# Stabilization of Loop-Free Redundant Routing

Jorge A. Cobb

Department of Computer Science, The University of Texas at Dallas, U.S.A
cobb@utdallas.edu

**Abstract.** Consider a network of processes that exchange messages via FIFO communication channels. Each process chooses a subset of its neighboring processes to be its successors. Furthermore, there is a distinguished process, called root, that may be reached from any other process by following the successor relation at each hop. Thus, under the successor relation, the processes are arranged as a directed acyclic graph that converges on the root process, i.e., a converging DAG (c-DAG). We present a network where each process may dynamically change its choice of successors, and during this change, the following two nice properties are satisfied. First, if the initial state of the network forms a c-DAG, then a c-DAG is preserved at all times. Second, if the protocol is started from an arbitrary state (i.e., where each variable has an arbitrary value), then a c-DAG is automatically restored.

## 1 Introduction

A network consists of a set of processes that exchange messages via FIFO communication channels. A common task in a network is the construction of a spanning tree. To build a spanning tree, each process chooses ones of its neighbors as its parent on the tree. The parent is also known as the successor of the process.

Spanning trees have multiple uses. Two of the most common are unicast and broadcast routing of data messages. In unicast routing [1, 2], a spanning tree is built with the destination as the root of the tree. When a process receives a message addressed to the destination, the message is forwarded to the parent on the tree. In broadcast routing [3, 4], when a process receives a broadcast message from a neighbor on the spanning tree, it forwards the message to all other neighbors that are also on this tree.

In both unicast and broadcast routing, the spanning tree is required to adapt to network conditions, such as congestion, and modify its structure. In doing so, temporary loops may be introduced, and processes may become disconnected from the tree. This is undesirable, since it reduces routing performance. Thus, loop-free spanning trees were developed [5–7]. These ensure that, even while the spanning tree is modifying its structure, no temporary loops are introduced, and no process is disconnected from the tree. Maintaining loop-freedom is of particular importance in ad-hoc networks, due to the frequent changes in network connectivity and low network bandwidth [8–10].

An alternative approach is to maintain *multiple* successors at each node. A single process, called, root, has no successors, and all processes lead to the root.

Thus, rather than maintaining a tree, a converging directed acyclic graph (c-DAG), is maintained, where all paths converge on the root process. This graph is used in unicast routing to provide multiple paths to the destination, i.e., to the root [11–13]. In broadcast routing, it provides alternative paths in the event of link failures.

All the works above assume a fail-safe model of fault-tolerance: if a process or channel fails, it simply stops functioning. This, however, does not cover some failures that are hard to detect. These include: transient hardware or software faults at lower layers, undetected corrupted messages, improper initialization of a node, or temporary disruptions from a network intruder. A broader fault-tolerant model that captures all of these transient faults is known as stabilization.

A network of processes is said to be *stabilizing* iff, starting from any arbitrary state (such as the state after an undetected fault), the network converges to a normal operating state within finite time. Stabilizing protocols are desirable due to their high degree of fault-tolerance [14]. They have the advantage of not requiring a global initialization, plus they tolerate all types of transient faults.

Multiple techniques to build loop-free and stabilizing spanning trees exist in the literature [15–18]. All of these techniques assume a shared memory model.

To our knowledge, only a single technique for constructing a loop-free and stabilizing c-DAG has been presented in the literature [19]. However, it suffers from the following drawbacks: a) a shared memory model is assumed, b) when a process chooses to change its successor set, this is restricted to occur only during a diffusing computation initiated by the root, and c) a temporary loop may be created in the event of a channel failure, even though the failed channel is not part of the c-DAG.

In this paper, we present a technique that solves the above problems. Processes exchange information via message passing, which is a more practical model than shared memory. A process is free to change its successor set without having to coordinate with the root process. Finally, loops are never introduced, even if channels fail.

We present our network of processes in three steps. First, we present processes that avoid loops at all times. However, the choice of successors for each process is limited. Then, we enhance our processes to have freedom in choosing their successors. Finally, we further enhance our processes to be stabilizing.

## 2 A Converging DAG of Processes

In this section, we present a general overview of the problem. We begin with some notation.

A *network* consists of a set of processes interconnected via communication channels. Two processes are *neighbors* if they are joined by a pair of channels. A *network path* is a sequence of processes where for each pair $(u, v)$ of consecutive processes in the path, $v$ is a neighbor of $u$.

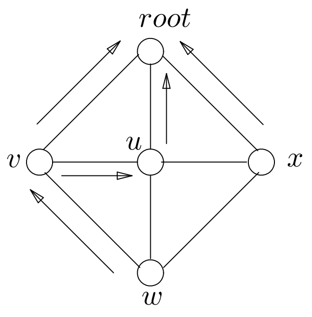

**Fig. 1.** Converging DAG of processes.

Each process $u$ maintains a variable, $u.S$, where it stores the identifiers of a subset of its neighboring processes. If process $v \in u.S$, then $v$ is said to be a *successor* of $u$ and $u$ is said to be a *predecessor* of $v$.

A path is *active* when, for each pair $(u, v)$ of consecutive processes in the path, $v$ is a successor of $u$. Process $v$ is *reachable* from a process $u$ when there is an active path from $u$ to $v$.

For example, consider Figure 1. In this figure, the neighbor relation is denoted by lines, and the successor relation is denoted by arrows. Thus, all processes are neighbors of $u$, $v$ has two successors, i.e., $v.S = \{u, root\}$, and $w$ has only one successor, $w.S = \{v\}$.

We require that all active paths be simple paths, i.e., loop-free. In consequence, the successor relation forms a directed acyclic graph.

