Efficient routing protocols in nameless networks

Ivan Lavallee, C. Lavault

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1254

Programme 2
Structures Nouvelles d'Ordinateurs

EFFICIENT ROUTING PROTOCOLS IN NAMELESS NETWORKS

Ivan LAVALLÉE
Christian LAVAULT

Juin 1990

RR_1254

EFFICIENT ROUTING PROTOCOLS IN NAMELESS NETWORKS*

DEUX PROTOCOLES DE ROUTAGE EFFICACES POUR DES RÉSEAUX ANONYMES QUELCONQUES

Ivan LAVALLÉE Christian LAVAULT

E_mail : lavallée/lavault@seti.inria.fr

*This work was supported in part by C^3, *COPARADIS Group.*

Abstract

Two types of distributed fully asynchronous probabilistic algorithms are given in the present paper which elect a leader and find a spanning tree in arbitrary anonymous networks of processes. Our algorithms are simpler than in [11] and slightly improve on those in [9,11] with respect to communication complexity. So far, the present algorithms are very likely to be the first fully and precisely specified distributed communication protocols for nameless networks. They are basically patterned upon the spanning tree algorithm designed in [7,8], and motivated by the previous works proposed in [9,11].

For the case where no bound is known on the network size, we give a message terminating algorithm with error probability ϵ which requires $O(m \log\log(nr) + n \log n)$ messages on the average, each of size $O(\log r + \log\log n)$, where n and m are the number of nodes and links in the network, and $r = 1/\epsilon$. In the case where some bounds are known on n ($N < n \leq KN$, with $K \geq 1$), we give a process terminating algorithm, with error probability ϵ, with $O(m + n \log n)$ messages of size $O(\log n)$ in the worst case. In either case, the (virtual) time complexity is $O(D \times \log\log(nr))$. In the particular case where the exact value of n is known, a variant of the preceding algorithm process terminates and always succeeds in $O(m + n \log n)$ messages of size $O(\log n)$.

Keywords : Probabilistic algorithm, Anonymous network, Election, Spanning tree, Analysis of probabilistic algorithms.

Résumé

Nous proposons dans ce rapport deux algorithmes probabilistes entièrement asynchrones qui permettent de réaliser une election et de construire un arbre couvrant dans des réseaux anonymes quelconques. Nos algorithmes sont plus simples que ceux présentés en [11] et améliorent légèrement la complexité en communication par rapport à [9,11]. Par ailleurs, ces algorithmes sont, à notre connaissance, les premiers du genre dont la specification est totalement et précisement réalisée. Fondamentalement, ils sont conçus a partir de l'algorithme de construction d'arbre couvrant proposé en [7] et la motivation de ce travail provient des problèmes soulevés dans [9] et [11].

Dans le cas ou le réseau est non borné (la taille du réseau est totalement inconnue des processus), nous présentons un algorithme à terminaison par message de probabilité d'erreur ϵ qui nécessite en moyenne $O(m \log\log(nr) + n \log n)$ messages de taille $O(\log r + \log\log n)$, ou n et m sont, respectivement, le nombres de sites et le nombres de liens de communication du réseau, avec $r = 1/\epsilon$. Lorsque le réseau est borné (i.e. n est connu et tel que : pour $K \geq 1$, $N < n \leq KN$), nous proposons un algorithme à terminaison par processus de probabilité d'erreur ϵ, dont la complexité en message dans le pire des cas est $O(m + n \log n)$ avec des messages de taille $O(\log n)$. Dans ces deux cas, la complexité en temps (virtuel) est $O(D \times \log\log(nr))$. Dans le cas particulier ou la valeur exacte de n est connue d' au moins un processus, une variante de l'algorithme précedent a terminaison par processus résout sans erreur les deux problèmes en $O(m + n \log n)$ messages de taille $O(\log n)$. Les algorithmes ont de bonnes mesures complexité en communication sur des toplogies particulières telles que l'anneau ou le réseau complet.

Mots Clés : Algorithme probabiliste, Réseaux anonymes, Election, Arbres couvrants, Analyses d'algorithmes probabilistes.

Contents

1 Introduction **4**

2 Preliminaries and results **4**
2.1 Definitions 4
2.2 Results 5

3 The Algorithm $\mathcal{A}_1$ **6**
3.1 High Level Description 6
3.1.1 The Merging Process 7
3.1.2 The Combination Rules 8
3.1.3 The Notion of Credit 9
3.1.4 Messages comb and Termination 10
3.2 Notations 10
3.2.1 Messages, Variables and Arrays 10
3.2.2 Specification in *CSAP* 11
3.3 Procedures 12

4 Correctness **13**

5 Analysis **14**

6 The Algorithm $\mathcal{A}_2$ **18**
6.1 The Modified Procedure ***INIT*** 18
6.2 Analysis 19

7 Open Problems **22**

8 Annex **25**

1 Introduction

In a distributed algorithm, a network of processes collaborate to solve a given problem. In this framework, each site or process acquires, via local interaction with its neighbours, some global information about the system : e.g. the size of the network, its location in a (minimum-weight) spanning tree, the distances to all other processes, etc. Typically, one assumes that the processes have *distinct* identification labels or *identities*, which means that some global coordination between the processes has taken place beforehand. What happens if this assumption is dropped, so that the processes are indistinguishable ? Consider for example regular nameless networks of some fixed degree. The executions of a deterministic algorithm may end with all processes in the same state, irrespectively of the network size or structure ; a deterministic algorithm cannot distinguish between processes of a regular distributed system, nor can it distinguish between distinct regular networks of the same degree (see [4,5]).

The situation is different if processes are assumed to make independent probabilistic choices ; probabilistic choices can be used to break symmetry in anonymous networks of indistinguishable, nameless processes. When designing election algorithms for the leader election problem (*LEP*) and spanning tree construction problem (*STP*) in anonymous networks, one has to consider the following issues : *relative information on the network size* and *termination detection* [1,2,3,5,6,8,9,11].

2 Preliminaries and results

2.1 Definitions

We consider here the standard model of *static asynchronous network*. This is a point-to-point communication network, described by an undirected *communication graph* $G = (V, E)$ where the set of nodes V represents processes of the network and the set of edges E represents bidirectional non-interfering communication channels (links) operating between neighbouring nodes ; $|V| = n$ and $|E| = m$. No common memory is shared by the processes. We confine ourselves only to *message-driven algorithms*, which do not have central controller and do not use time-outs, i. e. processes cannot access a global clock in order to decide what to do. In a transition, a process receives a message on one of its links, and changes state ; a transition may be probabilistic. We may also assume w.l.o.g. that the same algorithm resides at all nodes of the network and that each process simultaneously start executing its algorithm. Indeed, any process may actually start executing its algorithm either spontaneously at any arbitrary moment or upon receipt of a message which triggers the computation (see procedure ***INIT***). We finally assume throughout the processes and the communication subsystem to be error-free, and that the links operate in a FIFO-manner.

The course of any execution of an algorithm is determined by a *scheduler*, that chooses at each step the next message to be received, as a function of the current network state. An algorithm *process terminates* if in every execution all processes reach a special

halting state ; this corresponds to an algorithm *with termination detection.* An algorithm *message terminates* if in every execution the network reaches a quiescent state where there are no pending messages on the links [11]. In message termination, the processes may ignore that the computation is halted ; this corresponds to an algorithm *without termination detection.* An algorithm has *error probability* ϵ if, for any scheduler and any input, the probability that the algorithm terminates with the right answer is at least $1-\epsilon$.

Our algorithms in the present paper are "Monte Carlo" probabilistic algorithms which occasionally make a mistake, but find a correct solution with high probability whatever the instance considered. These algorithms are $(1-\epsilon)$-correct Monte Carlo algorithms.

