A Distributed Spanning Tree Algorithm for Topology-Aware Networks

Arjan J. Mooij  Niclae Goga  Wieger Wesselink
A.J.Mooij@tue.nl  goga@win.tue.nl  J.W.Wesselink@tue.nl

Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Keywords: Spanning trees, stabilization, distributed networks

Abstract
A topology-aware network is a dynamic network in which the nodes can detect whether locally topology changes occur. Modern networks, like IEEE 1394.1, are topology-aware networks. We present a distributed algorithm for computing and maintaining an arbitrary spanning tree in such a topology-aware network. Although usually minimal spanning trees are studied, in practice arbitrary spanning trees are often sufficient. Since our algorithm is not involved in the detection of topology changes, it performs better than the spanning tree algorithms in standards like IEEE 802.1. Because reasoning about distributed algorithms is rather tricky, we use a systematic approach to prove the correctness of our algorithm.

1. INTRODUCTION

A spanning tree of a connected graph is a connected acyclic subgraph that contains all the nodes of the graph. In the literature many algorithms are described for computing spanning trees. However, most of these algorithms consider the computation of the more specific minimum spanning trees. A minimum spanning tree of a connected weighted graph is a minimum-weight spanning tree of the graph. In many applications, it is sufficient to consider the weaker arbitrary spanning trees.

Such an application of arbitrary spanning trees can be found in (interconnected) Local Area Networks (LANs). A LAN, sometimes called a bus, is a network that interconnects a (usually) limited number of devices. To lift this limitation, the LANs themselves can be interconnected by a special device called bridge. It turns out to be useful to organize the LANs and bridges in a spanning tree, since it defines a (unique) path between any two nodes, which is handsome for e.g. routing messages between the LANs. Especially in case bridges can be added and removed dynamically, an distributed algorithm to (automatically) maintain the spanning tree is indispensable. Such an algorithm is sometimes called self-configuring in the sense that the algorithm maintains the spanning tree under additions and removals of bridges.

Examples of this bridge concept for interconnecting LANs can be found in the IEEE 802.1 MAC Bridge Standard and the upcoming IEEE 1394.1 Standard for High Performance Serial Bus Bridges. The spanning tree algorithm in the IEEE 1394.1 draft standard can (and is expected to) benefit from the IEEE 1394 standard feature that all devices on a bus are signalled whenever a device (e.g. a bridge) is added to or removed from the bus. This feature can be used to detect whether locally a topology change has occurred in the interconnected network by the dynamic addition or removal of a bridge. Since the IEEE 802 standard does not provide such a feature, the spanning tree algorithm in the IEEE 802.1 standard is also involved in the detection of topology changes in the network.

In this article we focus on networks in which nodes can locally detect whether additions and removals of a connected edge have occurred. We call such a network a topology-aware network. An IEEE 1394.1 interconnected network can be mapped on such a network by representing bridges as nodes, and by representing (physical) connections between pairs of bridges as edges.

We present a distributed algorithm for computing and maintaining an arbitrary spanning tree in a topology-aware network in which each node has a unique identity. The algorithm is not restricted to IEEE 1394.1 interconnected networks. Since the algorithm does not have to deal with the detection of topology changes, it performs better than the algorithm used in the IEEE 802.1 standard. Since we developed the algorithm from scratch, it is simpler than the algorithm in the IEEE 1394.1 draft standard. Since we consider arbitrary spanning trees, it is less centralized than the minimum spanning tree algorithms in the literature.

1 We studied this algorithm as a case study of the project “Improving the Quality of Protocol Standards”, funded by the NWO under project number 016.023.015.
We will pay much attention to the formalization of our algorithm. Our attempts [13, 14] to verify the spanning tree algorithm in the IEEE 1394.1 draft standard show that the verification of this kind of algorithms is non-trivial. A full verification using model-checking techniques (see [13]) turned out to be infeasible due to an enormous state space. But using manual formal development techniques, at least a variant of the algorithm could be proved (see [14]). Based on these experiences, we decided to proof the correctness of our algorithm in a systematic manual way.

Section 2 presents a summary of the literature regarding spanning trees. Section 3 describes our spanning tree algorithm for topology-aware networks in an informal way, including an example. Section 4 complements Section 3 by presenting an outline of a formal derivation of the algorithm, starting from its specification. Partial correctness, stabilization and complexity are also addressed in this section. Section 5 gives the conclusions.

