A Fault-contained Spanning Tree Protocol for Arbitrary Networks

J. El Haddad
Lamsade, UMR 7024
University of Paris Dauphine
Place de Lattre de Tassigny
75775 Paris Cedex 16, France

S. Haddad
Lamsade, UMR 7024
University of Paris Dauphine
Place de Lattre de Tassigny
75775 Paris Cedex 16, France

Abstract

Fault-containing self-stabilizing algorithms, introduced by A. Gupta, fulfill two requirements: they converge to a correct behavior starting from an arbitrary state and they *quickly* stabilize starting from a state corrupted by a single fault. Such algorithms are obtained either by an ad hoc transformation of a self-stabilizing algorithm or by a generic transformation which produces a slower stabilization in case of a single fault. In this paper, we transform a self-stabilizing algorithm for constructing a rooted spanning tree in an arbitrary network. This algorithm has two specific features with respect to previously adapted algorithms: there is no distinguished node (i.e. each site execute the same code), and the principle of stabilization involves a global coordination through requests and replies following paths of the communication graph.

1 Introduction

The design of self-stabilizing distributed algorithms has emerged as an important research area in recent years. It was first introduced in a well-known paper by Dijkstra [3]. Since then self-stabilizing protocols have been designed for a large variety of problems and underlying principles have been explored [4, 10, 9]. The principle of self-stabilization is to guarantee convergence of the system to some desired stable state from an arbitrary initial state arising out of an arbitrarily large number of faults. The first designed self-stabilizing protocols were global. That is, a small number of transient faults causes the entire system to start a global convergence activity. This recovery process does not scale to modern large networks. The above discussion motivates several approaches as *fault local*, *time adaptive* or *fault containment* to have fast convergence to a correct configuration of the system in a graceful way following the occurrence of a limited number of faults.

We present some of the research works that have introduced or used theses approaches. Kutten and Peleg introduced the *fault locality* approach in [8], as well as a persistent bit algorithm recovering from a corruption of one bit at some k nodes. Afek and Dolev presented in [1] a *time adaptive* algorithm that converts any algorithm (fixed-output, non-fixed-output, or even interactive) to cope with f faults within $O(f)$ cycles. Gupta introduced the *fault containment* approach in [5, 6, 7]. The goal of this technique is to ensure that the system is self stabilizing, and during recovery from a single transient fault, only a small number of processes will be allowed to execute recovery actions. Gupta presented three examples that illustrate the feasibility of designing efficient fault-containment self-stabilizing protocols for important problems: a leader election protocol that recovers in $O(1)$ time from a state with a single transient fault on oriented ring [5], a breadth-first search spanning tree construction protocol in an arbitrary network [7], and spanning-tree protocol in an undirected network [6].

In this paper, we carry on towards *fault containment* by presenting an algorithm for spanning-tree task. More precisely, we modify an earlier self-stabilizing algorithm for spanning tree construction on general networks by Afek, Kutten and Yung [2] to derive a fault-contained self-stabilizing algorithm. Starting from any arbitrary state, our algorithm converges to a state with a spanning tree rooted at the largest identity node, in a finite number of steps. The specific difficulty of this transformation lies on the nature of the original algorithm. That is, unique identities are used for the nodes to break symmetry and a global coordination is involved through requests and replies following paths of the communication graph.

The rest of the paper is organized as follows. Section 2 overviews the model of computation. Section 3 contains a short description of the original spanning tree algorithm of Afek, Kutten and Yung. Section 4 includes a detailed description of its transformation into a fault-contained self-stabilizing algorithm. Section 5 contains some concluding remarks and perspectives.

2 Model

We consider the model of a network of processors communicating via shared memory. The network is presented by a graph $G = (V, E)$, where V denotes the set of nodes representing the processors, and E denotes the set of edges representing the links. Each node $v \in V$ correspond to a process, and an edge $(u, v) \in E$ means that u and v directly communicate. Each processor has a unique identity, the only assumption that make our model *non uniform*. The total number of processors, n, is unknown to the processors of the network.

Each processor has a set of local variables. A local variable belonging to a process i can be read by i and any of its neighbors, but can written into only by i. Thus, a process can directly communicate with only its neighbors in the network by reading their local variables, and writing into its own local variables. Reads of neighbors memory as well as read and write on local memory are atomic operations. The local computation at each processor consist of a sequence of transitions where each transition consists of an operation that moves the processor from its state before the transition to another state.