We assume that there exists a distinguished process, which we call *root* (see Figure 1). In addition, we require that for every non-root process, there must exist an active path from the process to the root. This implies that the successor set of all non-root processes is non-empty. Also, the root process becomes a convergence point for the digraph, and hence, we refer to this structure as a converging DAG (c-DAG).

Contrary to earlier work [15, 19, 16–18], our processes do not choose which neighbors should be added to the successor set. We assume this is guided by a higher-layer application that chooses a particular structure. To capture the behavior of the application without imposing any restrictions, our processes simply choose non-deterministically whether or not to add a neighbor to the successor set. Our processes ensure that the requirements presented above are met at all times. Furthermore, if these requirements are not met initially, then the processes automatically converge to a state where the requirements are met.

We conclude this section with some path notation.

$$\begin{aligned}
|P| &: \text{number of processes in path } P \\
P_j &: j^{\text{th}} \text{ process in } P,\ 1 \le j \le |P| \\
L &: \text{maximum length of a simple path} \\
\mathit{active}(P) &: (\forall j : 1 < j \le |P| : P_j \in P_{j-1}.S) \\
\mathit{below}(u, v, x) &: (\exists P : \mathit{active}(P) \wedge |P| \le x : P_1 = u \wedge P_{|P|} = v)
\end{aligned}$$

## 3 Process Notation

Before presenting our processes, we first give a short overview of the notation that we use in specifying their behavior. This is similar to the notation introduced in [20]. Processes communicate with each other via the exchange of messages over FIFO channels. We use the following notation when referring to channels and messages.

| | |
|---|---|
| $Ch(u, v)$ | : channel from $u$ to $v$ |
| $m(u, v)$ | : message of type $m$ from $u$ to $v$ |
| $m(u, v).f$ | : value of field $f$ in message $m(u, v)$ |
| $neigh(u, v)$ | : function returning **true** iff both $Ch(u, v)$ and $Ch(v, u)$ exist. |

Without loss of generality, for every pair of distinct processes $u$ and $v$, either both $Ch(u, v)$ and $Ch(v, u)$ exist or neither of these two channels exist.

Each process is specified by a set of inputs, a set of variables, a parameter, and a set of actions. In general, a process is specified as shown below.

```
process <process name>
inp
  <input name>      : <type>,
      ...
var
  <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. To distinguish between variables of different processes, we prefix the variable names with the process name. For example, variable $u.r$ corresponds to variable $r$ in process $u$. If a variable does not have a process prefix, the process is understood from the context.

Every action in a process is of the form: <guard> $\rightarrow$ <statement>. The <guard> can be of three types: local, receiving, and timeout.

A local guard is a boolean expression over the inputs, variables, and parameter declared in the process. A receiving guard at process $u$ is of the form

**rcv** $m$ **from** $v$

where $v$ is a neighbor of $u$. Finally, a timeout guard is of the form

$$\textbf{timeout}\ m \notin Ch(u,v) \land m' \notin Ch(v,u)$$

where $v$ is a neighbor of $u$.

The <statement> is a sequence of message send statements or conditional statements. Conditional statements are of the following form.

<variable> := <expression> **if** <boolean expression>

If <boolean expression> is true before the conditional statement is executed, then <variable> is assigned the current value of <expression>.

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
   $g : 1 \,..\, 2$
```

then the following action

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

is a shorthand notation for the following two actions.

$$\begin{array}{l} \quad x = 1 \rightarrow x := x + 1 \\ \square \\ \quad x = 2 \rightarrow x := x + 2 \end{array}$$

An execution step of a protocol consists of choosing an action whose guard evaluates to true and executing the statement of this action. We assume all executions of a protocol are weakly fair, that is, an action whose guard is continuously true must be eventually executed.

We often refer to each element in an array variable $A$. With some abuse of notation, the expression $A = x$ is equivalent to $(\forall i :: A[i] = x)$. Similarly, the assignment statement $A := x$ assigns the value $x$ to each element of $A$.

## 4 Ranked Processes

In this section we present a network of simple processes that maintain a c-DAG. We assume such structure exists in the initial state. Thus, this network is not stabilizing.

Active paths are maintained loop-free as follows. Each process is assigned a rank value. We denote by $R$ the set of all possible rank values, and by $u.r$ the rank of process $u$. Whenever process $u$ adds a new successor to $u.S$, the new successor must have a rank greater than that of $u$. In consequence, for every pair of processes $u$ and $v$, where $v \in u.S$, $v.r$ is greater than $u.r$.

The reason all active paths are loop-free is simply as follows. Let $P$ be an active path with a loop, that is, $P_1 = u = P_{|P|}$. Then, because ranks increase from each process to its successor, this implies that $u.r$ is greater than $u.r$, which is not possible.

We next formalize process ranks and the relation on rank values. We are given a relation $\preceq$ on ranks. This relation satisfies the following properties.

**i. Transitive:**
For every $r, r'$, and $r''$,
$(r \preceq r' \wedge r' \preceq r'') \Rightarrow (r \preceq r'')$

**ii. Antisymmetric:**
For every $r$ and $r'$,
$r \preceq r' \wedge r' \preceq r \Rightarrow r = r'$

**iii. Bounded:**
There exists a value $\top$ (top) such that, for all $r$, $r \preceq \top$, and a value $\bot$ (bottom) such that, for all $r$, $\bot \preceq r$.

We denote the reflexive reduction of $\preceq$ as $\prec$.

The above general definition of rank allows for many possible choices of $R$ and $\preceq$. For example, $R$ could simply be the set of natural numbers, $\preceq$ be $\leq$, and $\prec$ be $<$. In addition, ranks could be based on the model of maximizable metrics introduced in [21, 22]

Ranks could be independent of the application that chooses the successor set. In this case, ranks would simply be used to prevent the application from violating the requirements on active paths. On the other hand, ranks could be intimately related to the application. E.g., assume the c-DAG is used for datagram routing in computer networks. Then, the successor set may be chosen to be those neighbors that provide the lowest metric to the root process. The metric could be as simple as the hop count to the root, or it could be a more complex metric, such as bottleneck bandwidth, queuing delay, or a combination of all of these. The rank in this case would simply be the metric used by the application.

