A Self-stabilizing Algorithm for Maintaining A Spanning Tree in a Distributed System

Vijay K. Garg *
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Anurag Agarwal
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233, USA
anurag@cs.utexas.edu

Abstract

Self stabilizing algorithms have always been of interest to researchers due to their attractive property of keeping a system in legal state even after the occurence of data fault. In this paper, we propose a self stabilization algorithm for maintaining a spanning tree using a labeled tree encoding. Our work is different from other self stabilizing algorithms for spanning trees in the way it deals with the notion of legal and illegal states. The *core part of the data structures always remains in the legal states and are used to regenerate the* noncore components. The algorithm proposed also gives efficient ways to locally check for consistency of common data structures which can be used for other algorithms as well.

Keywords: Fault Tolerance, Self Stabilization, Spanning Tree

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

1 Introduction

Spanning trees have many uses in computer networks. Once a spanning tree is established in a network, it may be used in broadcast of a message, convergecast, β synchronizer, and many other algorithms. In many of these applications, the spanning tree is rooted, i.e., one of the node is distinguished as a root and all edges point to the direction of the root. The goal of this paper is to propose a distributed algorithm to maintain a spanning tree in a network in spite of ”data” faults. A ”data” fault corresponds to change in value of one or more variable of a program due to some unforeseen error. For example, a spanning tree is usually implemented using *parent* variables. Every node in the computer network maintains the *parent* pointer which point to the parent in the spanning tree. What if one or more of the *parent* pointers get corrupted? Now, the *parent* pointers may not form a valid spanning tree. Such errors will be called ”data” faults. An algorithm to maintain a spanning tree is called self-stabilizing if it ensures that whenever a fault occurs, the system gets back into a *legal* state in a finite number of steps provided no faults occur for a sufficiently long time. In this application, by the legal state we mean that the *parent* pointers form a valid rooted spanning tree.

The idea of self-stabilizing algorithms first appeared in Dijkstra [Dij74], where three self-stabilizing algorithms were presented for mutual exclusion in a ring. The idea has received considerable attention in recent times and the reader is referred to [Dol00] for a book devoted to self-stabilizing algorithms. The idea of a self-stabilizing spanning tree is also not new and many versions are described in [Dol00]. For example, the algorithm to maintain a breadth-first-search (BFS) rooted spanning tree is typically based around the idea of using an additional variable *dist* with each node. The algorithm is given by the following two rules.

```
var dist:0..n; parent:node identifier;

Root:
     execute periodically
        dist := 0;
        parent := null;

All other nodes:
     Let x be a neighbor with the smallest distance
              dist := dist[x]+1;
          parent := x;
```

It can be easily shown that the algorithm will converge to a BFS tree irrespective of initial values of the variables. Our goal in this paper is two fold. We first show an alternative algorithm for maintaining a spanning tree (not necessarily a BFS tree) and more importantly a new technique by which self-stabilizing algorithms can be constructed for maintaining various data structures in a distributed system.

As usual, we assume that the system states can be divided into legal and illegal states. The definition of the legal state is dependent on the application. When a fault occurs, the system moves to an illegal state and if the system is not designed properly, it may continue to execute in illegal states. A system is called *self-stabilizing* if regardless of the initial state, the system is guaranteed to reach a legal state after a finite number of moves provided there are no faults for a sufficiently long period.

It is well known that one can design self-stabilizing algorithms with *detection* and *reset* strategy. In this strategy, the nodes periodically test if the system is in legal state and on detection of a fault, carry out the reset strategy. Many self-stabilizing algorithms have *local detection*, i.e., the detection by each node corresponds to evaluation of a boolean predicate only on its and its neighbors' variables. The reset procedure may be complicated depending upon the application.

Our method is an extension of the above strategy. We view the set of *global* states as cross-product of *core* states and *noncore* states. The core states satisfy the property: There exists a unique legal state for every *core* state. The *noncore* component of a global state is maintained only for performance reason. Given the *core* component, one could always recreate the *noncore* component. In our example of maintaining a spanning tree, we will use Neville's code of the tree as the *core* component and *parent* structure as the *noncore* component. Given any Neville's code, there exists a unique labeled spanning tree. Since nodes use the spanning tree using parent pointers, we maintain the *parent* variables. Now assume that our program suffers from a data fault. The data fault could be in the core component or the noncore component. The crucial property that we use is that every bit pattern in the core component results in a valid code. Therefore, in either case, we will assume that it is the noncore component that has changed. If there is an efficient method of detecting that the noncore component does not correspond to the core component, we simply reset the noncore component to the unique value corresponding to the core component. The challenge lies in efficient detection and reset of the state when information is distributed across the network.