2.2 Results

We address the problem of computing a function whose value depends on all the network processes, such as counting the number of nodes in the graph G associated to the network, or solving *LEP* and *STP* for G. In [6], Itai and Rodeh showed that these problems can be solved on a ring by an algorithm which processor terminates (or *distributively terminates*) and always succeeds if and only if the ring size n is known up to a factor of two (see also [1,2,5] for improvements). It was also shown in [6] that it is possible to solve *LEP* in an anonymous network, *with* termination detection and with error probability ϵ *only if* an upper bound on the network size is known.

In [8,9,11] and in the present paper, all of these results were extended to arbitrary networks, while improving some of the bounds. In [11], Schieber and Snir presented efficient schemes of algorithms for *LEP* and *STP* ; in [9], Matias and Afek proposed more detailed schemes of three types of simple algorithms which efficiently solve *LEP*.

Given some $0 < \epsilon < 1$, let $r = 1/\epsilon$. On the assumption that the exact value of n is a priori unknown to any process of the network – and thus without termination detection —, the probabilistic solutions given in [11] require $O(m \log\log(nr) + n \log n)$[1] messages of size $O(\log n + \log r)$, for fixed error probability ϵ. The probabilistic solutions given in [9] require $O(m \log n \times r \log r)$ messages of size $O(\log r + \log\log n)$ on the same assumptions. In the case when n is known up to a factor of $K \geq 1$, that is $N < n \leq KN$, [9] achieves $O(m \times r \log(Kr) \log r)$ messages of size $O(\log n)$, in the worst case, with $N < n \leq 2N$. The time complexity in [9,11] is $O(D)$ and $O(n)$, respectively, where D is the diameter of the network.

Our algorithms are simpler than in [11] and slightly improve on those in [9,11] with respect to communication complexity. Compared to [11], we give overall solutions with improved bit complexity and less bit information per node ; the message complexity in [9] is also higher than ours. Moreover, to the best of the authors' knowledge, the present algorithms are very likely to be the first fully and precisely specified probabilistic protocols to solve *LEP* and *STP* in nameless networks. On the other hand, our "time complexity" is slightly higher than in [9,11].

For the first algorithm $\mathcal{A}_1$, on the assumption that n is a priori unknown to the processes — and thus without termination detection —, the expected message complexity is

[1] Throughout the paper, log denotes the base two logarithm and ln the natural logarithm.

$O(m \times \log\log(nr) + n \log n)$, each message of size $O(\log r + \log\log n)$, with probability $> 1 - \epsilon$.

Assuming n is known up to a factor of $K \geq 1$, that is $N < n \leq KN$, the second algorithm $\mathcal{A}_2$ requires — in the worst-case, and with termination detection —, $O(m + n \log n)$ messages, of size $O(\log n)$, with probability $> 1 - \epsilon$.

Whenever the exact value of n at least known to one process — and with no error —, the complexity of $\mathcal{A}_2$ shrinks to $O(m + n \log n)$ messages, with message size $O(\lg n)$

Algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ run in (virtual) time $O(D \times \log\log(nr))$. $\mathcal{A}_1$ and $\mathcal{A}_2$ also require $O(\log r + \log\log n)$ and $O(\log n)$ bits of state information per node, respectively.

In section 3 we fully describe the message terminating algorithm $\mathcal{A}_1$ which solves *LEP* and *STP* in arbitrary (connected) anonymous networks with fixed error probability ϵ when n is unknown. In section 4 we give the correctness proof of algorithm $\mathcal{A}_1$, followed by a brief analysis of the communication and time complexity in Section 5. Section 6 is devoted to the algorithm $\mathcal{A}_2$. Assuming n is known up to a factor of $K \geq 1$, $\mathcal{A}_2$ is a process terminating protocol which solves *LEP* and *STP* in general (connected) anonymous networks with probability $> 1 - \epsilon$. Description, correctness proof and complexity analysis of algorithm $\mathcal{A}_2$ are given in section 6. The Annex gives a full specification of algorithms $\mathcal{A}_1$ in *CSAP*, an asynchronous variant of *CSP*.

3 The Algorithm $\mathcal{A}_1$

For algorithm $\mathcal{A}_1$, we assume that no bound is known on the size n of the input graph G. In the sequel, we show how the same algorithm $\mathcal{A}_1$ can be modified in $\mathcal{A}_2$ to handle the simpler case when there are bounds on the size n of G. For the sake of brevity, we consider only the case when G is connected. The algorithms and their analysis can be easily extended to the case when the input graph is not connected.

3.1 High Level Description

At each point in the algorithm a rooted forest of G is maintained which is composed of subtrees building a spanning tree of G. The algorithm start by taking the spanning forest which consists of each one of the nodes as a subtree or *fragment* of size one. Upon termination, there is a single tree spanning the whole network, the root of which is the elected leader, with probability $> 1 - \epsilon$.

At each node-process P_i, we maintain the variable id_i. This variable contains information on F_i to which P_i currently belongs. Define id_i, the identity of process P_i, as follows :

Each process tosses a fair coin until a "head" occurs. This experiment is repeated k times. Estimate $\log n$ as j, the longest waiting time for head in any of the k trials. Let t_i be a function of j defined as the number of tosses until P_i tossed a head for

the first time in any of the k trials. Finally denote this procedure of choosing t_i as ***ESTIMATE*** (t).

Each process P_i randomly selects a label s_i from some domain $\boldsymbol{I}$ of size d, where $d = O(r \log r)$ (recall $r = 1/\epsilon$, the value of d will be given in the analysis). These n initial random drawings in the range $[1, d]$ take place during the initialization procedure ***INIT*** . Notice that throughout the paper, the domain $\boldsymbol{I}$ is a *poset.*

Finally, let id_i, the identity of process P_i, be the ordered pair $id_i = (si, t_i)$ (see procedures ***INIT*** and ***ESTIMATE*** (t) in subsection 3.3). Note that a lexicographic order on id_i is defined in the natural way.

Initially, each fragment consists of a single node-process which is its own root. Similarly, In the course of the algorithm, the *identification number* of a current fragment F_i is the identity of its root : id_i.

3.1.1 The Merging Process

For any node a, its *father* is the next node on the path from node a to the root of the tree ; root has no father. In the course of the algorithm, every fragment F finds an *outgoing edge* (a, b) (a being an *outgoing son* of F). Eventually, either F gets absorbed or merged into the fragment on the other side of edge (a, b), becoming a subtree within bigger tree, or F captures the latter fragment, this according to Sollin's property [7,8], with probability $> 1 - \epsilon$.

However, in contrast to most distributed spanning tree algorithms which use the technique of *levels* to ensure a balanced growth of the current fragments (thus pseudo-synchronizing the algorithms), with merging process completed by means of *wars* between kings which result in *annexations of kings' territories*, our combination of fragments proceeds fully asynchronous as follows :

(i) A given fragment F is candidate for merging *into* some other fragment G : the root of F sends a message of combination request denoted by **comb** to G. The combination process never works in the (usual) reverse direction : viz. usually, the king of F asks the king of G to capture his territories, not to get captured.

(ii) Any request of F to get merged into G can only be initiated by the one privileged node in F which may send messages **comb**, but it may be accepted (or rejected) by the *first* node of G which receives the message **comb**.

(iii) Consider a chosen outgoing edge (a, b) in the network and suppose it is traversed by a first message **comb** from node a to node b. In our algorithms, rejection of that first message changes link (a, b) into the *virtually directed* link $(b \rightarrow a)$. Such a virtual link reversal works in such a way as to strictly ensure that the next message **comb** sent through this very link shall traverse in the reverse direction from node b to node a. This "flipping-the-edge" mechanism is completed here by updating the boolean array $PORT[\,]$ at every node (see subsection 3.2.1). Thus the combination process let the fragments grow possibly unbalanced and proceeds fully asynchronous.

