A Fault-Tolerant Distributed Sorting Algorithm in Tree Networks

Gianluigi Alari, Joffroy Beauquier, Joseph Chacko, Ajoy K. Datta, Sébastien Tixeuil

Abstract— **This paper presents a distributed sorting algorithm for a network with the tree topology. The distributed sorting problem can be informally described as follows: nodes cooperate to reach a global configuration where every node, depending on its identifier, is assigned a specific final value taken from a set of input values distributed across all nodes; the input values may change in time. In our solution, the system reaches its final configuration in a finite time after the input values are stable and the faults cease. The fault-tolerance and adaptive to changing input are achieved using Dijkstra's paradigm of self-stabilization. Our solution is based on continuous broadcast with acknowledgment along the tree network to achieve synchronization among processes in the system; it has $O(nh)$ time complexity and only $O(log(n)deg)$ memory requirement where deg is the degree of the tree and h is the height of the tree.**

Keywords— **Distributed algorithms, distributed sorting, self-stabilization, software fault-tolerance.**

I. Introduction

An important niche in the software fault tolerance area is occupied by self-stabilizing algorithms. The concept of self-stabilization was introduced by Dijkstra [3]. A self-stabilizing system, regardless of the initial values of the states of the nodes and initial messages in the links, is guaranteed to converge to the intended behavior within finite time. For a good survey on the self-stabilizing systems, please refer to [7].

An instance of the distributed sorting problem has a set of n connected nodes and each node i has a hard-coded identifier Id and an input value Val. The goal is to sort the input values of the system to correspond with Ids such that the pairing relationship works as follows: the node with the lowest Id in the system takes the lowest Val in the system, the second lowest Id in the system takes the second lowest Val in the system, etc. This pattern continues until the highest Id in the system takes the highest Val. A variable $Final\_Val$ is used to store the sorted value at any particular node. We call this sorted value the *final value* for a particular node. Distributed sorting has been previously solved [5], [6], [9] but without addressing the fault issues. It is trivial to verify that these algorithms are not self-stabilizing since they are correct only if the state variables of each node in the system are properly initialized. Furthermore, the fault-tolerant extensions to these algorithms are complicated by their inability to perform local checking.

While sorting is a fundamental problem in computer science both in sequential and parallel environments, several applications for distributed sorting can be emphasized. Let us consider a distributed system where the processes with unique identifiers are waiting for their corresponding computers (the highest identifier waiting for the fastest computer, and so on). We can label processes or threads to be executed in this system with a value reflecting the amount of CPU time needed to complete their jobs. With a distributed algorithm solving the sorting problem, one can easily transfer processes and threads among the computers in the network by maintaining the values associated with them in sorted manner. As the algorithm completes, the processes with the highest value (i.e., the longest time to complete) would be run in the fastest computer, and so on. If the algorithm is *input adaptive*, then the change of a process value on a single node would induce new processes and/or threads make moves for providing the optimal response. If the algorithm is, in addition, *self-stabilizing*, then it can deal with dynamic networks, or with networks where nodes/links may fail, stop, and be repaired later.

In this paper, we present an input adaptive self-stabilizing solution to the distributed sorting problem for tree-based networks[1]. It relies on continuous broadcast with acknowledgment over the tree network and uses the counter flushing paradigm of [8] to reset and synchronize the network nodes.

The algorithm presented here is highly robust since it adapts to the input value changes and it is able to mask the uncommon transient errors that

G. Alari is with Unité d'Informatique, Université catholique de Louvain, Belgium. J. Beauquier and S. Tixeuil are with L.R.I./C.N.R.S., Université de Paris-Sud, France. J. Chacko and A. Datta are with Department of Computer Science, University of Nevada Las Vegas.

[1]It is easy to transform the given algorithm to work in general networks. Self-stabilization can be achieved by composing the sorting algorithm with one of the well-known self-stabilizing spanning tree algorithms given in [1], [2], [4].

0-7803-4468-5/98 $10.00 © 1998 IEEE

lead to corruption of state variables, temporary link failures, duplication or loss of messages, permanent link failures, and node crashes that leave the system connected. It can also transparently deal with new or repaired nodes and links joining the system. Furthermore, all these exceptional events are handled automatically without any initialization or intervention due to the self-stabilizing characteristic of our solution.

The remainder of the paper is organized as follows. In the next section, we introduce the system model. The sorting algorithm is presented in Section III. The proofs of correctness are given in Section IV. Finally, we make our concluding remarks in Section V.

II. Model and Notation

A. System Model

We model a distributed system by an undirected, connected graph $G(V, E)$ where V is the set of nodes and E is the set of links or edges, $|V| = n$, and $|E| = l$. A link connecting node i to node j is uniquely identified by the two-tuple (i, j), and for every $(i, j) \in E$, nodes i and j are called *neighbors*. Neighbors are able to communicate by exchanging messages. Messages pending in edge (i, j) will be denoted as $M_{(i,j)}$.

The system is asynchronous, meaning that we cannot make any assumptions about the relative speeds of the nodes, and communication delays are finite but unbounded. The *link capacity* is the maximum number of pending messages for destination j over the link (i, j); it is finite and will be denoted by $Cap_{i,j}$. For the sake of simplicity, we assume that $Cap_{i,j} = Cap_{j,i}$ and we define $Cap_{Max} = maximum\ (Cap_{i,j} : (i, j) \in E)$, the maximum link capacity of the system.