2. SUMMARY OF THE LITERATURE

In this section we summarize some literature regarding spanning tree algorithms. There are general requirements about the existence of distributed algorithms for computing spanning trees. According to [5], it is possible to develop such a minimum spanning tree algorithm for graphs in which the minimum spanning tree is uniquely determined, and this is the case if all edge weights are distinct. [10] even states that for graphs with neither distinct edge weights nor distinct node identities, no such a spanning tree algorithm exists that uses a bounded number of messages. In what follows, we first discuss some algorithms for the two types of spanning trees, and afterwards we discuss the implications for our algorithm.

Many algorithms for minimum spanning trees are based on one of the two classical algorithms for computing spanning trees: Kruskal’s algorithm and Prim’s algorithm (see e.g. [5, 7]). Kruskal’s algorithm builds fragments of the spanning tree which are themselves spanning trees. Initially each node is a single node fragment; then the algorithm successively adds an edge with minimal weight that combines two disjoint fragments.

In Prim’s algorithm a single tree is formed. Initially a root node is chosen; then the algorithm successively adds an edge with minimal weight that extends the tree with a node that is not in the tree. Prim’s algorithm is more centralized than Kruskal’s algorithm. Because in practice distributivity is desired, Kruskal’s algorithm is usually preferred over Prim’s algorithm. Many distributed spanning tree algorithms have been derived from Kruskal’s algorithm (e.g. [6, 10]).

Many for arbitrary spanning trees are so-called self-stabilizing algorithms (e.g. [2, 15]). We will concentrate on the spanning tree algorithm in the IEEE 802.1 standard, which is described in [15, 16]. It is of special interest, because it is used on a large scale in practice for interconnecting networks; and thus it has the validation given by practice.

To compute (and maintain) the spanning tree, every few seconds each bridge broadcasts its unique identity. One bridge is elected as the root of the tree, namely the bridge with the minimal identity. Then the tree is constructed by including for each bridge a shortest path to the root. In the case of ties, the bridge with the smallest identity wins. Even after the spanning tree has been computed, the algorithm continues to periodically send messages in order to detect topology changes and to maintain the spanning tree.

The algorithm converges in time proportional to the diameter of the extended LAN, and requires a small amount of memory per bridge and communications bandwidth per LAN, independently of the size of the network. However, the actual performance of the algorithm depends on (worst-case) parameters of the underlying network. Some variants of this algorithm have been proposed in the literature (e.g.[4, 12, 9]).

Many proposed algorithms for building spanning trees in a distributed environment are not sufficiently detailed, e.g. [3]. These are described at a high level of abstraction based on the assumption that nodes are correctly informed about the failures of other nodes (the details of how this happens are not given). Exactly such skipped details are important to be given, because in practice it is not trivial to implement them.

For the many algorithms (e.g. [11, 17]) based on the Bellman-Ford algorithm there is another problem. This algorithm does not deal properly with loops and the counting to infinity-problem in distributed environments (see [11]). It is not easy to deal with this problem; it is even the main problem that we encountered in verifying the spanning tree algorithm from IEEE 1394.1. The existence of a loop can cause a node to increment its recorded distance to some destination node until it reaches a predefined maximum value. This is a detriment to the overall performance of the network. Although a number of attempts where made to solve this problem, it still receives attention from the scientific community.

For our algorithm, we do not consider a weighted graph, and hence we need to assume (and exploit) that all nodes have a unique identity. We developed an algorithm that is similar to the one used in the IEEE 802.1 standard, but targeted at modern topology-aware networks. This resulted in a better performance regarding the required number of messages and their size, and it avoids a continuous flux of messages from the root node to the other nodes like in [15, 16]. Our algorithm can also be considered as a variant of [4] such that it does not depend on the maximum possible network size.
3. THE ALGORITHM

In this section we present our spanning tree algorithm for topology-aware networks. The algorithm stabilizes if (during a sufficiently large period of time) no more topology changes occur. Since these topology changes can partition the network, we extend the notion of a spanning tree to a spanning forest, i.e. a (disjoint) collection of spanning trees.

We assume that the nodes in the network are computational units with a unique identity, and we assume that messages can be sent between neighbor nodes, i.e. nodes that are directly connected by an edge. To that end we consider the edges as buffered bidirectional first-in-first-out communication channels.

For simplicity reasons, we restrict the topology changes to additions and removals of edges such that between any two nodes there is at most one edge, and such that there are no self-loops. We expect that the algorithm can easily be extended for networks with self-loops and multiple edges, and for topology changes like the addition and removal of nodes.