Yet, it turns out that the fragments do combine and grow properly *altogether* since the combination rules enforce any "worst merging case" to change into a somewhat "best merging case" after one or only very few execution stages.

Note that for a given fragment, the first and second rules allow a fully decentralized handling of several messages **comb** simultaneously.

3.1.2 The Combination Rules

Consider two fragments F and G with identification numbers id_i for the root of F and id_j for the root of G, respectively. Let (x, y) be an outgoing edge between F and G such that $x \in F$ and $y \in G$, and assume that process x owns in fragment F the privilege of emitting the message **comb** (see subsection 3.1.4 for description of this property that there is only one process at a time having such a privilege in a given fragment).

Suppose x sends y a message **comb** asking G to let F get merged into G. Owing to the *partial ordering* in the domain I , two main cases may occur :

Either $id_i \neq id_j$; the merging operation is thus performed (or aborted) as usual, according to the order relation between id_i and id_j : the process y may locally decide to send back either a rejection message **nok** or an acceptance message **ok** to the process x (see [7,8]),

or $id_i = id_j$; in this second case, it is absolutely impossible to accept *a priori* the combination request of process x because of the possible risk of building a cycle. On the other hand, it is neither possible to simply reject *a priori* this (possibly fair) request.

In such a situation, two conditions may actually arise. Either both processes x and y belong to the same fragment ($F = G$), or not ($F \neq G$). Notice that, owing to their fully local information, no process can distinguish the first condition from the second. Let P_i be the *root* of fragment F. In order to break the symmetry,

- y sends back a message **equal** to x, which passes up the message **equal** to the root P_i.
- Now the behaviour of root P_i depends on the current value of its local variable *credit*, viz. the current number of remaining random changes of identities with which P_i is still credited at this point in the algorithm : initially, the counter variable *credit* is set to r in the procedure ***INIT*** (see subsection 3.1.3 for a precise definition of this notion of *credit*).

  If $credit > 0$, then the procedure ***RANDRAW*** is called, which makes P_i randomly select a new identity id'_i such that $id'_i > id_i$ (s'_i is randomly chosen in the range $[s_i + 1, s_i + d]$: see subsections 3.1.3 and 3.3). Thereafter, the messages **newroot** update each process' identity within F to the new value id_i chosen in ***RANDRAW*** . *credit* is also decremented by 1.

If *credit* = 0, there is no way for P_i to randomly select a new identity since the root is no more credited with any random draw. Thus P_i may regard **equal** as a message received (via x) from some process (y) which actually belongs to its own fragment F.

- P_i sends y a message **cousin** via the same path used by the message **equal** to traverse F. Note that a linking method must be used to keep track of this traversal. Each process thus maintains a local logical clock τ which is incremented by 1 each time a process passes a message up to its father. The message contains this updated time τ. Moreover, an array denoted by $TABLE[\tau]$ is also maintained at each one process which keeps the current values of τ (the local time of the given process), j (the number of the port through which the message was sent) and τ' (the local logical time for the process at the end of the port j). The method is fully described in [8] (see also subsection 3.2.1).

- The receipt of a message **cousin** allows x to clear up the ambiguity of **equal**. So, x locks its port labeled y, and updates its variables *PORT* and *open*. Then either x keeps on sending messages **comb** to its neighbours as long as there still remains one free neighbour (with *open* $\neq$ false), or x grants its father the privilege of emitting the message **comb** if none of its neighbours is free.

3.1.3 The Notion of Credit

The notion of credit and the variable *credit* arise here as a specific *control parameter* which is fixed in advance to *tune the precision* of the algorithm. At each one root and at each point in the algorithm, the local counter variable *credit* is the current number of remaining random drawings the root of the current fragment is granted to randomly choose a new identity if necessary, in using procedure ***RANDRAW*** . As to the procedure ***RANDRAW*** , it strictly increases the current random value of the identification number id_i of the fragment F_i : viz. a number a being randomly drawn in the range $[1, d]$, where $d = O(r \log r)$, s_i is set to $s_i + a$. We thus ensure that the new identity (s'_i, t_i) keeps the same estimate t_i (see subsection 3.3 for a full specification of ***INIT*** and ***RANDRAW***). Initially, every process is credited with r possible random changes of identity in the course of the algorithm : in the initialization procedure ***INIT*** , *credit* := r.

The larger the fixed credit, the better the algorithm. Namely, with fixed error probability $0 < \epsilon < 1$ and with *credit* credited with r possible random drawings in the range $[1, d]$, the algorithm solves (*LEP*) and (*STP*) with probability $> 1 - \epsilon$. In some executions, the algorithm may use only part of its drawings' credit and yet find a spanning tree with probability $> 1 - \epsilon$. On the other hand, some executions of the algorithm may consume the whole credit and yet terminate only with a spanning forest with fixed error probability ϵ.

3.1.4 Messages comb and Termination

Within each one fragment, only one single process at a time is allowed to emitt **comb** messages. When two fragments merge, the process which asked for merging, and was allowed to send messages **comb**, looses this privilege upon receipt of message **ok**. The receipt of either **nok** or **cousin**, causes the port used by the message **comb** to get locked with the logical array $PORT[\,]$ (see subsection 3.2.1). If no more port is available to messages **comb**, the process with the **comb** emission privilege grants its father the privilege, which in turn locks the corresponding port.

Suppose the privilege owner is the root of a fragment. If none of its ports is available any more, then the algorithm message terminates (see [8] for a full description). In the case where the exact value of n is known to at least one process, the process termination of the algorithm is fully completed whenever some process eventually learns that it belongs to some tree of size n which thus spans the whole network (see subsection 6.2).

3.2 Notations

3.2.1 Messages, Variables and Arrays

Messages are composed of four records and denoted by $< \alpha, \beta, \gamma, \delta >$.

The first record is the identification number of the sending fragment.

The second record is an element of the set {**comb, ok, nok, equal, cousin, merge, newroot, end**}. This record is the message *stricto sensu.*

The third record is a boolean variable which possibly sets the logical state of the consirered port.

The last record is a local logical time (see subsection 3.1.2 and [8]).

The algorithm uses three local arrays, $PORT$, SON, and $TABLE$, with the following definitions :

1. $PORT[\,]$ is a logical array with the ports' labels of the current process as indices. Note that each process is assumed to locally distinguish between its different ports. $PORT[\,]$ enables the current process to update the labels of the ports through which messages **comb** may still be sent : $PORT[i]$ = true or false, either if the port labeled i is still available, or not, respectively. Initially, $(\forall i)$ $PORT[i] :=$ true. When the procedure $\boldsymbol{UPDT}$ $(y, PORT, open)$ is called, $PORT[y]$ is set to false upon receipt of either a message **ok** or **nok**, or **cousin**.

2. $SON[\,]$ is a boolean array with the ports labels of the sons of the current process as indices. $SON[]$ enables the current process to know its sons :
$(SON[i] = 1) \iff$ (The process connected to port i is a son of the current process).

3. $TABLE[\,]$ is a list data structure with the values of the local logical clock as indices (see subsection 3.1.1, and [8]) :
$(TABLE[t] = (j, \tau')) \iff$ (At local time t, the process received a message via its port j, and the process connected to this port j had local time t').