We assume that each process i has a program that consists of a finite set of guarded statements of the form, $label$: $Precon \longrightarrow Action$ where, $Precon$ is a boolean predicate involving the local variables of i and the local variables of its neighbors, and $Action$ is an assignment that modifies the local variables in i. The $Action$ is executed only if the corresponding guard $Precon$ evaluates to true, in which case we say statement $label$ is enabled.

3 Spanning tree construction

Afek, Kutten, and Yung [2] proposed a stabilizing algorithm for computing a spanning tree in an asynchronous network of processors that communicate through shared memory. The processors have unique identifiers but are otherwise identical. Informally the algorithm can be described as follows. Every node tries to build a tree in the network rooted from itself. The construction initiated by a node with a larger identity overruns the one of a node with a lower identity. Eventually, the building initiated by the largest *id* node will overrun all the other trees constructions. A node leaves its tree when it detects a local inconsistency. However, to join another tree whose root identity is larger, a node has to propagate a request message along the new tree branches to the root and to receive a grant message back. This mechanism prevents the necessity to know additional information such as the network size. Of course these messages propagate through the shared memories of the nodes via a sequence of read and write operations. To establish self-stabilization, Afek *et al.* defined certain local conditions which enable a node to detect inconsistencies and to "restart" the building process by considering itself the root of a single node tree.

A formal description of the algorithm is shown in figure 1 for the read-all state model. Each node i maintains the variables $root_i$, par_i and dis_i to describe the tree structure with the following predicates :

$$
\begin{array}{lcl}
Root(i) & \equiv & root_i = i \wedge dis_i = 0 \\
Child(i,j) & \equiv & root_i = root_j > i \wedge \\
 & & par_i = j \in Neigh_i \wedge dis_i = dis_j + 1 \\
Tree(i) & \equiv & root(i) \vee \exists j : Child(i,j) \\
Lmax(i) & \equiv & \forall j \in Neigh_i : root_p \geq root_j \\
Sat(i) & \equiv & Tree(i) \ \wedge \ Lmax(i)
\end{array}
$$

Moreover each node maintains variables related to the passing of the request and the grant messages. Variable req_i contains the node whose join request i is currently processing, $from_i$ is the neighbor from whom i read the request, to_i is the neighbor to which i forwards it, and dir_i indicates whether the request is granted. As for the tree related variables, the following predicates describe the request forwarding mechanism :

$$
\begin{array}{lcl}
Idle(i) & \equiv & req_i = from_i = to_i = dir_i = \perp \\
Asks(i,j) & \equiv & ((root(i) \wedge req_i = i) \vee Child(i,j)) \wedge \\
 & & to_i = j \wedge dir_i = ask \\
Forw(i,j) & \equiv & req_i = req_j \wedge from_i = j \wedge to_j = i \wedge \\
 & & to_i = par_i \\
Grant(i,j) & \equiv & Forw(i,j) \wedge dir_i = grant
\end{array}
$$

To achieve self-stabilization of the spanning tree construction, all the nodes must satisfy the condition defined by $\forall i$: $Sat(i)$. A node i with $Sat(i)$ false attempts to establish it by becoming the child of its neighbor j with the highest value of $root_j$. Joining j's tree by i is performed in three steps. First, i becomes a root (action $\mathbf{B}_i$), then asks for join permission (action $\mathbf{A}_i$), and finally joins when j grants the permission (action $\mathbf{J}_i$). The remaining four actions ($\mathbf{C}_i$, $\mathbf{F}_i$, $\mathbf{G}_i$ and $\mathbf{R}_i$) implement the request forwarding mechanism

$\mathbf{B}_i$
Precon : $\neg Tree(i)$
Action : $root_i = par_i = i; dist_i = 0;$
$\quad req_i = from_i = to_i = dir_i = \perp;$

$\mathbf{A}_i$
Precon : $Tree(i) \wedge \neg Lmax(i)$
Action : Select $j \in Neigh_i$ with maximal value of $root_j$;
$\quad req_i = i; from_i = i; to_i = j; dir_i = ask;$

