DISTRIBUTED RANDOM WALKS AND THE DESIGN OF A SELF–STABILIZING RANDOM SPANNING TREE

HICHEM BAALA AND OLIVIER FLAUZAC

Abstract. We present a self–stabilized distributed algorithm for constructing a random spanning tree, making use of random walks as network traversal scheme. Our approach is novel and make use of distributed random walks, each one represented by a token annexing a territory over the underlying graph. These multiple random walks collapse into a final one, that defines the final territory and provides the random spanning tree. The scheme is parallel and make use of waves to merge very efficiently the spanning forest computed by the random walks into one final random spanning tree. The algorithm is said to be self–stabilized, because it uses a protocol that tolerate faults and adapt itself to dynamic topology changes. A proof of correctness is provided.

Keywords: random spanning tree, random walks, distributed algorithm, self–stabilizing algorithms.

1. Introduction

A distributed algorithm is an algorithm designed to run on a distributed system where many processes cooperate to solve parts of a given problem in parallel. The problem of efficiently constructing a spanning tree in distributed networks is a central one and is essential for structuring a distributed system. We address the problem of constructing such a structure with a protocol that tolerate faults and adapt itself to dynamic topology changes. In this paper, we introduce Distributed Random Walks (DRW) as a collection of random walks (RW) that cooperate in order to establish a computation. The technique uses a collection of RW that are coalescing into a final one which maintains the control structure. We apply this technique to compute a spanning tree (ST) which is randomly selected among all the possible ones for the network, and to gather informations, we use a wave scheme. We can informally describe the whole procedure as follows: several nodes initiate a RW, with an explorer token. Every node, upon receiving an explorer token, mark himself visited with the identity of the token, except if it has already been visited by another token, and then forwards at random to one of its neighbors the received explorer token. The network is thus, explored in parallel and decomposed into sub–regions, one per token.
Each token constructs a sub-tree of the network. When a node meets another one, or an already visited node, a wave is initiated. This wave is a backward propagation wave that merges one of the sub-trees with the other into one. This process is driven in parallel and eventually, the waves will cover the network, resulting in the ST definition and the protocol is ready for termination when a single explorer token remains and all nodes of the graph are visited. In this paper, we develop a technique for designing algorithms on graphs, especially for an efficient random spanning tree (RST). Our goal is the computation of a RST (i.e. a ST that is chosen randomly among all possible ST).

**The problem.** Let $G(V, E)$ be a connected graph representing a distributed system with real-valued weights $w : E \rightarrow R$ having $n$ vertices and $m$ edges. A spanning tree in $G$ is an acyclic sub-graph of $G$ that includes every vertex of $G$ and is connected; every ST has exactly $n - 1$ edges. We are interested in computing in a distributed way a RST (i.e. a RST is as ST selected among all possible ST on the underlying graph at random).

**Related Works.** In distributed computing, the design of a ST structures for distributed networks has a wide literature. Many papers [GHS83], [LR86], [GKP93] propose distributed solutions and identify interesting properties to construct efficiently and in parallel a ST. Else, the power of RW has also been demonstrated in distributed computing, several authors have successfully designed original solutions for many important control problems such as mutual exclusion [IJ90] or Unique Naming Problem [AEY91], within the self-stabilizing area where the goal is to cope with possible transient failures. We will make use of the attractive techniques found in this area in our solution. E. Chang [Cha92] and A. Segall [Seg83] have introduced the concept of wave in distributed computing with the *Propagation of Information with Feedback* scheme. More recently, [FGL93] and [Baa99] have respectively defined the Distributed Recursive Wave (DRW) and the Distributed Recursive Multi-Wave (DRMW) as a general programming paradigms for distributed systems. We couple these items with a derivation of ST construction and RW techniques to define our general modular technique that we call DRW.

