Stabilization of Routing in Directed Networks

Jorge A. Cobb[1] and Mohamed G. Gouda[2]

[1] Department of Computer Science (EC 31)
The University of Texas at Dallas, Richardson, TX 75083-0688
jcobb@utdallas.edu
[2] Department of Computer Sciences, The University of Texas at Austin
Austin, TX 78712-1188
gouda@cs.utexas.edu

Abstract. Routing messages in a network is often based on the assumption that each link, and so each path, in the network is bidirectional. The two directions of a path are employed in routing messages as follows. One direction is used by the nodes in the path to forward messages to their destination at the end of the path, and the other direction is used by the destination to inform the nodes in the path that this path does lead to the destination. Clearly, routing messages is more difficult in directed networks where links are unidirectional. (Examples of such networks are mobile ad-hoc networks and satellite networks.) In this paper, we present the first stabilizing protocol for routing messages in directed networks. We keep our presentation manageable by dividing it into three (relatively simple) steps. In the first step, we develop an arbitrary directed network where each node broadcasts to every reachable node in the network. In the second step, we enhance the network such that each node broadcasts its shortest distance to the destination. In the third step, we enhance the network further such that each node can determine its best neighbor for reaching the destination.

1 Introduction

The routing of messages from a source node to a destination node is a fundamental problem in computing networks. In general, routing protocols are divided into two broad categories [13]: link-state protocols and distance-vector protocols. In link-state protocols, each node broadcasts a list of its neighbors to all nodes in the network. Each node then builds in its memory the topology of the network, and computes the shortest path to each destination. A total of $O(n^2)$ storage is required to store this topology. Examples of link-state protocols include [7,14]. In distance-vector protocols, each node forwards to all neighboring nodes a vector with its distance to each destination. In this way, only $O(n)$ storage is required. Examples of distance-vector protocols include [1, 5, 6, 8, 12].

There is an implicit assumption in these protocols. It is assumed that each link, and therefore each path, is bidirectional. The forward path from a source node to a destination node is discovered by sending routing messages along this path, and the backward path is used to inform the source of the existence of

A.K. Datta and T. Herman (Eds.): WSS 2001, LNCS 2194, pp. 51–66, 2001.
© Springer-Verlag Berlin Heidelberg 2001

the forward path. However, new technologies are producing directed networks, that is, networks with unidirectional links. An example is networks with satellite links, since these links allow only unidirectional traffic [3, 4]. Another example is mobile wireless networks, in particular, ad-hoc networks [9, 10]. In these networks, nodes communicate with each other via radio links. These links may be unidirectional for several reasons. For example, there might be a disparity in the transmission power of two neighboring nodes. Thus, only one node may be able to receive messages from the other. In addition, if there is more interference at the vicinity of one node, then the node with the higher interference is unable to receive messages from its neighboring node.

Routing protocols that assume bidirectional links will fail in a directed network [9,10]. That is, the shortest path to a destination will not be found when this path contains unidirectional links. To remedy this shortcoming, new routing protocols have been developed that take unidirectional links into account [3,4,9,10]. Because routing protocols are essential in computing networks, it is desirable for them to be stabilizing. A protocol is said to be stabilizing iff it converges to a normal operating state starting from any arbitrary state [2,11]. Although stabilizing routing protocols exist for undirected networks, the protocols in [3, 4, 9, 10] for directed networks have not been shown to be stabilizing. In [3,4], "tunnels" need to be configured to go around single unidirectional links. Furthermore, this technique is not applicable to networks with an arbitrary number of unidirectional links. In [9,10], the stabilization of the protocol is not addressed (unbounded sequence numbers are used, which may require an unbounded stabilization time.)

In this paper, we present the first stabilizing protocol for routing messages in directed networks. We keep our presentation manageable by dividing it into three (relatively simple) steps. In the first step, we develop an arbitrary directed network where each node broadcasts to every reachable node in the network. In the second step, we enhance the network such that each node broadcasts its shortest distance to the destination. In the third step, we enhance the network further such that each node can determine its best neighbor for reaching the destination. In our network, each node is a process, and for simplicity, processes communicate with each other via shared memory. A message passing implementation is briefly discussed in Section 8. In addition, we assume that the cost of each link in the network is one. Thus, the shortest path (i.e., with the least number of links) is found to each destination. It is straightforward to modify our network to work with arbitrary positive costs assigned to each link.

2 Directed Networks

We consider a network of communicating processes that can be represented by a directed graph. In this directed graph, each node represents a distinct process in the network, and each directed edge from a process u to a process v represents a possible flow of information from u to v. Specifically, a directed edge from a process u to a process v indicates that each action of v can read the variables of both u and v, but can write only the variables of v. Thus, the existence of

a directed path from a process u to a process v indicates that information can flow from u to v (via the intermediate processes in the directed path), and the lack of a directed path from a process u to a process v indicates that information cannot flow from u to v.

If there is a directed edge from a process u to a process v, then u is called a *backward neighbor* of v and v is called a *forward neighbor* of u. In every process u, two constant sets, $B.u$ and $F.u$, are declared as follows.

const $B.u$: set of identifiers of all backward neighbors of u
$F.u$: set of identifiers of all forward neighbors of u

Without loss of generality, we assume that for each process u in a network, both $B.u$ and $F.u$ are non-empty. Each process in the network has a unique identifier in the range $0\,..\,n-1$, where n is the number of processes in the network. Process 0 is called the *network root*.