$\mathbf{J}_i$
Precon : $Tree(i) \wedge \neg Lmax(i) \wedge Grant(to_i, i)$
Action : $root_i = root_j; par_i = j; dist_i = dist_j + 1;$
$\quad req_i = from_i = to_i = dir_i = \perp;$

$\mathbf{C}_i$
Precon : $Sat(i) \wedge \neg\exists j : Forw(i,j) \wedge \neg Idle(i)$
Action : $req_i = from_i = to_i = dir_i = \perp;$

$\mathbf{F}_i$
Precon : $Sat(i) \wedge Idle(i) \wedge Asks(j,i)$
Action : $req_i = req_j; from_i = j; to_i = par_i; dir_i = ask;$

$\mathbf{G}_i$
Precon : $Sat(i) \wedge Root(i) \wedge Forw(i,j) \wedge dir_i = ask$
Action : $dir_i = grant;$

$\mathbf{R}_i$
Precon : $Sat(i) \wedge Grants(par_i, i) \wedge dir_i = ask$
Action : $dir_i = grant;$

Figure 1: Afek, Kutten and Yung's spanning tree algorithm

and are only executed by a node i with $Sat(i)$ true.

However, Afek, Kutten and Yung's algorithm has a poor fault-containment features during recovery from a single transient fault. In fact faults are detected locally but corrected by global operation. Consider the graph shown in figure 2(a). The solide lines show edges in the constructed spanning tree, with the arrows indicating the parent of each node. The root and the level of each node are shown beside the node. The dashed lines show edges in the graph that are not in the constructed spanning tree. Let a fault at node i set par_i to k, resulting in the state shown in figure 2(b). In this 1-faulty state, $\mathbf{B}_i$ is enabled since $\neg tree(i)$ is true, as $dist_i = dist_k + 1$. After node i executes action $\mathbf{B}_i$, it becomes a root and make a join request to j's tree. Then each node p between i and the root r will participate in the request/reply mechanism by executing the actions $\mathbf{C}_p$, $\mathbf{F}_p$, and $\mathbf{R}_p$. As for the root, it executes once actions $\mathbf{C}_r$ and $\mathbf{G}_r$ before i joins the tree by executing $\mathbf{J}_i$. Moreover, in the worst case (i.e. the tree is a chain), all nodes in the system may change their states before the system comes back to a correct state.

In the following section, we show how to modify Afek, Kutten, and Yung's algorithm to obtain

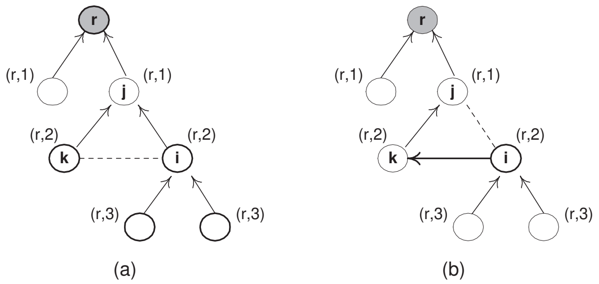

Figure 2: A fault not locally contained

a fault-contained self-stabilizing spanning tree algorithm. Starting from a 1-faulty state (a state resulting from a single transient fault occurring in a correct state), only the faulty node or one of its neighbors make any local state changes during recovery to a correct state.

4 Fault-contained spanning tree

Our algorithm requires that a node checks the state of some neighbors within distance 2 from it. Thereby, we will strengthen our model of computation by assuming that each node i, in addition to having read-access to neighbors' local variables, also has a read-access to the local variables of neighbors' neighbors.

The idea of the fault-contained spanning tree version of the algorithm is that when a node detects a local inconsistency, it does not necessarily make a correction move, instead it first attempts to determine the cause of this local inconsistency. If a node i finds that the local inconsistency may be due to a fault at i alone, then it corrects its local state appropriately. Otherwise, inconsistency may be due to a fault at a neighbor j, then the node i waits for j to correct its fault.

We now precise this idea. Let $\mathbb{N}_f$ be a finite interval of $\mathbb{N}$ that denotes the set of identities in the network and $Neigh_i$ be the set of neighbors of i. In the rest of this paper, we denote by i the identity of a node i that maintains two groups of variables :

1. The variables related to the tree structure:
   - $root_i$ holds the identity of the root of the tree to which node i belongs. $root_i$ is equal to some node in $\mathbb{N}_f$;
   - $dist_i$ is the distance from node i to its root. It is an integer in the range $[1 \ldots |\mathbb{N}_f|]$;

- par_i is the identity of a neighbor of i which is the parent of i in the tree. It is equal to some node in $Neigh_i \cup \{i\}$.