The nodes in a topology-aware network can detect whether locally topology changes occur, i.e. additions or removals of connected edges. Each node eventually detects topology changes between the node and any other node, but independently of that other node, and before the node uses the edge (if any) between them for communication. When an edge is removed, all messages in the corresponding bidirectional communication channel (and buffers) are removed. Similar assumptions can also be found in other papers (e.g. [1, 6]), but usually there are some subtle differences.

To get an idea of the potential of the topology changes, we briefly consider them using the example networks in Figure 1. These networks contain the four nodes $a$, $b$, $c$ and $d$. The edges in the networks are represented by lines between the nodes, and the edges in the spanning forests are represented by arrows from nodes to their parent node (if any).

If an edge is added to the network, it might be necessary to extend the forest; if this is necessary it suffices to extend the forest with that edge (see cases 1 and 2). Removal of an edge from the network that is no part of the forest does not involve a change in the forest (see case 3). Removal of an edge that is a part of the forest involves removing that edge from the forest, therewith splitting a tree into two trees (see case 4).

However, if these two trees are in the same component of the network, the forest must be extended (see case 5). Note that nodes $a$ and $b$ cannot locally detect whether the removal of the edge between nodes $a$ and $b$ is a case 4 or a case 5 removal (see the dashed lines). In general this also holds for the addition of an edge.

Although it turns out to be relatively simple to develop an algorithm that maintains a spanning forest as long as the topology changes are restricted to additions of edges, it becomes far more complicated if edges can also be removed. This turns out to be a common problem in spanning tree algorithms for “dynamic” topologies, e.g. [4, 6].

We define a spanning forest on a network as a directed subgraph of the network in which all nodes are contained, in which each node has at most one parent, in which there are no cycles, and in which each two connected nodes in the network belong to the same tree in the forest. This specification of a spanning forest can be expressed more locally (see also Section 4.1) in terms of the following three local variables per node $v$:

- $parent.v$ of type node identity extended with a special element $\perp$. This variable is used to ensure that each node has at most one parent in the directed subgraph, and to identify roots in the directed subgraph. It indicates the neighbor (if any) of the node that is its direct parent in the directed subgraph. In case a node has no parent, i.e. it is a root in the directed subgraph, this variable has the special value $\perp$.

- $dist.v$ of type natural. This variable is used to ensure that the directed subgraph (in which each node has at most one parent) is a forest (i.e. it contains no cycles) by requiring $dist.(parent.v) < dist.v$ for non-root nodes $v$. Then we could derive (using the transitivity of $<$) the contradiction $dist.v < dist.v$ for all nodes $v$ in a cycle; so there are no cycles. In our algorithm, $dist.v$ can even be interpreted as the distance to the root of the tree.

- $root.v$ of type node identity. This variable is used to ensure that the forest is spanning, i.e. any two neighbor nodes belong to the same tree in the forest. To that end we assign unique identities to the trees in the forest. Using that each node has a unique identity we choose to identify the trees by the identity of their one-and-only root-node. So we require $root.v = v$ for each root-node $v$ of a tree, and $root.v = root.w$ for each two neighbor nodes $v$ and $w$ in the network.

Our algorithm mainly consists of two parts: a basic algorithm and a removal algorithm. The basic algorithm is a (relatively simple) local algorithm that maintains a spanning forest as long as the topology changes are restricted to the

\[ \text{Figure 1. Topology changes} \]
addition of edges. Upon removal of an edge, the removal algorithm intervenes such that afterwards the basic algorithm can again behave correctly.

In [1] a similar algorithm structure is proposed, namely to use a basic algorithm for static networks and requiring that upon each topology change a reset procedure is started. Exploiting the specifics of our application, we use a basic algorithm that can deal with at least the edge additions, and hence we only start such a reset procedure in some specific edge removal cases.

In the coming sections we first describe the basic algorithm and afterwards we extend it with the removal algorithm.

### 3.1 Basic algorithm

The basic algorithm has to maintain a spanning forest as long as no edges are removed. To that end neighbor nodes will send each other so-called M-messages (for normal Maintenance of the spanning forest) containing their \((\text{root}, \text{dist})\) value. Upon receipt of such a message \((r, d)\) from node \(w\) by node \(v\), node \(v\) can decide to assign \((\text{parent}_v, \text{root}_v, \text{dist}_v)\) the value \((w, r, d + 1)\) such that node \(w\) becomes its parent. To guarantee that the algorithm will stabilize, this assignment will only be performed if the receiving node’s \((\text{root}, \text{dist})\) value will really decrease, i.e. its root becomes smaller, or its root remains equal and its dist becomes smaller. Whenever a node changes its \((\text{root}, \text{dist})\) value, it will send an M-message to its neighbors.