Variables are the following :

- *id* is a local, integer variable the value of which is the identity of the current process.
- *root* is the identity of the root of the fragment to which the current process belongs.
- *father* is a local integer variable with value the number of the port which leads to the current process' father within the subtree.
- τ is a local integer variable with value the last local time computed so far.
- *credit* is a local integer variable with value the number of random drawings with which the current root of a fragment is still credited.
- *req* is a local boolean variable which either allows a current root to send a message **comb** ($req = 1$), or do not ($req = 0$).
- *open* is a local logical variable which determines whether the current process still has free neighbours or not (i.e. neighbours which may possibly send back **ok** upon receipt of **comb**).
- *ambiguity* is a local logical variable which gives knowledge that the current process received back **equal** in answer to **comb**, and that some decision of its fragment's root is awaited.

3.2.2 Specification in *CSAP*

CSAP is the acronym for *Communicating Sequential Asynchronous Processes.* The syntax of *CSAP* is very close to the syntax of *CSP.* However, the modifications entail important repercussions with respect to the semantics of the language. Thus, the generation of a message is a non-blocking primitive. Denote $P_i!! <>$ the emission of message $<>$ via the port labeled i in a non-blocking way. In such a case, i may be regarded as the identity of the process to which the message is transmitted. This also entails the presence of a buffer at each one process for incoming messages. The corresponding queue must be bounded, which is the case in the present model. Similarly, denote $P_j?? <>$ the receipt of message $<>$ through the port labeled j. As an example of such notations, "$\forall x \in SON, P_x!! <>$" means that message $<>$ is sent from the current process to all its sons via the ports labeled with the indices of the array SON.

3.3 Procedures

The full specification of algorithm $\mathcal{A}_1$ is given in the Annex.

1. **The Procedure *ESTIMATE***
   Suppose one starts the algorithm by performing the following local probabilistic experiment to estimate $\log n$. The experiment is set so that with high probability at least one process deduces an estimate of a value $\leq \log n$, and no estimate is much larger than $\log n$. Each process P_i uses this estimate to randomly select its label s_i in a domain I whose size d depends on the largest estimate. Thus, in the procedure ***ESTIMATE*** , each process tosses a fair coin until a "head" occurs. This experiment is repeated k times, and $\log n$ is estimated as j, the longest waiting time for head in any of the k trials.

   Procedure ***ESTIMATE*** $(k, \mathbf{var}\ : max)$ ::
   $max := 0$;
   $*[k > 0 \rightarrow t := 0\ ;\ x := 0\ ;$
   $\quad *[x = 0 \rightarrow t := t + 1; x := \mathbf{random}(\{0,1\})]$;
   $\quad [t > max \rightarrow max := t]\ ;\ k := k - 1$
   ].

2. **The Procedure *INIT***
   In the case where a process P_i starts executing its algorithm upon receipt of a message sent by process P_j, P_i righaway considers itself as a node in F_j. Then, P_i gains its identity (without random selection), and wakes up and so act all processes which were not yet awakened. The variable $TABLE$ is a recursive list type with four records: $id, root, father, credit, \tau$ are integer-valued variables; $req, open, ambiguity$ are boolean variables; SON and $PORT$ are arrays of booleans; $NEIGHB$ is a variable which denotes the set of the neighbours of the current process and id is an ordered pair of integers.

   Procedure ***INIT*** $(id, \tau, root, father, credit, PORT, req, SON, open, ambiguity)$ ::
   /* performed upon waking up or reception of first message
   — whichever comes first */
   $[s := \mathbf{random}([1..d])\ ;$ ***ESTIMATE*** $(k,t)\ ;$
   $id := (s,t)\ ;\ root := id; father := id\ ;$
   $\tau := 0\ ;\ req := \text{false}\ ;\ SON := \emptyset\ ;$
   $open := \text{true}\ ;\ \forall x \in NEIGHB, PORT[x] := \text{true}\ ;$
   $credit := \tau\ ;\ ambiguity := 0\ ]$.
   /* $d = O(r \log r)$, with $r = 1/\epsilon$ for fixed ϵ */

3. **The procedure *RANDRAW***

   Procedure ***RANDRAW*** $(d, \mathbf{var}\ : id)$ ::
   $[s := id(1)\ ;\ a := \mathbf{random}([1..d])\ ;\ s := s + a\ ;\ id(1) := s]$.
   /* $id(1)$ is the first integer within the ordered pair (s,t) */

4. **The Procedure *SELECT***

Procedure ***SELECT*** $(\boldsymbol{var} : x, PORT)$ ::
$x := 0$;
$[PORT[1] = \text{true} \rightarrow x := 1$;
▪
$PORT[2] = \text{true} \rightarrow x := 2$;
▪
. . . . . .
▪
$PORT[n] = \text{true} \rightarrow x := n$;
].

5. **The Procedure *UPDT***

Procedure ***UPDT*** $(y, PORT, \boldsymbol{var} : open)$ ::
$[PORT[y] := \text{false}$;
$open := \bigvee_k PORT[k]$;
].
/* *PORT* is a boolean array which represents the state of the port. y is the label of the port through which the current process received a message **nok** or **cousin** */

6. **The procedure *TERM***

Procedure ***TERM*** (SON) ::
$[\forall x \in SON, P_x!! < \_, \textbf{end}, \_, \_ >$; **STOP**.].

4 Correctness

The correctness proof of the algorithm consists of two parts. First, at each point in the algorithm (termination included) a *rooted spanning forest* of G is maintained. Second, the algorithm eventually *message terminates* with a rooted spanning tree with probability $> 1 - \epsilon$.

Lemma 4.1 *In any execution and at each point in the algorithm, the set of current fragments constitute a rooted spanning forest of G, which is an invariant for the algorithm.*

Proof At the beginning of the algorithm, each subtree of the rooted spanning forest of G consists of a single node. Consider a rooted spanning forest of G at some point in the algorithm. Let F_1 and F_2 be two current fragments in the forest, F_1 and F_2 are connected with at most one edge. Denote p_1 and p_2 the number of nodes in F_1 and F_2, respectively : F_1 contains $p_1 - 1$ edges and F_2 contains $p_2 - 1$ edges. Merging F_1 and F_2 builds a new connected component in the forest, say F, which consists of $(p_1 + p_2)$ nodes and $(p_1 - 1) + (p_2 - 1) + 1$ edges. Set $(p_1 + p_2) = p$. F contains p nodes and $p - 1$ edges, and thus F is actually a tree. Therefore, as far as no cycle is built, the tree structure is preserved all along the course of the algorithm. Now to make sure that no cycle can be

built in case of equality, consider how the procedure ***RANDRAW*** works and breaks the symmetry (see subsection 3. 3). Hence, the structure of rooted spanning forest of G is always preserved and is an invariant of $\mathcal{A}_1$. □

Theorem 4.1 *In any execution, the algorithm eventually message terminates. Upon termination, all the fragments constitute a rooted spanning forest of G.*

Proof It is easy to see that the size of fragments never decrease. Since the size of the network and all values and parameters are finite in *LEP* and *STP* (in particular the variable *credit*), the network will reach a quiescent state in some finite stage of the algorithm. In the case where all processes have distinct identities, the proof of the theorem is given in [7]. Suppose at least two processes have the same identity. The termination is triggered either whenever all roots have consumed their whole drawings credit and all have the same identity, or whenever the privilege owner is the root of a fragment while none of its ports is available any more. In such a case, the root considers the algorithm terminated : viz. $root = i \wedge req = \text{false} \wedge ambiguity = 0$ holds. The latter formula implies that each process in the fragment will receive a termination message **end**. Therefore, the algorithm message terminates, and, by Lemma 4.1, upon termination all the fragments constitute a rooted spanning forest of G (see [8] for a detailed proof). □

Let ϵ be the error probability and let $d \geq \lceil 2/\epsilon \rceil$. Recall that the labels are randomly selected from a domain of size d in the procedure ***RANDRAW*** .