2. The variables related to passing the *request* and *grant* messages:
   - req_i is either the identity of a node that is currently requesting to join the tree to which i belongs, or equal to i if i itself is trying to join another tree. It is equal to some node in $\mathbb{N}_f \cup \{\perp\}$;
   - $from_i$ is either the identity of the neighbor from which i copied the value of req_i, or i if i has initiated a request in an attempt to join a new tree. It is equal to some node in $Neigh_i \cup \{i\} \cup \{\perp\}$;
   - to_i is the identity of neighbor of i through which it is trying to propagate the request message. It is equal to some node in $Neigh_i \cup \{i\} \cup \{\perp\}$;
   - dir_i is either ask, to signify that the node whose identity is in req_i wishes to join the tree, or $grant$ to signify that this request has been granted. It is equal to some value in $\{ask, grant, \perp\}$.

To describe the tree structure, we introduce the following predicates:

$$Child(i) \equiv \{j \in Neigh_i : \ par_j = i\}$$

$$\begin{aligned} Tree(i) \equiv\ & (root_i = i \ \wedge \ dist_i = 0 \ \wedge \ par_i = i) \vee \\ & (par_i \neq i \ \wedge \ root_i = root_{par_i} > i \wedge \\ & \ dist_i = dist_{par_i} + 1) \end{aligned}$$

$$\begin{aligned} Ownrmax(i) \equiv\ & if\ (\forall k \in Neigh_i : root_i \geq root_k)\ then\ i \\ & else\ j = \{Min_{k \in Neigh_i} : root_j \geq root_k\} \end{aligned}$$

$$Sat(i) \equiv Tree(i) \ \wedge \ Ownrmax(i) = i$$

To describe the request/reply mechanism, we introduce the following predicates :

$$Idle(i) \equiv req_i = \perp \wedge \ from_i = \perp \wedge \ to_i = \perp \wedge \ dir_i = \perp$$

$$\begin{aligned} Asks(i,j) \equiv\ & j \in Neigh_i \ \wedge \ req_i = i \ \wedge \ from_i = i \wedge \\ & to_i = j \ \wedge \ dir_i = ask \ \wedge \ Sat(j) \wedge \\ & root_i = i < root_j \ \wedge \ par_i = j \ \wedge \\ & \forall k \in Neigh_i : \\ & (root_k = root_j \Rightarrow dist_k = dist_j + 2) \end{aligned}$$

$$\begin{aligned} ForwNa(i,j) \equiv\ & j \in Neigh_i \cup \{i\} \ \wedge \ to_i = j \wedge \\ & par_i = j \ \wedge \ dir_i = ask \ \wedge \ Sat(i) \wedge \\ & from_i \neq i \ \wedge \ req_i \neq i \ \wedge \ req_i \neq \perp \wedge \\ & req_i = req_{from_i} \ \wedge \ \forall k \in Child(i) : \\ & (root_k = root_i \Rightarrow dist_k = dist_i + 1) \end{aligned}$$

$$\begin{aligned} ForwA(i,j) \equiv\ & j \in Neigh_i \cup \{i\} \ \wedge \ to_i = j \wedge \\ & par_i = j \ \wedge \ dir_i = grant \ \wedge \ Sat(i) \wedge \\ & from_i \neq i \ \wedge \ req_i \neq i \ \wedge \ req_i \neq \perp \wedge \\ & req_i = req_{from_i} \ \wedge \ \forall k \in Child(i) : \\ & (root_k = root_i \Rightarrow dist_k = dist_i + 1) \end{aligned}$$

$$\begin{aligned} Rgcorrect(i) \equiv\ & Idle(i) \ \vee \ \exists j : ForwNa(i,j) \vee \\ & (\exists j : ForwA(i,j) \ \wedge \ dir_{from_i} = ask) \end{aligned}$$

$$Grants(i,j) \equiv from_i = j \ \wedge \ dir_i = grant$$

Before we give a detailed description of the algorithm, we investigate the behavior of the algorithm under a single transient fault. Let a legitimate state of the system be defined as follows. A global state of the system is a legitimate state, if and only if the neighborhood of each node i in the graph is coherent, that is

$$\begin{aligned} Coherent(i) \equiv\ & Tree(i) \wedge \\ & \forall k \in Child(i) : dist_k = dist_i + 1 \wedge \\ & \forall k \in Neigh_i : root_k = root_i \end{aligned}$$