We next present the processes in this network. The rank given to each process is fixed. However, we relax this requirement in the next section. We first show the specification of a non-root process $u$.

```
process $u$
inp
  $G$      : set of pid's,   {neighbor set}
  $r$      : element of $R$  {rank}
var
  $S$      : subset of $G$   {successor set}
par
  $g$      : element of $G$  {any neighbor}
begin
  true →            $upd.r := r$;
                    send $upd$ to $g$
```

```
[]
  rcv upd from g →  $S := S \bigcup \{g\}$    if $r \prec upd.r \wedge$ any
[]
  any $\wedge\ |S| > 1$ →     $S := S - \{g\}$
end
```

Each process periodically sends an *upd* (update) message to each of its neighbors. The *upd* message contains the rank of the process.

Each non-root process contains three actions. In the first action, process $u$ sends an update to neighbor $g$, and includes its rank in this update.

In the second action, process $u$ receives an *upd* message from neighbor $g$. If the rank of $g$ is greater than that of $u$, then $u$ adds $g$ to its successor set. We model the application's choice of adding $g$ to the successor set by including the operator **any** in the statement's condition. The operator **any** nondeterministically returns **true** or **false**.

In the third action, process $u$ removes a neighbor $g$ from its successor set. This, however, is done only if the successor set of $u$ does not become empty. Again, we represent the choice of removing $g$ from the successor set of $u$ by including the operator **any** in the guard of the action.

The specification of the root process is given below.

```
process root
inp
  G       :  set of pid's,    {neighbor set}
const
  S       :  $\emptyset$                 {successor set}
  r       :  $\top$                 {rank}
par
  g       :  element of G  {any neighbor}
begin
  true →                  upd.r := r;
                          send upd to g
[]
  rcv upd from g →  skip
end
```

The root process consists of two actions. In the first action, the root sends an update message to a neighbor. In the second action, the root receives an update message from a neighbor. Since the root is not allowed to have successors, it simply discards the message. Note that the successor set and the rank are constant values, which are the empty set and the top rank, respectively.

## 5 Dynamically-Ranked Processes

Having a fixed rank at each process restricts significantly the set of neighbors from which the process can choose successors. In consequence, the overall structure of the c-DAG is also restricted. To allow a dynamic structure, the rank of each process must also be dynamic. We address dynamic ranks in this section, and show how loops are avoided. Our technique has some similarities with earlier non-stabilizing loop-free protocols [5–7].

In the previous section, loops were avoided by ensuring the following two conditions.

1. The rank of every process is less than the rank of each of its successors.
2. When a process adds a new successor, the rank of the new successor is greater than that of the process.

However, these conditions are stronger than necessary, and are a consequence of processes having a fixed rank. To support dynamic ranks, we replace the above conditions with the following.

**Definition 1. (Loop-Avoidance Conditions)**

1. *When a process adds a new successor, the rank of the new successor is greater than the rank of the process.*
2. *When a process $u$ adds a new successor, all processes below $u$ must have a rank at most the rank of $u$.*
3. *A process cannot increase its rank to a value greater than the rank of any of its successors.*
4. *If the rank of a process is greater than that of any of its successors, then the process must reduce its rank to be at most the rank of all of its successors.*

□

Note that the above conditions allow a process to reduce its rank at any time and by any amount.

The first three conditions imply that a new successor cannot be below the process, and thus loops are avoided. That is, if process $u$ adds a new successor, the rank of the successor is greater than that of $u$, but at the same time all processes below $u$ have a rank at most that of $u$. Hence, the new successor cannot be below $u$. The fourth condition aids in the implementation of the second condition, as will be shown later in this section.

As an example, consider Figure 2(a). The structure is the same as that in Figure 1, and each process is labeled with its rank. The rank of each process is an integer, and $\prec$ is simply $<$.

Assume $u$ attempts to add $x$ to its successor set. Since the rank of $u$ is greater than that of $x$, from the perspective of $u$, $x$ may be below $u$. To determine if this is the case, $u$ decreases its rank to be less than the rank of $x$, as shown in Figure 2(b). This in turn causes all processes below $u$ to decrease their ranks, as shown in Figure 2(c). Once this operation completes, if the rank of $x$ is still greater than that of $u$, then $x$ is not below $u$, and $u$ can add $x$ to its successor set. This is shown in Figure 2(d).

We next consider each of the first three loop avoidance conditions. For each, we show how violating the condition may result in a loop.

Consider the first condition, and consider Figure 2(a). If $u$ adds a successor with lesser rank, namely $w$, then a loop is formed. Consider the second condition,