3 Routing Trees

Information needs to flow from every process in a directed network to the network root. To achieve this goal, every process u maintains the identifier of one of its forward neighbors, the one closest to the network root, in a variable named $next.u$. When the network reaches a stable state, the values of the $next.u$ variables define a directed rooted tree where all the directed paths lead to the network root. This tree is called a *routing tree*.

Also, every process u maintains the length of (i.e., the number of edges in) the shortest directed path from u to the root in a variable called $dist.u$. When the network reaches a stable state, the value of each $dist.u$ variable defines the length of the directed path from u to the root in the routing tree.

In fact, not every process in the network can be in the routing tree for two reasons. First, if the network has no directed path from a process u to the root, then information cannot flow from u to the root, and u cannot be in the routing tree. Second, if the network has no directed path from any backward neighbor of the root to a process u, then u cannot be informed of whether there is a directed path from u to the root. In this case, no information flow will be attempted from u to the root, and u cannot be in the routing tree.

From this discussion, u is in the routing tree of a network iff there is a backward neighbor v of the network root such that the network has a directed path from u to v and a directed path from v to u. The first path (from u to v) can be used as a route for the information flow from u to the root, and the second path (from v to u) can be used to inform u about the existence of the first path. When the network reaches a stable state, if a process u is in the routing tree, then the value of variable $dist.u$ is in the range $0\,..\,n-1$; otherwise, the value of variable $dist.u$ is n.

As an example, consider the directed network in Fig. 1. There is no directed path from processes 3 and 7 to process 0 (the root); thus, processes 3 and 7 cannot be in the routing tree. Also, there is no directed path from a backward

neighbor of process 0 (i.e., from process 2) to processes 1 and 5; thus, processes 1 and 5 cannot be in the routing tree. For each of the other processes (i.e. processes 2, 4, and 6), there is a backward neighbor of process 0 such that there are a directed path from the process to the backward neighbor and a directed path from the backward neighbor to the process. Thus, each of the processes 2, 4, and 6 is in the routing tree.

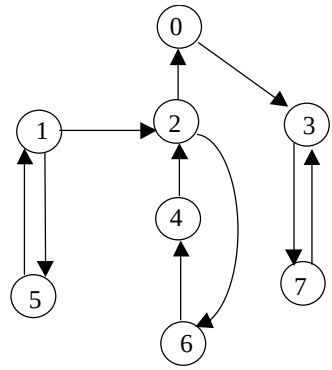

Fig. 1. A directed network.

The routing tree for this network is shown in Fig. 2. In this figure, the values of the two variables *next.u* and *dist.u* are written beside every process *u*. Also, the network edges that belong to the routing tree are shown as solid lines, whereas the other edges are shown as dashed lines.

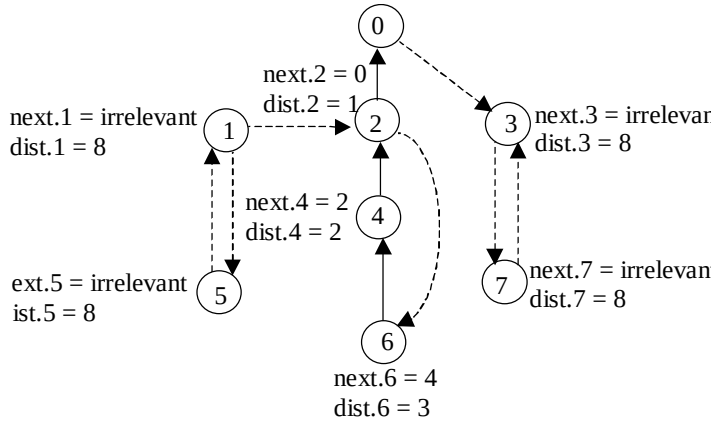

Fig. 2. The routing tree of the network in Fig. 1

Next, we discuss how to make the processes in a directed network maintain a routing tree. For simplicity, we divide our discussion into three steps. In the first step, we discuss how to make each process u broadcast a local value $x.u$ to every other process (that can be reached via a directed path from u) in the directed network. In the second step, we enhance the network process such that each value $x.u$, which is broadcasted by process u, is the shortest distance from u to the root. When the modified network reaches a stable state, each process u knows the network distance vector that stores, for every process v in the network, the shortest distance from v to the network root. In the third step, each process u computes from its network distance vector the two values $next.u$ and $dist.u$ needed for maintaining the routing tree. These three steps are discussed in more detail in Sections 5, 6 and 7.

4 Network Notation

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

process <process name>
const
    <constant name> : <type>,
    ...
    <constant name> : <type>
var
    <variable name> : <type>,
    ...
    <variable name> : <type>
par
    <parameter name> : <type>,
    ...
    <parameter name> : <type>
begin
    <action>
[]
    ...
[]
    <action>
end

The constants 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 by the actions of that process and the actions of forward neighbors of that process. The

variables declared in a process can be written only by the actions of that process. Parameters are discussed below.

Every action in a process is of the form <guard> → <body>. The <guard> is a boolean expression over the constants, variables, and parameters declared in the process, and also over the variables declared in the backward neighbors of that process. The <body> is a sequence of assignment statements that update the variables of the process.