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

Proof Suppose that the algorithm terminates with more than one fragment left. By Theorem 4.1, we have that all the roots have the same identity and that $credit = 0$ at each root. The probability that all drawings at $credit = j > 0$ selected the same label (given that more than one drawing occurred) is bounded from above by $1/d^j$. The probability that the algorithm fails at the last value of *credit* is the probability that all drawings at $credit = 1$ selected the same label (given that more than one drawing occurred). It is whence bounded from above by $\sum_j 1/d^j < 1/(d-1) < \epsilon$. □

5 Analysis

The *worst-case message complexity* of an algorithm (for a given input size) is the maximum over all networks of the given size and over all schedulers, of the largest number of messages sent in any execution of the algorithm. The expected message complexity (for a given input 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 in the network ; the expectation is over the random choices of the algorithm.

In the following, denote $T = max_i\{t_i\}$, and $M = max_i\{id_i\}$. We first make sure that at least one of the estimates t_i is $\geq \log n$ with probability $> 1 - \epsilon$, and then prove that $T < \log(nr)$ with probability $> 1 - \epsilon$.

Lemma 5.1 *Let t_i be one of the estimates.* $(\exists i \in [1,n])\ Pr\{t_i \geq \log n\} > 1 - \epsilon$.

Proof Assume that for all $i \in [1,n]$, we have $2^{t_i - 1} < n \leq 2^{t_i}$. Let $t_i = t$. The probability that the waiting time for head is $\geq t$ is $2^{-t} > 1/2n$. Now the probability that no waiting time of t or more occurred in kn trials at all nodes is $(1 - 2^{-t})^{kn} < (1 - 1/2n)^{kn} \leq e^{-k/2}$. Thus, it is sufficient to take $k = 2\ln(1/\epsilon)$, and for at least one $i \in [1,n]$, $Pr\{t_i \geq \log n\} > 1 - \epsilon$. □

Lemma 5.2 *Let t_i be one of the estimates.* $Pr\{(\exists i \in [1,n])\ t_i \leq \log(nr)\} > 1 - \epsilon$.

Proof For fixed $i \in [1,n]$, $Pr\{t_i \geq \log(nr)\} = 2^{-\log(nr)} = 1/nr$. Therefore,

$$Pr\{(\exists i \in [1,n])\ t_i \geq \log(nr)\} \leq 1/r = \epsilon.$$

□

Corollary 5.1 $(\forall \epsilon > 0)\ Pr\{\log n \geq T < \log(nr)\} > 1 - \epsilon$.

Proposition 5.1 *Let $T = max_i\{t_i\}$, $E[T] = O(\log(nr))$.*

Proof T is the highest estimate computed by a process, viz. T is the maximum of kn waiting times in independent Bernoulli sequences of trials with probability $1/2$ of success. Thus, $E[T] = \sum_j Pr\{T > j\}$, and

$$E[T] < \sum_{j < \log n} Pr\{T > j\} + \sum_{j \geq \log n} Pr\{T > j\}.$$

Since $(\forall \epsilon > 0)\ Pr\{T \geq \log(nr)\} < \epsilon$,

$$(\forall \epsilon > 0)\ E[T] < \sum_{j < \log n} Pr\{T > j\} + \sum_{j = \log n}^{\log(nr)} Pr\{T > j\} + \epsilon,$$

and

$$E[T] < (\log n) Pr\{T > \log n\} + (\log n + 1) Pr\{T \geq \log n\} + \epsilon.$$

Hence,

$$E[T] = O(\log n + \log r), \tag{1}$$

and the expected value of T is $O(\log(nr))$. □

The main idea in the procedure ***ESTIMATE*** (see subsection 3.3) is that there is one process with a t_i larger than the others. The following claim and lemma prove the existence, with high probability, of such a *unique* process.

Definition 5.1 *Denote as* ***candidate*** *a process P_i such that $t_i \geq \log n - t$, for some parameter t to be fixed later. The function $\varphi(t) = \log n - t$ is a* ***threshold*** *function which identifies processes with large identities, denoted as candidates, from all other processes. The function φ is an integer-valued, nonnegative, monotone nondecreasing function of t.*

Claim 5.1 *Let $M = max_i\{id_i\}$. If the number of candidates is $\leq C$, and s_i was randomly selected from a domain of size Cr, then M is unique, with probability $> 1 - \epsilon$.*

Sketch of Proof Assume that all candidates have the same t_i. If such is not the case, then the definition of a *candidate* should be refined as follows : the candidates are the only processes whose estimate is T. C drawings are completed from a domain $\boldsymbol{I}$ of size $d = Cr$. Now, let p be the probability that in C drawings from $\boldsymbol{I}$ the largest s_i (among those which were drawn) is drawn only once. We know that $p > 1 - C/d$ (see [9, Claim 12] for a detailed proof). Hence, we have that $max_i\{s_i\}$ (over the P_i which are candidates), is unique with probability $> 1 - C/Cr = 1 - \epsilon$. □

Lemma 5.3 *There is a unique process P_i with identity $id_i = M$, with probability $> 1 - \epsilon$.*

Proof The proof of the lemma consists of two parts.

- We first show that (a) there are $\Theta(\log r)$ candidates, with probability $> 1 - \epsilon$. This ensures that, with high enough probability, there is at least one candidate but not too many.
- Second we show that (b) with probability $> 1 - \epsilon$, $max_i\{s_i\}$, where i is over all the candidates, is randomly selected in $\boldsymbol{INIT}$ by only one among the candidates. This ((a) and (b)) is sufficient to show that there exists a unique process with identity M, with probability $> 1 - \epsilon$.

(a) is proven in using Tchernov's bounds [10]. Let X be the number of candidates. The probability of a process to be a candidate in any of the k trials is $2^{t-\log n} = 2^t/n$: i. e. the probability to obtain $(\log n - t)$ tails in $(\log n - t)$ tosses. Since, the number of trials is k at each one of the n processes, $c = E[X] = k2^t$.
Now if we let $c = 12\ln(2r)$, by Tchernov's inequalities [10, p. 121],

$$Pr\{c/2 \leq X \leq 3c/2\} > 1 - [exp(-c/12) + exp(-c/8)] > 1 - 2/2r > 1 - \epsilon,$$

and the number of candidates X is $\Theta(\log r)$ with probability $> 1 - \epsilon$.

(b) is a straightforward consequence of Claim 5.1 : Let $C = 18\ln(2r)$, then by Claim 5.1, we have that there exists one unique process P_i with $id_i = M$ (over all the candidates), with probability $> 1 - \epsilon$. This concludes the proof of the lemma. □

Remarks

- If we let $C = 18\ln r$, then by the tight inequalities of Tchernov, we know at the same time that
  - the number of candidates is $\Theta(\log r)$ with probability $> 1 - \epsilon$,
  - if the number of candidates is $\leq C$, then M is unique with probability $> 1 - \epsilon$.
- As a direct consequence of Lemma 5.3, we are now able to compute the size d of domain $\boldsymbol{I}$: since the number of candidates is $\leq C = 18\ln r$, d needs to be $2Cr = 36r\ln r = O(r\log r)$.

- Our average number of candidates is larger than in [9] : $c = k2^t$. This is due to the fact that k initial tosses are experimented at each one process in our procedure ***ESTIMATE*** , which improves the number of processes with estimate closer to $\log n$.

We now turn out to compute the (virtual) running time and expected message complexity of the algorithm, and the size of messages, with probability $> 1 - \epsilon$.