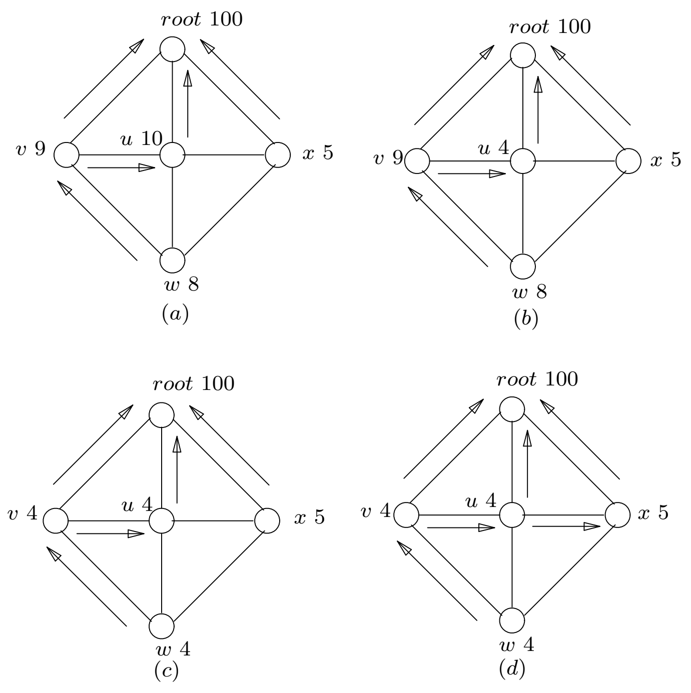

**Fig. 2.** Avoiding a loop while decreasing the rank.

and consider Figure 2(b). If $u$ adds a successor, again $w$, before the rank of $w$ has been decreased to be less than that of $u$, then a loop is formed, even though the rank of $w$ is greater than that of $u$. Finally, consider the third condition and Figure 2(c). If $w$ were to increase its rank to a value greater than the rank of $v$, its rank would be greater than the rank of $u$. This would allow $u$ to add $w$ to its successors and cause a loop.

We next address how to implement the conditions above. In particular, each process must lower its rank to be at most the rank of each of its successors. In addition, each process must be able to determine that each process below it has a rank no greater than its own. We consider each of these in turn.

As in the previous section, each process $u$ periodically sends an *upd* message to all its neighbors. The message contains the rank of the process. Process $u$ maintains two additional variables, $u.\widetilde{r}$ and $u.\widetilde{S}$. Variable $u.\widetilde{S}$ is a set containing those neighbors from whom $u$ has received an *upd* message. Variable $u.\widetilde{r}$ contains a lower bound on the ranks of the successors of $u$ from whom $u$ has received an *upd* message, i.e., from successors in $u.\widetilde{S}$. When $u$ has received an *upd* message from all successors, i.e., $u.S \subseteq u.\widetilde{S}$, $\widetilde{r}$ contains a lower bound on the rank of all successors. At this time, $u.\widetilde{r}$ is assigned to $u.r$. Furthermore, to prepare for another round of *upd* messages from each neighbor, $u.\widetilde{S}$ is set to the empty set, and $u.\widetilde{r}$ is assigned the top rank.

We next address how a process can determine that the ranks of all processes below it are at most its own rank. Each process maintains an array $D$ with the depth of rank ordering. That is, $D$ has an entry per neighbor, and the value of the entry is in the range $0 \mathinner{..} L$. Let $g$ be a predecessor of $u$. If $u.D[g] = i$, then all processes that are below both $u$ and $g$ up to $i$ hops below $u$ have a rank that is at most the rank of $u$.

More formally, we have the following rank ordering property.

**Definition 2. (Rank Ordering Property)**
*Consider any active path $P$, where: $t = P_1, g = P_{|P|-1}, u = P_{|P|}$, and $2 \leq |P| \leq u.D[g]$. Then, the following holds:*

$$t.r \preceq u.r \ \wedge\ (t.\widetilde{r} \preceq u.r \ \vee\ P_2 \notin t.\widetilde{S}) \ \wedge\ (\forall x : \mathit{neigh}(x,t) : \mathit{upd}(t,x).r \preceq u.r)$$

*In addition, if $u.D[g] = 1$, then $\mathit{upd}(u,g).r \preceq u.r$.* □

Note that when $u.D[g] = L$, the rank of all processes below $u$ is at most the rank of $u$, and $u$ is free to add a new successor.

Finally, consider how $D$ should be updated. When a neighbor $g$ receives an *upd* message from process $u$, $g$ returns an *ack* message to $u$. This *ack* message contains two values. The first value, $\mathit{ack}(g,u).r$, is the current rank of $g$. The second value, $\mathit{ack}(g,u).d$, contains the minimum of all the elements in array $D$ at $g$. This indicates to $u$ the depth at which processes below $g$ have a rank at most that of $g$. However, if $u$ is not a successor of $g$, then $\mathit{ack}(g,u).r = \bot$ and $\mathit{ack}(g,u).d = L$.

When process $u$ receives an *ack* message from neighbor $g$, it checks the rank of the message and its own rank. If the rank of $g$ is at most the rank of $u$, then the depth along $g$ is increased by one. That is, $u.D[g] := \mathit{max}(u.D[g], \mathit{ack}(g,u).d{+}1)$.

We have yet to address when the value of $u.D[g]$ is decreased. Note that as long as $u.r$ increases, then the value of $u.D[g]$ need not decrease, since the rank ordering property is not violated. However, if $u.r$ decreases, this property may no longer hold. Thus, whenever $u.r$ is decreased, $u.D[g]$ is assigned zero for all $g$.

We are now ready to present the specification of a network with dynamic rank. Below, we present the specification of a non-root process $u$.

```
process $u$
inp
   $G$        :  set of pid's           {neighbor set}
   $L$        :  integer                {max. path length}
var
   $S, \widetilde{S}$     :  subset of $G$          {successor set and its iteration set}
   $r, \widetilde{r}$     :  element of $R$         {rank and its iteration value}
   $D$        :  array [$G$] of 0 .. $L$   {rank depth}
par
   $g$        :  element of $G$         {any neighbor}
```

```
begin
  timeout $upd \notin Ch(u,g) \wedge ack \notin Ch(g,u)$ $\rightarrow$
                        $D[g] := max(1, D[g])$;
                        $upd.r := r$;
                        send $upd$ to $g$
[]
  rcv $upd$ from $g$ $\rightarrow$   $\widetilde{S} := \widetilde{S} \bigcup \{g\}$;
                        $S := S \bigcup \{g\}$   if $r \prec upd.r \wedge D = L \wedge$ any;
                        $\widetilde{r} := any\{x \mid x \preceq min(\widetilde{r}, upd.r)\}$   if $g \in S$;
                        $reply(g)$
[]
  rcv $ack$ from $g$ $\rightarrow$   $D[g] := max(D[g], ack.d + 1)$
                              if $ack.r \preceq r \wedge D[g] > 0$
[]
  $S \subseteq \widetilde{S}$ $\rightarrow$            $D := 0$  if $\widetilde{r} \prec r$;
                        $r, \widetilde{r}, \widetilde{S} := \widetilde{r}, \top, \emptyset$
[]
  any $\wedge$ $|S| > 1$ $\rightarrow$       $S := S - \{g\}$
end
```

