May 1987 LIDS-P-1692

TOPOLOGY BROADCAST ALGORITHMS

Pierre A. Humblet
Stuart R. Soloway

ABSTRACT

This paper describes efficient distributed algorithms to broadcast network topology to all nodes in a network. They also build minimum depth spanning trees, one rooted at each node. The broadcast of the topology takes place simultaneously with the building of the spanning trees, in a way that insures that each node receives information exactly once. The algorithms are extended to work in presence of link and node failures.

Pierre Humblet is with the Laboratory for Information and Decision Systems, Rm 35-203, Massachusetts Institute of Technology, Cambridge, MA 02139. His work on this research was supported in part by Codex Corporation and in part by the National Science Foundation under contract ECS 8310698. Stuart Soloway performed this research while he was with Codex Corporation; he is now with Digital Equipment Corporation, LTN2-2/h17, 305 Foster Street, Littleton, MA 01460.

Address correspondence about this paper to: Stuart R. Soloway

1 INTRODUCTION

It is often necessary for all the nodes in a communication network to be aware of the network topology for routing and control purposes. This requires a distributed algorithm, as the presence or absence of a communication link is detected locally at the end points of the link and must be communicated to the other nodes. The way communication takes place between the nodes depends on the topology itself, leading to an interesting problem.

Some networks (e.g. the Arpanet [McQ80]) solve the problem by flooding, i.e. transmitting information about each node over each link in the network, insuring that it will be received by all nodes. This is inefficient as typically a node receives the same information many times. An alternative is to first build a spanning tree in the network, and then to broadcast the topology on the spanning tree. Distributed algorithms to build spanning trees have been developed, e.g. see [Dal77], [Seg83], [Gal83], [Hum83].

It has not been widely recognized that spanning trees can be built while the topology is being broadcast. This paper describes such an integrated approach where minimum depth spanning trees, one rooted at each node of the network, are built at the same time as the network topology is broadcast on the spanning trees, insuring that each node receives information about each link only once. This class of algorithm is thus efficient and fast.

In sections 2 and 3 we describe two such algorithms, and we prove their correctness, assuming that the network topology remains unchanged. We then discuss the communication cost of the algorithms, and compare them with those of alternate approaches in section 4. Finally in section 5 we discuss methods to handle changing topologies.

Before starting with the algorithms we present the part of the model that is common to all sections. The network consists of a set of nodes with distinct identities, and of directed edges. For the sake of notation simplicity we allow only one edge between any two nodes, so that (I,J) denotes the edge from I to J. We assume that if there is an edge (I,J) there is also an edge (J,I), or, in communication terms, that communication takes place in both directions on the links; some of the link characteristics may depend on the direction of traffic. Initially each node knows the identities of its neighbors.

The nodes execute in parallel a distributed algorithm consisting of exchanging messages over links and processing. The nodes have no access to shared memory or to a global clock. A node starts the distributed algorithm on receiving a local signal or a message from an adjacent node.

2 TOPOLOGY BROADCAST ALGORITHM: VERSION 1

2.1 Introduction

The purpose of this algorithm is to build in a distributed fashion minimum depth spanning trees rooted at each node of the network, and to broadcast the network topology (at the minimum the set of nodes and their adjacent edges, optionally other information like link delay) on

those spanning trees.

A minimum depth spanning tree rooted at a node is a tree of shortest paths from the root node to the other nodes in the network, where the length of an edge is 1. Ties are broken in some fashion dependent on the algorithm.

The general idea behind the algorithm is the following: Assume that node I has received the topological information from all nodes at distance HOP or less. By this we mean that it knows those nodes, AND ALSO who the neighbors of those nodes are. Consequently node I knows the identities of all other nodes at distance HOP + 1 or less and all length HOP+1 paths originating at node I. It now faces the question whether it should pass information about a node K at distance HOP to a neighbor J. It only wants to do it if the distance between K and J is HOP+1 and if no other neighbor of J will provide information about K.

