Calling Names on Nameless Networks*

BACHU SCHIEBER AND MARC SNIR

IBM Research Division, T. J. Watson Research Center,
P.O. Box 218, Yorktown Heights, New York 10598

We consider the problem of constructing a rooted spanning tree in an anonymous (connected) network. In case no upper bound on the network size is known, we give the following algorithms, all of which have error probability ε: (1) A message terminating algorithm that runs in \( O(n) \) time and \( O(m \log(n^7/m)) \) messages, each of size \( O(\log(n/\varepsilon)) \), where \( n \) and \( m \) are the number of nodes and links in the network. (2) A message terminating algorithm with expected running time \( O(n \log \log(n/\varepsilon)) \) and expected message complexity \( O(n \log n + m \log \log(n/\varepsilon)) \), each of size \( O(\log(n/\varepsilon)) \). For any fixed \( \varepsilon \), this algorithm can be modified to run in \( O(n f(n)) \) expected time and \( O(n \log n + mf(n)) \) expected message complexity, where \( f(n) \) is any slowly-growing function. However, this requires the use of longer messages. In case an upper bound on the network size is known, we give a processor terminating algorithm with error probability \( \varepsilon \) that runs in \( O(n) \) time, and \( O(n \log n + m) \) messages. Finally, in case the network size is known within a factor of 2, we give an algorithm that processor terminates and always succeeds, in expected \( O(n) \) time and \( O(n \log n + m) \) messages. © 1994 Academic Press, Inc.

1. Introduction

In a distributed algorithm a network of processors collaborate to solve a given problem. In this process, each processor acquires, via local interaction with its neighbors, some global knowledge of the network: e.g., the size of the network, its location in a minimal spanning tree, the distances to all other nodes, etc. Typically, one assumes that the nodes have distinct labels, which means that some global coordination of the processors has taken place beforehand. What happens if this assumption is dropped, so that the processors are indistinguishable? Consider, for example, regular networks of some fixed degree. The execution of a deterministic algorithm may end with all nodes in the same state, regardless of the network size or structure; a deterministic algorithm cannot distinguish between nodes of a regular network, nor can it distinguish between distinct regular networks of the same degree. (See Angluin [Ang80], Attiya et al. [ASW88], and

* A preliminary version of this paper appeared in “Proceedings of the 8th ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, August 1989.”
Yamashita and Kameda [YK88].) The situation is different if one assumes that each processor can make independent probabilistic choices; probabilistic choices can be used to break symmetry in a network of anonymous processors (i.e., processors with no distinct labels). For example, a probabilistic algorithm is used by Itai and Rodeh [IR81, IR90] to elect a leader in a ring of anonymous processors. The leader can next assign each processor on the ring a distinct label. In this work, we extend this result to arbitrary networks. We show that a rooted spanning tree can be computed with high probability in an anonymous network by a probabilistic distributed algorithm, with almost the same (expected) number of messages as needed in a labeled network. Thus, distinct labels can be assigned to the processors, and each node can acquire complete knowledge of the network configuration. In particular, any two non-isomorphic networks can be distinguished by a distributed probabilistic algorithm with high probability.

2. Model

We use the usual model of an asynchronous network. (See, e.g., Gallager et al. [GHS83], Segall [Seg83], and Awerbuch [Awe85].) The network is described by an undirected graph \( G = (V, E) \), where each node represents a processor and each edge represents a bidirectional communication link; \( |V| = n \), and \( |E| = m \). (For simplicity, parallel edges are not considered.)

The computation is message-driven. In a transition, a processor receives a message on one of its links, possibly sends messages on some of its links, and changes state. The transition may be probabilistic. All processors associated with nodes of the same degree have the same initial state. Startup messages may be initially present on some links. Messages are transferred on links in FIFO order, with no errors.

The course of an execution is determined by an adversary, or scheduler, that chooses the initial startup messages, and chooses at each step the next message to be received, as a function of the current network state. An algorithm processor terminates if in every execution all processors reach a special halting state. An algorithm message terminates if in every execution the network reaches a quiescent state where there are no pending messages on the links. In message termination, the processors may not "know" that the computation has halted. Define the propagation delay of a message as the difference between its arrival time and its transmission time, and the inter-message delay of a link as the difference between transmission times of any two consecutive messages on this link. Note that if an upper bound on both the propagation delay and the inter-message delay are given (as is usually the case in practical systems), message termination can be
transformed into processor termination. (However, this seems to be an unreasonable assumption in case an upper bound on the network size is not given.)

The worst-case message complexity of an algorithm (for a given network size) is the maximum over all networks of the given size of the largest number of messages sent in any execution of the algorithm on the network. The expected message complexity (for a given network size) is the maximum over all networks of the given size and over all schedulers of the expected number of messages sent in an execution with this scheduler on the network; the expectation is over the random choices of the algorithm. Note that the expectation is taken with respect to the worst-case network (of the given size) and scheduler.

The time complexity of an execution of an algorithm is the worst-case number of time units from the start to completion of the execution, assuming that the propagation delay and the inter-message delay of each link are at most one time unit. This assumption is introduced only for the purpose of performance evaluation; the algorithm must operate correctly with arbitrary delays.

The worst-case (expected) time complexity of an algorithm (for a given network size) is the maximum over all networks of the given size and over all schedulers of the (expected) time complexity of the execution of the algorithm on this network and with this scheduler; the expectation is over the random choices of the algorithm.

An algorithm has error probability (at most) \( \epsilon \) if for any scheduler and any network the probability that the algorithm terminates with the right answer is at least \( 1 - \epsilon \). An algorithm is a Monte Carlo algorithm if it has some positive error probability, but its complexity is bounded by a function of the network parameters and the error probability. An algorithm is a Las Vegas algorithm if it is always correct upon termination, and terminates with probability 1. This type of algorithm is useful if its expected running time is bounded.

3. Results

We consider the problem of computing two functions whose value depends on all the network nodes: counting the number of nodes in the network, and choosing a unique leader. Itai and Rodeh [IR81, IR90] considered this problem for the special class of a ring networks. They proved the following results. (1) These functions can be computed on a ring by a Monte Carlo algorithm that message terminates. However, they cannot be computed by a processor terminating Monte Carlo algorithm. (2) If an upper bound on the ring size is known, then these functions can
be computed by a Monte Carlo algorithm that processor terminates. (3) These functions can be computed by a Las Vegas algorithm that processor terminates if and only if the ring size is known within a factor of 2. (See also, Abrahamson et al. [AAHK86], and Attiya et al. [ASW88].)