The process consists of five actions. In the first action, a new *upd* message is sent to a neighbor $g$. The message is sent only if the previous *upd* message has been received (or is lost) and its corresponding *ack* has been received (or is lost). Furthermore, since $upd(u,g).r = u.r$, we can safely assign a value of at least one to $u.D[g]$.

In the second action, an *upd* message is received from a neighbor $g$. Neighbor $g$ is added as a successor if the rank ordering property is not violated, and in addition, the higher layer application chooses $g$ as a successor. We represent this by the operator **any**, which nondeterministically returns **true** or **false**. In this action, $reply(g)$ is a shorthand for the following sequence of statements.

```
$ack.r, ack.d := r, min\{D\}$ if $g \in S$;
$ack.r, ack.d := \perp, L$      if $g \notin S$;
send $ack$ to $g$
```

In the third action, an *ack* is received from a neighbor $g$. Variable $u.D[g]$ is increased provided the rank of $g$ is at most the rank of $u$ and $u.D[g] > 0$. The reason for $u.D[g] > 0$ is as follows. If $u.D[g] = 0$, then is possible that the *ack* received is in response to an *upd* message sent *before* $u.r$ was decreased. This would cause synchronization problems between $u$ and $g$, and the rank ordering property may be violated.

In the fourth action, process $u$ has finished receiving an *upd* message from all neighbors. It then updates $u.r, u.\widetilde{r}$, and $u.\widetilde{S}$ as discussed earlier.

In the fifth action, process $u$ removes neighbor $g$ from its successor set, provided the successor set does not become empty, and provided that the higher-layer application, which we model by the operator **any**, chooses to remove $g$.

We present below the specification of the root process.

```
process $root$
inp
  $G$       :  set of pid's          {neighbor set}
  $L$       :  integer               {max. path length}
const
  $S, \widetilde{S}$  :  $\emptyset, \emptyset$              {successor set and its iteration set}
  $r, \widetilde{r}$  :  $\top, \top$              {rank and its iteration value}
  $D$       :  array $[G]$ of $L$       {rank depth}
par
  $g$       :  element of $G$          {any neighbor}
begin
  timeout $upd \notin Ch(u,g) \wedge ack \notin Ch(g,u) = 0$  $\rightarrow$
                          $upd.r := r$;
                          send $upd$ to $g$
[]
  rcv $upd$ from $g$  $\rightarrow$   $reply(g)$
[]
  rcv $ack$ from $g$  $\rightarrow$   skip
end
```

The root process consists of three simple actions. In the second action, *reply*$(g)$ is a shorthand for the following sequence of statements.

$ack.r, ack.d := \top, L$;
**send** *ack* **to** $g$

Notice that the value of $root.D$ is always $L$, and that the value of $root.r$ is always $\top$. This is because the root does not need to decrease its rank, since it has no successors.

## 6 c-DAG Restoration

The processes in the previous section ensure that the network is maintained loop-free at all times. However, they are not stabilizing. In particular, if a loop exists at the initial state of the execution, then the loop may be maintained throughout the execution. In this section, we enhance our processes with the ability of automatically breaking any existing loop, and restoring the integrity of the c-DAG. Loops are detected using an extension of the spanning-tree technique we presented in [16].

Although the dynamically-ranked processes of the previous section are not stabilizing, they have an interesting property. Starting from any arbitrary state, the rank ordering property will eventually hold and continue to hold. That is, the processes stabilize to the rank-ordering property. Therefore, even though loops that exist at the initial state may not be broken, there is a point in the execution after which no new loops may be created.

[0] THs

Given that the rank-ordering property is stabilizing, the main obstacle in the stabilization of our processes is the removal of existing loops. Thus, processes must be able to detect the presence of a loop. In addition, the loop must be broken, and any processes that become separated from the c-DAG must rejoin it.

To detect loops, each process maintains an estimate of the number of hops to the root process. This estimate is maintained in variable $u.h$. Each *upd* message from $u$ now contains two values: the rank $u.r$ and hop count $u.h$. Process $u$ assigns to $u.h$ the largest hop count of each of its successors plus one.

To collect the hop counts from each neighbor, process $u$ maintains a variable $u.\widetilde{h}$. This variable contains the maximum hop count (plus one) of every neighbor in $u.\widetilde{S}$, i.e., of every neighbor from whom an *upd* message has been received. When an *upd* has been received from every neighbor, $u.\widetilde{h}$ is assigned to $u.h$ and $u.\widetilde{h}$ is assigned zero.

Since the maximum length of a simple network path is $L$, we expect the value of $u.h$ to never increase beyond $L - 1$. Thus, a straightforward way to detect a loop is to check if $u.h \geq L$. If so, process $u$ concludes that it is involved in a loop. However, this is not accurate due to the dynamic nature of the network, as we demonstrate below.