Each parameter declared in a process is used as a shorthand to write a set of actions as one action. For example, if we have the following parameter definition,

par g : 1 . . 3

then the following action

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

is a shorthand notation for the following three actions.

$x = 1 \quad \rightarrow \quad x := x + 1$
[]
$x = 2 \quad \rightarrow \quad x := x + 2$
[]
$x = 3 \quad \rightarrow \quad x := x + 3$

An execution step of a network consists of evaluating the guards of all the actions of all processes, choosing one action whose guard evaluates to true, and executing the body of this action. An execution of a network 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 network to be weakly fair, that is, an action whose guard is continuously true is eventually executed.

5 Directed Broadcast

In this section, we discuss a directed network where each process u computes an array $X.u$ that has n elements, where n is the number of processes in the network. When the network reaches a stable state, the v^{th} element $X[v].u$ in array $X.u$ has the value $x.v$ that is local to process v.

In this network, each process u maintains, along with each element $X[v].u$, two corresponding elements:

$b[v].u$ = identifier of a backward neighbor of u from which u has read the latest value of $X[v].u$

$d[v].u$ = length of (i.e., number of edges in) the directed path along which the value $x.v$ (in v) is transmitted to $X[v].u$ (in u)

Note that if the value of $d[v].u$ ever becomes n, then process u recognizes that it has not yet found a directed path from v to u and that the current value

of $X[v].u$ is probably incorrect. Thus, if the network has no directed path from process v to process u, then the value of $d[v].u$ stabilizes to n and the value of $X[v].u$ stabilizes to a probably incorrect value.

Each process u in the directed broadcast network is defined next. In this definition, we use the expression $a \oplus b$ to mean $\min(a + b, n)$.

process $u : 0 \,..\, n-1$
const
$B.u$: set of identifiers of all backward neighbors of u
$F.u$: set of identifiers of all forward neighbors of u
$x.u$: $0 \,..\, n$ {local constant in u}
var
$X.u$: **array** $[0 \,..\, n-1]$ **of** $0 \,..\, n$,
$b.u$: **array** $[0 \,..\, n-1]$ **of** $B.u$,
$d.u$: **array** $[0 \,..\, n-1]$ **of** $0 \,..\, n$
par
v : $0 \,..\, n-1$, { any process in the network }
w : $B.u$ { any backward neighbor of u }
begin
$X[u].u \neq x.u \ \vee \ d[u].u \neq 0 \quad \rightarrow$
$X[u].u := x.u;$
$d[u].u := 0$
[]
$v \neq u \ \wedge \ b[v].u = w \ \wedge \ (X[v].u \neq X[v].w \ \vee \ d[v].u \neq d[v].w \oplus 1) \quad \rightarrow$
$X[v].u := X[v].w;$
$d[v].u := d[v].w \oplus 1$
[]
$v \neq u \ \wedge \ d[v].w \oplus 1 < d[v].u \quad \rightarrow$
$X[v].u := X[v].w;$
$b[v].u := w;$
$d[v].u := d[v].w \oplus 1$
end

Process u has three actions. In the first action, u ensures that $X[u].u$ equals its local constant $x.u$ and $d[u].u$ is zero. In the second action, u recognizes that it has read the latest value of $X[v].u$ from a backward neighbor w and so ensures that $X[v].u$ equals $X[v].w$ and $d[v].u$ equals $d[v].w \oplus 1$. In the third action, u recognizes there is a shorter directed path from v to u along its backward neighbor w. In this case, u assigns to $X[v].u$ the value of $X[v].w$, assigns to $b[v].u$ the value of w, and assigns to $d[v].u$ the value $d[v].w \oplus 1$.

This network maintains for every process u, a stabilizing, rooted, shortest-path spanning tree $T.u$. The root of $T.u$ is process u itself, and $T.u$ contains every process that is reachable from u via a directed path. The value of the constant $x.u$ flows from u over $T.u$ to every process in $T.u$.

6 Distance Vectors

In this section, we modify the network in the previous section such that the value of each element $X[v].u$ in each process u, when the network reaches a stable state, is the shortest distance (i.e., the smallest number of edges in a directed path) from process v to the network root. The modification is slight. The second and third actions in every process remain the same as before. Only the first action of each process u is modified to become as follows.

$$\begin{aligned} X[u].u \neq f(u, F.u, X.u) \ \vee \ d[u].u \neq 0 \quad \rightarrow \\ X[u].u := f(u, F.u, X.u); \\ d[u].u := 0 \end{aligned}$$

Above, $f(u, F.u, X.u)$ computes the shortest distance from u to the network root, and is defined as follows.

$$\begin{aligned} f(u, F.u, X.u) = 0 \ \text{ if } u = 0, \\ 1 \ \text{ if } u \neq 0 \ \wedge \ 0 \in F.u, \\ (\text{min over } v,\ v \in F.u \wedge d[v].u < n, \text{ of } X[v].u) \oplus 1 \ \text{ otherwise} \end{aligned}$$

In the appendix, we present a proof of the stabilization of this network.