We extend all these results to arbitrary networks, while improving some of the bounds. We consider the problem of computing a rooted spanning tree in a connected network. The root of the spanning tree can be elected leader; while the tree is built, one can also count the number of nodes in the network, and assign distinct labels to the nodes, according to their position in the tree.

1. We give a linear time message terminating Monte Carlo algorithm that computes a rooted spanning tree in a (connected) network (Section 4). The algorithm runs in $O(n)$ time and $O(m \log(n^2/m))$ messages, each of size $O(\log(n/e))$, where $e$ is the error probability. This implies a message terminating leader election algorithm on a ring of unknown size with error probability $e$ in $O(n \log n)$ messages, an improvement on the $O(en^2)$ bound of Itai and Rodeh [IR81, IR90]. (Since the algorithm is message terminating the leader does not "know" that it is the leader. It is impossible to achieve a processor terminating algorithm when no upper bound on the network size is known.)

2. We present a second message terminating spanning tree algorithm (Section 5) which is slower than the previous, but has a better expected message complexity. Its expected running time is $O(n \log \log(n/e))$ and its expected message complexity is $O(n \log n + m \log \log(n/e))$. The messages are of size $O(\log(n/e))$. For any fixed error probability $e$, the expected running time and message complexity can be reduced to $O(nf(n))$ and $O(n \log n + mf(n))$, respectively, where $f(n)$ is some slowly growing function, such as $\log^*(n)$, or $\alpha(n)$ ($\log^*(\cdot)$ is the number of times the log function has to be iterated to get a constant result, and $\alpha(\cdot)$ is the functional inverse of Ackermann's function). However, this requires the use of longer messages of size $O(\log(x/e))$, where $x$ is the maximum integer for which $\lceil f(x) \rceil = \lceil f(n) \rceil$. (Note that this algorithm is neither Monte Carlo nor Las Vegas since it may give a wrong answer, and its running time is not guaranteed.)

3. We consider networks with bounded size (Section 6). In case an upper bound $N$ on the network size is known, we give a Monte Carlo spanning tree algorithm with error probability $e$ that processor terminates in $O(n)$ time, and $O(n \log n + m)$ messages each of size $O(\log(N/e))$. If the

---

1 Throughout the paper "log" denotes the base 2 logarithm and "ln" denotes the natural logarithm.
network size is known within a factor of 2, we give a Las Vegas spanning tree algorithm that processor terminates in expected $O(n)$ time and $O(n \log n + m)$ messages.

4. The Linear Time Spanning Tree Algorithm

We assume in this section that no bound is known on the size of the graph $G$. Throughout the paper, we consider only the case when $G$ is connected. The algorithms and their analysis can be easily extended to handle disconnected graphs.

We start by giving some motivation for our algorithm. Our algorithm is an adaptation of the counting algorithm of Awerbuch [Awe87]. The counting algorithm elects a leader and computes a spanning tree in a labeled network (i.e., a network in which each node has a distinct label). The counting algorithm maintains a rooted spanning forest of $G$. Following the terminology of Gallager et al. [GHS83], each rooted tree in the forest is called fragment, and the root of each fragment is called the core of the fragment. The initial spanning forest consists of each of the nodes as a fragment of size 1. Upon termination the forest consists of only one fragment (tree). The core of this fragment is taken to be the leader of $G$.

Had the network been labeled, we could just use the counting algorithm. Since the network is anonymous, we randomly assign a label to each node in the graph. Had the probability of assigning the same label to two distinct nodes been very low, we could again use the counting algorithm with a high probability of success. However, this cannot be achieved a priori, since the network size is not known. Instead, we recompute labels for fragments as they grow; whenever the fragments' labels change, nontree links are scanned anew. This method has the disadvantage that nontree links have to be scanned more than once, resulting in a high message complexity. Our second algorithm alleviates this problem.

Several subtle points have to be taken care of. (1) The links of the graph cannot be assigned names that are guaranteed to be distinct; thus, a fragment cannot select, say, its minimum outgoing link. Instead, it selects an outgoing link according to the identity of its adjacent fragments. (2) Special care has to be taken to prevent deadlocking cycles. These cycles are avoided in the counting algorithm of Awerbuch [Awe87] using the distinct labels. (3) Nontree links that have been scanned and found to lead to a fragment with the same identity cannot be discarded as internal; they may have to be rescanned, since they may lead to a different fragment that happened to have the same signature at the time of the scan. "Eager" rescanning decreases the error probability, but increases the number of
messages; these two measures have to be carefully balanced. (4) A component that has exhausted all its nontree links may be reactivated.

In the next subsections we give a full description of the algorithm, prove its correctness, and analyze its performance.

4.1. High Level Description

The algorithm maintains a rooted spanning forest of $G$. Initially, the spanning forest consists of each of the nodes as a fragment of size 1. At each node $v$ two variables are maintained: $\text{Level}(v)$ and $\text{Label}(v)$. These variables contain information about the fragment to which $v$ currently belongs. Suppose that $v$ belongs to fragment $T$.

$\text{Level}(v)$ is the integer part of the (base two) logarithm of the (estimated) number of nodes in $T$.

$\text{Label}(v)$ is the label of $T$.

Remark. We sometimes refer to the above variables as the variables associated with a fragment, namely, $\text{Level}(T)$, and $\text{Label}(T)$.

Define the signature of a node $v$ to be the pair $(\text{Level}(v), \text{Label}(v))$. By our definition, two nodes that belong to the same fragment $T$ have the same signature, unless either $\text{Level}(T)$ or $\text{Label}(T)$ is being updated, in which case some of the nodes may still have the old values, while others have the updated ones. However, since the labels are not guaranteed to be distinct, two nodes from different fragments may also have the same signature.

We define a lexicographic order on the signatures in the natural way.

Initially, for each node $v$, $\text{Level}(v) = 0$, and $\text{Label}(v)$ is drawn independently and randomly from the uniform distribution on some domain of size $d$. (The value of $d$ is given in the analysis.)

In the course of the algorithm fragments are expanding and clashing with adjacent fragments. In such clashes higher level fragments invade lower level fragments, capturing their nodes. Captured nodes become part of the higher level fragment. A fragment may be invaded by more than one fragment; then each captures some of its nodes.

The algorithm performed by each fragment $T$ can be viewed as an application of three procedures: Link-Scan, Update, and Marriage. In the procedure Link-Scan a link outgoing from a node in $T$ to a node in another fragment is sought. In the procedure Marriage fragment $T$ attempts to merge with fragment $T'$ (where $\text{Level}(T) = \text{Level}(T')$) through a selected feasible link. In the procedure Update the number of nodes in $T$ is counted and $\text{Level}(T)$ is updated accordingly; that is, $\text{Level}(T)$ is set to be the integer part of the logarithm of the number of nodes in $T$. During the update the fragment is locked. When a fragment is locked it does not respond to any messages from adjacent fragments. These messages are queued for a possible later response.
Below, we describe each of these procedures. The terminology used in the description is taken from Gallager et al. [GHS83].