We assume that initially the network and the spanning forest contain no edges, so each node is a root-node and for each node \(w\) we have \(\text{parent}_w, \text{root}_w, \text{dist}_w = \perp, v, 0\). During execution of the algorithm edges can be added. When an edge has been added, the nodes that are connected to that edge will be signalled by the topology-aware network. This signalling of a node must eventually result in the node sending an M-message over the edge to start the algorithm over the edge. So we obtain the following basic algorithm:

\[
\begin{align*}
\text{parent}_v, \text{root}_v, \text{dist}_v & = \perp, v, 0 \\
\text{do forever} & \\
& | \left( \left( \text{I arrived M-message from } w \right) \land \left( \text{no topology change with } w \text{ occurred} \right) \right) \rightarrow \text{receive the oldest M-message } (r, d) \text{ from neighbor } w \\
& \text{if } (r, d + 1) < (\text{root}_v, \text{dist}_v) \text{ then} \\
& \quad \text{parent}_v, \text{root}_v, \text{dist}_v := w, r, d + 1 \\
& \quad \text{send an M-message } (\text{root}_v, \text{dist}_v) \text{ to all neighbors except } w \\
& \text{else skip} \text{ fi} \\
& | \text{I a neighbor } w \text{ has been connected } \rightarrow \\
& \quad \text{send an M-message } (\text{root}_v, \text{dist}_v) \text{ to } w \\
\text{od}
\end{align*}
\]

If two actions are surrounded by a pair \(<\text{ and }\rangle\), these actions must be executed as a single atomic action. We use a repetition construct that is in fact a combination of an infinite repetition and a non-deterministic selection. The alternatives of this selection are separated by \([\text{-}]\)-markers and they have the following shape: guard \(\rightarrow\) statement. Note that after the initializing assignment, a node only becomes active upon an event like an arriving M-message or a topology change.

To improve the performance of the algorithm, and to keep the buffers of the incoming M-messages small, all M-messages from a node except the most recent message may be removed. Note that the \((\text{root}, \text{dist})\) value of a node never increases, and that handling an arrived M-message results in a decrease of this value or a decrease in the number of messages. Hence this algorithm stabilizes if (during a sufficiently large period of time) no more topology changes occur. We will discuss this in more detail in section 4.1.3.

### 3.2 Removal algorithm

In this section we extend the set of topology changes that we consider with the removal of edges. We generalize the procedure for handling the addition of an edge as described above, such that it can handle a mixture of additions and removals of an edge between two nodes. In this way it remains sufficient to detect whether at least one topology change has occurred between the node and another node, as long as it can also be detected whether there is currently an edge between them. So it is not necessary to detect each individual topology change.

Only when a node has handled all topology changes between the node and another node, the node may use the edge (if any) between them for communication. In particular, if an edge has been removed, attempts to send or receive messages via the edge must be ignored.

Upon a (physical) topology change between two nodes, all messages between those nodes are automatically removed. When the nodes handle a topology change, they must ensure that the edge between them is not in the spanning forest. If there is currently an edge between the two nodes, the nodes must send an M-message to each other to (re-)start the algorithm over the edge. So we generalize the corresponding part of the algorithm into:

\[
\begin{align*}
\text{if topology changes have occurred between nodes } v \text{ and } w \rightarrow \\
& \quad \text{"ensure that } w \text{ is not the parent of } v" \\
& \quad \text{send an M-message } (\text{root}_v, \text{dist}_v) \text{ to } w
\end{align*}
\]

Then we continue with “ensure that \(w\) is not the parent of \(v\)”. If \(w\) is currently the parent of node \(v\), the node must find another suitable parent or it must become a root-node. If the node “knows” (e.g. by storing the last arrived M-message per neighbor) a neighbor with a smaller \((\text{root}, \text{dist})\), the node can restore the parent relation by updating its \((\text{parent}, \text{root}, \text{dist})\) value such that the neighbor becomes its parent, while maintaining the descendence of its \((\text{root}, \text{dist})\) value. If a node must become a root-node, it must increase its \((\text{root}, \text{dist})\) value to \((v, 0)\), which potentially endangers stabilization. Below we will describe this case in more detail; so we obtain:
If a node must become a root-node (i.e. the last alternative), it must increase its \((\text{root}, \text{dist})\) value. This turns out to be allowed (see also section 4.2) if:

- all neighbors will send an M-message to the node;
- there are no children of the node; and
- all messages that were previously sent by the node have been removed.

In particular, all current children of the node must ensure that this node is not their parent. This recursive behavior leads to a removal algorithm that has a more global nature than the basic algorithm.