It is easy to show that if a single transient fault occurs at a node i and this node is the root of the tree, then a stable state is reached again by a single move of node i setting its parent and root variables values to itself, its distance variable value to 0 and resetting its requests variables. Otherwise, node i tries to determine the reasons of the local incoherence by checking predicate $Onelocalfault(i, d, j)$ with d the current value of the distance variable of i, j a neighbor of i as well as its parent with the value $d - 1$ in $dist_j$. Note that, if node i is a leaf then many couples (d, j) are possible. Otherwise, only one value of d is possible and hence the number of couples (d, j) is restrained. First, node i checks if all its neighbors have the same root value r, what is greater than i and greater than or equal to their identities, a parent different from it and a positive distance value. If root node r is a neighbor of i, then it checks whether r has a distance value equal to 0 and a parent id equal to r. In this case, node i concludes that incoherency may be due to either a false value in its root variable or an incorrect, value in its distance variable, towards its parent or one of its children. A formal and detailed description of this predicate follows.

$$Incoherent(i) \equiv \neg Coherent(i)$$

$Onelocalfault(i, d, j) \equiv j \in Neigh_i \ \wedge \ \exists r > i :$

1. $Incoherent(i)$ (*one fault at i*)
2. $\forall k \in Neigh_i :$ (*coherent neighborhood*)
   - $root_k = r \ \wedge \ k \leq r \ \wedge \ (k = r \Rightarrow dist_r = \ 0 \ \wedge \ par_r = r)$

- $k \neq r \Rightarrow par_k \neq k \wedge dist_k > 0$

3. $\exists k : Neigh_i = \{k\} \Rightarrow$ (*case of a unique neighbor*)
   - $\forall k' \in Neigh_k \setminus \{i\} : root_{k'} = r \wedge k' \leq r \wedge (k' = r \Rightarrow dist_r = 0 \wedge par_r = r)$
   - $(k \neq r \wedge par_k \neq i) \Rightarrow dist_k = dist_{par_k} + 1$
   - $\forall k' \neq i \in Child(k) : dist_{k'} = dist_k + 1$
4. $\forall k \in Child(i) : dist_k = d + 1 \wedge dist_j = d - 1$ (*eventual correction with the help of j*)
5. (*elimination of false detection due to a unique fault at a child*)
   $root_i = r \wedge dist_i = dist_{par_i} + 1 \wedge Idle(i) \Rightarrow$
   - $|Child(i)| > 1 \vee$
   - $\exists k : Child(i) = \{k\} \wedge \exists k' \neq i \in Child(k) \wedge (\forall k' \neq i \in Child(k) : dist_{k'} = dist_k + 1 \vee par_i = k)$
6. (*elimination of false detection due to a unique fault at the parent*)
   $(root_i = r \wedge \forall k \in Child(i) : dist_k = dist_i + 1 \wedge Idle(i) \wedge par_{par_i} \neq i) \Rightarrow$
   $(dist_{par_i} = dist_{par_{par_i}} + 1 \wedge \forall k' \neq i \in Child(par_i) : dist_{k'} = dist_{par_i} + 1)$
7. (*elimination of cross parent*)
   $(root_i = r \wedge Idle(i) \wedge par_{par_i} = i \wedge par_i \neq i) \Rightarrow \forall k \in Child(par_i) \setminus \{i\} : dist_k = dist_{par_i} + 1$

In order to achieve fault containment, in a state where a single transient fault occurred at a node i, we must inhibit each child of i to presume that it is the faulty node of the system and hence to leave the tree by becoming a root. This allows us to define a new predicate $Onelocalparfault(i, d, j)$ under which each child of a node may check if its parent estimates to be the faulty node of the system.

