root along the tree is a minimum metric path.

3. Protocol Notation

We next present the notation that we will use to describe our routing protocols. The network model consists of a set of processes, which read but do not write the variables of their neighboring processes.

Each process is defined by a set of local constants, a set of local inputs, a set of local variables, and a set of actions. We denote with $p.v$ the local variable v of process p . We omit the prefix if the process is clear from the context.

The actions of a process are separated by the symbol $[]$, using the following syntax:

begin *action* [] *action* [] . . . [] *action* **end**

Each action is of the form *guard* \rightarrow *command*. A guard is a Boolean expression. A command is a sequence of executable statements. Similar notations for defining network protocols are discussed in [4] and [5].

An action in a process is *enabled* if its guard evaluates to **true** in the current state of the network. An *execution step* of a protocol consists of choosing any enabled action from any process, and executing the action's command. Executions are maximal and fair.

If multiple actions in a process differ by a single value, we abbreviate them into a single action with parameters. For example, let j be a parameter whose type is the range $0 \dots 1$. The action

$$x[j] < 0 \rightarrow y[j] := \mathbf{false}$$

is a shorthand notation for the following two actions.

$$x[0] < 0 \rightarrow y[0] := \mathbf{false} \quad [] \quad x[1] < 0 \rightarrow y[1] := \mathbf{false}$$

4. The Basic Protocol

We next present a basic protocol to obtain a minimum metric tree. The protocol introduces temporary loops, but it eventually converges, and a minimum metric tree is obtained. This protocol serves as a stepping stone for the loop-less protocols that we present in the next sections.

To represent the edges in the tree, each process has a variable pt indicating its parent in the tree. Also, each process has a local constant N indicating its set of neighbor processes. Thus, pt is always chosen from the set N .

For each neighbor q of p , p contains an input $(p, q).V$, which is the metric vector of the link from p to q . Furthermore, each process stores in variable V its best estimate of the metric vector of the path to the root.

To build the spanning tree, we use an algorithm similar to the Bellman-Ford algorithm, but tailored towards metric vectors rather than a single metric. It consists of two steps. First, each process periodically compares its metric vector V with that of its parent, $pt.V$, and updates V if necessary. This is necessary because, at any time, the metric of the path to the root via the parent may change.

The second step consists of choosing a new parent, if possible. For each neighbor q of p , p reads the current values of $q.V$ and $(p, q).V$, and chooses q as its new parent iff the metric of the path to the root via q is better than the metric of the path to the root via its current parent. That is, p chooses q as its new parent iff

$$\mathbf{vupd}(q.V, (p, q).V) \prec p.V$$

The basic protocol for a non-root process is given below. References to local variables have no prefix, while references to neighbor's variables have the neighbor's identifier as a prefix.

process p

constants

N : { x | x is a neighbor of p }

n : **integer**

parameters

q : **element of** N (* q is any neighbor of p *)

inputs

$(p, q).V$: **array** [$1 \dots n$] **of integer**

variables

pt : **element of** N

V : **array** [$1 \dots n$] **of integer**

begin

vupd $(pt.V, (p, pt).V) \neq V \rightarrow$
 $V := \mathbf{vupd}(pt.V, (p, pt).V)$

[] **vupd** $(q.V, (p, q).V) \prec V \rightarrow$
 $pt := q;$
 $V := \mathbf{vupd}(pt.V, (p, pt).V)$

end

The process contains two actions. In the first action, the process checks if its estimate, V , of the metric of the path to the root is an accordance with the metric of the parent and the metric of the link to the parent. If they do not match, the estimate V is updated.

In the second action, a neighbor q is checked to see if it offers a path to the root with a better metric than that of the current path to the root. If this is true, q is chosen as the parent, and the metric vector V is updated accordingly.

The root process may be specified as follows.

process $root$

constants

n : **integer**

variable

pt : **integer**

V : **array** [$1 \dots n$] **of integer**

begin

$V \neq \perp \vee pt \neq root \rightarrow$
 $pt := root;$
 $V := \perp$

end

There is only a single action. Since the root has no parent, we set pt to the root itself. Also, its metric vector should be set to a vector of bottom values, since no process can have a metric better than the root's metric.