Recall that stabilization of the basic algorithm depends on the fact that the \((\text{root}, \text{dist})\) value is descending as long as there are no topology changes. To maintain stabilization under increasing \((\text{root}, \text{dist})\) values, such an increase is only allowed in a terminating algorithm that is initiated upon handling a topology change. Since the removal algorithm is initiated upon handling a topology change, what remains is to show its termination. We will deal with this after describing this removal algorithm.

One may wonder whether handling the removal of an edge can also be done using a local algorithm (like the basic algorithm). This turns out to be complicated without seriously endangering stabilization. Global behavior related to the removal of an edge that is in the spanning tree also occurs in algorithms like [6] (contributed to inconsistent identification of the trees) and [15] (contributed to \text{root} values that are nodes which are not in the tree).

If a node must become a root-node, it starts the removal algorithm by sending an R-message (for starting the Removal algorithm) to all non-parent neighbors. Then it waits to receive an ER-message (for indicating the End of the Removal algorithm) from all non-parent neighbors, indicating that the node can safely increase its \((\text{root}, \text{dist})\) value according to that node. After increasing its \((\text{root}, \text{dist})\) value, the basic algorithm must be restarted by sending an M-message to all neighbors. Thus we obtain:

\[
\begin{align*}
\text{If a node receives an R-message from its parent, it first completes handling the currently handled event (if any), and then it starts a similar algorithm, which is described below. If a node receives an R-message from a non-parent node, it delays handling the currently handled event for handling the R-message. In particular it must be able to handle such an R-message when it is executing the removal algorithm for its parent-node, even when the node has not yet handled a topology change on the edge (which can then easily be combined).}
\end{align*}
\]

Upon arrival of an R-message or an ER-message via an edge all previously arrived M-messages via the edge must be removed. This enables the removal algorithm to respond to these messages without being hindered by earlier arrived M-messages. Removing older M-messages is not harmful for the correctness of the algorithm, because both an R-message and an ER-message “promise” a new M-message.

Note that the removal algorithm is executed within a finite tree, since the R-messages are propagated over the child-relation, all nodes have at most one parent (in fact fixed during execution of the removal algorithm), and the initiator has no parent. Then termination is guaranteed since eventually the leaf-nodes are reached, which initiates the ER-messages being sent over the (former) parent-relation to the initiator. This completes the removal algorithm.

It is possible to combine some M-messages (by sending the M-message earlier) with the R- and ER-messages. This can reduce the number of messages that have to be sent at the cost of some extra buffering. In doing so the algorithm can even be adapted such that it saves some parts of the structure of the tree, by declaring the node that initiates the removal algorithm to be the root of the underlying tree structure, instead of decomposing the tree structure into singleton trees.

### 3.3 Example

To get an operational idea of how this algorithm can behave, Figure 2 describes some possible behavior of the algorithm. It is based on the topology changes that were described in Figure 1. We define the following total order on the node-identities: \(a \leq b \leq c \leq d\). The three-tuples represent \((\text{parent}, \text{root}, \text{dist})\) values (in which “⊥” is denoted by “\(\sim\)”), and the M-messages are labelled with their \((r, d)\) content. For the topology changes (and corresponding signalling) the following kind of abbreviations are used: “add \(a \sim d'\) for the “addition of an edge between nodes \(a\) and \(d'\), and “rem \(a \sim d'\) for the “removal of an edge between nodes \(a\) and \(d'\).”
3.4 Messages

We introduced three types of messages: M-, R- and ER-messages. We already noted that we can reduce the number of messages that are sent by the algorithm, by combining the R- and ER-messages with an M-message. Then we only need one type of message which contains the following fields:

- a type of the (combined) message;
- a root of type node identity;  and
- a distance of type natural.

To compare our algorithm with [15], recall that that algorithm continues to send messages periodically in order to detect topology changes. Since sending messages in our algorithm is always triggered by a topology change, the total amount of messages that sent by for our algorithm is lower in case there is a relatively small number of topology changes. Furthermore, the messages that are used by our algorithm are smaller than the ones used by [15].

3.5 Forwarding data

If a node is not involved in the removal algorithm, it can possibly do some higher-level functionality (e.g. forwarding data) using the currently existing forest. The “waiting time”, i.e. the period during which it cannot do so, is the duration of the removal algorithm. In our algorithm this is dynamically determined for each node that is involved in the removal algorithm using the ER-messages. In [15], for example, the waiting time depends on the (theoretical) maximum number of nodes in the network. Such a static waiting time usually involves a larger overhead than a dynamic one.