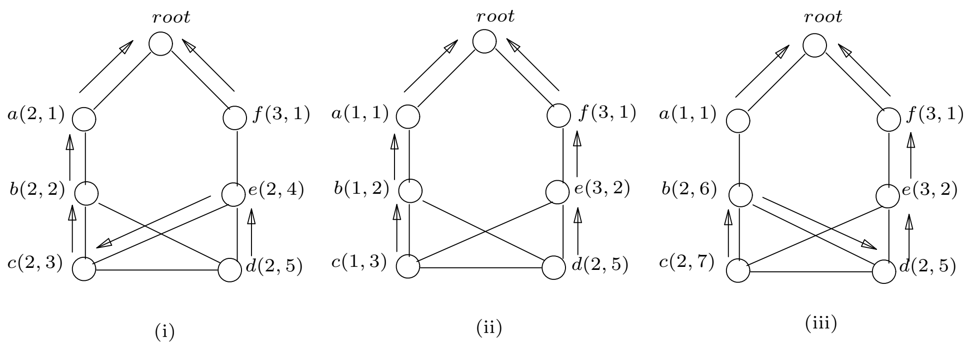

**Fig. 3.** Incorrect loop detection.

Consider the network in Figure 3. The rank of each process is an integer, and $\prec$ is simply $<$. In this network, $L = 7$. Alongside each process are its rank and its distance, in that order. The initial state of the network is given in Figure 3(i).

Assume process $e$ adds $f$ to its successor set, and then removes $c$ from its successor set. For the moment, assume the channel from $e$ to $d$ is slow, so $d$ does not update its values from those of $e$ for some time. After $e$ changes successors, the rank of $a$ drops to one, and this new rank is propagated to $b$ and $c$. Still, $d$ has not updated its values from those of $e$. This is shown in Figure 3(ii). Next, assume process $b$ chooses $d$ as a successor, and then removes $a$ from its successor set. The new rank of $b$ is then propagated to $c$. Still, $d$ has not updated its values from those of $e$. This is shown in Figure 3(iii).

Note that in Figure 3(iii), $c.h = 7 = L$. Thus, $c.h$ indicates the presence of a loop, even though none exists. (The scenario in Figure 3 can be extended further to show that $c.h$ grows without bound even though a loop is never present.) Therefore, a simple hop count cannot be used as a method of loop detection.

The above problem of erroneous loop detection is due to the flexibility in adding and removing successors. These operations need to be restricted, but not to the extent of making the structure inflexible. We choose to restrict them as follows.

**Definition 3. (Loop Detection Conditions)**

1. *A process $u$ cannot add a neighbor $v$ to its successor set if $v.h \geq L$.*
2. *A process $u$, where $u.h \geq L$, cannot add nor remove neighbors from its successor set unless all of processes $v$ below it have $v.h \geq L$.*
3. *A process $u$ cannot decrease $u.h$ to less than $L$ until all processes $v$ below it have $v.h \geq L$.*

□

From the above restrictions, when process $u$ reaches a hop count of at least $L$, it stops adding or removing successors. In addition, no process will choose $u$ as its successor. Then, a hop count of at least $L$ propagates to all descendants of $u$. In this way, the structure below $u$ will cease to change. Thus, since the maximum length of a simple path is $L$, no process below $u$ should obtain a hop count of $2L$ unless a loop exists.

When $u.h \geq 2L$, process $u$ assumes that either itself or a process above it is part of a loop. Process $u$ will empty its successor set (thus breaking the loop) and then choose as a successor the first neighbor which indicates that its hop count is less than $2L$. As in the previous section, process $u$ ensures that the new successor is not below $u$, and thus, no new loops are be formed.

What remains to be addressed is the method by which process $u$ determines that all its descendants have a hop count of at least $L$. We present a property similar to the rank ordering property of the previous section. Previously, $u.D = i$ implied that all processes at most $i$ hops below $u$ have a rank that is at most $u.r$. We now strengthen the meaning of $u.D = i$ to also imply that, if $u.h \geq L$, then all processes at most $i$ hops below $u$ have a hop count of at least $L$.

We refer to the pair of values $(u.r, u.h)$ as the *extended-rank* of $u$. For terseness, we will write this pair as $u.(r, h)$. Below, we define a relation $\preceq$ on extended-ranks[1]. The loop-avoidance conditions (Definition 1) of the previous section also hold for extended-ranks.

We define $\preceq$ on extended-ranks as follows: $(r, h) \preceq (r', h')$ iff

$$r \preceq r' \wedge (h' \geq L \Rightarrow h \geq L)$$

[1] We overload the symbol $\succeq$ to be a relation on ranks and a relation on extended-ranks. Which of these two meanings is appropriate is evident from the context.

We define $(r,h) \prec (r',h')$ similarly, except that $r \preceq r'$ is replaced by $r \prec r'$.

The rank-ordering property of the previous section (Definition 2) can now be replaced by the following extended-rank-ordering property.

**Definition 4. (Extended-Rank Ordering Property)**
*Consider any active path $P$, where: $t = P_1, g = P_{|P|-1}, u = P_{|P|}$, and $2 \leq |P| \leq u.D[g]$. Then, the following holds:*

$$t.(r,h) \preceq u.(r,h) \wedge (t.(\widetilde{r},\widetilde{h}) \preceq u.(r,h) \vee P_2 \notin t.\widetilde{S}) \wedge$$
$$(\forall x : neigh(x,t) : upd(t,x).(r,h) \preceq u.(r,h))$$

*In addition, if $u.D[g] = 1$, then $upd(u,g).(r,h) \preceq u.(r,h)$.* □

Using the above property, each process $u$ can deduce that, if $u.D = L \wedge u.h \geq L$, then all processes below $u$ have a hop count of at least $L$. Once this happens, $u$, can reduce its hop count to less than $L$ (if allowed by the hop counts of its successors) and make changes to its successor set.