The Procedure Link-Scan. This procedure is activated when the algorithm starts and after each successful termination of the procedure Update. The procedure consists of three stages: In the first stage each node in \( T \) tries to invade all its adjacent fragments with lower level. In the second stage each node searches for an outgoing link called a feasible link. In the third stage the core of each fragment selects a single link among the feasible links found by the nodes.

We start with the first stage. Suppose that a node \( v \) receives a message from the core of its fragment to start the procedure. Node \( v \) maintains a queue with its nontree links. The first links in the queue are links that have not been scanned yet. The last links are links that have been scanned and found to lead to fragments with the same signature. These links are ordered in the order in which they were scanned.

Node \( v \) scans the links in its queue one by one. The scan of a link \( (v \rightarrow u) \) is done by sending a test message along the link asking for the signature of \( u \). Whenever a link \( (v \rightarrow u) \) such that \( \text{Level}(u) < \text{Level}(v) \) is found, the lower level fragment containing \( u \) is invad by \( T \) (the fragment containing \( v \)). That is, a connect message is sent along \( (v \rightarrow u) \). When this message is received (and acknowledged) by \( u \) link \( (v \rightarrow u) \) becomes a tree link of \( T \), \( u \) is appended to \( T \) (and removed from its previous fragment), and starts executing Link-Scan. In case a link \( u \) is locked for update when it receives the connect message, the acknowledgment of this message is postponed until the update is done. Then, the connect message is acknowledged only if \( \text{Level}(u) \) is still less than \( \text{Level}(v) \).

In the second stage \( v \) scans its links again. This time it searches for links \( (v \rightarrow u) \) such that \( \text{Level}(u) \geq \text{Level}(v) \). Suppose that such a link \( (v \rightarrow u) \) is found. There are two possibilities:

1. The signature of \( u \) is identical to the signature of \( v \). In this case \( v \) moves the link \( (v \rightarrow u) \) to the rear of the queue, and continues to the next link. (Since it is possible that \( u \) and \( v \) belong to different fragments that happen to have the same signature, the link \( (v \rightarrow u) \) is not discarded from \( v \)'s queue.)

2. The signature of \( u \) is different from the signature of \( v \). In this case the scan is stopped. The link is defined to be a feasible link of \( v \).

Node \( v \) terminates the second stage when either a feasible link is found or all links have been scanned at \( v \)'s current level.

The first two stages are implemented using the communication pattern described in Gallager et al. [GHS83]. We remark that although the
network considered by Gallager et al. [GHS83] is labelled, while the network considered here is anonymous, the implementation of Gallager et al. [GHS83] is still valid here since the tree and nontree links are well defined, and this is the only requirement of the implementation.

In the third stage the core selects a feasible outgoing link if such exists as follows. Whenever a leaf finishes the second stage it sends a report message with the results of its search to its parent. The message consists of the feasible outgoing link if such was found and “failure” otherwise. An internal node \( v \) collates the results of its own search with the report messages of all its descendants. If no feasible link was found either by it or by its descendants, then a failure message is sent to the parent. Otherwise, a message is sent that reports a feasible link with maximal signature. The core collates the results of its own search with the results of all its descendants. If no feasible link is found by the core, or a feasible link to fragment \( T' \) such that \( \text{Level}(T') > \text{Level}(T) \) is detected, then \( T \) becomes idle. Otherwise, that is, all the feasible links are outgoing to fragments with the same level as \( T \), the link with maximal signature is selected as the feasible link of \( T \), and the procedure Marriage is activated.

In this way, the core eventually gets information and selects a feasible link with maximal signature among the feasible links scanned, if such a link exists. Suppose that this link is outgoing to a fragment \( T' \). Note that the signature of \( T' \) is different from the signature of \( T \). However, \( T' \) is not necessarily the fragment with the maximal signature that is connected to \( T \), since it may be that the links outgoing to the fragment with the maximal signature have not been scanned.

The third stage is implemented using a communication pattern similar to the one described in Gallager et al. [GHS83]. The same pattern is used also by Awerbuch [Awe85, Awe87] and referred to as convergecast.

The Procedure Marriage. Suppose that the feasible link of \( T \) is outgoing to fragment \( T' \) (where \( \text{Level}(T) = \text{Level}(T') \)). In the procedure Marriage fragment \( T \) attempts to merge with fragment \( T'' \). The behavior of \( T \) depends on the relation between \( \text{Label}(T) \) and \( \text{Label}(T') \). If \( \text{Label}(T) < \text{Label}(T') \), then \( T \) sends a “marriage proposal” on the feasible link that connects it to \( T' \). Otherwise, it becomes inactive, waiting for a “marriage proposal.” The marriage message is sent along the tree links to the core of \( T' \). If the core of \( T' \) is not executing the procedure Marriage when it gets the message, it just buffers the message until it executes the procedure. Otherwise, the core accepts the proposal (and replies to it), if it is waiting for a proposal (and has not already accepted another proposal). If the proposed of \( T \) is accepted, then \( T \) invades \( T' \); that is, a connect message is sent along the feasible link and the tree links of \( T' \) to the core \( T' \). When the core of \( T' \) receives this message, it broadcasts a message to
all its descendants, indicating that the new fragment signature is the signature of $T$ and the new core is the core of $T$. When the receipt of this message is acknowledged by all nodes in $T'$, the procedure Update is activated.

**The Procedure Update.** When this procedure is activated, the number of nodes in $T$ is counted and $\text{Level}(T)$ is updated accordingly; that is, $\text{Level}(T)$ is set to be the integer part of the logarithm of the number of nodes in $T$. The label of (all the nodes in) $T$ is redrawn from the domain $d^{2 \cdot \text{Level}(T)}$. The procedure is aborted without any updating, if $T$ is being invaded by other fragments at the time Update is executed; in this case $T$ becomes *idle*.

The procedure is implemented using a locking mechanism as follows. First, an attempt is made to lock all the nodes. This is done by broadcasting a *lock* message along the tree links from the core. A node becomes locked if it has not yet been invaded by another fragment. A locked node does not respond to any messages sent from adjacent fragments, and thus, it cannot be invaded until it is unlocked. When a leaf gets a lock message it sends a *report* message to its parent reporting whether it is locked or not. An internal node $v$ collates the report messages of all its descendants and sends a report message to its parent reporting whether all its descendants (including itself) are locked. The core collates the report messages from its children. If all the nodes are locked, it computes the size of the fragment and the new level, and broadcasts it to all the nodes. Upon changing their level the nodes also become unlocked, and start executing the procedure Link-Scan. If not all the nodes become locked, then an unlock message is broadcast, unlocking the nodes.