**Contributions** In this paper, we propose a multiple RW scheme combined with a generalized diffusing feedback scheme (waves) that allow a fast RST construction. The main result of this paper, is that the simulation of multiple RW on a connected undirected graph $G$ coupled with some (adequate) path reversal scheme (that we call *waves*) can be used to generate a ST of $G$ at random. The advantages of our scheme are the following: we generate a RST structure that is less subject to failures compare to a deterministically
predetermined ST; the solution is adaptive and deals with topology changes and can be adapted to ad–hoc wireless networks; it can be derived to obtain a self–stabilizing solution; it is parallel, uses the whole power of distributed resources and exhibits a good average running time.

**Outline of the Paper.** The paper is organized as follows: in the next section, we describe the model of distributed computation assumed. In section 3, we describe the algorithm illustrating the use of multiple random walks for selecting a random spanning tree. Thereafter, we show how we implement this and discuss why it is an efficient solution and discuss the key points on creating a RST. Section 4 addresses the informal correctness proof of the algorithm. Finally, in section 5, we give some concluding remarks and open questions.

### 2. Preliminaries

In this section, we give the definitions needed and we introduce some of the tools we use.

**The system.** We model the network as an undirected connected graph $G = (V, E)$ with $V$ the set of nodes ($|V| = n$) and $E$ the set of edges. Each node represents a computer and each link represents a bi–directional communication channel. Each node is associated to a unique identifier. A communication link $(i, j)$ exists iff $i$ and $j$ are neighbors, and is associated to a cost which can vary in time but is always positive. A change in the status of a node is implicitly recognized by the change in the status of its links. We consider the network to be asynchronous. Each node $i$ maintains its set of neighbors, denoted as $N_i$. The degree of $i$ is the number of neighbors of $i$, it is equal to $|N_i|$.

**Processes.** Every process of the distributed systems executes the same code. The program consists of a set of variables and a finite set of rules. A process proceed to an internal action (for example, write to its own variables, compute something or send a message) upon reception of a message.

**Random Walk.** Let us consider a token that moves on a connected undirected graph $G = (V, E)$. At each step, the token goes from the current vertex to one of its neighbors, chosen uniformly at random. This stochastic process is a Markov chain; it is called (simple) random walk on the graph.

**Failures and Self–Stabilization.** A transient fault is a fault that causes the state of a process (its local state, program counter, and variables) to change arbitrarily. An algorithm is called self–stabilizing if it is resilient to transient in the sense that, when started in an arbitrary system state, if no other transient
faults occur, the processes converge to a global legal state after which they perform their task correctly (see [Dij74]).

3. Algorithms description

**Note.** rule to compare tokens:
A token contains two main fields: *color* and *root*. Let $T_1$ and $T_2$ two tokens, the relation between those tokens is:

- $T_1 > T_2$ if $[T_1.color > T_2.color] \lor [T_1.color = T_2.color \land T_1.root > T_2.root]$
- $T_1 = T_2$ if $[(T_1.color = T_2.color \land T_1.root = T_2.root]$

On timeout, flip an unbiased coin and generate a token identified by a color and the node identity, and send *local_state* to all neighbors.
Once all neighbors *local_state* received, if *test_validity_state* is not correct reset(*node*)

**Data Structure.** Each node $p$ maintains
- *color*, the identity of a token
- *master*, the (sub–)tree root which the node belongs to
- *father*, the node father within the (sub–)tree
- *sons* {set of sons}, (optional) is the set of the node sons

The algorithm RST is specified in a pseudo–code form as in [KKM90] for a better understanding. We specify the algorithm behavior by means of overall actions driven by tokens and waves.
Some sites randomly generate token identified by a color and characterized by the initiators of the token.

<table>
<thead>
<tr>
<th><strong>(TA) Token Annexing Mode</strong> whenever a $token_i(color_i, rac_i)$ issued from a node $q$ is annexing (or generated at) node $p$, which belongs to a (sub–)tree [i.e. a $token_j(color_j, rac_j)$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>(TA1)</strong> if $(color_i &lt; color_j)$, the annexing stopped and the token is destroyed.</td>
</tr>
<tr>
<td>• <strong>(TA2)</strong> if $(color_i &gt; color_j)$, one of the 2 conditions holds:</td>
</tr>
<tr>
<td>– (i) one (or more) token(s) are present on node $p$</td>
</tr>
<tr>
<td>– (ii) no other token on node $p$</td>
</tr>
</tbody>
</table>
(TA2-i) if test collision is true:

- if \( \text{token}_i \) is the unique biggest, it continue its traversal and all others are destroyed. Node \( p \) marked himself with \( \text{color} \leftarrow \text{color}_i, \text{master} \leftarrow \text{rac}_i, \text{father} \leftarrow q \)
- if \( \text{token}_i \) is biggest but not unique (others \( \text{token}_{j_1}, \ldots, \text{token}_{j_d} \) has respectively \( \text{color}_{j_1}, \ldots, \text{color}_{j_d} \) equal to \( \text{color}_i, \text{token}_i \) and \( \text{token}_{j_1}, \ldots, \text{token}_{j_d} \) are merged to form the unique token of identity \( i + 1 \) rooted in \( p \) [i.e. \( \text{token}(i + 1, p) \) is generated]
- if \( \text{token}_i \) is not the biggest, it is destroyed.

(TA2-ii) \( \text{token}_i \) continue its traversal scheme. \( p \) marked himself with \( \text{color} \leftarrow \text{color}_i, \text{master} \leftarrow \text{rac}_i, \text{father} \leftarrow q \)

(WU) Wave Update Mode Whenever a \( \text{token}_i(\text{color}_i, \text{rac}_i) \) reaches a node \( p \) with its variable \( \text{color} \) such that \( \text{color} < \text{color}_i \) a wave is generated.

- **WU1** the wave is propagated applying a path reversal scheme over the domain identified by \( \text{color} \) (the domain which \( p \) belongs to)
- **WU2** the wave stops itself when it reaches the \( p \) domain limit.

**Termination** of the algorithm is realized with a derivation of the Dijkstra–Scholten scheme [DS80] known as diffusing computation. This termination detection is periodically initiated by nodes which have initiated an annexing token.

Example. The following example illustrates RST’s construction.
Figure 3.1: Example network. 3 tokens proceed to random walks (token’s id 51, 54 and 31).

Figure 3.2: Evolution of random walks: three explorer tokens are annexing regions of the graph, (token id 51 enter a node within the territory marked 31)
Figure 3.3: Waves: a wave is initiate when a token meets another one or enter the region of the graph already explored by a token. In this example, the meeting of the 51 id token and the 31 id territory initiate a wave (that will flood the 31 id region. Token 51 is killed because it meets a greater territory — 54 —).

Figure 3.4: Evolution of waves: the wave issued from the token with the largest identity merges the subtrees and updates the variables of each node (here the 54 id token wins).
4. Proof

The previous algorithm is a self–stabilizing algorithm. To give the validity proof of such an algorithm, we need (i) to prove the convergence of the system (from any state the system reaches a legal configuration in a finite number of steps) and (ii) to prove the closure (without any failure, any legal transition leave the system in a legal state).

4.1. Results on Random Walks

In order to give such a proof, we need first, to remind some classical results on Random Walks.

**Theorem 1.** Let $R_1$ and $R_2$ be two random walks on a network $G$, in a finite number of steps, $R_1$ and $R_2$ collide :

1. the token of $R_1$ enter a node of $R_2$, or
2. conversely, the token of $R_2$ enter a node of $R_1$, or
3. both situations 1 and 2 hold at the same time.

**Theorem 2.** Let $R$ a random walk on a network $G$, $R$ covers the entire network, visits all the nodes of the network, in a finite time.

**Theorem 3.** Let $R$ a random walk on a network $G$, $R$ visits infinitely often each node $n$ of $G$.

4.2. Specification

We present the specification of our algorithm in two predicates :

$\mathcal{T}S$ which give the set of the legal state for the tokens

$\mathcal{N}S$ which give the set of the legal state for the nodes.