We may now present the specification of a non-root process $u$ in the c-DAG-forming network of processes.

```
process $u$
inp
  $G$        :  set of pid's            {neighbor set}
  $L$        :  integer                 {max. path length}
var
  $S, \widetilde{S}$     :  subset of $G$            {successor set and its iteration set}
  $r, \widetilde{r}$     :  element of $R$           {rank and its iteration value}
  $h, \widetilde{h}$     :  $0 .. 2L$                {hop count and its iteration value}
  $D$        :  array [$G$] of $0 .. L$    {rank depth}
par
  $g$        :  element of $G$           {any neighbor}
begin
  timeout $upd \notin Ch(u,g) \wedge ack \notin Ch(g,u)$ →
                          $D[g] := max(1, D[g])$;
                          $upd.r, upd.h := r, h$;
                          send $upd$ to $g$
[]
  rcv $upd$ from $g$ →     $\widetilde{S} := \widetilde{S} \bigcup \{g\}$;
                          $S := S \bigcup \{g\}$   if $new\_succ(g)$;
                          $\widetilde{r} := any\{x \mid x \preceq min(\widetilde{r}, upd.r)\}$
                                  if $g \in S$;
                          $\widetilde{h} := max\{\widetilde{h}, upd.h + 1\}$
                                  if $g \in S$;
                          $S, \widetilde{r}, \widetilde{h} := \{g\}, upd.r, upd.h + 1$
                                  if $break(g)$;
                          $reply(g)$
[]
  rcv $ack$ from $g$ →     $D[g] := max(D[g], ack.d + 1)$
                                  if $ack.(r,h) \preceq (r,h) \wedge D[g] > 0$
```

```
[]
  $S \subseteq \widetilde{S}$ $\rightarrow$        $\widetilde{h} := max(\widetilde{h}, L)$   if $max(h, \widetilde{h}) \geq L \wedge D < L$;
                       $\widetilde{r} = \perp$ if $\widetilde{h} = 2L$;
                       $D := 0$              if $(\widetilde{r}, \widetilde{h}) \prec (r, h)$;
                       $r, h, \widetilde{r}, \widetilde{h}, \widetilde{S} := \widetilde{r}, \widetilde{h}, \top, 0, \emptyset$
[]
  any $\wedge$ $|S| > 1$ $\rightarrow$    $S := S - \{g\}$       if $\neg(max(h, \widetilde{h}) \geq L \wedge D < L)$
end
```

Process $u$ consists of five actions. In the first action, process $u$ sends an *upd* message to a neighbor $g$. This action is the same as before except that the message also contains the hop count.

In the second action, an *upd* message is received from a neighbor $g$. The first two statements are similar to those in the previous section. In this action, $new\_succ(g)$ is equivalent to the following.

$$(r, h) \prec upd.(r, h) \wedge D = L \wedge upd.h < L$$

Thus, the only difference from before is that extended-ranks are used when comparing the values of $u$ against those of the received message, and furthermore, $upd.h < L$ is necessary to satisfy the loop-detection conditions.

The next two statements in the action remain the same. The fifth statement breaks away from a loop. Here, $break(g)$ is defined as follows.

$$(r, h) = (\perp, 2L) \wedge D = L \wedge (r, h) \prec upd.(r, h)$$

That is, if $h = 2L$, then $u$ is involved in a loop, and it may choose $g$ as its sole successor (thus breaking the loop) provided the loop avoidance conditions are not violated, i.e., the extended-rank of $g$ is greater than that of $u$ and $D = L$. The reason we chose $r = \perp$ is explained below.

Finally, $reply(g)$ in the second action is a shorthand for the following sequence of statements.

```
$ack.r, ack.h := r, h$;
$ack.d := L$          if $g \notin S$;
$ack.d := min\{D\}$ if $g \in S$;
send $ack$ to $g$
```

In the third action, an *ack* message is received from a neighbor $g$, and $D[g]$ is increased. The only difference between this action and that of the previous section is that the decision to increase $D[g]$ is based on process extended-ranks.

In the fourth action, $r$ and $h$ are updated from $\widetilde{r}$ and $\widetilde{h}$ after an *upd* message has been received from every neighbor. The action differs from the previous section by not allowing $h$ to be reduced below $L$ until all descendants of $u$

have a hop count of at least $L$. This is necessary to satisfy the loop detection conditions. In addition, if $h = 2L$, i.e., if a loop is detected, the rank is set to the bottom value. This is done to "poison" all the descendants of $u$ also with a bottom rank, and thus the successor which will be used to break the loop must have a rank higher than the bottom value.

In the fifth action, a successor is removed. This operation is not allowed if the hop count of $u$ is at least $L$ and there are still neighbors whose hop count is less than $L$. This is also necessary to satisfy the loop detection conditions.

The specification of the root process is shown below.

```
process $root$
inp
   $G$       : set of pid's            {neighbor set}
   $L$       : integer                 {max. path length}
const
   $S, \widetilde{S}$    : $\emptyset, \emptyset$                  {successor set and its iteration set}
   $r, \widetilde{r}$    : $\top, \top$                  {rank and its iteration value}
   $h, \widetilde{h}$    : $0, 0$                  {hop count and its iteration value}
   $D$       : array $[G]$ of $L$          {rank depth}
par
   $g$       : element of $G$            {any neighbor}
begin
   timeout $upd \notin Ch(u, g) \wedge ack \notin Ch(g, u)$ $\rightarrow$
                                $upd.r, upd.h := r, h$;
                                send $upd$ to $g$
[]
   rcv $upd$ from $g$ $\rightarrow$        $reply(g)$
[]
   rcv $ack$ from $g$ $\rightarrow$        skip
end
```

The process consists of four simple actions. In the first action, the root sends an *upd* message to a neighbor $g$. In the second action, the root receives an *upd* message and it returns an *ack* message. In this action, $reply(g)$ is a shorthand for the following sequence of statements.

```
$ack.r, ack.h, ack.d := \top, 0, L$;
send $ack$ to $g$
```

In the third action, the root receives (and discards) an *ack*.