2 Neville's third encoding

We assume that processes are numbered as $P_1,\ldots,P_n$. Each process knows its number and the number never changes. As mentioned earlier, to maintain a spanning tree (in a non-stabilizing manner), it is sufficient for each process to maintain the *parent* variable. We will assume that P_n will always stay as the root of the spanning tree with its parent pointer as 0. The program is not self-stabilizing because if the parent pointer of any process gets corrupted, the resulting structure may not be a spanning tree.

To make the program self-stabilizing, we first define the set of legal structures as all spanning trees that are rooted at P_n. For this purpose, we would be making use of an encoding for labeled trees called the *Neville's third encoding* [E.H53] (The reader can find a discussion on other labeled tree encodings and their properties in [DM01]). In this paper whenever we use the term "Neville's code", it refers to Neville's third code. Each spanning tree has a one-to-one correspondence with a Neville's code. This code is a sequence of $n-2$ numbers from $\{1\ldots n\}$. Let Neville's code of the tree be denoted by $code[i]$ for $i \in \{1..n-2\}$. For completeness sake we show how Neville's code for a labeled spanning tree is derived. Given a labeled spanning tree with n nodes, the Neville's code can be obtained by deleting $n-1$ edges in the tree as follows.

```
x[1] = least node with degree 1
for(i=1;i<n;i++){
    y[i] = neighbor of x[i]
```

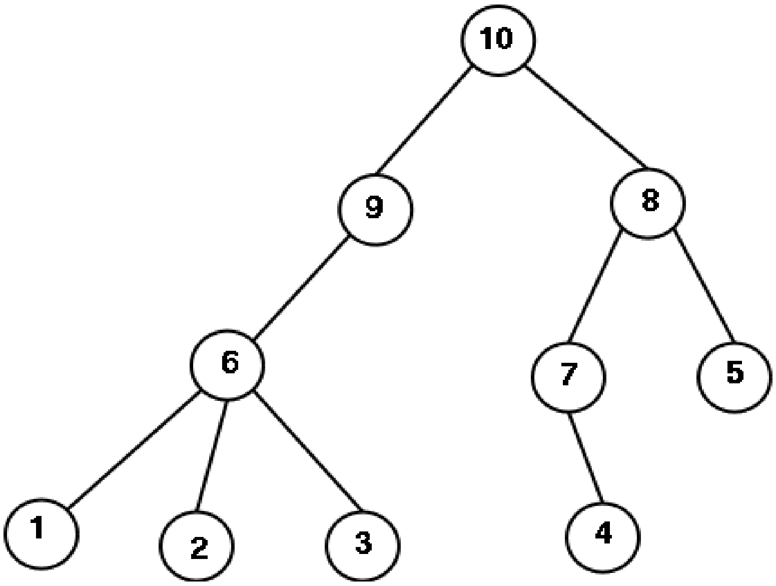

Figure 1: A labeled spanning tree with 10 nodes

```
    delete edge between x[i] and y[i]
    If degree[y[i]]==1
    then
      x[i+1] = y[i]
    else
      x[i+1] = least node with degree 1
  }
```

In the above algorithm, we start by deleting the leaf node with least label ($x[1]$). In the iteration i, on deleting $x[i]$, we record its neighbor $y[i]$ and delete the edge between $x[i]$ and $y[i]$. If $y[i]$ has degree 1, then it is deleted in the $(i+1)^{th}$ iteration otherwise the least leaf node is chosen. The sequence $\{y[i] | 1 \leq i \leq n-2\}$ is called Neville's code.