7 Maintaining a Routing Tree

To make the network in the previous section maintain a routing tree (as defined in Section 3), each process u, $u \neq 0$, is modified as follows. First, the following two variables $next.u$ and $dist.u$ are added to process u.

$$\begin{array}{lll} \textbf{var} \ next.u & : & F.u, \\ \quad dist.u & : & 0\,..\,n \end{array}$$

Second, the following action is added to process u.

$$\begin{aligned} next.u \neq h(F.u, X.u) \ \vee \ dist.u \neq X[u].u \quad \rightarrow \\ next.u := h(F.u, X.u); \\ dist.u := X[u].u \end{aligned}$$

where

$$\begin{aligned} h(F.u, X.u) = \text{the smallest identifier } w \text{ in } F.u \text{ such that} \\ X[u].u = X[w].u \oplus 1 \end{aligned}$$

8 Message Passing Implementation

In order to simplify our presentation, processes in our network communicate with their neighbors using shared memory. In this section, we discuss a message passing implementation of our network.

We first address the construction of the shortest spanning trees discussed in Section 5. In our shared memory model, each process reads array d from each of its backward neighbors. To implement this, each process places the contents of its d array into a message, and periodically sends this message to all its forward neighbors. We will refer to this message as the spanning-tree message. Since array d has n entries, the size of the spanning-tree message is $O(n)$.

We next address the broadcast of the distance to the root (i.e. $X[u].u$) by every process u. To implement this, each process u places its distance to the root and its process identifier into a message, which we refer to as the broadcast message. This message is periodically sent to all forward neighbors of u. When a process v receives a broadcast message whose process identifier is u, this message is forwarded to all the forward neighbors of v, provided the message was received from neighbor $b[u].v$. In this way, the broadcast message is forwarded only along the spanning tree $T.u$, and the propagation of this message is cycle free. Note that the size of the broadcast message is $O(1)$.

The above two messages, namely the spanning-tree and broadcast message, are all that is required to implement the process network in a message passing model. We next address the storage requirements of each process.

Each process is required to store its d array, whose size is $O(n)$. With respect to the distances to the root (i.e., array X), note that when a process computes its distance to the root, it only requires the distance to the root of each of its forward neighbors. Therefore, in a message passing implementation, the distance to the root broadcasted by any other process is simply forwarded as soon as it is received. Thus, only the distances of the forward neighbors need to be stored.

Furthermore, with a more careful implementation, only the distance to the root of the next hop neighbor ($next.u$) needs to be stored. If process u receives a broadcast from a forward neighbor indicating a smaller distance to the root than that of neighbor $next.u$, then $next.u$ is updated to this neighbor, and the new distance is recorded. Thus, we require $O(1)$ storage for the distance to the root.

In our network, we considered only a single process (the root process) as the destination. To allow any process to be a destination, each process needs to maintain the distance to each destination. Thus, instead of $O(1)$ storage for the distance to the root mentioned above, we require $O(n)$ storage, i.e., $O(1)$ storage for each of the n possible destinations. Note, however, that array d remains as before, since our original network allows every process to perform a broadcast. Since array d requires $O(n)$ storage, the storage remains $O(n)$. In addition, the broadcast message would include the distance to each destination, and thus would be of size $O(n)$. The spanning-tree message remains $O(n)$. Therefore, since vectors of size n are sent to each neighbor, this network falls into the category of distance-vector routing networks.

One final issue remains to be addressed, and that is the detection of channel failure. Thus far, we have assumed that if a process v is in the forward neighbor set $F.u$ of a process u, then the channel from u to v is in working order. This is possible in networks where the channel is implemented by a lower layer, and the

status of this channel is monitored by the lower layer (e.g., the channel could be an ATM circuit, and the status of the channel is maintained by the ATM layer). If the lower layer at u detects that the channel from u to v has failed, then v is removed from $F.u$. However, if the lower layer does not provide the capability of monitoring the status of the channel, process u can monitor the status as follows. When process v sends a broadcast message, in addition to including its distance to each destination, it also includes a list of its backward neighbors from whom it has recently received messages. When process u receives a broadcast from v, it checks if u is in the list of backward neighbors of v. If so, then the channel from u to v is in working order.

Note that if the broadcast message includes the list of backward neighbors, we may choose to only include this list in the message, and not include the distance to each destination. In this case, each process would have to collect the list of neighbors from each process in the network, build in its memory a graph representing the network topology, and choose its next hop neighbor using Dijkstra's [13] shortest path algorithm. In this case, the storage required would increase to $O(n^2)$, and the network would fall into the category of link-state routing networks.

Finally, note that if a list of neighbors is required in the broadcast message, either because we have a link-state network or we require to detect the status of the channel in a distance-vector network, then the requirements for a process u to be in the routing tree are different than those presented in Section 3. In this case, a path must exist from u to the root and a path must also exist from the root to u.

9 Concluding Remarks

In this paper, we presented a network of processes that constructs a routing tree to a given destination, even though the network is directed, i.e., communication between neighboring processes may be unidirectional. We presented our network in three steps. First, we presented a network that allows each process to broadcast a value to all other processes. Next, we presented a network where each process can compute its distance to the destination, and broadcast this distance to all processes. Finally, we presented a network where a routing tree is constructed by having each process choose its parent in the routing tree in accordance to its distance to the destination.