## 7 Protocol Correctness

Due to space restrictions, we present the proof of correctness in [23]. Here, we very briefly outline the proof for the interested reader.

A network stabilizes to a predicate $Z$ iff every computation of the network contains a suffix where each state of the computation satisfies $Z$ [14]. Thus, the system will reach a state after which it will continuously satisfy $Z$.

Starting from any initial state, the first property that is restored automatically is the rank-ordering property (in the dynamic-rank network), and the extended-rank-ordering property (in the c-DAG-forming network). Since the structure of the proof is similar for both networks, in [23], we abstract both of these proofs into a single one by introducing a network of abstract processes, where each abstract process has a general behavior that captures the behavior of both the dynamically-ranked processes and the stabilizing processes.

**Theorem 1. (Restoring Ranks)**

- *The rank-ordering property (Definition 2) is stabilizing in the dynamic-rank network of processes (Section 4).*
- *The extended-rank-ordering property (Definition 4) is stabilizing in the c-DAG-forming network (Section 6).* $\square$

Once ranks between nodes have the correct relationship, new loops cannot be formed, and we have the following.

**Theorem 2. (Loop-Freedom)**
*The c-DAG-forming network stabilizes to the following:*

$$(\forall\, u \;::\; \neg(\exists\, P \;::\; \mathit{active}(P) \,\wedge\, (P_1 = u) \,\wedge\, \mathit{loop}(P)))$$

$\square$

That is, loops are broken when the hop-count of processes reaches $2 \cdot L$, and no new loops are formed due to the label-ordering property. Finally, the desired state is then reached.

**Theorem 3. (c-DAG Restoration)**
*The c-DAG-forming network stabilizes to the following:*

$$(\forall\, u \;::\; (\exists\, P \;::\; \mathit{active}(P) \,\wedge\, P_1 = u \,\wedge\, P_{|P|} = \mathit{root})$$

$\square$

We therefore have that all active paths are loop-free, and each node has at least one active path to the root, i.e., a c-DAG is restored and maintained.

## References

1. Hedrick, C.: Routing information protocol. RFC 1058 (1988)
2. Moy, J.: Ospf version 2. RFC 1247 (August 1991)
3. Cobb, J.A., Gouda, M.G.: The request-reply family of group routing protocols. IEEE Transactions on Computers **46**(6) (June 1997) 659–672
4. Deering, S., Cheriton, D.: Multicast routing in datagram networks and extended lans. ACM Transactions on Computer Systems **8**(2) (May 1990)
5. Garcia-Luna-Aceves, J.J.: Loop-free routing using diffusing computations. IEEE/ACM Transactions on Networking **1**(1) (February 1993)

6. Garcia-Luna-Aceves, J.J., S., M.: A path-finding algorithm for loop-free routing. IEEE/ACM Transactions on Networking **5**(1) (February 1997)
7. Segall, A.: Distributed network protocols. IEEE Transactions on Information Theory **IT-29**(1) (January 1983) 23–35
8. Garcia-Luna-Aceves, J., Soumya, R.: On-demand loop-free routing with link vectors. In: Proceedings of the 12th IEEE International Conference on Network Protocols. (2004)
9. Johnson, D.B., Maltz, D.A., Hu, Y.C.: The dynamic source routing protocol for mobile ad hoc networks (dsr). work in progress, draft-ietf-manet-dsr-09.txt (2003)
10. Perkins, C.E., Royer, E.M.: Ad hoc on-demand distance vector routing. In: Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications. (1999) 90–100
11. Vutukury, S., Garcia-Luna-Aceves, J.J.: An algorithm for multi-path computation using distance-vectors with predecessor information. In: Proceedings of the ICCCN Conference. (1999)
12. Vutukury, S., Garcia-Luna-Aceves, J.J.: A distributed algorithm for multi-path computation. In: Proceedings of the IEEE GLOBECOM Conference. (1999)
13. Zaumen, W., Garcia-Luna-Aceves, J.J.: Loop-free multi-path routing using generalized diffusing computations. In: Proc. of the INFOCOM Conference. (1998)
14. Gouda, M.G.: The triumph and tribulation of system stabilization. In: Proceedings of the 9th International Workshop on Distributed Algorithms, (LNCS Vol 972). (1995) 1–18
15. Arora, A., Gouda, M.G., Herman, T.: Composite routing protocols. In: Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing. (1990)
16. Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. Journal of Parallel and Distributed Computing **62** (2002) 922–944
17. Cobb, J.A., Waris, M.: Propagated timestamps: A scheme for the stabilization of maximum-flow routing protocols. In: Proceedings of the Third Workshop on Self-Stabilizing Systems. (1997) 185–200
18. Gouda, M.G., Schneider, M.: Maximum flow routing. In: Proceedings of the Second Workshop on Self-Stabilizing Systems. (1995)
19. Cobb, J.A.: Convergent multipath routing. In: Proceedings of the International Conference on Network Protocols. (2002)
20. Gouda, M.: The Elements of Network Protocols. Wyley (1997)
21. Gouda, M.G., Schneider, M.: Maximizable routing metrics. In: Proceedings of the IEEE International Conference on Network Protocols. (1998) 71–78
22. Gouda, M., Schneider, M.: Stabilization of maximal metric trees. In: Proceedings of the Workshop on Self-Stabilizing Systems at the International Conference on Distributed Computing Systems. (June 1999)
23. Cobb, J.A.: Stabilization of loop-free redundant routing. The University of Texas at Dallas technical report (2007)