As node I knows all length HOP+1 paths from itself it knows all length HOP paths from J; it also knows all the neighbors of K. Thus I can find all paths of length HOP+1 or less between K and J and determine, based on some arbitration rule, which of the neighbors of J is responsible for passing to J information about K, for example the neighbor of J on the lexicographically smallest minimum hop path from K to J, using as path name the sequence of nodes in the path, starting with node K.

The situation is illustrated in figure 1 for HOP = 3: I1, I2 and I3 are all candidates to send information about K to J. They all have enough information to recreate the figure and to arbitrate among themselves, without requiring any extra coordinating messages and without time-consuming computation. The lexicographically minimal path is

(K,M1,Q,I2,J). It can be recognized by the following two remarks:
- M1 and M2 are at distance HOP from J, and M1 < M2 (this eliminates M2)
- the lowest path between M1 and J is (M1,Q,I2,K) (this eliminates I3)

The method uses the idea of iterating on HOP, which has been used before, e.g. in [Gal76] to find shortest paths. The following section defines the algorithm and shows its correctness.

2.2 Precise Description Of The Algorithm

A description of the messages and data structure follows. Pseudo code for the algorithm appears in figure 2 (the notation A \ B on sets denotes the elements in A that are not in B). On starting, a node first executes the initialization code in figure 2. We assume that a message sent on a link arrives after a finite delay; we make no assumption on the order of message delivery. Recall that the topology was assumed to be fixed.

2.2.1 Messages And Data Structures

Only one type of message is transmitted during the course of the algorithm. It is a record containing the destinations (and possibly other information not used by the topology update) about the edges outgoing from some node; it is denoted Neighbors(K) when it is about node K.

Each node I keeps the following variables:

- An integer counter, called HOP.
- Sets Paths(K,dist) where K is node I or one of its neighbors, and dist is an integer. Paths(K,dist) will be the set of all nodes reachable from K by a path with dist links (possibly containing

cycles; some authors would use the word "walk" instead of "path").

-For each adjacent node, the set SPT(J) will contain all nodes K for which link (I,J) is part of the minimum hop spanning tree rooted at K; if there are ties, the spanning tree will contain lexicographically minimum paths.

2.2.2 Proof of correctness

We prove the following theorem about the algorithm of figure 2:

Theorem:
If any node in a connected component of the network starts the algorithm, then eventually all nodes in the component will stop, having received Neighbors(K) for all connected nodes K and having set SPT(J) for all local links J according to its definition.

Note first that if any node starts the algorithm, all its neighbors will receive a message and will also start. Eventually all connected nodes will start. The proof of correctness continues with an induction on HOP. The following is true with HOP = 1 after the algorithm has been initialized at node I:

a) Node I has Neighbors(K) from all nodes K at distance less than HOP.
b) Paths(I,n) has been set according to its definition for n <= HOP and Paths(J,n) has been set according to its definition for all neighbors J and n < HOP.
c) For all J, SPT(J) is the set of all nodes K at distance less than HOP from I such that I lies on the lexicographically shortest path between K and J.
d) If K is in SPT(J), node I has sent Neighbors(K) to J.

e) Node I will not stop before it has received and transmitted Neighbors(K) for all nodes on the correct spanning trees.

Assuming the previous statements to eventually hold for HOP=n at all nodes that have not stopped with a smaller HOP, we now proceed to show that at those nodes HOP will eventually reach n+1 and that assertions a) to e) will still hold.

By b), c) d) and e), all nodes have sent (or will send) the messages Neighbors(J) for the nodes at distance HOP-1 from them and by a) Paths(I,HOP) has been correctly set; thus the test in line 2 will eventually be satisfied, HOP will be incremented on line 11 and

a) will remain valid.

Paths(I,HOP+1) is set in line 3 according to its definition, as Paths(I,HOP) is assumed to be correct, and all paths of length HOP + 1 from K to I must consist of a link out of K to a neighbor L, followed by a path of length HOP from L to I. Thus b) will remain valid after HOP is incremented.