$Onelocalparfault(i, d, j) \equiv par_i \neq i \wedge j \in Neigh_{par_i} \wedge \exists r > par_i :$

1. $Incoherent(par_i)$ (*one fault at par_i*)
2. $\forall k \in Voisins_{par_i}$: (*coherent neighborhood*)
   - $root_k = r \wedge k \leq r \wedge (k = r \Rightarrow dist_r = 0 \wedge par_r = r)$
   - $k \neq r \Rightarrow par_k \neq k \wedge dist_k > 0$
3. $Voisins_{par_i} = \{i\} \Rightarrow$ (*case of a unique neighbor*)
   - $\forall k' \in Neigh_i \setminus \{par_i\} : root_{k'} = r \wedge k' \leq r \wedge (k' = r \Rightarrow dist_r = 0 \wedge par_r = r)$
   - $\forall k' \neq par_i \in Child(i) : dist_{k'} = dist_i + 1$
4. $\forall k \in Child(par_i) : dist_k = d + 1 \wedge dist_j = d - 1$ (*eventual correction with the help of j*)
5. (*elimination of false detection due to a unique fault at a child*)
   $(root_{par_i} = r \wedge dist_{par_i} = dist_{par_{par_i}} + 1 \wedge Idle(par_i)) \Rightarrow$
   - $|Child(par_i)| > 1 \vee$
   - $Child(par_i) = \{i\} \wedge \exists k' \neq par_i \in Child(i) \wedge (\forall k' \neq par_i \in Child(i) : dist_{k'} = dist_i + 1 \vee par_{par_i} = i)$
6. (*elimination of cross parent*)
   $(root_{par_i} = r \wedge Idle(par_i) \wedge par_{par_i} = i \wedge par_i \neq i) \Rightarrow \forall k \in Child(i) \setminus \{par_i\} : dist_k = dist_i + 1$

Next, we introduce a predicate analogous to $Onelocalparfault(i, d, j)$ where the parent of the node i is the faulty node and the root of the tree

$$
\begin{aligned}
Onerootparfault(i) \equiv\ & par_i \neq i \wedge \\
& \forall k \in Neigh_{par_i} : (k < par_i) \wedge (root_k = par_i) \wedge \\
& \forall k \in Neigh_i \setminus \{par_i\} : k < par_i \wedge root_k = par_i \wedge \\
& \forall k \in Child(par_i) : dist_k = 1
\end{aligned}
$$

The following predicate prevents a node i from leaving the tree in case of a faulty parent $par_i \neq i$

$$
\begin{aligned}
Oneparfault(i) \equiv\ & Onerootparfault(i) \vee \\
& \exists (d, j) : Onelocalparfault(i, d, j)
\end{aligned}
$$

A formal description of the algorithm is given in Figure 3. A more detailed description of the guarded statements follows. After the occurrence of a fault, if a non-root node i estimates being the only faulty node in the network, it executes the statement $\mathbf{O}_i(j)$, j being its parent. Joining j's tree is done in three steps. First, if node i estimates having an incoherent tree structure, it attempts to establish $Tree(i)$ by becoming a root (action $\mathbf{B}_i$) if it doesn't estimates that its parent is the only faulty node in the network. This action consists of becoming a root and resetting request variables. Then node i, with a coherent tree structure and a *root* value smaller then any *root* value of its neighbors, attempts to join a tree by becoming the child of its neighbor j with the highest root value and asking it for a join permission (action $\mathbf{A}_i$). In order to execute this action, node i must not assume that j is the only faulty node in the network, and all its children with the same *root* value as j must have a correct distance value. Finally, node i joins the tree when j grants the permission (action $\mathbf{J}_i$).

The remaining actions ($\mathbf{C}_i$, $\mathbf{F}_i(j)$, $\mathbf{G}_i$ and $\mathbf{R}_i$) implement the request/reply mechanism and are executed by a node i with $Sat(i)$ true. A node i clears

$\mathbf{O}_i(j)$ (*One fault begins to be corrected*)
Precon : $\exists d : Onelocalfault(i,d,j) \wedge \forall k : \neg Asks(i,k)$
Action : $root_i = i; par_i = j; req_i = i;$
$from_i = i; to_i = j; dir_i = ask;$