4.2. Correctness

**Theorem 4.1.** The algorithm eventually message terminates. Upon termination all the fragments are idle and have the same signature.

*Proof.* The level of a node never decreases; a fragment of level $i$ contains at least $2^i$ nodes, so that level is bounded by $\log n$. Thus, the computation eventually reaches a point when no node further increases its level. Consider the computation at the earliest such point. It is easy to see that from this point no fragment executes Update, since this procedure is eventually followed by an increase in the level of the fragment.

Next, we show that at this point no fragment $T$ has a link outgoing to a fragment of lower level. To obtain a contradiction, suppose that at this point there exists a link $(v \rightarrow u)$ (for $v$ in $T$), such that $\text{Level}(u) < \text{Level}(v)$. If $T$ is currently executing the procedure Link-Scan, then link $(v \rightarrow u)$ will eventually be scanned by $v$, followed by an increase in the level
of \( u \), a contradiction. If \( T \) is currently executing another procedure, then it already executed the procedure Link-Scan at its current level. (Note that whenever a level of a node changes it executes Link-Scan.) However, in this case, when \( v \) scanned the link \((v \rightarrow u)\) during the execution of Link-Scan at its current level, \( \text{Level}(v) \) must have been greater than \( \text{Level}(u) \) (since the levels never decrease). This implies that the level of \( u \) would have been increased, contradicting our assumption. Since the graph is connected, we conclude that \textit{all fragments have the same level}.

Now, we show that from this point no fragment executes the procedure \textit{Marriage}. Suppose that some fragment \( T_1 \) is executing the procedure \textit{Marriage}, after selecting a feasible link outgoing to \( T_2 \). Since \( \text{Label}(T_1) \neq \text{Label}(T_2) \), \( T_2 \) also has a feasible link (leading to \( T_1 \)), and is not idle. It follows that \( T_2 \) eventually selects a feasible link outgoing to a fragment of the same level (since at this point all fragments have the same level). Thus \( T_2 \) eventually executes the procedure \textit{Marriage}. We distinguish between two cases:

\textbf{Case 1.} \( \text{Label}(T_2) > \text{Label}(T_1) \). In this case, fragment \( T_1 \) has sent a marriage proposal to \( T_2 \) while executing the procedure \textit{Marriage}. This proposal will be considered by fragment \( T_2 \) while it is executing the procedure \textit{Marriage}. If the proposal of \( T_1 \) is accepted by \( T_2 \) then a marriage eventually occurs, and the level of some node increases, a contradiction. The proposal of \( T_1 \) is not accepted only if \( T_2 \) either has accepted a proposal, or has an outstanding proposal to a fragment \( T_3 \) where \( \text{Label}(T_3) > \text{Label}(T_2) \). In the first case, a marriage eventually occurs, again, a contradiction. In the last case, a marriage eventually occurs, unless the proposal of \( T_2 \) is not accepted. The proposal of \( T_2 \) is not accepted only if \( T_3 \) either has accepted a proposal, or has an outstanding proposal to a fragment \( T_4 \) where \( \text{Label}(T_4) > \text{Label}(T_3) \). We thus obtain that if a marriage does not occur eventually, there must be an infinite chain \( T_1, T_2, T_3, \ldots \), such that \( \text{Label}(T_1) < \text{Label}(T_2) < \text{Label}(T_3) < \cdots \), a contradiction.

\textbf{Case 2.} \( \text{Label}(T_2) < \text{Label}(T_1) \). In this case, fragment \( T_1 \) becomes inactive, waiting for a marriage proposal. Consider the feasible link of \( T_2 \). If this link is outgoing to a fragment with a higher label, then we are back to Case 1, where \( T_2 \) plays the role of \( T_1 \). Otherwise, this link is outgoing to a fragment \( T_3 \) such that \( \text{Label}(T_3) < \text{Label}(T_2) \); then consider the feasible link of \( T_3 \) while it eventually executes the procedure \textit{Marriage}. Again, if this link is outgoing to a fragment with a higher label, then we are back to Case 1 where \( T_3 \) plays the role of \( T_1 \). Otherwise, this link is outgoing to a fragment \( T_4 \) such that \( \text{Label}(T_4) < \text{Label}(T_3) \), then consider the feasible link of \( T_4 \), and continue in the same manner. We thus obtain that if are not back to Case 1, there must be an infinite chain
$T_1, T_2, T_3, \ldots$, such that $Label(T_1) > Label(T_2) > Label(T_3) > \cdots$; a contradiction.

It follows that at this point in the computation, each fragment is either idle, or is executing the procedure Link-Scan. However, if it is executing Link-Scan it must eventually become idle or otherwise, it would execute the procedure Marriage in contradiction to our assumption (since all feasible links must be outgoing to fragments with the same level). Thus, eventually all fragments are idle. □

Below, we consider the error probability of our algorithm. Let $d = \lceil 2/e \rceil$. Recall that the labels are chosen from a domain of size $2^\text{Level}(T)$.

**Theorem 4.2.** When the algorithm terminates, the probability that a spanning tree is found is at least $1 - \epsilon$.

**Proof.** At each step of the algorithm each node belongs to some fragment. Thus, if upon termination there is only one fragment, then this fragment must be spanning tree. Suppose that the algorithm terminates with more than one fragment left. By Theorem 4.1 we have that all these fragments have the same level and the same label. The probability that all drawings at level $j$ selected the same label (given that more than one drawing occurred) is bounded by $1/(d2^j)$; the probability that all drawings at the last level selected the same label (given that more than one drawing occurred) is bounded by $\sum_{j=0}^{\infty} 1/(d2^j) = 2/d \leq \epsilon$. □

### 4.3. Complexity

**Theorem 4.3.** The number of messages sent by the algorithm is bounded by $O(m \log n)$.

**Proof.** Following Awerbuch [Awe87], we divide the messages into two categories: *exploration* messages are those that are sent in the procedures Link-Scan and Marriage over the nontree links; *control* messages are all the rest. We show that the number of *control* messages is $O(n \log n)$ and the number of *exploration* messages is $O(m \log n)$.

To bound the number of *control* messages observe that all the *control* messages are sent over tree links. Since the number of broadcasts and convergences in each procedure is constant, the total number of *control* messages sent over tree links in all fragments with the same level is linear in $n$. Since the number of levels is bounded by $\log n$, the number of *control* messages is bounded by $O(n \log n)$.