A similar reasoning shows that line 5 constructs Paths(J,HOP) so that c) will remain valid after HOP is incremented.

Line 6 selects all nodes K at distance HOP+1 from J for which I lies on a shortest path, as there is a path of length HOP from K to I, but no path of length HOP or less from K to J. For all the nodes selected on line 6, line 7 first finds the set of neighbors lying on a path of length HOP + 1 to J, and selects the neighbor M that is on the lexicographically minimum such path; in step 8, K is added to SPT(J) if and only if (by the induction hypothesis) I lies on the

(lexicographically) shortest path between M and J. Together lines 7 and 8 insure that K is added to SPT(J) if and only if I lies on the lexicographically shortest path between K and J. Thus c) will remain valid after HOP is incremented.

Line 9 and the properties of SPT insure that d) will remain valid after HOP is incremented.

Finally if the test on line 12 is satisfied, there are no nodes at distance HOP, thus no node at distance greater than HOP because the node identities are distinct. This insures that e) holds. The node must eventually stop because Paths() is correctly set and the network is finite.

3 TOPOLOGY BROADCAST ALGORITHM: VERSION 2

3.1 Introduction

The second algorithm requires slightly more communications and more time to complete. However it avoids long computations (e.g. finding lexmin inside nested "for loops" on line 7, Fig. 2) and uses less memory (e.g. no Paths() sets for neighboring nodes). It operates by finding the identities of the other nodes of the network in order of increasing distance. When a new identity K is first received at I, on a link (J,I) say, node I selects link (J,I) as part of the spanning tree rooted at K. This choice is not directly communicated to J; rather node I signals to its other neighbors that their links to I should not be part of the spanning tree rooted at K. Eventually node J realizes that information about K should be transmitted to I and it proceeds to do so.

3.2 Precise description of the algorithm

A description of the messages and data structures follows while pseudo code appears in figure 3. Contrary to the previous algorithm we require that messages arrive on a link in the order they were sent.

3.2.1 Messages and data structures

In addition to the Neighbors(.) messages which are as previously described, this algorithm use messages denoted by NOSPT=Set, where Set is a set of nodes. If that message is sent on a link (J,I), the spanning trees of the nodes in Set do not include link (I,J).

When a message is received on a link it is first placed in a first-in-first-out queue associated with the link and the code specified in figure 3 is executed. We assume that the type of message at the head of a queue can be examined without removing the message from the queue.

Each node I also keeps the following variables:

An integer counter called HOP.

Sets Nodes(dist) where dist is an integer. Nodes(dist) will contain the set of nodes at distance dist from I.

For each link J, sets of nodes In_SPT(J) and Not_SPT(J). If a node is in In_SPT(J) then link (J,I) is part of its spanning tree, while if it is in Not_SPT(J) then link (I,J) is not part of its spanning tree (note the distinction between (I,J) and (J,I)). The spanning trees are minimum hop spanning trees, with ties broken arbitrarily, depending of the order of the processing in line 4.

3.2.2 Proof of correctness

We prove the following theorem:

Theorem:
If any node in a connected component of the network starts the algorithm, then eventually all nodes in the component will stop, having received Neighbors(J) for all connected nodes and having set In_SPT(J) and Not_SPT(J) according to their definition.

As for the previous algorithm we first note that all connected nodes will start the algorithm if any node starts and we prove the following statements by induction on HOP, starting with HOP=0 just after initialization.

a) Node I has received Neighbors(K) for all nodes at distance less than HOP.
b) For all nodes K at distance not exceeding HOP, K is included in exactly one In_SPT(J). For that J, link (J,I) is on a shortest path between K and I.
c) Nodes(n) is the set of nodes at distance n from I, for all n <= HOP.
d) For all nodes K at distance less than HOP, node I has included K in Not_SPT(J) if I does not lie on the selected shortest path between K and J.
e) For all nodes K at distance n less than HOP, node K has sent Neighbors(K) to J if I is on the selected path between K and J. That message was sent before the n th NOT_SPT=. message (we number those messages 0,1,....) .
f) Node I has sent HOP + 1 NOT_SPT=. messages to each neighbor J. A node identity K is in the nth NOT_SPT= message to J if the distance

between I and K is n and if J does not lie on a selected shortest path between K and I.

g) Node I will not stop until it has transmitted all Neighbors(.) messages and set In_SPT(.) and Not_SPT(.) according to their definitions.