Theorem 5.1 *A rooted spanning tree can be built with probability $> 1 - \epsilon$ in expected message complexity $O(m \log\log(nr) + n \log n)$, with message size $O(\log r + \log\log n)$, and expected (virtual) running time $O(D \times \log\log(nr))$ (where D is the diameter of the network).*

Proof Given $E[T]$, we can compute the expected complexity of the algorithm as a function of $E[T]$ and $M = max_i\{id_i\}$. All processes are waken up with $O(m)$ messages in time $O(n)$. The k trials can take place simultaneously, although they have to be triggered by messages, so that the expected number of messages per node is $O(T)$. Hence, ***ESTIMATE*** requires an expected number of messages of $O(m + n \log(nr))$ in expected running time $O(n + \log(nr))$.

The highest message cost is the number of messages **newroot**, since all identities of processes within *one* of two merging fragments are updated upon receipt of messages **newroot**. Indeed, the total number of other messages is $O(m + n \log(nr))$ on the average. Since the number of candidates is $\Theta(\log r)$ with probability $> 1 - \epsilon$, the upper bound on the number of times a fragment is updated is $\varphi(T \log r)$. Thus, the number of messages **newroot** used in the algorithm is $O(m \times \varphi(T \log r))$. It follows that the total number of messages required by the algorithm is $O(n \log n + m \times \varphi(T \log r))$. Similarly, the (virtual) running time is bounded by $O(D \times \varphi(T \log r)$. Assuming that the threshold function φ is concave and by equation (1), we have $E[\varphi(T \log r)] \leq \varphi(E[T \log r]) = \varphi(\log r \times E[T]) = O(\varphi(\log r \times \log(nr)))$. Hence, the expected message complexity is $O(n \log n + m \times \varphi(\log r \times \log(nr)))$, and the expected (virtual) running time is $O(D \times \varphi(\log r \times \log(nr)))$.

Using the threshold function $\varphi(x) = \lceil \log x \rceil$ we obtain the results given in the algorithm. The expected number of messages used by the algorithm is $O(n \log n + m \log(\log r \log(nr))) = O(n \log n + m \log\log(nr))$, with probability $> 1 - \epsilon$. The expected (virtual) running time is $O(D \times \log\log(nr))$, with error probability ϵ. The maximum number of bits per message is the number of bits in T plus the maximum number of bits in s_i (selected from $[1, d]$, where $d = r \log r$). Therefore, the size of messages is bounded by $O(\log r + \log\log n)$. □

Remarks

- As a corollary, one can see that *LEP* and *STP* can be solved with error probability ϵ on a *ring* of unknown size with $O(n \log\log r + \log n)$ messages, each of size $O(\log\log n + \log r)$. Thus the bit complexity is $O(n \log n \log\log n)$, for fixed ϵ, which meets the result in [9] and is an improvement on the $O(n \log^2 n)$ bound in [11].
- Besides, *LEP* and *STP* can be solved with error probability ϵ on a *complete network* of unbounded size with $O(n^2 \log\log(nr) + \log n)$ messages, each of size $O(\log\log n + \log r)$. Thus, the bit complexity is $O(n^2(\log\log n)^2)$, for fixed ϵ, which improves the results in [9] and [11].
- Algorithm $\mathcal{A}_1$ requires $O(\log\log n + \log r)$ bits of state information per node, which again meets the result in [9].

6 The Algorithm $\mathcal{A}_2$

We now address the case when the network is bounded. Assume n is known up to a factor of K : viz. $N < n \leq KN$, with $K \geq 1$. Then Algorithm $\mathcal{A}_2$ is process terminating and solves *LEP* and *STP* with fixed error probability ϵ if $K > 1$, and with no error if n is exactly known.

To handle the case where bounds are known on the size n of the network, algorithm $\mathcal{A}_2$ is designed with slight modifications of $\mathcal{A}_1$, mainly within the procedure ***INIT*** , with the new variable *size*, and also regarding the termination of the algorithm in the subcase where the exact value of n is known to at least one process.

The procedure ***ESTIMATE*** is not changed in algorithm $\mathcal{A}_2$. By contrast, the procedure ***INIT*** takes all the additional information about n into account. In words, ***INIT*** uses the new parameters μ and φ (computed in the analysis, subsection 6.2) to ensure that each process can check if it is a candidate or not. Now, no process but each one *candidate* process P_i can randomly choose its label s_i, and can thereafter trigger the computation by sending a message **comb** to a selected neighbour. Upon receipt of a message **comb**, any noncandidate process gains the identity $id_i = (s_i, t_i)$ of the emitting process P_i, and is "merged" into its fragment. In the case where two candidates exchange messages, algorithm $\mathcal{A}_2$ and algorithm $\mathcal{A}_1$ are similar.

6.1 The Modified Procedure *INIT*

The variable *size* maintained at each process P_i is merely a local integer-valued variable, which counts the number of nodes in the current fragment F_i to which P_i belongs.

Procedure ***INIT*** $(id, r, root, father, credit, PORT, req, SON, open, ambiguity, size)$ ::
/* performed upon waking up or upon reception of first message — whichever comes first */
[***ESTIMATE*** (k, t)
$[(d, t) > \mu \rightarrow \forall x \in NEIGHB, P_x!! < \_, \mathbf{end}, \_, \_ >$ /* the algorithm fails with probability $< \epsilon$ */

```
    ■
    (d,t) ≤ μ → t := t − φ ;
        [t > φ → s := random([1..d]) ; id := (s,t) ;
        size := 1 ; root := id ; father := id ;
        τ := 0 ; req := false ; SON := ∅ ;
        open := true ; ∀x ∈ NEIGHB, PORT[x] := true ;
        credit := r ; ambiguity := 0 ]
    ]
].
/* d = O(Kr log r), μ = (r log r) × log(12Kr log r), and φ = log N − log(12 ln r) */
```

6.2 Analysis

Lemma 6.1 *Let $N < n \leq KN$. The number of candidates is, for some setting of t, $\Theta(K \log r)$ with probability $> 1 - \epsilon$.*

Sketch of Proof Set $t = \log c$, where $c = (n/N)12 \ln r$. Thus the threshold for the candidates is now $\varphi = \log n - \log c = \log N - \log(12 \ln r)$. Similarly to the proof of Lemma 5.3, for $K \geq n/N$, the number of candidates is $c = \Theta(K \log r)$, with probability $> 1 - \epsilon$. □

Let $t'_i = t_i - \varphi$, and $T' = max_i\{t'_i\}$.

Claim 6.1 *$T' = O(\log(Kr))$ with probability $> 1 - \epsilon$.*

Proof $t'_i = t_i - \varphi = t_i - (\log N - \log(12 \ln r)$. Therefore,

$$T' = \log(nr) - \log N + \log(12 \ln r) = O(\log(Kr)).$$

□

Whenever we have this additional information about n, we can use the initial knowledge on N, K, and r to improve the complexity. The threshold $\varphi = \log N - \log(12 \ln r)$ can also be given in advance to all processes. Thus, the identities can be reduced by φ to yield smaller size messages.
By Claim 6.1, for $K = n/N$, we have

$$M = max_i\{id_i\} < (r \log r) \times (\log(12Kr \ln r)) = \mu.$$

The value of μ can thus be also given to all processes when the algorithm starts, together with N, K, r and φ. All of this information is used to modify the procedures ***INIT*** in subsection 6. 1.

We can now turn out to compute the complexity of algorithm $\mathcal{A}_2$ whenever n is exactly known, and in the cases where $N < n \leq 2N$ and $N < n \leq KN$, with $K > 2$.

Lemma 6.2 *Suppose $N < n \leq K$, with $K \geq 2$. (a) If there are $(l-1)$ current fragments whose size is larger than that of some fragment F, then the latter is bounded by (n/l). (b) The total number of nodes in all fragments such as F is at most $O(n \log n)$.*

Sketch of Proof (a) is immediate. (b) is straightforward, since

$$\sum_{l=0}^{KN} n/l = nH_{KN} = O(n \log n).$$

H_k being the k^{th} harmonic number, with asymptotic expansion : $H_k = \ln k + O(1)$. □

We now give the computation of the size d of the domain $\boldsymbol{I}$ whenever the exact value of n is known to the processes, or in the case where $N < n \leq KN$, with $K \geq 2$.