Note that even though there are $n-1$ iterations, we only consider $n-2$ entries. It can be easily verified that in the iteration $n-1$, the value of $y[i]$ is always n. For example, the Neville's code of the 10-node tree given in 1 is $(6,6,6,9,10,7,8,8)$. Given Neville's code, one can compute the labeled spanning tree easily. It is easy to verify that the degree of each node in the labeled spanning tree equals the number of times it appears in the Neville's code sequence plus one. Once degree of each node is known, the following procedure can be used. In the following procedure $n-1$ edges are constructed. Although Neville's code has only $n-2$ entries, we have used $code[n-1]$ as n because that edge is implicit in Neville's code.

```
j = least node with degree 1
for (i=1; i<n; i++) {
     parent[j] = code[i];
```

```
    degree[j]--;
    degree[code[i]]--;
    If (degree[code[i]]==1)
       j = code[i];
    else
       j = least degree node with degree 1
}
```

We require P_i to maintain $code[i]$. If efficiency were not an issue, this would be sufficient for a self-stabilizing algorithm. Periodically, all nodes would send their *code* to P_n. P_n would calculate $parent[i]$ for each node and send it back. The node would reset *parent* to the value received from P_n. If any parent pointer was corrupted, then this would result in resetting of the spanning tree to the value given by the Neville's code. If the variable $code[i]$ gets corrupted for processes, then by the property of Neville's code, we know that it still represents a legal spanning tree rooted at P_n. The parent pointers will then be reset corresponding to the new code.

This method would be wasteful when there is no error. The transmission of code to the root node and the transmission of the parent pointer to various nodes will not change anything. In a large network, it is desirable to have a local detection of error followed by recalculation of the parent pointers only when there is an error.

3 Data Structures for Maintaining Spanning Trees isomorphic with Neville's Code

For this we introduce another variable at each process f. This variable $f[j]$ gives us the iteration i in the Neville's code algorithm in which the node P_j is deleted. Since P_n is not deleted in first $n-1$ iterations, we assume that $f[n] = n$. Thus, in iteration i the edge from P_j to $code[f(j)]$ is deleted. Assuming all data structures are correct, we get that $code[i]$ corresponds to $parent[j]$. For simplicity, we extend *code* to all processes by fixing $code[n-1]$ to n and $code[n]$ to 0. Now, we have the following variables in the program: *parent*, *code*, f. Assuming all data structures are correct, it can be verified that

(R1) $\forall i : code[f[i]] = parent[i]$

(R2) $code[n-1] = n$ and $code[n] = 0$.

(R3) f is a permutation such that $f[n] = n$

However, we still do not have all the constraints to guarantee that if the constraints are met, then the structure corresponds to a valid rooted spanning tree. As a simple example consider the following example E1 for n equals 4

i	1	2	3	4
parent	2	1	4	0
code	2	1	4	0
f	1	2	3	4

Here the parent structure does not form a rooted spanning tree because parent of node 1 is 2 and parent of node 2 is 1. However, it satisfies all the constraints so far. We now introduce an additional variable z. Let $z[j]$ be the largest value of i such that $code[i] = j$. If there is no such i, then $z[j] = 0$. From the properties of Neville's code, a node is not deleted in any iteration until it becomes a leaf node. This implies,

$$f[i] \geq z[i] + 1$$

If a node j, which at the starting of the algorithm was not a leaf node, becomes a leaf node during the iteration i of the algorithm, then it is deleted in the iteration $i+1$. Thus, for $1 \leq i \leq n-1$, we have ,

$$z[i] \neq 0 \Rightarrow \{f[i] = z[i] + 1\}$$

By fixing $z[n]$ to be $n-1$, we get the following two additional constraints on data structures:

(R4)

$$\forall j : z[j] = \max\{i | code[i] = j\} \cup \{0\}$$

(R5)(i)

$$\forall i : f[i] \geq z[i] + 1$$

(R5)(ii)

$$\forall i : z[i] \neq 0 \Rightarrow \{f[i] = z[i] + 1\}$$

We now show the surprising result that if *code*, *parent*, f and z satisfy constraints (R1)-(R5), then *parent* forms a valid spanning tree isomorphic to tree corresponding to Neville's code. Later, we will show that violation of (R1) and (R5) can be detected efficiently in a distributed manner.

Just to get the feel of z in an invalid *parent* data structure, let us reconsider example E1.

i	1	2	3	4
parent	2	1	4	0
code	2	1	4	0
f	1	2	3	4
z	2	1	0	3

In this example, z satisfies (R4) but violates (R5). We now have the following result.