4. A PROOF

Complementary to the informal presentation of the algorithm in Section 3, in this section we prove the algorithm by sketching how it can be derived from its specification. The techniques that we use are based on [8], which in turn is based on Owicki-Gries theory. For proving the algorithm, we will annotate (versions of) the algorithm with assertions. An assertion is a predicate on the state space, and it cannot influence the control flow of the algorithm. To proof that an assertion is correct, additional assertions can be introduced (which correctness must afterwards be proved). According to the Owicki-Gries theory, an assertion in a node is correct if it is established by the preceding statement of the node and it is maintained by all statements in the other nodes. Progress issues are not covered by the Owicki-Gries theory; to guarantee stabilization of our algorithm we will use a variant function (sometimes called bound function).

We will derive the algorithm in different chunks than the ones we used to present it in section 3. We will first focus on handling the M-messages, which is the “core” of the algorithm. Then we will focus on initialization and handling the topology changes, including the removal algorithm. Note that the basic algorithm from section 3 consists of the core of the algorithm and handling the addition of edges.

To make the proof more compact, we will sometimes deviate from good derivation practice by making early design decisions, and skipping details of the proofs. Usually we will not explicitly mention the type of the variables, but as a convention we use $r, r', u, v$ and $w$ for nodes, $m$ and $m'$ for messages, and $d$ and $d'$ for natural numbers.
4.1 Core of the algorithm

In this section we will develop the core of the algorithm. We start by specifying formal requirements. Then we will develop a partially correct algorithm, i.e. an algorithm for which upon stabilization the requirements have been established. And finally we will adapt this partially correct algorithm to guarantee that it stabilizes, without (undesired) deadlocks.

4.1.1 Specification

We start the development with a formal specification. Recall that we consider a network in which each node has a unique identity, and in which there are no self-loops nor multiple edges. In section 3 we already informally explained how the variables parent, root and dist can be used to specify a spanning forest on such network. To formalize this, we introduce a binary irreflexive symmetric relation ∼, such that v ∼ w ≡ “v and w are neighbors in the network”. Then using (∀v : v ≠ ⊥) and parent.v =⊥ ≡ “v is a root-node”, the specification of a spanning forest can be rephrased as (the conjunction of):

0 : (∀v, w : parent.v ≠ w ∨ v ∼ w)
1 : (∀v, w : parent.v ≠ w ∨ dist.w < dist.v)
2 : (∀v : parent.v ≠ ⊥ ∨ root.v = v)
3 : (∀v, w : v ≠ w ∨ root.v ≤ root.w)

In requirement 3 one might expect the term root.v = root.w instead of the term root.v ≤ root.w. Both versions of requirement 3 are equivalent, but our choice is formally weaker and simplifies the rest of the derivation.

A distributed algorithm must be developed that stabilizes in a state satisfying these requirements if (during a sufficiently large period of time) no more topology changes occur.

We assume the following behavior of the (physical) topology changes: Upon addition or removal of an edge between nodes v and w, the corresponding bidirectional communication channel (and its buffers) are cleared, and both nodes v and w are signalled, all in one atomic step. Note that it is still the case that the two nodes independently detect whether they have been signalled.

To make this signalling more explicit, we introduce for nodes v and w boolean variables S.v.w and S.w.v in nodes v and w respectively, such that signalling corresponds to assignment S.v.w, S.w.v := true, true. A send or a receive over a channel v ∼ w by a node v reduces to a skip as long as S.v.w holds, i.e. as long as node v has received a signal from a topology change between nodes v and w which it has not yet “handled”. Node v can re-establish ∼S.v.w, by handling the topology changes between nodes v and w.

4.1.2 Partial correctness

The algorithm we want to develop has to establish these requirements upon stabilization. Recall that the algorithm we presented in section 3 stabilizes when all topology changes have been handled and all messages have been removed.

Before we continue, we will first introduce the following abbreviation for messages: v → w denotes that “there exists an M-message m from node v to node w”. We assume that a sent message will eventually arrive, unless a topology change occurs between the two nodes. To simplify the (future) system invariants, we split “receiving a message” from a communication channel into “reading the message” and “removing the message” from the communication channel.

By reading a message, a node can obtain the contents of the message, and by removing the message it is removed from the communication channel.

To develop an algorithm that establishes the requirements upon stabilization, we will weaken the requirements into a set of system invariants such that upon stabilization the requirements are implied. System invariants are assertions that always hold (after initialization).