To bound the number of *exploration* messages observe that for each distinct value of $\text{Level}(v)$, each nontree link $(v \rightarrow u)$ is scanned at most twice by the procedure Link-Scan, and at most once by the procedure Marriage. The bound follows since the maximum level of a fragment is $\log n$. □
Theorem 4.4. The algorithm reaches message termination is $O(n \log n)$ time.

Proof. The proof is similar to the proof in Awerbuch [Awe87]. Let $t_k$ be the last time that an active fragment (i.e., a fragment that is neither idle nor inactive while executing the procedure Marriage) at level $k$ exists in the network. We claim that $t_k = O(kn)$. Since the maximum level of a fragment is $\log n$, the claim implies that at $t_{\log n} = O(n \log n)$ no active fragments exist. The theorem follows. To prove the claim we show by induction that $t_k - t_{k-1} \leq cn$, for some constant $c$, and any $k \geq 1$, where $t_0 = 0$. When a fragment $T$ of level $k$ is created it first executes the procedure Link-Scan. Then, if $T$ has not been invaded, either it becomes idle, or one of the procedures Marriage, Update is activated. Since each node is incident to fewer than $n$ links, the procedure Link-Scan takes $O(n)$ time for exploring the nontree links, and an additional $O(n)$ time for broadcast and convergecast over the tree links, if we ignore the time spent waiting for locked fragments. Notice that fragments are waiting only for locked fragments of lower level, thus after $t_{k-1}$ time fragments of level $k$ never wait. The procedures Update and Marriage take $O(n)$ time for broadcast and convergecast over the tree links. After this time $T$ either becomes inactive, or the level of $T$ increases.

4.4. The Modified Algorithm

To get linear running time we would like $t_{\log n}$ to be $O(n)$ and not $O(n \log n)$. For this $t_k - t_{k-1}$ should be bounded by $c2^k$, for some constant $c$. Below, we show how to modify the procedures Link-Scan and Marriage so that the execution of each of the procedures by fragment $T$ takes $O(2^{\text{Level}(T)})$ time. These modifications also slightly improve the message complexity of the algorithm.

The Modified Procedure Link-Scan. In order to make the execution time of the procedure $O(2^{\text{Level}(T)})$ we make two changes. First, the scan is stopped at nodes whose distance from the core is $2^{\text{Level}(T)} + 1$. In this way, the amount of time spent in the communication between the core and the nodes of $T$ is bounded. Second, each node in $T$ scans up to $2^{\text{Level}(T)} + 1$ nontree links. Thus, the amount of time spent during the scan is also bounded. Because of the second change, fragment $T$ may not scan all of its nontree links at its current level. To guarantee that before $T$ becomes idle it scanned at least one link outgoing to another fragment (if such exists) in its current level, the number of nodes is recomputed after an unsuccessful scan. Fragment $T$ becomes idle only if this number is less than $2^{\text{Level}(T)} + 1$. In this case, since each node in $T$ scans up to $2^{\text{Level}(T)} + 1$ nontree links, at least one of these links has to go to another fragment (if such exists).
Below, we give a detailed description of the changes. In the description we italicized all the modifications to the original procedure. The procedure is activated when the algorithm starts and after each successful termination of the procedure Update. The procedure consists of three stages: In the first stage each node in \( T \) tries to invade all its adjacent fragments with lower level. In the second stage \( v \) searches for an outgoing link called a feasible link. In the third stage the core of each fragment selects a single outgoing link among the outgoing links found by the nodes.

We start with the first stage. Suppose that a node \( v \) receives a message from the core of its fragment to start the procedure. If its distance from the core is \( 2^{\text{Level}(v)+1} \), then node \( v \) returns a message to its parent indicating this fact, and stops. Suppose that this is not the case. Node \( v \) maintains a queue with its non-tree links. The first links in the queue are links that has not been scanned yet. The last links are links that has been scanned and found to lead to fragments with the same signature. These links are ordered in the order in which they were scanned.

Node \( v \) scans up to \( 2^{\text{Level}(v)+1} \) links in its queue one by one. (In case its queue contains more than \( 2^{\text{Level}(v)+1} \) links it scans only the first ones.) The scan of a link \((v \rightarrow u)\) is done by sending a test message along the link asking for the signature of \( u \). Whenever a link \((v \rightarrow u)\) such that \( \text{Level}(u) < \text{Level}(v) \) is found, the lower level fragment containing \( u \) is invaded by \( T \) (the fragment containing \( v \)). That is, a connect message is sent along \((v \rightarrow u)\). When this message is received (and acknowledged) by \( u \), link \((v \rightarrow u)\) becomes a tree link of \( T \), \( u \) is appended to \( T \) and starts executing Link-Scan. In case a link \( u \) is locked for update when it receives the connect message, the acknowledgment of this message is postponed until the update is done. Then, the connect message is acknowledged only if \( \text{Level}(u) \) is still less than \( \text{Level}(v) \).

In the second stage \( v \) scans up to \( 2^{\text{Level}(v)+1} \) links again. This time it searches for links \((v \rightarrow u)\) such that \( \text{Level}(u) \geq \text{Level}(v) \). Suppose that such a link \((v \rightarrow u)\) is found. There are two possibilities:

1. The signature of \( u \) is identical to the signature of \( v \). In this case \( v \) moves the link \((v \rightarrow u)\) to the rear of the queue, and continues to the next link. (Since it is possible that \( u \) and \( v \) belong to different fragments that happen to have the same signature, the link \((v \rightarrow u)\) is not discarded from \( v \)'s queue.)

2. The signature of \( u \) is different from the signature of \( v \). In this case the scan is stopped. The link is defined to be a feasible link of \( v \).

Node \( v \) terminates the second stage when either a feasible link is found or up to \( 2^{\text{Level}(v)+1} \) links have been scanned at the current level.
In the third stage the core collates the information from its descendants, and either selects a feasible outgoing link (if such exists), or activates Update, as follows. Whenever a leaf finishes the second stage it sends a report message with the results of its search to its parent. The message consists of the feasible outgoing link if such was found and "failure" otherwise. Whenever a node \( v \) at distance \( 2^{\text{level}(v)+1} \) from the core is activated it does not scan its links but sends a message indicating its distance to its parent. Any other node \( v \) collates the results of its own search with the messages of its descendants. If at least one of its descendants is at distance \( 2^{\text{level}(v)+1} \) from the core it sends a message indicating this fact to its parent. Else, if no feasible link was found either by it or by its descendants, then a failure message is sent to its parent. Otherwise, a message is sent that reports a feasible link with maximal signature. The core collates the results of its own search with the results of all its descendants. If a node at distance \( 2^{\text{level}(T)+1} \) from the core is detected, then the procedure Update is activated. Else, if a feasible link outgoing to fragment \( T' \) such that \( \text{Level}(T') > \text{Level}(T) \) is detected, then \( T \) becomes idle. Else, if all the feasible links are outgoing to fragments with the same level as \( T \), the link with maximal signature is selected as the feasible link of \( T \), and the procedure Marriage is activated. Otherwise, neither a node whose distance from the core is \( 2^{\text{level}(T)+1} \) nor a feasible link is found; then the number of nodes in \( T \) is counted (as in the procedure Update). If this number is at least \( 2^{\text{level}(T)+1} \), the procedure Update is activated. If the number is less than \( 2^{\text{level}(T)+1} \), \( T \) selects a new label randomly from the domain \( 2^{2\text{level}(T)} \) and repeats the second stage of the procedure. If no feasible links are found this time also, then \( T \) becomes idle.