Theorem 1 *If code, parent, f and z satisfy constraints (R1)-(R5), then parent forms a valid spanning tree rooted at P_n.*

Proof: First we prove that the structure generated by *parent* is a forest i.e. it has no cycles. Suppose the structure generated by the procedure has a cycle. Let the nodes involved in the cycle be $\{i_1, i_2, \ldots, i_k\}$, such that $i_{j+1} = parent[i_j], \forall j \in (1, k-1)$ and $i_1 = parent[i_k]$. Now let us consider each of the constraints one by one and see what they imply.

(R1) $\Rightarrow \forall j : code[f[i_j]] = i_{j+1}$.

(R2) and (R3) $code[n] = 0$ and $f[n] = n \Rightarrow parent[n] = 0$. This means that the process P_n has no parent. $code[n-1] = n \Rightarrow$ Process P_n has atleast one child.

(R4) and (R1) $\Rightarrow \forall j : z[i_{j+1}] \geq f[i_j]$.

(R5) $\Rightarrow \forall j : f[i_{j+1}] > z[i_{j+1}]$

Therefore, $f[i_{j+1}] > f[i_j]$. So we have $f[i_1] < f[i_2] < \ldots < f[i_k]$. But by similar reasoning we would also have that $f[i_1] > f[i_k]$ as $i_1 = parent[i_k]$. So there is a contradiction. Hence there cannot be any cycles in the structure generated if the constraints (R1)-(R5) are satisfied. So the structure generated by *parent* must be a forest. Since every process except P_n has a *parent* value between 1 and n, so they cannot be isolated. P_n is also not isolated as it is always a parent of some process. So all the processes are connected to each other and hence the forest must consist of only a tree. Moreover, since the process P_n was shown to have no parent so it must be the root of the spanning tree. Hence the structure generated by *parent* is a spanning tree rooted at P_n.

■

Corollary 1 *If code, parent, f and z satisfy constraints (R1)-(R5), then the rooted spanning tree formed by parent is isomorphic to the tree generated by code.*

Proof: Let us denote the parent structure for the tree generated by the *code* by $parent'$, such that $parent'[i]$ represents the parent of node i. Let us also define a function f', such that $f'[i]$ gives us the iteration in which the node i is made the child. This corresponds to the function f for the tree given by *parent* structure. Let the set of all the nodes be N. Now we define an isomorphism function, $M : N \rightarrow N$ in the following way:

$$M(i) = \{j : f[i] = f'[j]\}$$

Suppose M is not one-one. Therefore, $\exists i_1, i_2$ such that $M(i_1) = M(i_2)$. Therefore, $f[i_1] = f[i_2]$. But this is not possible as f is a permutation. Hence M must be one-one. Since M is a mapping from N to N, therefore $|domain(M)|$ = $|range(M)|$. Moreover, since M is one-one, so M is onto as well. The way Neville's code, z and f are defined, it is clear that when $z[i] \neq 0$, $f[i] = f'[i]$ and hence for these i, $M(i) = i$. To show that tree formed by *parent* is isomorphic to the tree generated by *code*, we just need to show that if $j = parent[i]$ then $M(j) = parent'[M(i)]$. Note that all the nodes having children have $z[i] \neq 0$ and hence $\forall j : j = parent[i]$, $M(j) = j$. So now we need to show that if $j = parent[i]$, then $j = parent'[M(i)]$. By the definition of M, $f[i] = f'[M(i)]$. Since the tree generated by the *code* also follows the constraints (R1)-(R5), so $parent[i] = code[f[i]] = code[f'[M(i)]] = parent'[M[i]]$. Hence the two trees are isomorphic.

■

4 Efficient Distributed Algorithm to Check Global Constraints

Our strategy to get a self-stabilizing algorithm will be as follows. Each node i will maintain *parent*, *code*, f and z. Furthermore, these nodes will cooperate in ensuring that (R1)-(R5) hold. In that case, we are guaranteed that we have a valid rooted spanning tree. Again, it is easy to perform this check in a centralized manner. Our goal is to come up with an algorithm which requires $O(1)$ messages and $O(1)$ time per node to ensure above constraints. We deal with each constraint one at a time.