Again we assume the previous statements to hold for HOP=n at all nodes that have not stopped with a smaller HOP and show that at those nodes HOP will eventually reach n+1 and that assertions a) to g) still hold.

If the algorithm has not stopped, Nodes(HOP) is not empty (line 14) thus Nodes(HOP-1) is not (or will not be) empty at all neighbors. By (f) they will all send at least HOP messages and the test on line 3 will eventually be satisfied, leading to incrementing HOP on line 13. e) and the processing on line 17 guarantee that a) will remain valid after HOP is incremented.

If the Set included in the nth NOTSPT=Set messages included all nodes at distance n, it would be clear that b) and c) and the processing on lines 7 and 8 keep b) and c) valid when HOP is increased. However nodes at distance n that are missing from the nth NOTSPT= message to J must be in In_SPT(I) at J (line 11) and thus must be at distance less than HOP from I (by b)).

Line 9 and the validity of statement e) insure that of d) even after HOP is incremented. The validity of e) and f) is guaranteed by the processing on lines 11 and 12 and the validity of b), c) and d).

The validity of g) follows from the previous statement and a reasoning similar to the one for e) in section 2.2.2.

4 COMMUNICATION AND TIME COMPLEXITIES

We first examine the communication resources used by the algorithms. When topology is broadcast by flooding the network with Neighbors(I) messages (containing only I and the identities of the neighbors I) a total of E (N + E) node identities are transmitted in the network (i.e. about E**2) where N denotes the number of nodes and E the number of (one way) edges (those numbers and others below are only upper bounds if the network has disconnected components).

In the first version of the algorithm information about each remote edge is received by each node exactly once. The communication cost is (N-1) * (N + E) node identities, i.e. about N E, a substantial decrease. Although each node receives information about each edge exactly once, this algorithm does not achieve the minimum possible communication cost. The reason for this is that we require the edges to come in pairs ((I,J) and (J,I)), so that some redundant information is transmitted; the algorithm uses that redundancy to select the neighbors to whom information should be forwarded (in practice the messages would also contain information like link delay which is specific to each direction and will not be redundant).

An "two phase" algorithm could be used, where spanning trees are built using an efficient algorithm (e.g. following [Gal83] or [Hum83]) during the first phase, and topology is broadcast during the second phase. Their communication complexity in phase 2 would be inferior by about .5 * N * E to the one just derived, thus about .5 N E. The communication

requirements in phase 1 are x N log(N) for [Gal83] and x N**2 for [Hum83], where x denotes a small constant.

The second version requires the extra transmission of each node identity on E - N + 1 edges, so that the new total is E * (2 N - 1), about 2 N E. For dense networks this is about twice the cost of the original algorithm, but the relative increase is much smaller for sparse networks and for the case where the Neighbors(.) messages contain more information than just the edge destinations.

We now turn our attention to the time it takes for the algorithms to complete. The algorithms themselves do not require timing assumptions, but for the purpose of comparing them we will make some.

If one assumes that the transmission and processing of a message take constant time (a good model when the overheads involved are considered), then flooding, the two previous algorithms and phase two of the "two phase" method all require time of about x N, where x is a small constant (between 1 and 3, depending on the details of the model, e.g. if all nodes start simultaneously). For [Gal83] phase 1 takes about x N log(N). This has been lowered to O(N) in [Awe87] (this reference contains others on the subject), however the proportionality constant in that last result is unspecified and likely to be too large for N of practical interest. For [Hum83] the time involved is O(N**2).