It can be shown that if the first metric of the metric vector is an increasing metric, then the protocol is correct and converges to a minimum metric tree. However, if the first metric is a non-decreasing metric, then the parent variables may initially form a loop, and the minimum metric tree is never obtained.¹

5. The Propagated Metric Protocol

Is easy to show that, as the metrics of edges change, the protocol above suffers from temporary routing loops. That is, a process p will choose one of its descendants r as its

¹For a non-decreasing initial metric, if at least one metric in the vector metric is increasing, and if an upper bound on this metric is known for any path, a correction is possible. It consists of choosing a different parent if the increasing metric reaches the upper bound.

However, we will see in a later section that this trick is inappropriate in the later loop-avoiding protocol. Since the trick is not suitable for this later case, we chose not to include it in our basic protocol.

parent, forming a loop. This occurs because the propagation of routing updates from p to its descendants is not instantaneous, and p believes that its descendant r offers a better path to the root than p's current path.

To avoid routing loops, we use a technique similar to the one originally proposed in [9]. Variations of this technique have been used to prevent loops in leader election protocols [0] and minimum cost routing protocols [3] [11].

Consider the case of a single metric ($n = 1$). Each process has a Boolean bit, called *clean*, which if true implies that all descendants of p in the tree have a metric worse than p's metric. We refer to this requirement as the *clean bit invariant*.

If p's *clean* bit is true, then p may choose any neighbor as its parent, provided p's metric improves. This is because all descendants of p have metrics worse than p's, and hence, they will not be chosen as p's parent, avoiding the formation of loops.

What remains to be decided is when to set the *clean* bit to true or false. When p updates its metric from its parent's metric, and its new metric is worse, *clean* is set to false. When p notices that all its children have metrics worse than its own, and all its children have *clean* equal to true, then p may set its *clean* bit to true. It is easy to show by induction that these actions preserve the *clean bit invariant*.

We next expand the above loop-avoidance technique to metric vectors. One way to do so is simply the following: p's *clean* bit becomes false when p.V becomes worse, and p's *clean* bit becomes true when all of its children are *clean*, and their metric vector is no better than p.V.

However, this is too restrictive, and should be relaxed. For example, assume V has two metrics, and p is *clean*. If p updates p.V becoming worse in the second metric, and the first metric remains equal, then p sets p.*clean* to false, and cannot change parents until it becomes *clean* again. However, a neighbor may offer p a path whose first metric is better than p's first metric. In this case, p should be able to change parents, but it cannot since p.*clean* is false.

Thus, we choose instead to relax when process p can change parents as follows. Let p.*clean* be an integer variable in the range $0 \dots n$. If p.*clean* = 0, then process p can make no assumptions about the metrics of its descendants. If p.*clean* > 0, then for each descendant r of p,

$$p.V \upharpoonright_{p.\text{clean}} \preceq r.V \upharpoonright_{p.\text{clean}} \quad (2)$$

We refer to (2) as the *clean vector invariant*.

Furthermore, p can choose neighbor q as its parent if

$$\mathbf{vupd}(q.V, (p, q).V) \upharpoonright_{p.\text{clean}} \prec p.V \upharpoonright_{p.\text{clean}}$$

No loops are created because, if the above holds, then

$$q.V \upharpoonright_{p.\text{clean}} \prec p.V \upharpoonright_{p.\text{clean}}$$

and from (2), we have

$$p.V \upharpoonright_{p.\text{clean}} \preceq r.V \upharpoonright_{p.\text{clean}}$$

and thus $q \neq r$.

What remains is choosing how to change the value of p.*clean* such that the *clean vector invariant* is preserved.

Let p.V' be the new value of p.V after p.V is updated from the parent of p. Assume first that p.*clean* = 0 or

$p.V' \upharpoonright_{p.\text{clean}} = p.V \upharpoonright_{p.\text{clean}}$. In either case, p.*clean* should remain unchanged, since nothing new may be deduced about the metrics of p's descendants.

Assume next that p.*clean* > 0 and $p.V' \upharpoonright_{p.\text{clean}} \prec p.V \upharpoonright_{p.\text{clean}}$. In this case, since (2) holds, then from Observation 1(a), $p.V \prec r.V$ for any descendant r of p, and thus p.*clean* can be set to n.

Finally, assume p.*clean* > 0 and $p.V \upharpoonright_{p.\text{clean}} \prec p.V' \upharpoonright_{p.\text{clean}}$. In this case, p.*clean* needs to be set to the maximum i, $1 \leq i < p.\text{clean}$, such that $p.V \upharpoonright_i = p.V' \upharpoonright_i$. From Observation 1(b), this is the largest value of p.*clean* that preserves the *clean vector invariant*. If no such i exists, p.*clean* is set to zero.