We first weaken requirements 1 and 3, which contain variables of both node v and node w, with a disjunct (∃m : v m v w). Then for (the weakened) requirements 0, 1 and 3, which can be endangered by a topology change, we introduce a disjunct S.v.w if node v can easily restore the requirement upon an assignment S.v.w := false, and similarly for a disjunct S.w.v. So we will use the following system invariants:

P₀ : (∀v, w : parent.v ≠ w ∨ v ∼ w ∨ S.v.w)
P₁ : (∀v, w : parent.v ≠ w ∨ dist.w < dist.v ∨ (∃m : v m w v) ∨ S.w.v ∧ S.v.w)
P₂ : (∀v : parent.v ≠ ⊥ ∨ root.v = v)
P₃ : (∀v, w : v ≠ w ∨ root.v ≤ root.w ∨ (∃m : v m w v) ∨ S.w.v)

Note that when there are no more messages, and all topology changes have been handled, these invariants imply the requirements. Under maintenance of these invariants, the algorithm we will develop must remove messages and handle topology changes.

We first focus on removing (arrived) M-messages. Upon removing an M-message, maintenance of invariants P₁ and P₃ can be endangered due to disjunct (∃m : v m w v); this is our starting point of the development. By applying standard techniques and by making design decisions, we developed an algorithm that removes one M-message at a time by imposing the following invariant (related to P₁ and P₃) on the contents of the M-messages:
that the set of possible node identities is finite (it is contained within each of
its four components is bounded from below and as-

To assume that the addition is defined on node identities, in
fact we are only using addition as an abbreviation of concate-
tion with the lexicographical order.

We require this as condition on the arrived M-messages
in the second guard of the selection for future invariant

To be able to check the partial correctness proof more easily, we annotated the algorithm with assertions between
brackets \(\{\ldots\}\). Note that for maintenance of the invariants under assignment \(\text{parent} v, \text{root} v, \text{dist} v := w, r, d + 1\), we introduced three extra pre-assertions, and the \text{send}-statement. It turns out that we need disjunct \(\text{S} v w\) in the second guard of the selection.

\subsection{Stabilization}

We will modify the algorithm such that it is guaranteed to sta-
libilize when there are no more topology changes. Of course
this modification must be such that partial correctness is
maintained, which is guaranteed when we only restrict the
possible behavior of the algorithm. We will do so by strength-
ening the guards of the selection.

To ensure stabilization we can impose a well-founded
function on the state space of the system and adapt the algo-
rithm such that the function is a variant function (sometimes
called bound function) for the algorithm, i.e. it decreases in
each execution of the algorithm.

Using these techniques we will guarantee stabilization of
this algorithm. We impose as variant function the four-tuple
\((\text{\#v}, v : \text{S} v w), (\text{\sum} v : \text{root} v), (\text{\sum} v : \text{dist} v), (\text{\# messages in the system})\) with the lexicographical order, in which
\(\#\) is used as "the number of"-quantifier. Although we seem to
assume that the addition is defined on node identities, in
fact we are only using addition as an abbreviation of concatene-
tation with the lexicographical order.

To ensure that this function is well-founded we require that
each of its four components is bounded from below and as-
sume that there is a total order on the node identities. Note
that the set of possible node identities is finite (it is contained
in the set of node-identities and current \(\text{root} v\) values) and
hence bounded from below, and all \(\text{dist} v\) values are bounded
from below if we require system invariant:

\[ P_3 : (\forall v : 0 \leq \text{dist} v) \]

For maintenance of this invariant we require the following
system invariant:

\[ P_6 : (\forall d, r, v, w : v \rightarrow (r, d) \vdash \text{root} v, \text{dist} v < \text{root} v, \text{dist} v) \]

Thanks to invariant \(P_5\), this invariant is maintained by the
algorithm.

Before we ensure that the function is decreasing under ex-
ecution of the algorithm, note that the \text{remove}-statement in isolation decreases the function. So we only have to ensure \text{descendence} under the other statements. Note that the only
possibly increasing statement is the first guarded command
of the selection. Because in that case the number of mes-
cs can possibly increase, we require that \((\text{root} v, \text{dist} v)\)
decreases, by strengthening its guard \(r \leq \text{root} v\) into \((r, d + 1) < (\text{root} v, \text{dist} v)\). Thus we obtain:

\[ P_6' : (\forall d, r, v, w : v \rightarrow (r, d) \vdash \text{root} v, \text{dist} v < \text{root} v, \text{dist} v) \]

\subsection{Deadlock}

What remains is to ensure that the algorithm cannot deadlock
when it is handling an M-message. Therefore we have to en-
sure that we have as a pre-assertion of the selection that one of
its guards holds, i.e. the disjunction of the guards holds. Us-
ing some (elementary) calculation this can be simplified into
\(\text{parent} v \neq w \vee r < \text{root} v \vee d < \text{dist} v \vee \text{S} v w\). It turns out to be easier to deal with this condition by slightly strengthen-
ing it into \(\text{parent} v \neq w \vee (r, d) < (\text{root} v, \text{dist} v) \vee \text{S} v w\).