4.1 Constraint (R1) and (R2)

First consider (R1)

$$\forall i : code[f[i]] = parent[i]$$

This constraint is trivial to check locally. Each node i inquires node $j = f[i]$ for $code[j]$. If this value does not match $parent[i]$, then the constraint $(R1)$ is violated. If the constraint is not met, then it is also easy to satisfy it by setting $parent[i]$ to $code[j]$.

Now consider (R2).

$$(code[n-1] = n) \wedge (code[n] = 0)$$

This constraint is also trivial to check locally and to fix it if it is not met.

4.2 Constraint (R3)

The constraint (R3) requires that f is a permutation such that $f[n] = n$ Assuming that $f[n] = n$, this problem simply reduces to checking if f is a permutation on $1..n-1$. So for simplicity, we will simply ignore the constraint $f[n] = n$, and now our problem is: that every node maintains a number from 1..n. These numbers denote a permutation of 1..n and therefore required to be different. Therefore, the requirement is the following constraints (C1,C2):

(C1)

$$\forall i : 1 \leq f[i] \leq n$$

(C2)

$$\forall i, j : f[i] \neq f[j]$$

The fault may change any of the numbers such that the above integrity constraint is not met. The goal is to detect efficiently when this is the case.

The violation of (C1) is easy to detect. Every node i checks the value $f[i]$ periodically. If it is not between 1 and n, then a fault has occurred. At first glance it seems counter-intuitive that we can detect violation of (C2) in $O(1)$ messages. After all, every process needs to ensure that its number is not the same as any of the numbers of other processes. However, by adding auxiliary variables the above task can indeed be accomplished. We maintain $g[i]$ at each process P_i such that in a legal global state the following holds:

$$f[i] = j \equiv g[j] = i$$

Thus, g represents the inverse of the array f. Note that the inverse of a function exists iff it is 1-1 and onto. If each process P_i maintains $f[i]$ and $g[i]$, then it is sufficient for a node to check periodically the constraints (D1,D2,D3):

(D1)

$$\forall i : 1 \leq f[i] \leq n$$

(D2)

$$\forall i : 1 \leq g[i] \leq n$$

(D3)

$$g[f[i]] = i$$

It is easy to show that (C2) is implied by (D1-D3). If for some distinct i and j, $f[i]$ is equal to $f[j]$, then $g[f[i]]$ and $g[f[j]]$ are also equal. This means that $(g[f[i]] = i)$ and $(g[f[j]] = j)$ cannot be true simultaneously.

(D3) can be checked by P_i by sending a "checkA" message to $P_{f[i]}$ periodically. If a process P_j receives a "checkA" message from P_i and $g[j] \neq i$, then we know that (D3) is not met.

Note here that by introducing additional variables we have also introduced additional sources of data faults. It may happen that requirements of A (C1,C2) are met but due to fault in g, constraints D1-D3 are not met. We believe that the advantage of local detection of a fault outweighs this disadvantage.

The above scheme has an additional attractive property: If we assume that there is a single fault then it can also be automatically corrected as shown next. We first exploit the symmetry between f and g. g is inverse of f also implies that f is inverse of g. This implies that the following constraint (D4) is also met for a fault-free data structure:

(D4)

$$f[g[i]] = i$$

Now assume that a node i discovers that $g[f[i]] \neq i$. This means that either $f[i]$ or $g[f[i]]$ got corrupted. To detect which of the case has happened, it is sufficient to check whether

$$f[g[f[i]]] = f[i]$$

If the above equation does not hold, then $g[f[i]]$ is corrupted and is set back to i. If the above equation holds, but $g[f[i]] \neq i$ then $f[i]$ is corrupted and it needs to be reset. What value should $f[i]$ be set to? We need to set it to k such that $g[k]$ equals i. This correction would be done by node k because node k will find that $f[g[k]] \neq k$. Hence, by the similar reasoning as above it will deduce that $f[g[k]]$ is corrupted and will reset it to k. Thus, the program for P_i is shown in Figure 4.2. For simplicity, we let process P_i simply read and write variables of other processes. In practice, this may be translated into messages.

Note that in our scheme a permutation may undetectably change into another permutation (when there are multiple faults). However so long as f is not a valid permutation, it will be detected.

4.3 Constraint (R4)

For this constraint, we need to check that whenever $z[i]$ not equal to zero, $\forall j : code[z[j]] = j$ and $\forall k > z[j] : code[k] \neq j$.