We are now ready to present the specification of any non-root process p in our loop-less routing protocol. The root process remains the same as before.

process p

constants

N : { x | x is a neighbor of p }

n : **integer**

parameters

q : **element of** N (* q is any neighbor of p *)

j : 1 .. n

inputs

(p, q).V : **array** [1 .. n] **of integer**

variables

pt : **element of** N

V : **array** [1 .. n] **of integer**

clean : 0 .. n

begin

vupd(pt.V, (p, pt).V) \neq V \rightarrow

if $V \upharpoonright_{\text{clean}} \prec \mathbf{vupd}(pt.V, (p, pt).V) \upharpoonright_{\text{clean}}$
 $\wedge \text{clean} > 0 \rightarrow$
clean := **maxclean**

\square $\mathbf{vupd}(pt.V, (p, pt).V) \upharpoonright_{\text{clean}} \prec V \upharpoonright_{\text{clean}}$
 $\wedge \text{clean} > 0 \rightarrow$
clean := n

\square $\mathbf{vupd}(pt.V, (p, pt).V) \upharpoonright_{\text{clean}} = V \upharpoonright_{\text{clean}}$
 $\vee \text{clean} = 0 \rightarrow$
skip

fi;

V := **vupd**(pt.V, (p, pt).V)

\square

$\text{clean} > 0 \wedge \mathbf{vupd}(q.V, (p, q).V) \upharpoonright_{\text{clean}} \prec V \upharpoonright_{\text{clean}} \rightarrow$
pt := q;
V := **vupd**(q.V, (p, q).V);
clean := n

\square

$\text{clean} < j \wedge (\forall r : r \in N \wedge r.pt = p : j \leq r.\text{clean} \wedge$
 $V \upharpoonright_j \preceq r.V \upharpoonright_j) \rightarrow$
clean := j

end

Above, **maxclean** is the maximum i such that

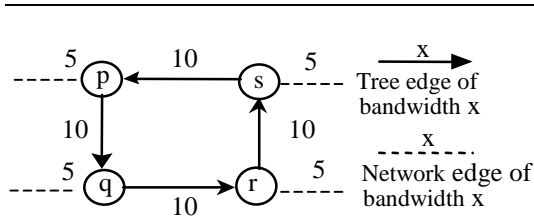


Figure 2: Routing loop

$$1 \leq i < \text{clean} \wedge \forall i = \mathbf{vupd}(\text{pt.V}, (\text{p}, \text{pt}).\text{V}) | i$$

or zero if no such i exists.

The process has three actions. The first action is similar to the first action in the previous section. However, the value of p.clean is updated as explained earlier.

In the second action, a new parent is chosen, but only if it will improve the metric vector of p , and if the clean variable is large enough to ensure the clean vector invariant is preserved.

In the last action, the process increases its clean value provided all its children have at least the same clean value, and a metric no better than p 's metric.

6. The Propagated Timestamp Protocol

The propagated metric protocol above is very sensitive to the initial state of the system. For example, consider Figure 2, where there is a single non-decreasing metric, namely, the bottleneck bandwidth of the path to the root.

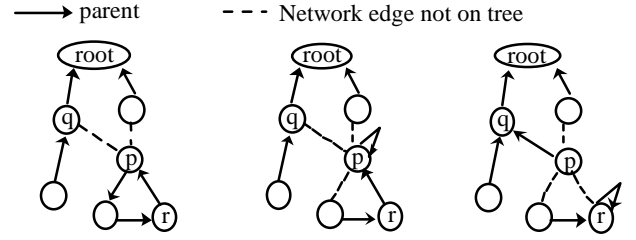
The arrows indicate the parent relationships, e.g., p has chosen q as its parent. The numbers indicate the bandwidth of each edge. If $V = 10$ for each process in the loop, the loop will not be broken, because all edges not included in the loop have a lower bandwidth than the loop edges.

Our goal is to present a protocol that will reach a normal operating state (loop-less routing tables) even though it started from an arbitrary initial state. To do so, initial loops like the one in Figure 2 must be broken.

One is tempted to attempt to break such loops by ensuring that the metric vector contains at least one increasing metric, such as a hop count. When the hop count reaches a value that exceeds the network diameter, then the process abandons its current parent and chooses another neighbor as its parent.