The implementation of the modified procedure is analogous to the one of the original procedure.

**The Modified Procedure Marriage.** Suppose that the feasible link of \( T \) is outgoing to fragment \( T' \) (where \( \text{Level}(T) = \text{Level}(T') \)). In the modified procedure a fragment \( T \) sends a "marriage proposal" through its feasible link even if \( \text{Label}(T') < \text{Label}(T) \), provided that \( T' \) is idle. In this case, \( T' \) accepts the proposal (unless it has already accepted another marriage proposal). This change is done since in the modified algorithm a fragment may become idle although it is connected to fragments with a different signature.

The modified procedure Link-Scan takes \( O(2^{\text{level}(T)}) \) time for both exploring the nontree links, and broadcast and convergecast over the tree links, if we ignore the time spent for waiting for locked fragments of lower levels. This implies that \( t_k - t_{k-1} \leq c2^k \), and thus \( t_{\log n} = O(n) \).
Let \( m_v \) be the degree of node \( v \); \( \sum_v m_v = 2m \). The number of exploration messages sent by node \( v \) at each level \( k \) are bounded by \( O(2^k) \) and also by \( m_v \). Since the maximum level of a fragment is \( \log n \), the total number of exploration messages sent by node \( v \) throughout the algorithm is bounded by

\[
\sum_{i=1}^{\log n} \min(m_v, 2^i) = O(m_v(1 + \log(n/m_v))).
\]

The total number of exploration messages is

\[
O\left( \sum_{v \in \mathcal{V}} m_v(1 + \log(n/m_v)) \right).
\]

This sum is maximized when all degrees are identical, and is then equal to

\[
O(m(1 + \log(n^2/m))).
\]

Thus, the message complexity of the modified algorithm is

\[
O(n \log n + m \log(n^2/m)) = O(m \log(n^2/m)).
\]

Next, we prove the correctness and the error probability of the algorithm.

**Theorem 4.5.** The algorithm eventually message terminates. Upon termination all fragments are idle.

**Proof.** The level of a node never decreases; a fragment of level \( i \) contains at least \( 2^i \) nodes, so the level is bounded by \( \log n \). Thus, the computation eventually reaches a point when no node further increases its level. Consider the computation at the earliest such point. It is easy to see that from this point no fragment is invaded and no fragment executes Update, since both these operations are eventually followed by an increase in the level of the fragment.

Next, we show that from this point no fragment executes the procedure Marriage. Suppose that some fragment \( T_i \) is executing the procedure Marriage, after selecting a feasible link outgoing to \( T_2 \). We claim that in this case a marriage eventually occurs, and thus the level of some node increases; a contradiction. If \( T_2 \) is eventually idle then a marriage eventually occurs because of the modification in the procedure Marriage. Assume that \( T_2 \) is not eventually idle, in this case \( T_2 \) will eventually execute the procedure Marriage. Using arguments similar to those in the proof of Theorem 4.1, we obtain a contradiction by generating an infinite sequence of nodes with either decreasing or increasing labels. The only difference is that here the feasible link outgoing from fragment \( T_i \) may go to an idle fragment, in which case a marriage eventually occurs.
So far we have shown that from the earliest point when no node further increases its level, each fragment is either idle, or is executing the procedure Link-Scan. Suppose that fragment $T$ is executing Link-Scan. During this execution no node whose distance from the core is greater than $2^{\text{Level}(u)+1}$ is detected, since this would activate the procedure Update. Also, no link $(v \to u)$ such that $\text{Level}(u) < \text{Level}(v)$ is found, since this is followed by invading $u$'s fragment, and an (eventual) increase in the level of $u$. Furthermore, no feasible link is found by $T$, since the finding of such a link causes the (eventual) activation of the procedure Marriage. Thus, fragment $T$ must become idle eventually.

To consider the error probability of the modified algorithm, let $d = \lceil 2/e \rceil$, and recall that the labels are chosen from a domain of size $d2^\text{Level}(T)$.

**Theorem 4.6.** When the algorithm terminates, the probability that a spanning tree is found is at least $1 - \varepsilon$.

**Proof.** By Theorem 4.5, upon termination all fragments are idle. Let $T$ be an idle fragment of maximal level. First, we show that if $T$ is not a spanning tree, some node $v$ in $T$ scanned a link outgoing to another fragment $T'$ twice, using two independent randomly chosen labels, and in both cases the signatures of $T$ and $T'$ were found to be identical. Since $T$ is the fragment with the maximum level, the only way it may become idle is when no feasible link has been detected the first time its links were scanned; then the number of nodes in $T$ has been counted and found to be less than $2^{\text{Level}(T)+1}$, a new label has been redrawn, and again no feasible link has been detected in a second scan of the links. We claim that in both scans some node $v$ scanned a link outgoing to another fragment $T'$. To see this, consider two cases.

**Case 1.** All nodes in $T$ exhausted their adjacency list in both scans. In this case, if $T$ is not a spanning tree for $G$, then at least one node $v$ in $T$ scanned a link $(v \to u)$, outgoing from $T$ to another fragment.

**Case 2.** Some node $v$ in $T$ scanned only $2^{\text{Level}(T)+1}$ outgoing links from its queue in both scans. When the nodes in $T$ were counted at the end of the first scan, the number of nodes has been found to be less than $2^{\text{Level}(T)+1}$. In this case, if $T$ is not a spanning tree for $G$, then one of the links scanned by $v$ at its current level has to go to a node $u$ that is not in $T$.