Another timimg model is that processing is instantaneous but that the transmission of a message takes time linear with the message length. With that model the worst time for flooding, the two versions of the broadcast algorithms and phase 2 of the two phase algorithms become O(E), because information about all links may have to go through some

"bottleneck" link. Thus all algorithms considered here require time O(E) in this model, except the two phase version with [Hum83] which still requires O(N**2).

We conclude that the algorithms considered here and the two phase approach are substantially better than flooding. The two phase method

has somewhat better communication complexity than the broadcast algorithms introduced here, at the expense of much larger time complexity in some models. We note also that two phase algorithms where a single spanning tree (rather than one rooted at each node) is built in phase 1 result in concentrating traffic on a few edges, an undesirable feature. Also having minimum depth spanning trees help insure rapid delivery of messages to all nodes. This can be important when the spanning trees serve to transmit commands or requests from a node to all other nodes).

5. IMPLEMENTATIONS FOR CHANGING TOPOLOGIES

Often the network topology changes due to failures, repairs, introductions or removals of links and nodes. It is desirable that within a reasonable time interval without any topological change all nodes be made aware of the current topology. As there are many paths between any two nodes, one cannot necessarily count on the fact that the latest information received about a node was sent most recently, even if messages are received in first-in-first-out fashion on each link; methods are needed to distinguish recent information from old information. Most known methods involve a numbering of the update (for an exception, see [Spi86]). We discuss some issues associated with this technique.

There are basically two ways of numbering updates: the update number can be global for the whole network (e.g. [Fin79] or [Seg83]), or it can be associated with the identity of the node issuing the update.

In either case the update number is an unbounded function of time. Early network designs addressed this perceived problem by numbering modulo some number M that should be large enough to insure that all updates except the M/2 most recent ones have disappeared from the network (otherwise it becomes impossible to distinguish "new" from "old" modulo M). That is hard to guarantee, specially when parts of the network become temporarily disconnected and other time-based mechanisms must be added. That approach is implemented in the ARPANET and it has been reasonably successful [McQ80],[Per83].

A simpler approach is to just number messages sequentially, without modulo operation. A 32 bit number should be large enough even under the most optimistic system lifetime assumptions, while not adding that much communication overhead; a variable length format capable of representing arbitrarily large numbers could also be used.

With algorithms like those of sections 2 and 3, the mode of operation in networks with changing topology would be to restart the algorithm when a local change is detected, using a new global update number. Reception of an update with a number larger than the current one triggers all other nodes to also restart. One might object that rebuilding spanning trees from scratch and broadcasting the entire topology whenever a local topology change occurs is extremely wasteful. In fact a careful counting of the messages involved shows that the extra information is of the same order N E as is routinely exchanged between nodes in many

networks to keep track of the link loading conditions for routing purposes. Having spanning trees cuts down on the amount of those routine messages and may be beneficial overall.

Rather than building the spanning trees and broadcasting the topology on them as they are built, one might consider distributing information about changes to the topology by flooding whenever a change occurs, and using the topology tables to find spanning trees. Those would be used to broadcast the routine link loading information. There are two problems with this approach:

First, it essential that the spanning trees at all nodes be consistent (otherwise messages may never be delivered or may loop "forever"). Insuring consistency is possible, but doing so reliably necessitates algorithms similar to those building spanning trees !

Second, when flooding the network with new local information, a node typically uses a local (increasing) sequence number. When a node is restarted following a crash, or when replacing another node with the same identity, it is important the last sequence number used be correctly recalled. This can be done by relying on hardware features (such as saving the sequence number in non volatile memory, or having a time of day clock and using day and time as sequence number). Another approach is to use the distributed algorithm outlined below where upon restarting a node effectively learns from the other nodes its last used sequence number ([Per83] has offered a similar method).