However, this simple technique does not always work, depending on the metric vector. For example, if the first metric is bottleneck bandwidth, and the second metric is hop count, we can obtain a scenario where the hop count estimate at some nodes grows beyond bound during *normal* execution of the protocol. Therefore, the protocol cannot distinguish between an unbounded hop count due to a loop, and an unbounded hop count due to normal execution. Hence, a more powerful technique is needed.

We break loops using a novel timestamp propagating technique. Using timestamps to avoid the formation of loops in network protocols has been done in the past [1] [2]. However, our method of propagating timestamps is novel because it breaks existing loops, and in addition, it does not interfere with the normal behavior of the protocol



a) $p.ts = 8, q.ts = 10$ b) $p.ts = 10, q.ts = 10$ c)

Figure 3: Breaking loops with propagated timestamps

when loops are not present. This timestamp propagation will allow our routing protocol to be self-stabilizing.

The strategy is as follows. The root process has a timestamp variable, root.ts . Periodically, the root increases its timestamp. When a child p of the root notices that the timestamp of the root, root.ts , is greater than its own timestamp, $p.ts$, p assigns root.ts to $p.ts$. Similarly, a child q of p will assign $p.ts$ to $q.ts$ provided $q.ts > p.ts$, etc.. Thus, the root timestamp propagates down the tree.

We would like to bound the difference between the timestamp of the root and that of its descendants to at most one. To do so, each process p maintains a Boolean variable, p.tclean , with the following meaning.

$$\text{p.tclean} \Rightarrow (\forall q : q \text{ descendant of } p : q.ts \geq p.ts) \quad (3)$$

We say that p is *timestamp clean* iff p.tclean equals **true**, otherwise p is *timestamp dirty*. If p is timestamp clean, it implies that all its descendants have a timestamp at least the timestamp of p . To guarantee Relation (3), p makes itself timestamp dirty when it increases its timestamp by copying its parent's timestamp into its own. Process p makes itself timestamp clean when each child of q is timestamp clean and has a timestamp at least that of p .

If the root process does not increase its timestamp until it is timestamp clean, then its timestamp is always at most one greater than that of its descendants. I.e., timestamps propagate in "waves". When all processes have a timestamp equal to k , and the root is timestamp clean, the root increases its timestamp to $k+1$ and becomes timestamp dirty. Then, timestamp $k+1$ propagates down the tree until it reaches the leaves. The leaves then become timestamp clean (since they have no children), which in turn allows their parents to become timestamp clean, and so on, until the root becomes timestamp clean once again.

To detect loops, consider the following. Assume process p notices that it has a neighbor q , where $q.ts \geq p.ts + 2$. This implies that p is disconnected from the root, and is most likely involved in a loop. That is, p never received timestamp $p.ts + 1$ from its parent, and hence its parent does not lead to the root.

If this is the case for process p , then p must choose a different parent. The steps to do so are illustrated in Figure 3. Initially (Figure 3(a)), $q.ts \geq p.ts + 2$. Thus, p detects a loop, and breaks it by setting $p.pt = p$, (Figure 3(b)), and also sets $p.ts = q.ts$, because the timestamp of the root should be at least $q.ts$. Since p no longer has a parent, each child of p , e.g. r , also sets $r.pt = r$. Thus, eventually p is

childless. Then, p may choose any parent whose timestamp is at least p.ts (Figure 3(c)), rejoining the tree.

We next present the non-root process p. Here, p.clean, which indicates the clean level of metric vector p.V, is renamed p.vclean, to distinguish it from p.tclean.

process p

constants

N : { x | x is a neighbor of p }

n : integer

parameters

q : element of N (* q is any neighbor of p *)

j : 1 .. n

inputs

(p, q).V : array [1 .. n] of integer

variables

pt : element of N

V : array [1 .. n] of integer

vclean : 0 .. n-1 (* clean level of V *)

ts : integer (* timestamp *)

tclean : boolean (* timestamp clean bit *)

begin

vupd(pt.V, (p, pt).V) ≠ V →

if V |_{vclean} < **vupd**(pt.V, (p, pt).V) |_{vclean}
 \wedge vclean > 0 →

vclean := **maxclean**

□ **vupd**(pt.V, (p, pt).V) |_{vclean} < V |_{vclean}
 \wedge vclean ≠ 0 \wedge →

vclean := n

□ **vupd**(pt.V, (p, pt).V) |_{vclean} = V |_{vclean}
 \wedge vclean > 0 \vee →

skip

fi;

V := **vupd**(pt.V, (p, pt).V)