Consider fragment $T'$, to which the link is outgoing. By our assumption $\text{Level}(T') \leq \text{Level}(T)$. However, fragment $T'$ cannot have either a level lower $\text{Level}(T)$, or a different signature, since both these situations are followed by an increase of some level. Thus, $T'$ has the same signature as $T$, as claimed.
To get the failure probability, suppose that the algorithm terminates and a spanning tree is not found. Consider the fragments of the highest level \( j \) found in the component. Each such fragment \( T \) is connected to at least one other fragment with the same level by a link \((v \rightarrow u)\) that has been scanned twice at level \( j \), and both times it was found that \( \text{Label}(u) = \text{Label}(v) \). The probability that the second scan of \((v \rightarrow u)\) fails (because the same label was chosen at both endpoints) is \( 1/(d2^j) \). Thus, the probability that the scans failed on all such connecting links at level \( j \) is bounded by \( 1/(d2^j) \); the probability that the algorithm fails at any level is bounded by \( \sum_j 1/(d2^j) = 2/d \leq \epsilon. \)

5. The Second Spanning Tree Algorithm

In case the graph \( G \) is sparse (i.e., \( m < n^{2-d} \)) the message complexity of the last algorithm is \( O(n \log n + m \log n) \). One would like to reduce it to, say, \( O(n \log n + m \log \log n) \). In order to do so one should decrease the frequency of link relabeling and rescanning. Rather than rescanning whenever the fragment level increases, one would rescan only when the fragment level doubles. This simple idea seems difficult to implement (we failed). Instead, we use the following approach.

Suppose one starts the algorithm by running some local probabilistic experiment, to estimate \( \log n \), where \( n \) is the number of nodes in the network. The experiment is set so that with high probability at least one node deduces an estimate of at least \( \log n \), and no estimate is much larger than \( \log n \). Each node uses its estimate to choose a random label. The choice is set so that, assuming the estimate is correct, there is a low probability that two nodes pick the same label. We then run the spanning tree algorithm described in Section 4 omitting the label update part in the procedure Update. Instead, whenever a fragment invades another fragment that has an estimate much larger than its own, it chooses a new label using the larger estimate, and restarts the procedure Link-Scan. Eventually, all links are scanned using labels drawn according to an estimate close to the highest, thus making an error unlikely. To prevent too frequent updates of the labels, and renewed scanning of the links, the estimates are divided into consecutive subranges. Relabeling and rescanning is triggered when an estimate in a higher subrange is found. The number of times a link is scanned is at most the number of subranges up to the highest estimate.

We now describe the modifications to the previous algorithm in more detail. A threshold function \( g \) is used to divide the integers into subranges. The function \( g \) is integer-valued, nonnegative, and monotonically non-decreasing. The \( r \)th subrange consists of \( \{ e : g(e) = r \} \). We define \( g(e) \) to be the order of the estimate \( e \). In addition to \( \text{Label}(T) \) and \( \text{Level}(T) \), each
fragment is also tagged with $\text{Order}(T)$, which is the order of the highest estimate of the nodes in the fragment. The initial value of $\text{Order}(u)$, for each node $u$, is computed by the procedure $\text{Estimate}$ (given below). The initial value of $\text{Label}(T)$ is randomly chosen from a domain whose size is a function of $\text{Order}(T)$. (The exact definition of this function is postponed to the analysis.)

The algorithm proceeds as previously, except for the label updating part in the procedure $\text{Update}$. Each link, when scanned by procedure $\text{Link-Scan}$, is tagged with the current order of the fragment. $\text{Order}(T)$ is updated whenever a fragment with a higher order is invaded (or merged). When such an update occurs, a new label for $T$ is randomly chosen from a domain whose size is a function of $\text{Order}(T)$. All nontree links that are tagged with a lower order are untagged, and rescheduled for scanning.

**The Procedure Estimate.** Each node $u$ tosses a fair coin until a "head" occurs. This experiment is repeated $k$ times. (The parameter $k$ is a fixed in the analysis.) $\text{Estimate}$ $\log n$ as $j$, the longest waiting time for "head" in any of the $k$ trials. Set $\text{Order}(u)$ to be equal to $g(j)$.

5.1. *Analysis*

We first select $k$ so that with probability at least $1 - \varepsilon/2$ at least one of the estimates is greater or equal to $\log n$. Assume that $2^{s-1} < n \leq 2^s$. The probability that the waiting time for "head" is $\geq s$ is $2^{-s} > 1/(2n)$; the probability that no waiting time of $s$ or more occurred in $kn$ trials at all nodes is bounded by $(1 - 1/2n)^{kn} \leq e^{-k/2}$. Thus, it is sufficient to take $k = 2 \ln(2/\varepsilon)$.

Let $d = d(r)$ be the size of the domain from which labels are drawn when the order of the estimate is $r$. We assume that $d(r)$ is monotonically non-decreasing, and that $d(r) \geq 2^{r+1}$. The probability of success, given that the highest estimate has value $s$, is bounded from below by the probability $p$ that in $n$ drawings from a domain of size $d = d(g(s))$ no label is drawn twice. We want to ensure that $p \geq 1 - \varepsilon/2$, when $s \geq \log n$. Note that

$$p = \frac{d(d-1) \cdots (d-n+1)}{d^n}.$$

For $0 < x \leq \frac{1}{2}$ we have

$$1 - x > e^{-2x}.$$ (i)

Thus, assuming that $d \geq 2n$,

$$p = \prod_{i=0}^{n-1} \left(1 - \frac{j}{d}\right) > \prod_{i=0}^{n-1} e^{-2i/d} > e^{-n^2/d}.$$
The term $e^{-n^2/d}$ is at least $1 - \varepsilon/2$, whenever $n^2/d \leq -\ln(1 - \varepsilon/2)$, or $d \geq -n^2/\ln(1 - \varepsilon/2)$. For $0 < x \leq \frac{1}{2}$ we have $x < -\ln(1 - x)$. It follows that $p \geq 1 - \varepsilon/2$ when $d \geq 2n^2/\varepsilon$.