In the procedure $\boldsymbol{INIT}$, an initial sequence of random drawings is completed. Let $d = d(r)$ be the size of the domain $\boldsymbol{I}$ from wich the labels are drawn, and where $r = 1/\epsilon$.

1. Assume the exact value of n, the size of the network, is known to at least one process. We can compute the exact value of d which is sufficient to draw n distinct identities with high probability. Let id_i and id_j be the identities of the two processes P_i and P_j. Let $p = Pr\{id_i \neq id_j\}$. Notice that

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

For all real $x \leq 1/2$, $1 - x > e^{-2x}$. Assuming that $n/d \leq 1/2$, or $d \geq 2n$ yields

$$p = \prod_{i=0}^{n-1}(1 - i/d) > \prod_{i=0}^{n-1} e^{-2i/d} > e^{-n^2/d}.$$

Since $(\forall 0 < \epsilon < 1)$ $\ln(1-\epsilon) < -\epsilon$, we have that $p > 1 - \epsilon$ when $d \geq n^2/\epsilon$. Hence, it is sufficient to randomly draw the n initial identities from a domain $\boldsymbol{I}$ of size $d = O(n^2 r)$, for fixed ϵ. In that case, the identities randomly selected in $\boldsymbol{INIT}$ will all be distinct with probability $> 1 - \epsilon$.

2. Assume now n, the size of the network, is only known up to a factor of K, i.e. $N < n \leq KN$. Suppose also that *only the candidates* can randomly select their own labels. Using the fact that $C = \Theta(K \log r)$ (where $K = n/N$, with probability $> 1 - \epsilon$), we can compute the value of d such that two identities are distinct with probability $> 1 - \epsilon$. By Claim 5.1, we know that the size of $\boldsymbol{I}$ can be $d = Cr$. Therefore, it is sufficient to take $d = O(Kr \log r)$ to have all the identities randomly selected in $\boldsymbol{INIT}$ be distinct with probability $> 1 - \epsilon$.

Remark Taking only the lower bound N into account would yield the following. Let s_i and s_j be the labels of two processes. Let $p = Pr\{s_i \neq s_j\}$. Assuming $d \geq 2N$ yields

$$p = \prod_{i=0}^{N-1}(1 - i/d) > \prod_{i=0}^{N-1} e^{-2i/d} > e^{-N^2/d}.$$

Hence, it is sufficient to take $d = O(N^2/\ln(1-\epsilon) = O(N^2/\epsilon$ to have all identities distinct with probability $> 1 - \epsilon$ in the initial random drawings of procedure $\boldsymbol{INIT}$.

Following the above calculations, we can conclude :

Lemma 6.3 *For fixed error probability ϵ, (i) If n is known in the range $N < n \leq KN$, with $K \geq 2$, then the size of domain I is $d = O(Kr \log r)$. (ii) If the exact value of n is known, then the size of domain I is $d = O(n^2 r)$.*

By Lemmas 6.1 and 6.2, the number of messages is at most $O(Km \log r + n \log n)$. However, this is a very pessimistic upper bound. The main point is that we can use our knowledge on the size of fragments at any point in the algorithm and the upper bound on n to update processes with only a *constant number of messages per link.* This shrinks the message complexity down to $O(m + n \log n)$

The algorithm $\mathcal{A}_2$ solves *LEP* and *STP* with termination detection and with no error provided the variable *credit* is not used as defined for $\mathcal{A}_1$ — or simply removed. Indeed, the algorithm may be restarted repeatedly (with new labels chosen in each run) till some process eventually learns that its fragment's size is n, and thus spans the whole network. The *process termination* is then completed *with no error*, and with $O(m+n \log n)$ messages each of size $O(\log n)$. This yields the result :

Theorem 6.1 *If the exact value of n is known to at least one process, then $\mathcal{A}_2$ solves LEP and STP with termination detection and with no error. The worst-case message complexity is $O(m + n \log n)$ and each message is of size $O(\log n)$.*

Remarks

- Removing *credit* is not really necessary, since one can set the initial value of the variable to a sufficiently large fixed value, say V, where $V = \Omega(n^2)$ for example. The algorithm shall be rerunned till $size = n$, and shall thus eventually succeed.
- In such a case, $\mathcal{A}_2$ is a "Las Vegas" probabilistic algorithm in the sense that it reacts by either returning a correct solution, or admitting that its random decisions have led to an impasse. In the latter case, it suffices to resubmit the same instance to $\mathcal{A}_2$ to have a second, independent chance of arriving at a correct solution. The overall probability of success therefore increases in each round of $\mathcal{A}_2$.

Theorem 6.2 *If n is known in the range $N < n \leq 2N$, then $\mathcal{A}_2$ solves LEP and STP with probability $> 1 - \epsilon$ and with termination detection. The worst-case message complexity is $O(m + n \log n)$ and each message is of size $O(\log n)$, with fixed error probability ϵ.*

Proof If algorithm $\mathcal{A}_2$ is executed with $\epsilon = 1/2$, the labels are randomly selected from a domain of size $d = 2K \log 2 = 2n/N$.

When a fragment F_i merges with another fragment, the new root P_i knows its new fragment's size : for example, $size_i := size_i + size_j$.

If $size_i \leq N$, then P_i knows for sure that it is not the leader (possibly not yet).

If $size_i > N$, then P_i is still not necessarily the leader, since F_i may be surrounded with other fragments (all of size $\leq N$) with the same identity. It is also possible that there exists a fragment with larger identity elsewhere in the network.

However, F_i can enlarge till either reaching a larger identification number, or possibly till it spans the whole network. In the latter case, F_i wins the game.

With probability $> 1 - \epsilon$, the algorithm eventually process terminates with a spanning tree. Altogether, the algorithm requires a *constant* number of messages **newroot** per link, and therefore, since the total number of other messages is $O(m + n \log n)$ from Lemma 6.1 and 6.2, the worst-case message complexity is $O(m + n \log n)$. The size of messages is $O(\log(Kr \log r)) = O(\log n)$, for fixed ϵ. □

Theorem 6.3 *If n is known in the range $N < n \leq K$, with $K > 2$, then $\mathcal{A}_2$ solves LEP and STP with probability $> 1 - \epsilon$ and with termination detection. The worst-case message complexity is $O(m + n \log n)$ and each message is of size $O(\log n)$, with fixed error probability ϵ.*

Proof The problem with the case $K > 2$ is that there might exist several fragments with the same identification number, all of size $> N$. However, there are at most K such fragments. Thus, in the same conditions that in the previous case $K = 2$, we are guaranteed with probability $> 1 - \epsilon$ that the labels are all distinct after at most $K/2$ calls of the procedure ***RANDRAW*** . Hence, a fragment (with new label s') can be surrounded with similar fragments only with probability $< \epsilon$, and if there is a unique leader the algorithm will eventually process terminate.

As to the (worst-case) message complexity, it remains the same as in the case $K = 2$, up to a constant factor, with probability $> 1 - \epsilon$. The size of each message is again $O(\log n)$. □

Remarks

- As a consequence, *LEP* and *STP* can be solved with error probability ϵ on a *ring* of bounded size, $N < n \leq KN$ $(K \geq 2)$, with bit complexity $O(n \log^2 n)$.
- In *complete networks* of bounded size, these problems can be solved with bit complexity $O(n^2 \log n)$. This also improves some of the bounds in [9,11].
- Algorithm $\mathcal{A}_2$ requires $O(\log \log n)$ bits of state information per node, wich again meets the result in [9].