We require this as condition on the arrived M-messages \((r, d)\)
by requiring invariant \(P_7\):

\[ P_7 : (\forall d, r, v, w : v \rightarrow (r, d) \vdash \text{parent} v \neq w \vee (r, d) < (\text{root} v, \text{dist} v) \vee \text{S} v w) \]

Maintenance of this invariant under an assignment to
\((\text{root} v, \text{dist} v)\) is guaranteed since the M-messages are
received in descending order of their content, namely by re-
ceiving the oldest message first. For maintenance under the
creation of a new M-message (by node \(w\)) we require an ad-
ditional invariant
In case $S.v.w$ holds, node $v$ must establish $\neg S.v.w$ by handling the topology changes between nodes $v$ and $w$, but again under maintenance of the invariants. If there is currently an edge between those two nodes, invariant $P_3$ (and the corresponding assertions) can be maintained by sending an M-message over the edge. Invariants $P_0$, $P_1$, $P_7$, $P_8$ are maintained if the node ensures that $parent.v \neq w$ holds and by introducing an additional invariant $(\forall v, w : S.v.w \Rightarrow parent.w \neq v \lor S.w.v)$ for invariant $P_4$.

What remains is when a node $v$ has to establish $parent.v \neq w$, while currently $parent.v = w$ holds. If there is a known (from previously handled M-messages) neighbor $u$ with a smaller $(root.u, dist.u)$ than $(root.v, dist.v)$, that neighbor can become the parent after the node sent an M-message to all neighbors (invariants $P_0$, $P_3$, $P_8$ and $P_9$). Otherwise the node must become a root-node and hence (by invariant $P_2$) it must increase its $root$ value to $v$. Since this only occurs upon handling a topology change, it maintains descendence of the variant function; but it possibly endangers some invariants, so we require as pre-assertions:

- for $P_3$: $(\forall u : v \neq u \lor (\exists m : u \rightarrow v) \lor S.u.v)$
- for $P_8$: $(\forall u : parent.u \neq v \lor S.u.v)$
- for $P_9$: $(\exists m, u : v \rightarrow m)$

For the edge $v \sim w$ on which the topology change occurred that is being handled by node $v$, we introduce invariants $(\forall v, w : S.v.w \Rightarrow v \neq w \lor (\exists m : w \rightarrow v) \lor S.w.v)$ and $(\forall v, w : S.v.w \Rightarrow (\exists m : v \rightarrow m \lor S.w.v))$. Note that these invariants are both initialized and maintained.

Note that for invariant $P_8$ the current children of the node must also possibly increase their $root$. For stabilization this is only allowed upon handling a topology change; it is the removal algorithm that establishes the link with the topology change.

Since for invariant $P_0$ some M-messages might have been removed, the node must, after increasing its $root$, send its neighbors an M-message (invariant $P_2$). The rest of the details of the removal algorithm can be found in section 3.2.

4.3 Complexity

If there are no more topology changes or executing removal algorithms, the algorithm will stabilize from the root-nodes (i.e. the nodes with minimal identity). Note that a node with minimal identity will never change its $(root, dist)$ and hence will never send an M-message. When all messages sent by this node have been received by its direct neighbors, these neighbors will never send a message and hence the first node has stabilized; and so on. Thus the algorithm stabilizes in a time proportional to the maximal minimum distance from the root-node to another node. As with [15] this is proportional to the diameter of the network.
5. CONCLUSIONS

In this article we presented an algorithm for computing and maintaining a spanning tree in a topology-aware network, i.e. a network in which nodes can locally detect the addition and removal of a connected edge. The topology-awareness was used to clearly separate the detection of topology changes and the spanning tree algorithm. We included a formalization and a proof of the correctness of the algorithm.

Compared to similar algorithms that are used in practice for interconnecting LANs, the algorithm improves on performance, and in the waiting time in which data cannot be forwarded through the network. The algorithm does not require any configuration or tuning of network dependent parameters.

A number of assumptions were made about the topology changes in the network. For example, a recently added edge may not be used for communication by a node until the node has handled all topology changes. A topic for further work is to investigate whether these assumptions can be weakened. We expect that the algorithm can at least easily be extended for dealing with self-loops and multiple edges.

References