The following description assumes that there is a reliable link protocol, but it can be easily extended should this assumption not hold. For the sake of simplicity we are only concerned with numbering the updates originating at a designated node, called the "source" node. We

assume that each update message carries an update number and we order the updates according to that number. It may also occur that two different update messages have the same number; in that case we order them in some way, e.g. lexicographically.

a) When the "source" node comes up it selects an update number (say 0). Anytime it wishes it can increment its update number, build an update message, save it in memory and send it on all adjacent links.
b) When a link comes up, the nodes at both sides exchange the update from the "source" (if any) that is stored in their memory.
c) When a node different from the "source" receives an update message it compares it to the one stored in memory (if any). If the new update is greater than the one in memory (or if there is none in memory), the new update is saved in memory and it is also transmitted to all other neighbors, otherwise the new update is just discarded.
d) An extra step is executed at the "source" node: if it receives an update greater than the last one sent (this will typically be the case when a node comes up again) it immediately selects an update number greater than the one in the new update and builds, stores and broadcasts a new update message.

It is easy to see that if, when the "source" comes up again, an obsolete version of its local topology (possibly with a larger number than what the source uses, or even the same number but a different content) exists somewhere in the network, then the source will learn of its existence and will issue a new update with a larger number that will eventually be adopted at all nodes.

REFERENCES

B. Awerbuch, "Optimal Distibuted Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and related problems", Extended Abstract, M.I.T. Laboratory for Computer Science, undated (1987 ?).

Y.K. Dalal, "Broadcast Protocols in Packet Switched Computer Networks", Pd.D. Dissertation, Digital Syst. Lab., Stanford Univ., Stanford CA, Tech. Rep. 128, Apr 1977.

S.G. Finn, "Resynch Procedures and a Fail-Safe Network Protocol", IEEE Trans. Commun., vol. COM-27, pp. 840-845, June 1979.

R.G. Gallager, "A Shortest Path Algorithm With Automatic Resync", unpublished note, March 1976.

R.G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees", ACM Trans. Program. Lang. Syst., vol. 5, pp. 66-77, Jan. 1983.

P.A. Humblet, "A Distributed Algorithm for Minimum Weight Directed Spanning Trees", IEEE. Trans. Commun., vol. COM-31, pp. 756-762, June 1983.

P.A. Humblet and S.R. Soloway, "Algorithms for Data Communication Networks - Parts 1 and 2", submitted for publication, 1986.

J.M. McQuillan, I. Richter and E.C. Rosen, "The New Routing Algorithm for the ARPANET", IEEE Trans. Comm., Vol. COM-28, May 1980.

R. Perlman, "Fault-Tolerant Broadcast of Routing Information", Proc. IEEE Infocom '83, San Diego, 1983. A. Segall, "Distributed Network Protocols", IEEE Trans. on Info. Theory, Vol. IT-29, no. 1, Jan. 1983.