Since an undirected network is a special case of a directed network, the network of processes we presented in Section 7 will correctly build a routing tree in an undirected network. However, it will not do so in the most efficient way, since processes are tailored towards a directed network. In future work, we will investigate networks of processes whose behavior will vary depending on the number of processes that have unidirectional communication with their neighbors. That is, processes will adapt to the "level" of unidirectional communication in the network, and adapt their behavior accordingly to improve performance.

Routing in directed networks is a fertile area of research, and much is yet to be done. Existing approaches assume bidirectional communication between neighbors, and thus will fail or exhibit different behaviors in directed networks. In addition, other distributed algorithms, in addition to routing, may be affected by directed networks. Two reasons for this may be routing asymmetry, i.e., for two nodes u and v, the path from u to v is not necessarily the same as the path from v to u, and one-way reachability, that is, there is a path from u to v but there is no path from v to u.

References

1. Cobb J., 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) pp. 185-200
2. Dolev S.: Self-Stabilization. MIT press, Cambridge, MA (2000)
3. Duros E., Dabbous W.: Supporting Unidirectional Links in the Internet. In: Proceedings of the First International Workshop on Satellite-Based Information Services (1996)
4. Duros E., Dabbous W., Izumiyama H., Fujii N., and Zhang Y.: A Link Layer Tunneling Mechanism for Unidirectional Links. Internet Request for Comments (RFC) 3077 (2001)
5. Garcia-Luna-Aceves, J.J.: Loop-Free Routing Using Diffusing Computations. IEEE/ACM Transactions on Networking. Vol 1 No. 1 (Feb. 1993)
6. Hedrick C.: Routing Information Protocol. Internet Request for Comments (RFC) 1058 (1988)
7. Moy J: OSPF Version 2. Internet Request for Comments (RFC) 1247 (1991)
8. Perkins C., Bhagwat P.: Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In: Proceedings of the ACM SIGCOMM Conference on Communication Architectures, Protocols and Applications (1994)
9. Prakash R.: Unidirectional Links Prove Costly in Wireless Ad Hoc Networks. In: Proceedings of the ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL M for Mobility) (1999)
10. Prakash R., Singhal M.: Impact of Unidirectional Links in Wireless Ad Hoc Networks. DIMACS Series in Discrete Mathematics and Computer Science, Vol. 52 (2000)
11. Schneider M.: Self-Stabilization. In: ACM Computing Surveys Vol. 25 No. 1 (1983)
12. Schneider M., Gouda M.: Stabilization of Maximal Metric Trees. In: Proceedings of the International Conference on Distributed Computing Systems, Workshop on Self-Stabilizing Systems (1999)
13. Tanenbaum A.: Computer Networks (3rd edition). Prentice Hall (1996)
14. Vutukury S., Garcia-Luna-Aceves J.J.: A Simple Approximation to Minimum Delay Routing. In: Proceedings of the ACM SIGCOMM Conference on Communication Architectures, Protocols and Applications (1999)

Appendix: Proof of Stabilization

Let $P_{min}(v, u)$ be a shortest path from v to u. Let $G(v)$ be the graph obtained from all edges of the form $(b[v].u, u)$ for every process u, $u \neq v$. Let $P_G(v, u)$ be

the path from v to u in $G(v)$[1]. If no such path exists, then $P_G(v,u)$ is the empty path. Let *btree* denote the following predicate.

$$
\begin{array}{l}
(\forall\, u :: d[u].u = 0) \wedge \\
(\forall\, v,u : v \neq u \ \wedge\ P_{min}(v,u) = \emptyset : d[v].u = n) \wedge \\
(\forall\, v,u : v \neq u \ \wedge\ P_{min}(v,u) \neq \emptyset : \\
\qquad\qquad P_G(v,u) \neq \emptyset \ \wedge\ |P_G(v,u)| = |P_{min}(v,u)| = d[v].u)
\end{array}
$$

Lemma 1.

1. *btree is stable*
2. *true converges to btree*

Proof. We focus only on arrays $d.u$ and $b.u$ of each process u, since only these variables are involved in *btree*. We first show the stability of *btree*.

Consider the first action. This action affects only $d[u].u$. From *btree*, $d[u].u = 0$ before the action. Thus, executing the action does not change $d[u].u$.

Consider the second action. This action affects only $d[v].u$. We have two cases.

1. Consider first $d[v].w < n$. In this case, from *btree*, $P_G(v,w) \neq \emptyset$, and $d[v].w = |P_G(v,w)| = |P_{min}(v,w)|$. Since $P_G(v,w) \neq \emptyset$ and $b[v].u = w$, then $P_G(v,u) = P_G(v,w);(w,u)$, and thus $P_{min}(v,u) \neq \emptyset$. This, along with *btree*, implies that $d[v].u = |P_{min}(v,u)| = |P_G(v,u)| = |P_G(v,w)| + 1 = d[v].w + 1$. Also, since $|P_{min}(v,u)| < n$, then $d[v].w + 1 < n$, and hence, $d[v].w + 1 = d[v].w \oplus 1$. Thus, $d[v].u$ is not changed when it is assigned $d[v].w \oplus 1$.
2. Consider instead $d[v].w = n$. In this case, from *btree*, $P_{min}(v,w) = \emptyset$. Hence, $P_G(v,w) = \emptyset$. From $b[v].u = w$, we have $P_G(v,u) = \emptyset$. From *btree*, if $P_{min}(v,u) \neq \emptyset$, then $P_G(v,u) \neq \emptyset$. Thus, we must have $P_{min}(v,u) = \emptyset$. Again, from *btree*, $d[v].u = n$, and thus $d[v].u$ does not change when it is assigned $d[v].w \oplus 1$.