**Definition 1.** Let $i$ be a node of $G$. The state of $i$ is coherent iff the state of $i$ holds for the three following statements :

$\mathcal{N}S_1 : \neg \text{intree}(i) \Rightarrow \text{father} = \text{sons} = \text{color} = \text{NIL}$

$\mathcal{N}S_2 : \begin{align*}
[\text{intree}(i) \land \text{root}(i) \land \neg \text{update}(i, \text{Neigh}_i)] \\
\Rightarrow \text{father} = \text{NIL} \land \forall k \in \text{sons}(i) \; \text{color}(i) = \text{color}(k) \\
\land \text{level}(k) = 1 \land \text{level}(i) = 0
\end{align*}$

$\mathcal{N}S_3 : \begin{align*}
[\neg \text{intree}(i) \land \text{root}(i) \land \neg \text{update}(i, \text{Neigh}_i)] \\
\Rightarrow \text{father} \neq \text{NIL} \land \forall k \in \text{sons}(i) \; \text{color}(i) = \text{color}(k) \\
\land \text{level}(k) = \text{level}(i) + 1 \land \text{color}(i) = \text{color}(\text{father}(i)) \\
\land \text{level}(i) = \text{level}(\text{father}(i)) + 1
\end{align*}$
Specification.

\( TS \ \forall t \ \text{a token present in the system, } \exists i \in V \ \text{such that } root(t) = i \land color(t) = color(i) \land root(i) = true. \)

\( TS \) implies that all tokens present in the system are “correctly linked” to a root node.

\( NS \ \forall n \ \text{a node of } G, \ \text{the state of } n \ \text{is coherent : } NS_1, NS_2, NS_3 \ \text{hold for } n. \)

\( NS \) implies that each construct tree is valid.

The global state of the system is legal if \( NS \) and \( TS \) hold.

4.3. Convergence

In that part, we prove that the system reaches a legal state in a finite number of steps, without any external intervention.

**Lemma 4.1.** An incoherent node eventually detects its incoherent state.

*Proof.* The test of the local states is done by the exchange of the local values. This exchange is driven by a timeout, so the executions of the algorithm cannot prevent the exchange on the local states. Even more, the test of the local state is driven by the reception of the states of all neighbors. So each node is able to compute its state in a finite time.

**Lemma 4.2.** An incoherent node restores eventually itself in a coherent state.

*Proof.* According to lemma 4.1, each node is able to compute the validity of its local state. So if that state is illegal, the node corrects it in a finite time. The new valid state of the node is the initial state: the node does not belong to any tree.

**Remark 1.** As each node is able to compute, in a finite time, the validity of its local state, and as timeouts are reasonnably set, if the system is in an illegal configuration, at least one node will be able to create a valid new token in a finite time.

**Lemma 4.3.** Each token \( t \) non associated to a root node \( i \) is eventually destroyed.

*Proof.* Let \( IT \) be the set of non associated root node token circulating in the system. According to remark 1, new legal token, *i.e.* token associated to new valid root node, will be created in a finite time. We can claim that in a finite time a legal token \( LT \) will be created, such that \( LT > \max(IT) \). Theorem 1 allow us to claim that all illegal tokens will be destroyed in a finite time.

So in a finite time, all non associated root node token will be destroyed.
Lemma 4.4. Each token $t$ associated to a root $i$ such that $\text{color}(t) \neq \text{color}(i)$ is eventually destroyed

Proof. According to theorem 3 each token visits eventually often each node of the system. So each time a token visits its associated root node, this one is able to test the equality of the colors, and if it’s not the case, the node destroy the token.

Corollary 4.1. There exists a region $\mathcal{R}_1$ where $TS$ holds.

In the following, we place ourselves in $\mathcal{R}_1$.

Lemma 4.5. All node eventually becomes coherent.

Proof. According to lemmas 4.1 and 4.2 each node of the system detects and corrects, if needed, its local state. Even more, corrolary 4.1 guarantee us that all tokens, present in the system, are legal tokens. So the system become coherent.

Corollary 4.2. There exists a region $\mathcal{R}_2$ where $TS$ and $NS$ hold.

4.4. Closure

To show the closure we place ourselves in $\mathcal{R}_2$

Lemma 4.6. The annexion of a new node do not provoke any detectable and correctible incoherence.