7 Open Problems

The open problems raised in [11] still remain open. In particular, can one compute spanning trees in anonymous networks of unbounded size with $O(m + n \log n)$ messages, i.e. with a *constant* number of messages per link ? Or, conversely, is it possible to give a lower bound showing that the number of messages per link is *not constant* ?

In the analysis of algorithm $\mathcal{A}_1$ (Section 5), we used the threshold function $\varphi(x) = \lceil \log(x) \rceil$. But the number of messages can be drastically reduced in using a very slowly growing threshold function φ, such as, for example, the iterated logarithm function $\varphi(x) = \lceil \log^*(x) \rceil$, or the inverse of the single-valued function of Ackermann $\varphi(x) = A^{-1}(x)$.

Then, the expected message complexity becomes $O(m \log^*(nr) + n \log n)$, or $O(mA^{-1}(\log nr) + n \log n)$, respectively : e.g. $A^{-1}(\log nr) \leq 5$ for $\log(nr) \leq 2^{65,536}$,

or for $nr \leq 2^{2^{65,536}}$ (recall $65,536 = 2^{16}$). In other words, $A^{-1}(\log nr) \leq 5$ for all n and r one is ever likely to encounter. However, the size of each message is here at least linear in n, and the average bit complexity is as large as $O(mn \log^* n + n^2 \log n)$, or $O(mnA^{-1}(\log nr) + n^2 \log n)$, respectively.

Hence, the conjecture that one can compute spanning trees in nameles networks of unbounded size with a *constant* number of messages per link is not at all unreasonnable. However, we should also conjecture that the size of messages is small enough : e.g the bit complexity remains of order $O(m \log n + n^2 \log n)$.

References

[1] **K. ABRAHAMSON, A. ADLER, L. HIGHAM and D. KIRKPATRICK**, Probabilistic solitude verification on a ring, *Proc. of the 5th ACM Symp. on Principles of Distributed Computing, 161-173, Calgary*, August 1986.

[2] **K. ABRAHAMSON, A. ADLER, L. HIGHAM and D. KIRKPATRICK**, Randomized function evaluation on a ring, *Distributed Computing* **3**, *107-119, Springer-Verlag*, 1989.

[3] **Y. AFEK, M. SAKS**, Detecting global termination conditions in the face of uncertainty, *Proc. of the 6th ACM Symp. on Principles of Distributed Computing, 109-124, Vancouver*, August 1987.

[4] **D. ANGLUIN**, Local and global properties in networks of processors, *Proc. 12th Annual ACM Symposium on Theory of Computing, 82-93, Los Angeles*, May 1980.

[5] **H. ATTIYA, M. SNIR, M. K. WARMUTH**, Computing on an Anonymous Ring, *J. ACM, Vol. 35, No 4, 845-875*, October 1988.

[6] **A. ITAI, M. RODEH**, Symmetry breaking in distributive networks, *Proc. of the 22nd IEEE Symp. on the Foundation of Computer Science, 150-158, Nashville*, October 198.

[7] **I. LAVALLÉE, C. LAVAULT**, Yet another distributed election and spanning tree algorithm, *R.R. INRIA No 1024*, April 1989.

[8] **I. LAVALLÉE, C. LAVAULT**, Un algorithme probabiliste d'élection et d'arbre couvrant sur des réseaux anonymes, *R.R. INRIA No 1151*, December 1989.

[9] **Y. MATIAS, Y. AFEK**, Simple and Efficient Election Algorithms for Anonymous Network, *Proc. of the 3rd International Workshop on Distributed Algorithms, 183-194, Nice, LNCS 392, Springer-Verlag*, September 1989.

[10] **A. PAPOULIS**, *Probabilities, Random variables, and Stochastic Processes*, McGraw-Hill, 2nd Edition, 1984.

[11] **B. SCHIEBER, M. SNIR**, Calling Names on Nameless Networks, *Proc. of the 8th ACM Symp. on Principles of Distributed Computing, 319-328, Edmonton*, August 1989.

8 Annex

Specification of Algorithm $\mathcal{A}_1$

```
Proc ::
INIT ;
*[ (root = id ∧ req = false ∧ ambiguity = 0) → [open = false → TERM (SON)] ;
                                    SELECT (x,PORT[ ],-,-) ; τ := τ + 1; req := true ; P_x !! <id, comb,-,τ>
    ▮
P_y ??<α,β,γ,δ> →
    [ β = end → TERM
    ▮
    β = comb ∧ y = father → [ open = true → SELECT (x,PORT[ ],-,-) ; P_x!!<α,β,γ,δ>
                              ▮
                              open = false → P_y !!<-,nok,open,->
                            ]
    ▮
    β = comb ∧ y ≠ father → [α = root → P_y!!<-,equal,-,->
                              ▮
                              α < root →P_y!!<root,ok,-,-> ; SON := SON ∪ {y}
                              ▮
                              α > root → P_y !!<-,nok,-,->
                            ]
    ▮
    β = ok → [y ∈ SON → SON := {SON − {y}}∪{father};
                        [root = id → root := α ; UPDT ; P_y!!<-,merge,open,-> ;
                                     ∀x∈ SON, P_x !!<α,newroot,-,->
                          ▮
                          root ≠ id → root := α ; P_father !!<α,β,-,->
                        ] ; father := y
        ]
```

▮

β = **nok** → [γ = false → *UPDT*] ;

[*root* ≠ *id* → [*open* = true → *SELECT* (*x*,*PORT*[ ],*y father*) ; P_x !!<*root*,**comb**,-,-> ▮ *open* = false → P_{father}!!<α,**nok**,false,->] ▮ *root* = *id* → *req* := false]

▮

β = **equal** → [*y* ∈ *SON* → [*id* = *root* → [*credit* > 0 → *RANDRAW* ; ∀*x*∈ *SON*, P_x !!<*id* ,**newroot**,-,->; *root* := *id* ; *credit* := *credit* − 1 ▮ *credit* = 0 → P_y!!<-,**cousin**,-,δ>] ▮ *id* ≠ root → τ := τ + 1 ; *TABLE*[τ] = (*y*, *e*) ; P_{father}!!<α,β,γ,τ>] ▮ *y* ∉ *SON* → *ambiguity* := 1 ; τ := τ + 1 ; *TABLE*[τ] = (*y*, -) ; P_{father}!!<α,β,γ,τ>]

▮

β = **cousin** → [*ambiguity* = 1 → (*z*,-) := *TABLE*[*e*] ; *UPDT*(*z*, *PORT*[ ],*open*) ; [*open* = false → P_{father}!!<-,**nok**,false,-,-> ▮ *open* ≠ false → *SELECT* (*x*,*PORT*[ ]) ; P_x!!<*root*,**comb**,-,->] ; *ambiguity* := 0 ▮ *ambiguity* = 0 → (*k*, *l*) := *TABLE*[*d*] ; P_k!!<α,β,γ,τ>]

▮

β = **newroot** → *root* = α ; ∀*x*∈ *SON*, P_x !!<α, **newroot**,-,-> ; [*ambiguity* = 1 → *ambiguity* := 0 ; *SELECT* (*x*,*PORT*[ ]) ; P_x!!<*root*,**comb**,-,->]

▮

β = **merge** → *SON* := *SON* ∪ {*y*} ; *PORT*[*y*] := γ ; *open* := $\bigvee_k$ *PORT*[*k*] ; ∀*x*∈ *SON*, P_x !!<α, **newroot**,-,-> ; [*root* ≠ *id* → P_{father}!!<α,β,*open*,->]

]

].

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399