We ensure these constraints with the following checks:

```
Pi::
 var
     f, g: array[1..n] of integer;

 Periodically do
     if (g[f[i]] ≠ i) ∧ f[g[f[i]]] ≠ f[i] then
            g[f[i]] = i;
     if (f[g[i]] ≠ i) ∧ g[f[g[i]]] ≠ g[i] then
            f[g[i]] = i;
```

Figure 2: Implementation of Permutation with local fault correction

- Every node j checks
$$code[z[j]] = j$$
- Also every node k with $code[k] = j$ checks
$$z[j] \geq k$$

4.4 Constraint (R5)

This constraint can be ensured by the following check:

- Every node j with $z[j] \neq 0$ checks
$$f[j] = z[j] + 1$$
- Every node j with $z[j] = 0$ checks
$$f[j] \geq z[j]$$

4.5 Complete Algorithm

The complete algorithm for process i is given in the figure 4.5. It is assumed that the underlying network is synchronous and there are no failures. It is also assumed that the processes start at almost the same time, so that there rounds are synchronized.

Theorem 2 *With code, parent, f, g, last with the above properties can be checked in O(1) time and O(1) messages per node.*

Proof:

■

```
Pi::
 var
      code, parent, f, g, z: integer;
      isFaulty : boolean;
           initialize isFaulty = true

 Periodically do
      if (i = n − 1) ∧ (code ≠ n) then
           isFaulty = true
      if (i = n) ∧ (code ≠ 0) then
           isFaulty = true

      get value of z.code from node z
      if z.code ≠ i then
           isFaulty = true
      send ("check z", i) to node code

      if (z ≠ 0) ∧ (f ≠ z + 1) then
           isFaulty = true
      if (f ≤ z) then
           isFaulty = true

      get f.g from node f
      get g.f from node g
      if (f.g ≠ i) ∨ (g.f ≠ i) then
           isFaulty = true
      get f.code from node f
      if (f.code ≠ parent)
           isFaulty = true
      return isFaulty

 Upon receiving ("check z", j)
      if z < j then
      isFaulty = true
```

Figure 3: Algorithm for checking the constraints (R1)-(R5)

5 Additional Data Structure for Maintaining the exact Neville's Code Tree

In the section 3, we presented data structures for maintaining a tree isomorphic to the Neville's code tree. Some applications may have more stringent requirements and need to maintain the exact tree corresponding to the Neville's code. This can also be easily accomplished by adding another constraint.

$$(R6) \qquad \forall i, j : (z[i] = 0) \wedge (z[j] = 0) \wedge (i < j) \Rightarrow (f[i] < f[j])$$

This constraint ensures that the nodes which have zero z value, are considered in order to become leaf nodes.

Theorem 3 *If code, parent, f and z satisfy constraints (R1)-(R6), then the rooted spanning tree formed by parent is same as the tree generated by code.*

Proof: From corollary 1, we know that the spanning tree formed by *parent* is isomorphic to the tree generated by *code*. So for proving the theorem, we need to prove that the isomorphism function in this case is the identity function i.e. $\forall i : M(i) = i$.

The positions in the f sequences for i where $z[i] \neq 0$ is the same in both the trees. So the nodes with zero z can permute themselves in the remaining positions. The constraint (R6) defines a total order for these nodes based on the process number which is the same order as used by the algorithm for constructing the Neville's code tree. So $\forall i : z[i] = 0 \Rightarrow f[i] = f'[i]$. This means that $\forall i : z[i] = 0 \Rightarrow M(i) = i$. This combined with the fact $\forall i : z[i] \neq 0 \Rightarrow M(i) = i$, implies that $\forall i : M(i) = i$. Hence proved.

■

5.1 A distributed algorithm for checking constraint (R6)

Constraint (R6) can be checked by maintaining a circular doubly linked list with the nodes having $z[i] = 0$ as its elements. Let the set of all such elements ordered according to the process numbers be denoted by L. Each element of the link list would have a *next* and *prev* pointer pointing to the next and the previous element, respectively, in the list. The next element in the list would be the node which occurs next to the node i in L. Similarly, the previous node would be the one to occur before node i in L. The only exceptions to the above rules are the *prev* pointer of first node in the list and the *next* pointer of the last node. To prevent a fragmented list to go undetected, we use the process P_1 as the sentinel node. So the *prev* pointer of the first node in the list and the *next* pointer of last node points to P_1. Similarly, P_1 also maintains its *prev* and *next* pointers. Using this link list (R6) can be checked in the following way:

- Every node j checks

$$(z[next[j]] = 0) \wedge ((f[next[j]] > f[j]) \vee (next[j] = 1))$$

$$(z[prev[j]] = 0) \wedge ((f[prev[j]] < f[j]) \vee (prev[j] = 1))$$

- Every node j also checks

$$prev[next[j]] = j$$

$$next[prev[j]] = j$$

By the first check, we ensure that the condition (R6) is checked and also accommodate the presence of the sentinel. The second check along with the check for presence of the sentinel ensures that the linked list is consistent. If there are any breaks in the linked list, they would be detected by the second condition. Due to the presence of the sentinel node, we would also be able to detect the cases when a single linked list might be split into two linked lists due to data faults. Also note that since (R6) would be checked last among all other constraints, so every node which has actually got a zero z value will have it set correctly. This would ensure that no error goes undetected. As all the checks are local checks, so check (R6) requires $O(1)$ time and $O(1)$ messages per node. Hence, by just checking for (R6), we can also maintain exactly same tree as Neville's code tree. Note that the method used in this case is a general method for maintaining a linked list in an efficient way and can be used for other algorithms as well.

6 Distributed Correction of Data Structure

So far we have discussed, how node can detect if they do not form a spanning tree isomorphic to one given by the Neville's code. On such a detection, one could start a correction algorithm. The centralized strategy would be trivial where each node would send their code to the root which can generate the data structures $parent$, f,g and z again. We can first try correcting the errors by a simple algorithm given in the figure 6. It takes $O(T)$ time where T is the time taken for a synchronization round to complete in the algorithm. This algorithm can correct most of the errors but some of the errors need a more comprehensive algorithm for correction.More specifically, this algorithm can correct the following type of errors:

- $code[n-1] \neq n$ and $code[n] \neq 0$
- All errors in z
- Errors in f for the nodes where $z \neq 0$
- Errors in g for the nodes where $g.z \neq 0$
- $parent$ for the nodes where $z \neq 0$

We now propose an algorithm for correction which will correct all the data faults provided no faults occur during the execution of the algorithm. This algorithm takes $O(\log n)$ time and $O(\log n)$ messages per node. The algorithm for process P_i is given in the figure 6

In the above algorithm, the step involving the calculation of the prefix sum takes $O(\log n)$ time and $O(\log n)$ messages per node. The rest of the steps are $O(1)$ time and $O(1)$ messages per node. The algorithm for calculating the prefix sum is an adaptation of the parallel prefix algorithm [Jaj92] and is given in the appendix 8 for the sake of completion. All other steps take $O(1)$ time and messages per node. So the algorithm takes $O(\log n)$ time and $O(\log n)$ messages per node. Here we are assuming one synchronous round to complete in $O(1)$ time.

```
Pi::
 var
     code, parent, f, g, z: integer;
 if i = n − 1 then
     code = n
 if i = n then
     code = 0
 Start round 1
 get value of z.code from node z
 if z.code ≠ i then
     z = 0
 send ("check z", i) to node code
 while (round 1 is not over)
     upon receiving ("check z", j)
         if z < j then
             z = j
 End round 1
 if z ≠ 0 then
     f = z + 1
     send ("update g", i) to node f
 upon receiving ("update g", j)
     g = j
 Start round 2
     get f.g, (f.g).f, g.f and (g.f).g from respective nodes
     if (f.g ≠ i) ∧ ((f.g).f ≠ f then
         send ("update g", i) to node f.g
     else if (g.f ≠ i) ∧ ((g.f).g ≠ g) then
         send ("update f", i) to node g.f
     upon receiving ("update g", j)
         g = j
     upon receiving ("update f", j)
         f = j
 End round 2
 get f.g and g.f from respective nodes
 if (f ≤ z) ∨ (f.g ≠ i) ∨ (g.f ≠ i) then
     start global correction algorithm
 get f.code from node f
 if f.code ≠ parent
     parent = f.code
```

Figure 4: Algorithm for correcting some errors

P_i::

var

$code, parent, f, g, z, next, prev$: integer;

$zeroz, zerof, pre_z, pre_f$: integer;