$\mathbf{B}_i$ (*Become root*)
Precon : $\forall(k,d) : \neg Onelocalfault(i,d,k) \wedge \neg Tree(i) \wedge$
$\forall k : \neg Asks(i,k) \wedge \neg Oneparfault(i)$
Action : $root_i = par_i = i; dist_i = 0;$
$req_i = from_i = to_i = dir_i = \perp;$

$\mathbf{A}_i$ (*Ask permission to join*)
Precon : $Tree(i) \wedge Ownrmax(i) \neq i \wedge Sat(Ownrmax(i)) \wedge$
$\forall k \in Child(i) : (root_k = root_{Ownrmax(i)} \Rightarrow$
$dist_k = dist_{Ownrmax(i)} + 2)$
Action : $req_i = i; from_i = i; to_i = Ownrmax(i);$
$dir_i = ask; root_i = i; par_i = Ownrmax(i)$

$\mathbf{J}_i$ (*Join tree*)
Precon : $\exists j : Asks(i,j) \wedge Grants(j,i) \wedge dist_j + 1 \in \mathbb{N}_f$
Action : $root_i = root_j; par_i = j; dist_i = dist_j + 1;$
$req_i = from_i = to_i = dir_i = \perp;$

$\mathbf{C}_i$ (*Clear request variables*)
Precon : $\forall(k,d) : \neg Onelocalfault(i,d,j) \wedge Sat(i) \wedge$
$\neg Rgcorrect(i)$
Action : $req_i = from_i = to_i = dir_i = \perp;$

$\mathbf{F}_i(j)$ (*Forward request*)
Precon : $Sat(i) \wedge Idle(i) \wedge (Asks(j,i) \vee ForwNa(j,i))$
Action : $req_i = req_j; from_i = j; to_i = par_i; dir_i = ask;$

$\mathbf{G}_i$ (*Grant join request*)
Precon : $Sat(i) \wedge i = root_i \wedge ForwNa(i,i)$
Action : $dir_i = grant;$

$\mathbf{R}_i$ (*Relay grant*)
Precon : $Sat(i) \wedge \exists j : (ForwNa(i,j) \wedge Grants(j,i))$
Action : $dir_i = grant;$

Figure 3: The fault-contained algorithm

the variables for request processing (action $\mathbf{C}_i$) if it is not currently processing a request or a grant has been received by the descendant of i and the variables are not already undefined. A non-root node participates in the process of forwarding requests and grants in its tree in order to enable addition of new nodes to the tree. More precisely if a node i is idle but has a child j with a request for i (a root that wants to join, or a child of i forwarding a request), node i may start forwarding the request (action $\mathbf{F}_i(j)$). If a node i is a root and forwards a request, it will grant it (action $\mathbf{G}_i$), and if i forwards the request to its parent and the parent grants it, then i relays the grant (action $\mathbf{R}_i$).

5 Summary

In this paper, we have presented a self-stabilizing algorithm for building a spanning tree in an arbitrary asynchronous network. The interest of this work is that the effects of a single fault are tightly contained in a very small neighborhood around it. The proposed algorithm runs under the strong assumption that a node i can directly read the local variables of it's neighbor's neighbors. The next stage of our research is to prove that it is possible to extend our result by a simple scheme that removes this additional assumption.

References

[1] Y. Afek and S. Dolev. Local stabilizer. *In Proceedings of the 5th Israel Symposium on Theory of Computing and Systems*, 1997.

[2] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilizing protocols for general networks. *In WDAG90: Distributed Algorithms 4th International Workshop Proceedings, LNCS*, 486:15–28, 1990.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[4] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[5] S. Ghosh and A. Gupta. An exercise in fault-containment : self-stabilization leader election. *Information Processing Letters*, 59:281–288, 1996.

[6] S. Ghosh, A. Gupta, and S. V. Pemmaraju. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing and Information*, 2:322–338, 1996.

[7] S. Ghosh, A. Gupta, and S. V. Pemmaraju. Fault-containing network protocols. *In Proceedings of the 12th Annual ACM Symposium on Applied Computing*, 1997.

[8] S. Kutten and D. Peleg. Fault-local distributed mending. *In Proceedings of the 14th Annual ACM Symposium on Principles of Distributecd Computing*, 1995.

[9] M. Schneider. Self-stabilization. *ACM Symposuim Computing Surveys*, 25:45–67, 1993.

[10] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000.