Consider now the third action. Again, we have two cases.

1. Consider first $d[v].u < n$. From *btree*, $d[v].u = |P_{min}(v,u)| = |P_G(v,u)|$. From the action's guard, $d[v].w \oplus 1 < d[v].u$, which implies $d[v].w < n$, and from *btree*, $d[v].w = |P_G(v,w)| = |P_{min}(v,w)|$. Note, however, that since w is a backward neighbor of u, $d[v].w \oplus 1 < d[v].u$ implies that $|P_G(v,w);(w,u)| < |P_G(v,u)|$, which is impossible, since $P_G(v,u)$ is the shortest path from v to u. Thus, the guard must be false if *btree* holds.
2. Consider now $d[v].u = n$. From *btree*, there is no path from v to u. However, if $d[v].w \oplus 1 < d[v].u$, then $d[v].w < n$, and *btree* implies there is a path from v to w, and thus there is also a path from v to u. Thus, again, the guard must be false if *btree* holds.

[1] Note that at most only one such path may exist, since u has only one incoming edge in $G(v)$, and v has no incoming edge.

Since no action falsifies *btree*, *btree* is stable. We now show that true converges to *btree*. First, note that the first action of u assigns zero to $d[u].u$. No other action modifies $d[u].u$, thus, for all u, true converges to $d[u].u = 0$, and $d[u].u = 0$ is stable.

Next, define $btree(v, i)$, where $1 \leq i < n$, as follows:

$$(\forall\, u : u \neq v \ \wedge\ (P_{min}(v,u) = \emptyset \ \vee\ |P_{min}(v,u)| > i) : d[v].u > i) \ \wedge$$
$$(\forall\, u : u \neq v \ \wedge\ P_{min}(v,u) \neq \emptyset \ \wedge\ |P_{min}(v,u)| \leq i : d[v].u = |P_{min}(v,u)| = |P_G(v,u)|)$$

We show that eventually, for all i, $btree(v, i)$ holds and continues to hold. We show this by induction.

As a base case, consider $i = 1$. Consider any process u, $u \neq v$. The second and third actions assign at least 1 to $d[v].u$. Thus, we can assume we reach a state where $d[v].u \geq 1$ holds and continues to hold for all processes u, $u \neq v$.

Consider a process u which is *not* a forward neighbor of v. Thus, for any backward neighbor w of u, $d[v].w \geq 1$, and thus $d[v].u \geq 2$ after the second or third action. Thus, $d[v].u \geq 2$ will hold and continue to hold, as desired in $btree(v, 1)$.

Consider now a process u which *is* a forward neighbor of v. We have two cases.

1. Assume u executes its second or third action with parameter $w = v$. Then, since $b[v].v = 0$, we obtain $b[v].u = v \wedge d[v].u = 1$, thus, $d[v].u = |P_{min}(v,u)| = |P_g(v,u)|$ as desired by $btree(v, 1)$. Note that this continues to hold for the following reason. First, the second action of u does not change $b[v].u$, and it assigns $d[v].v \oplus 1 = 1$ to $d[v].u$. The third action of u is not enabled, since for any backward neighbor w of u (including $w = v$), $d[v].w \oplus 1 \geq 1 = d[v].u$.
2. Assume u executes its second or third action with parameter $w \neq v$. Since $d[v].w \geq 1$, it results in $b[v].u = w \ \wedge\ d[v].u > 1$. Since $d[v].u > 1$, then the third action of u is enabled with $w = v$, and must be eventually executed, after which $b[v].u = v \wedge d[v].u = 1$, as shown above. Also, this continues to hold as shown above.

For the induction hypothesis, we assume $1 < i < n$, and $btree(v, i-1)$ holds.

Consider first any process u, where $u \neq v \wedge P_{min}(v,u) = \emptyset$. For any backward neighbor w of u, $P_{min}(v,w) = \emptyset$, and from $btree(v, i-1)$, $d[v].w \geq i-1$. Hence, when process u executes its second or third action, $d[v].u \geq i$, as desired. Since $d[v].w \geq i-1$ will continue to hold, then so will $d[v].u \geq i$.

Consider next any process u, $u \neq v$, where $P_{min} \neq \emptyset \wedge |P_{min}(v,u)| > i > i-1$. In this case, any backward neighbor w of u must have $P_{min}(v,u) = \emptyset \vee |P_{min}(v,w)| \geq i > i-1$, and from $btree(v, i-1)$, $d[v].w \geq i$, and this continues to hold. When u executes its second or third action, $d[v].u > i$ will result. Thus, $d[v].u > i$ will hold and continue to hold.

Consider now a process u with $P_{min} \neq \emptyset \wedge |P_{min}(v,u)| = i$. This implies that for all backward neighbors w of u, $P_{min}(v,w) = \emptyset \vee |P_{min}(v,w)| \geq i-1$.