Initialize:

$parent = f = g = z = next = prev = 0$

$zeroz = zerof = 1$

Start round 1

Send ("set z",i) to node $code$

Keep waiting for messages till round is over

Upon receiving("set z",j) :

if $z < j$ then

$z = j \wedge zeroz = 0$

Go back to waiting

End round 1

If $z \neq 0$ then

$f = z + 1$

send ("set g",i) message to node f

Calculate prefix sum pre_z of sequence $zeroz$ for the nodes

Calculate prefix sum pre_f of sequence $zerof$ for the nodes

If $z = 0$ then

Send ("zero z",i) to pre_z

If $f = 0$ then

Send ("zero f",i) to pre_f

Upon receiving ("zero z", j) and ("zero f", k)

Send ("set f", j)" message to k

Send ("set g", k) message to j

Upon receiving a ("set f",j) message

set $f = j$

Upon receiving a ("set g",j) message

set $g = j$

Set $prev$ and $next$ as well if (R6) is obeyed

Figure 5: Correction algorithm for data structures

7 Discussion

7.1 Maintaining particular types of spanning trees

Uptil now we have discussed the case when all the spanning trees form a legal structure.In some cases we might be interested in maintaining spanning trees with some special properties e.g. spanning trees with no nodes having degree greater than k. Some of these properties could be easily checked by using additional constraints. For example, to check that whether a spanning tree has any nodes with degree greater than k, the algorithm in figure 7.1 can be used.

P_i::
var
    degree: integer;

Send “increment degree” to node *code*
Upon receiving “increment degree”:
    degree++

Figure 6: Algorithm for calculating the degree of the nodes

The above algorithm takes an average of $O(1)$ time per node and $O(1)$ messages per node. After calculating the degree, each node can check whether its degree is greater than k or not. So we have a very efficient method for checking the validity of the constraint. In this way the algorithm described in the previous sections could be easily augmented to satisfy certain properties.

7.2 Additional fault correction algorithms

The fault correction algorithms suggested in the previous section could be used in conjunction with other techniques for fault correction. A very simple example would be a fault correction strategy in which we maintain a duplicate copy of all the data structures. Each set of the data structures could be checked for consistency within itself using the fault detection algorithm given in section 4. So if there is only one fault within the system at any time, then we can correct that by just adopting the fault free copy of the data structures. This idea can be adopted even if the number of faults within the system is not 1 but is bounded by a certain small number. Such an algorithm may be useful in certain applications and our correction algorithm in no way precludes use of such other methods.

8 Conclusion

In this paper we presented a new technique using *core* and *noncore* states to design self stabilizing algorithms. The technique was demonstrated in maintaining spanning trees in a distributed system. As part of the algorithm, we showed ways of maintaining some common structures like permutations and link lists which can be used in other algorithms as well.

References

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[DM01] N. Deo and P. Micikevicius. Prufer-like codes for labeled trees. *Congressus Numerantium*, 151:65–73, 2001.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000. 6 x 938 illus., cloth.

[E.H53] E.H.Neville. The codifying of tree-structure. *Proceedings of Cambridge Philosophical Society*, 49:381–385, 1953.

[Jaj92] Joseph Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

A Distributed Algorithm for Prefix Sum

```
Pi::
 var
     A, k:integer;
     B, C: array [1..n] of integers
 Initialize:
     B[0] = A

 for round = 1 to log n
     k = n/2^round
     if (1 ≤ i ≤ k)
         get P_{2i-1}.B[round - 1] from process P_{2i-1}
         get P_{2i}.B[round - 1] from process P_{2i}
         set B[round] = P_{2i-1}.B[round - 1] + P_{2i}.B[round]

 for round = log n to 0
     k = n/2^round
     if (1 ≤ i ≤ k)
         if (i%2 = 0)
             get P_{i/2}.C[round + 1] from P_{i/2}
             set C[round] = P_{i/2}.C[round + 1]
         if (i = 1)
             set C[round]=B[round]
         if ((i > 1) ∧ (i%2 = 1))
             get P_{(i-1)/2}.C[round + 1] from P_{(i-1)/2}
             set C[round] = P_{(i-1)/2}.C[round + 1] + B[round]

 return C[0]
```

Figure 7: Distributed algorithm for calculating prefix sum of variable A