Define $\beta(i) = \max \{ r : g(r) = i \}$; in words, $\beta(i)$ is the largest value in the $i$th subrange defined by $g$. By our assumption, $2^{\beta(g(i))} \geq 2^i \geq n$. We take

$$d(r) = \frac{2^{2\beta(r)} + 1}{\varepsilon}.$$

To compute the expected running time and message complexity of the algorithm, we first compute the expected value of the highest estimate over all nodes in the network, denoted by $Max$. The value $Max$ is the expected value of the maximum of $kn$ waiting times in independent Bernoulli sequences of trials with probability $\frac{1}{2}$ of success. Thus

$$\Pr(Max > i) = 1 - (1 - 2^{-i})^{kn}.$$

It follows that

$$E(Max) = \sum_{i=0}^{\infty} \Pr(Max > i) = \sum_{i=0}^{\infty} (1 - (1 - 2^{-i})^{kn}).$$

Each of the summands is less than 1. Also,

$$(1 - 2^{-i})^{kn} > 1 - kn2^{-i}.$$

Bounding the first $\log n$ summands by 1, we get

$$\sum_{i=0}^{\infty} (1 - (1 - 2^{-i})^{kn}) < \log n \sum_{i=\log n}^{\infty} (1 - (1 - 2^{-i})^{kn})$$

$$< \log n + \sum_{i=\log n}^{\infty} \frac{kn}{2^i} = O(\log n + k).$$

Substituting $k = 2 \ln(2/\varepsilon)$,

$$E(Max) = O(\log n + 2 \ln(2/\varepsilon)) = O(\log(n/\varepsilon)).$$

Given $E(Max)$, we can compute the expected running time and message complexity of the algorithm, as follows. All processors can be woken up in $O(n)$ time and $O(m)$ messages. The initial estimates are computed with no communication at all.

A fragment is relabeled at most $g(Max)$ times; this is an upper bound on the number of times a link is scanned. It follows that the total number of messages used by the algorithm is $O(n \log n + mg(Max))$. By an analysis similar to that given in the previous section, it can be shown that
the running time is bounded by $O(ng(\text{Max}))$. Assuming that the threshold function $g$ is concave, i.e., $ag(x) + (1 - a) g(y) \leq g(ax + (1 - a) y)$, for any $0 \leq a \leq 1$; we get

$$E(g(\text{Max})) \leq g(E(\text{Max})) = O(g(\log(n/\varepsilon))).$$

Thus, the expected message complexity is

$$O(n \log n + mg(\log(n/\varepsilon)))$$

and the expected running time is

$$O(ng(\log(n/\varepsilon))).$$

Using the threshold function $g(x) = \lceil \log x \rceil$ we obtain the following result:

**Theorem 5.1.** A rooted spanning tree can be computed with error probability $\varepsilon$ in expected running time $O(n \log \log(n/\varepsilon))$, and expected message complexity $O(n \log n + m \log \log(n/\varepsilon))$.

For $g(x) = \lceil \log x \rceil$ we have $\beta(g(i)) < 2i$; i.e., estimates that are in the same range differ by a factor of at most 2. Thus, if the largest estimate obtained in a computation is $\text{Max}$, then all labels are drawn from a domain of size $\leq 2^{A\text{Max}/\varepsilon}$, and the number of bits per message is at most $O(\text{Max} - \log \varepsilon)$. It follows that the expected message size is bounded by $O(\log(n/\varepsilon))$.

The number of messages can be reduced, using a slower growing threshold function $g$; any message complexity of the form $O(n \log n + mg(\log(n/\varepsilon)))$ can be obtained. However, for this the labels have to be drawn from a larger domain (to cover all values in a subrange) and, hence, more bits have to be used per message. For example, if one uses the iterated logarithm function $g(x) = \log^* x$, then the expected running time becomes $O(n \log^*(n/\varepsilon))$, and the expected message complexity is $O(n \log n + m \log^*(n/\varepsilon))$. However, a subrange contains values from $x$ to $2^x - 1$, and thus, the expected message size is (at least) linear in $n$.

6. **Bounded Size Networks**

Suppose that the network size is bounded by $N$. For this case, we use a variation of the algorithm given in Section 4. The only difference is that now the labels are fixed for the whole execution and are not changed in the procedure Update. The initial labels are drawn from a domain of size $d \geq 2N \geq 2n$. 


The algorithm fails only if at least two nodes draw the same label. The probability of success is bounded from below as in the previous section by

\[ p = \prod_{i=0}^{N-1} \left( 1 - \frac{i}{d} \right) > \prod_{i=0}^{N-1} e^{-2i/d} > e^{-N^2/d}. \]

Thus, it is sufficient to take \( d = -N^2/\ln(1-\varepsilon) = O(N^2/\varepsilon) \). We obtain

**Theorem 6.1.** A rooted spanning tree can be computed with error probability \( \varepsilon \) in a connected network of size bounded by \( N \), by a processor terminating algorithm that runs in \( O(n) \) time and sends \( O(n \log n + m) \) messages, each of size \( O(\log(N/\varepsilon)) \).

Assume that the network size is known within a factor of 2; i.e., \( N/2 < n < N \). Furthermore, assume that the network is connected. We execute the previously described algorithm with \( \varepsilon = \frac{1}{2} \) (i.e., labels are chosen from a domain of size \( O(N^2) \)). Whenever the computation within a fragment terminates, the nodes in the fragment are counted. If their number is less than \( N/2 \), a new label is drawn for the fragment and the computation is restarted. Otherwise, a message containing the fragment size is sent over each nontree link. If the same message is also received on all nontree links, then the processors in the fragment terminate. Otherwise, new labels are chosen and the computation is restarted.

If a fragment does not contain all the nodes, then either it has no more than \( N/2 \) nodes, or it has a neighbor with fewer than \( N/2 \) nodes. In either case, the fragment detects the error. Thus, the algorithm never errs. The expected number of messages sent is twice the number used by the previous algorithm. We obtain

**Theorem 6.2.** A rooted spanning tree can be computed with no error in a connected network of size \( n \), \( N/2 < n < N \), in expected \( O(n) \) time and \( O(n \log n + m) \) messages, each of size \( O(\log n) \).

Note that because of the symmetry, even in this case it is impossible to have a deterministic algorithm that never errs. (See Itai and Rodeh [IR81, IR90].) The algorithm must be either Las Vegas (as shown above) or Monte Carlo (as shown in Section 4).

7. **Open Problems**

This paper provides upper bounds on the message complexity of the computation of spanning trees in anonymous networks. Many questions remain open. In particular, can one compute spanning trees in anonymous networks of unbounded size with \( O(n \log n + m) \) messages, i.e., with a
constant number of messages per link? Or, conversely, can one give a lower bound showing that the number of messages per links is not constant? What is the bit complexity of this problem? Our results give an upper bound of $O(n \log^2 n + m \log n \log \log n)$, for fixed error probability, but it is likely that this bound can be improved. Our upper bound suggests a trade-off between message complexity and bit complexity. Is that the case? What is the trade-off between error probability and bit complexity? Finally, our algorithms require $O(\log(n/\epsilon))$ bits of state information per link. An interesting question is what nontrivial global knowledge on a network can be acquired using only a constant amount of memory per link. Recently, Matias and Afek [MA89] gave several algorithms for the same problem with improved bit complexity and less bits of state information per link; however, their message complexity is higher than ours.

Received June 19, 1989; final manuscript received February 24, 1992

REFERENCES