From $btree(v, i-1)$, $d[v].w \geq i-1$. In addition, u must have a backward neighbor y such that $P_{min}(v,y) \neq \emptyset \wedge |P_{min}(v,y)| = i-1$. From $btree(v, i-1)$, $i-1 = d[v].y = |P_G(v,y)| = |P_{min}(v,y)|$ holds and will continue to hold. We have two cases.

1. Assume the second or third action of u is executed where $w = y$. Then, after the action is executed, $b[v].u = y \wedge d[v].u = i$, and hence, $P_G(v,u) = PG(v,y);(y,u) \wedge |P_G(v,u)| = i = |P_{min}(v,u)|$ as desired for $btree(v,i)$. Furthermore, executing the second action does not change these values, and note that the third action cannot execute, since all backward neighbors w of u must have $d[v].w \geq i-1$, and thus, $d[v].w \oplus 1 \geq d[v].u$. Hence $i = d[v].u = |P_{min}(v,u)| = |P_G(v,u)|$ will continue to hold forever.
2. Assume the second or third action of u is executed for some backward neighbor w, where $w \neq y$. In this case, $P_{min}(v,w) = \emptyset \vee |P_{min}(v,w)| = i$. From $btree(v, i-1)$, $d[v].w \geq i$. Then, after the action, $d[v].u \geq i+1$. Note that in this case, the third action is enabled with $w = y$, and it will eventually be executed. As argued above, $i = d[v].u = |P_{min}(v,u)| = |P_G(v,u)|$ will then hold and continue to hold.

This finishes the induction step, and thus the induction proof.

Note that $btree = (\forall v, i : 1 \leq i < n : btree(v,i)) \wedge (\forall u :: d[u].u = 0)$. Thus, from the above, eventually we reach a state where *btree* holds and continues to hold.

Corollary 1. *For all u, v, and y, where $u \neq v$,*

$$btree \wedge b[v].u = y \text{ is stable}$$

Proof. Only the third action affects $b[v].u$, so we focus on this action.

Assume first that $P_{min}(v,u) = \emptyset$. Thus, $P_{min}(v,w) = \emptyset$. From *btree*, $d[v].u = n \wedge d[v].w = n$. Thus, the third action is not enabled.

Assume next that $P_{min}(v,u) \neq \emptyset$. From *btree*, $d[v].u = |P_{min}(v,u)| < n$. For any backward neighbor w of u, if the guard $d[v].w \oplus 1 < d[v].u$ is true, then this implies $d[v].w < n-1$, and from *btree*, $d[v].w = |P_{min}(v,w)|$. Combining the above, $|P_{min}(v,w)| \oplus 1 < |P_{min}(v,u)|$. This is not possible since $P_{min}(v,u)$ is a minimum path. Thus, the third action is not enabled.

Lemma 2.

1. *Consider a computation in which $btree \wedge X[v].v \leq k$ holds for some k and continues to hold. Let $u \neq v$ and $P_G(v,u) \neq \emptyset$. Then, for any y, where $y \neq v$ and y is a process in $P_G(v,u)$, $X[v].y \leq k$ will hold and continue to hold.*
2. *Consider a computation in which $btree \wedge X[v].v \geq k$ holds for some k and continues to hold. Let $u \neq v$ and $P_G(v,u) \neq \emptyset$. Then, for any y, where $y \neq v$ and y is a process in $P_G(v,u)$, $X[v].y \geq k$ will hold and continue to hold.*

Proof. We consider only part 2 above. The proof for part 1 is similar.

From Lemma 1 and Corollary 1, *btree* continues to hold, and $P_G(v,u)$ does not change. Also, only the second action modifies $X[v].u$, so we focus on this action.

The proof is by induction on the length of $P_G(v,u)$. Assume that $P_G(v,u) = (v,u)$, i.e., $b[v].u = v$. If $X[v].u \neq X[v].v$, then the second action of u is enabled with $v = w$. When this action executes, $X[v].u = X[v].v \geq k$. If this action becomes disabled before being executed, then from its guard it must also be that $X[v].u = X[v].v \geq k$. Thus, $X[v].u \geq k$ will hold and continue to hold.

For the induction step, let $w = b[v].u$, and let $|P_G(v,u)| = i, i > 1$. We assume the lemma holds for all paths in $G(v)$ of length $i-1$, in particular, $P_G(v,w)$. Thus, eventually, $X[v].w \geq k$ holds, and it continues to hold. A similar argument as the one above, except that process u assigns $X[v].w$ to $X[v].u$, shows that $X[v].u \geq k$ will hold and continue to hold. This concludes the proof.

Let S_i, where $0 \leq i < n$, be a set of processes defined as follows.

$$
\begin{aligned}
S_0 &= \{\ 0\ \} \\
S_1 &= \{\ u \mid u \text{ is a backwards neighbor of process } 0\} \cup S_0 \\
S_i &= \{\ u \mid \text{there is a path from a backwards neighbor } v \\
&\qquad \text{of the root to } u, \text{ and a path of length at most } i \\
&\qquad \text{from } u \text{ to the root via } v\ \} \cup S_1,\ 1 < i < n
\end{aligned}
$$

Note that a process u is in the routing tree iff $u \in S_{n-1}$.