□ tclean \wedge ts = q.ts \wedge (q.pt ≠ q \vee q = root) \wedge vclean > 0
 \wedge **vupd**(q.V, (p, q).V) |_{vclean} < V |_{vclean} →
 pt := q; clean := n;
 V := **vupd**(pt.V, (p, pt).V)

□ ($\forall r : r \in N \wedge r.pt = p : j \leq r.vclean \wedge V |_j \preceq r.V |_j$)
 \wedge vclean < j →
 vclean := j

□ pt.ts > ts → tclean, ts := **false**, pt.ts

□ ($\forall r : r \in N \wedge r.pt = p : r.tclean \wedge r.ts \geq ts$) →
 tclean := **true**

□ (q.ts ≥ ts + 2 \vee (pt.pt = pt \wedge pt ≠ root)) \wedge pt ≠ p →
 pt, ts, tclean := p, **max**(ts, q.ts), **false**

□ ($\forall r : r \in N : r.pt \neq p$) \wedge pt = p \wedge (q.pt ≠ q \vee q = root)
 \wedge q.ts ≥ ts →
 pt, vclean, tclean, ts := q, n, **true**, q.ts;
 V := **vupd**(pt.V, (p, pt).V)

end

Process p contains seven actions. As before, the first action updates V and p.vclean from pt.V and (p, pt).V.

The second action changes parents. We have strengthened the guard to ensure that p is timestamp clean, that the new parent has a timestamp equal to p' s, and that the new parent is not disconnected from the tree.

The third action is the same as in the propagated metric protocol. The fourth action updates the timestamp to that of the parent, and makes p timestamp dirty. The fifth action makes p timestamp clean. The sixth action detects that a neighbor q has q.ts ≥ p.ts + 2, or that the parent of p has no parent. In this case, p could be in a loop, so it sets pt to p. The timestamp of p is increased to at least the timestamp of the neighbor.

In the last action, p rejoins the tree. If p has no parent nor children, and it finds a neighbor who has a parent and whose timestamp is at least p' s timestamp, then p chooses that neighbor as its new parent, it updates its timestamp and metric accordingly, and becomes timestamp clean and vector clean (i.e., vclean = n).

The specification root process is similar to before, except for the following additional action:

($\forall q : q \in N \wedge q.pt = \text{root} : q.tclean \wedge q.ts \geq ts$) → ts := ts + 1

This protocol can be shown to be self-stabilizing, to remain loop-less under a fault-free execution, and to achieve the minimum metric tree.

References

- [0] A. Arora and A. Singhai, "Fault-Tolerant Reconfiguration of Trees and Rings in Networks", *IEEE International Conference on Network Protocols*, 1994, page 221.
- [1] A. Arora, M. G. Gouda, and T. Herman, "Composite Routing Protocols", Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990.
- [2] J. Cobb, M. Gouda, "The Request-Reply Family of Group Routing Protocols", to appear in *ACM ToC*, 1998.
- [3] J. J. Garcia-Luna-Aceves, "Loop-Free Routing Using Difussing Computations", *IEEE/ACM Transactions on Networking*, Vol 1, No. 1., Feb. 1993, page 130.
- [4] M. Gouda, "Protocol Verification Made Simple", *Computer Networks and ISDN Systems*, Vol. 25, 1993.
- [5] M. Gouda, *The Elements of Network Protocols*, Wiley, 1998.
- [6] M. Gouda, "The Triumph and Tribulation of System Stabilization", *Int'l Workshop on Dist. Algorithms*, 1995.
- [7] M. Gouda and M. Schneider, "Stabilization of Maximum Flow Trees", *Proceedings of the third Annual Joint Conference on Information Sciences*, 1994.
- [8] M. Gouda and M. Schneider, "Maximum Flow Routing", *Second Workshop on Self-Stabilizing Systems*, 1996.
- [9] P. M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol", *IEEE Transactions on Communications*, Vol. COM-27, No. 9, 1979.
- [10] M. Schneider, "Self-Stabilization", *ACM Computing Surveys*, Vol. 25, No. 1, March 1993.
- [11] A. Segall, "Distributed Network Protocols", *IEEE Trans. on Inf. Theory*, Vol. IT-29, No. 1, pp. 23-35, Jan. 1983.
- [12] W. Zhao, S. Tripathi, "Routing Guaranteed Quality of Service Connections in Intergrated Service Packet Networks", *Proceedings of the 1997 IEEE Int'l Conf. on Network Protocols*.