Proof. When a node $i$ is annexed, the annexing operations suspend the local testing operations. $i$, and the neighbors of $i$ are not able to compute the state, and so, not able to correct themselves if they detect an incoherent state. It’s only when the wave update guard will be executed that the node will be able to compute its state.

Lemma 4.7. The wave update operations do not provoke any detectable and correctible incoherences.

Proof. Similarly to the proof of lemma 4.6 the wave update operations suspend the corrections and the tests of the state. Once the wave finished, the system is coherent and nodes are able to compute and correct their states.

Lemma 4.8. Once an annexion and a wave update executes, the system is in a coherent state.

Proof. Once all the operations of a wave update complete, a new sub–tree has been computed. The local state of each node of that new sub–tree has been set up to a coherent state by the algorithm.
Corollary 4.3. The execution of any part of the algorithm leaves the system in a valid state.

4.5. Construction of an unique tree

Lemma 4.9. Each node of $G$ eventually belongs to a tree.

Proof. Let $V_F$ be the set of nodes of the system that do not belong to any tree. At regular intervals, each node $i$ of $V_F$ can create a new token. In that case the size of $V_F$ decrease, because a node creating a new token belongs to its own tree. Even more, if $i$ does not create a new token theorem 2 insure that $i$ will be reached, at least, by a token in a finite time, and in that case, the size of $V_F$ decrease. So in all cases the size of the set $V_F$ decrease strictly to zero. So in a finite time, all nodes belong to a tree.

Lemma 4.10. There eventually remain only the greatest color token in the system.

Proof. According to lemma 4.9, the system reaches, in a finite time, a state where all nodes belong to a tree. In that state, there remain in the network, only a finite number of tokens.

If in the remaining tokens, there is a unique greater one, theorem 1 insure that all tokens of the system will collide with in a finite time, lastly it will remain only one token in the system.

If the greater token is not unique, in all cases, if the associated node of the different greatest token are the same or not, those tokens will collide in only one in a finite time. The resulting token will be the greatest in the system.

So there eventually remains only one token in the system, and the color of the remaining unique token is the greatest one present in the system.

Lemma 4.11. All nodes of the system eventually belongs to the same tree.

Proof. According to lemma 4.10 there eventually remain only the token with the greatest color in the system. That token will annexe all the trees of the system.

Theorem 4. Without any fault, the constructed tree is never modified.

Proof. If the resulting unique tree is modified, that means that :

1. the structure of the network has changed
2. a new tree is being constructed
3. the structure of the constructed tree is modified.
All precedent cases are impossible because, the case 1 implies a network modification so a fault: impossible. The case 2 implies that a new tree is in construction, so a new token with a greater color has been generated, but only root nodes or non associated nodes are able to create a new token, so it’s impossible because all nodes belongs to the unique tree, and if the root node creates a new token, the color of that new token is the one of the current token of the tree. Finally, the case 3 implies the annexion of a set of nodes that belongs to the tree: impossible.

Without the occurrence of a fault, the constructed tree is never modified. □

5. Conclusions and Future Works

We have presented DRW an efficient scheme for constructing a uniform spanning tree over a distributed network. It is based on a new family of algorithms for distributed computing called the distributed random walks scheme algorithms. The key advantage of this novel approach is that it has a very moderate and the best balanced impact on network and computer resources. RW are expected to converge to one that keeps a surveillance function to relaunch computation in case of links or nodes failures.

However, the estimate of the complexity of DRW algorithms for the general case is more complicated than for usual distributed computing algorithms the difficulty is due to the interactions between the random walks.

References


Distributed Random Walks and the Design of a Self–Stabilizing RST


Authors addresses:

Hichem Baala
Université de Technologie de Belfort–Montbéliard
Département Génie Informatique
Laboratoire Système et Transport
90010 Belfort Cedex
Email. Hichem.Baala@utbm.fr

Olivier Flauzac, Université de Technologie de Compiègne
Laboratoire Heudiasyc, Centre de Recherches de Royallieu
60205 Compiègne Cédex, France
Email. Olivier.Flauzac@univ-reims.fr