A. Segall, "Distributed Network Protocols", IEEE Trans. on Info. Theory, Vol. IT-29, no. 1, Jan. 1983.

J. Spinelli, "Broadcasting Topology and Routing Information in Computer Networks", submitted for publication, 1986.

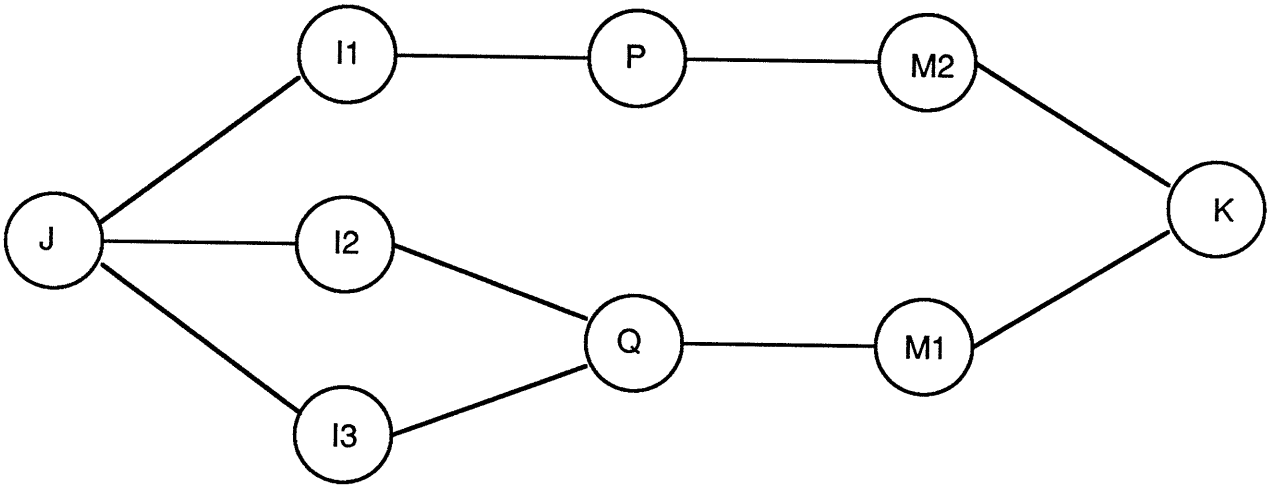

Figure 1.

Topology broadcast algorithm, version 1.

FIGURE 2: TOPOLOGY BROADCAST ALGORITHM, VERSION 1

```
Initialization at node I {

Neighbors(J) = {} V nodes J
Paths(I,0) = {I}
V links J {
        Neighbors(I) = Neighbors(I) U {J}
        SPT(J) = {I}
        Paths(J,0) = {J}
        }
Paths(I,1) = Neighbors(I)
HOP =1
send  Neighbors(I) on all links
if ( Neighbors(I) = {} ) Stop
}

On receiving Neighbors(J) at node I {

1 Save it
2 If (Neighbors(J) is not {} for all J in Paths(I,HOP)) {
3       Paths(I,HOP+1) = U Neighbors(K) for all K in Paths(I,HOP)
4       V links J do {
5               Paths(J,HOP) = U Neighbors(K) for all K in Paths(J,HOP-1)
6               V K in (Paths(I,HOP) \ ( U Paths(J,n) for n <= HOP ) do {
7                       M = lexmin ( Neighbor(K) Paths(J,HOP) )
8                       If (M is in SPT(J)) {
9                               SPT(J) = SPT(J) U {K}
10                              send Neighbors(K) to J
                }       }       }
11      HOP = HOP + 1
12      If (Paths(I,HOP) \ ( U Paths(I,n) for n < HOP) == {} ) stop
   }    }
```

FIGURE 3: TOPOLOGY BROADCAST ALGORITHM, VERSION 2

```
Initialization at node I {

Nodes(0) = {I}
Neighbors(I) = {}
for all links J {
       In_SPT(J) = Not_SPT(J) = {}
       Neighbors(I) = Neighbors(I) U {J}
       }
HOP = 0
send  NOTSPT={I} on all links
if ( Neighbors(I) = {} ) Stop
}

On receiving a message on link J at node I {

1 Place the message in queue J
2 Loop:
3 If ( ALL queues have a NOTSPT=. message at their heads ) {
4      Nodes(HOP+1) = {}
5      V link J {
6              dequeue the message NOTSPT=Set from queue J
7              In_SPT(J)  = In_SPT(J)  U  (Set \ ( U In_SPT(L) V L ))
8              Nodes(HOP+1) = Nodes(HOP+1) U  (Set \ ( U In_SPT(L) V L ))
9              Not_SPT(J) = Not_SPT(J) U Set
               }
10     V link J {
11             send to J Neighbors(K) V K in ( Nodes(HOP) \ Not_SPT(J) )
12             send to J NOTSPT=Nodes(HOP+1) \ In_SPT(J))
               }
13     HOP = HOP + 1
14     if ( Nodes(HOP) = {} ) STOP
15     goto Loop
       }
16 if ( NEIGHBORS(K) is at the head of a queue ) {
17     dequeue the message and save it
18     goto loop
   }   }
```