Lemma 3. *For every i, $1 < i < n$,*

$$u \in S_i \Leftrightarrow (u \in S_1\ \vee\ (\exists\, v : v \in F.u : v \in S_{i-1}\ \wedge\ P_{min}(v,u) \neq \emptyset))$$

Proof. Consider first the following implication.

$$u \in S_i \Leftarrow (u \in S_1\ \vee\ (\exists\, v : v \in F.u : v \in S_{i-1}\ \wedge\ P_{min}(v,u) \neq \emptyset))$$

If u is in S_1 then, from the definition of S_i, u is in S_i for any i, $i > 1$. Instead, assume u is not in S_1. Thus, assume there exists a v satisfying the quantification. We are given that $v \in S_{i-1}$. Note that $i-1 > 0$, since otherwise, v would be the root, and u would be in S_1.

Assume first that $i-1 = 1$. In this case, v is a backward neighbor of the root, and from $P_{min}(v,u) \neq \emptyset$, there is a path from v to u. Thus, u is in S_i. Assume instead that $i-1 > 1$. Then there is a path of length $i-1$ from v to the root via a backward neighbor w of the root, and a path from w to v. Since $v \in F.u$ and $P_{min}(v,u) \neq \emptyset$, there is a path of length i from u to the root via w (and via v), and also a path from w to u (via v). Thus, u is in S_i.

Consider now the other implication.

$$u \in S_i \Rightarrow (u \in S_1\ \vee\ (\exists\, v : v \in F.u : v \in S_{i-1}\ \wedge\ P_{min}(v,u) \neq \emptyset))$$

If u is in S_1, then we are done. Assume instead that u is in S_i, but u is not in S_1. Then, since u belongs to S_i, there is a path from u to the root via a backward

neighbor w of the root, and the length of this path is i. Let v be the next hop from u to w along this path. Thus, there is a path of length $i-1$ from v to the root via w. Also, since there is a path from w to u, then there is a path from v to u. Hence, $P_{min}(v,u) \neq \emptyset$ and v is in S_{i-1}.

Combining both implications we obtain the desired result.

Theorem 1. *For every i, $0 \leq i < n$, the distance vector network stabilizes to the following predicate.*

$$(\forall\, u :: u \in S_i \Leftrightarrow X[u].u \leq i)$$

Proof. $X[u].u$ is changed only in the first action, so we focus only on this action. Also, from Lemma 1, the network stabilizes to *btree*. Thus, consider a computation starting from a state in which *btree* holds.

Consider first $i = 0$. The only process in S_0 is the root process 0. The definition of function f ensures that $X[0].0$ is assigned 0 regardless of the network state. Thus, $X[0].0 = 0$ is stable. For any process u, $u \neq 0$, f assigns at least 1 to $X[u].u$. Thus, $u \in S_0 \Leftrightarrow X[u].u \leq 0$ will hold and continue to hold.

Consider next $i = 1$. Let $u \in S_1$. By the definition of S_1, the root is a forward neighbor of u. From the definition f, $X[u].u$ will be assigned 1 regardless of the network state. Thus, $X[u].u = 1$ will hold and continue to hold. Consider now any process u, where $u \notin S_1$. Thus, for every forward neighbor v of u, $v \neq 0$, i.e., $v \notin S_0$, and from above, $X[v].v \geq 1$ will hold and continue to hold. If there is a path from v to u then, from Lemma 2 part 2, eventually $X[v].u \geq 1$ holds and continues to hold. If there is no path from v to u, then from *btree*, $d[v].u = n$ holds and continues to hold. Thus, from the definition of f, $X[u].u$ will always be assigned at least 2, i.e., $X[u].u > 1$ will hold and continue to hold.

The remainder of the proof is by induction over i, where $1 \leq i < n$. We show that for each i, the network stabilizes to $(\forall\, u :: u \in S_i \Leftrightarrow X[u].u \leq i)$. The base case, $i = 1$, was shown above. Thus, consider $1 < i < n$, and assume we have a computation where we have reached a state where $(\forall\, u :: u \in S_{i-1} \Leftrightarrow X[u].u \leq i-1)$ holds and continues to hold.

Consider a process u, $u \in S_i$, and $u \notin S_{i-1}$. From the definition of S_i, and from Lemma 3, a forward neighbor v belongs to S_{i-1} and there is a path from v to u. From the induction hypothesis, $X[v].v \leq i-1$, and from *btree*, $d[v].u < n$. Furthermore, from Lemma 2 part 1, eventually $X[v].u \leq i-1$ holds and continues to hold. Thus, from the definition of f, $X[u].u$ is assigned a value at most i, and this continues to hold.

Consider now a process u, $u \notin S_i$. From Lemma 3, $u \notin S_1$, and for all forward neighbors v of u, $v \notin S_{i-1} \ \vee \ P_{min}(v,u) = \emptyset$. If $v \notin S_{i-1}$, then from the induction hypothesis, $X[v].v > i-1$, i.e., $X[v].v \geq i$, holds and continues to hold, and from Lemma 2 part 2, eventually $X[v].u \geq i$ holds and continues to hold. If $P_{min}(v,u) = \emptyset$, then from *btree*, $d[v].u = n$. Hence, from f, eventually $X[u].u > i$ holds and continues to hold.

Thus, by induction, for all $i, 0 \leq i < n$, the network stabilizes to $(\forall\, u :: u \in S_i \Leftrightarrow X[u].u \leq i)$.