A node communicates with its neighbors through an unreliable communication layer which provides a minimal service consisting of at least two basic message passing primitives $send(d, variablelist)$ and $receive(s, variablelist)$. The primary reasons why the communication might be unreliable are the failures and faults of the network components. But, after the faults cease to occur, communication between system nodes is reliable. Both of these basic communication operations are nonblocking, i.e., the issuing node does not wait for the information to be relayed to the destination, or to be received if no information has been sent to it.

When a node i executes $send(d, variablelist)$, the values of the variables of its internal state specified in $variablelist$, along with the intended destination neighbor d, are passed to the communication layer that will take care of relaying the information to d. Provided that the link capacity is not already reached, the implementation of the *send* and *receive* operations preserves the FIFO ordering. If the number of pending messages for destination d on the link (i, d) is already $Cap_{i,d}$, then the message is simply lost or discarded by the communication layer, and no exceptional or further action is taken.

Execution of $receive(s, variablelist)$ returns a boolean value: *true* indicates that some information is waiting to be relayed, *false* otherwise. When a *true* value is returned, the waiting information is assigned to the variables specified in $variablelist$ and the identifier of the sender is returned in the variable s.

All nodes cooperate to maintain a distributed spanning tree $T = (V, E')$ of G with $E' \subseteq E$ through a self-stabilizing spanning tree algorithm [1], [2], [4] run by a *tree layer*. This algorithm provides every node with a set $Children$ consisting of the Ids of its children in T and of a $Parent$ identifier which is set to *null* for the root node r of T. T_i will denote the subtree rooted at node i while $Children_i$ and $Parent_i$ represent the set $Children$ and the identifier $Parent$ of node i, respectively.

B. Distributed Sorting

Each node maintains a three-tuple $\langle Id, Val, Final\_Val \rangle$ consisting of a non-corruptible node identifier Id, a variable Val containing a specific value, and a variable $Final\_Val$ ranging over the Val domain. All Ids are distinct, but there can exist equal Vals. In the rest of this paper, we will denote the components of the above three-tuple at node i as Id_i, Val_i, and $Final\_Val_i$, respectively.

We define L to be the set of legitimate states, with respect to the distributed sorting problem. For any instance $S \in L$, there exists a function f:

$$\begin{aligned} f \ &: \ V \to V \\ &\quad i \mapsto f(i) \end{aligned}$$

such that f is *bijective* and satisfies the two following conditions:

a. Sort Condition

$$\forall i, j \in V, i \neq j, (Id_{f(i)} < Id_{f(j)}) \wedge (Final\_Val_{f(i)} \leq Final\_Val_{f(j)})$$

b. Value Condition

$$\forall i \in V, Final\_Val_{f(i)} = Val_i$$

The sort condition states the final legitimate relation between the system identifiers and final values (the node having the lowest Id will get the lowest $Final\_Val$, and so on) while the value condition forces the values of the $Final_Val$s to be a permutation of the system Vals.

In our model, the input values, Vals, may change in time. So, we require that a self-stabilizing solution to the distributed sorting problem is such that a system will reach and maintain itself in L in a finite time after the input transitions and faults cease.

The self-stabilization property with respect to the sorting problem can be stated formally using the set $\mathcal{L}$ of legitimate configurations as follows:

Definition 1: A sorting algorithm $\mathcal{A}$ is *self-stabilizing* for the specification of the sorting problem S if (i) $\mathcal{L}$ is closed in S, i.e., any computation of $\mathcal{A}$ starting from a configuration in $\mathcal{L}$ preserves S (**closure**) and (ii) starting from any initial configuration, any computation of $\mathcal{A}$ reaches a configuration in $\mathcal{L}$ (**convergence**).

III. Ordering by Broadcast with Acknowledgment

A. Dynamics of the System

The basic idea is that the root r sends a sequence of data to all its children. Then these children forward the data to their children, and so on, until the data reaches the leaves of the tree. We will call this a *broadcast phase.* On receiving data from the parents, the leaves acknowledge to their parents. The parents receiving the acknowledgments from all their children acknowledge to their parents, and so on, until the acknowledgment messages reach the root r. We will call this an *acknowledgment phase.* A broadcast phase and the corresponding acknowledgment phase form one *cycle.* The cycle of broadcast and acknowledgment phase is used to solve the sorting problem (defined in Section II-B).

The k^{th} acknowledgment phase after a reset is used to compute the k^{th} lowest pair $\langle Id, Val \rangle$ in the system while the k^{th} broadcast phase after a network reset is used to diffuse this two-tuple. The alternating broadcast and acknowledgment phases continue until all Id's and Val's are chosen and removed from the tree, and at that time, the sorting process completes.

The root initiates the reset process when it finds that all Id's or Val's of the system are chosen and removed. But, due to the transient faults, r may think that all Id's or Val's are chosen and removed when actually they are not. This anomalous state may be reached when some nodes in the tree have some wrong information in some variables and send this information up to the root in the acknowledgment phase. The effect of this incorrect information at r is that it initiates the reset process earlier since there is no point in continuing with the sorting process when some information is corrupted.

So, the reset process is initiated by the root in two situations: (i) the sorting process completes, all Id's and Val's are chosen and removed; and (ii) due to some transient errors, the root finds/thinks all Id's or Val's are chosen and removed when actually they are not. The system goes through the sorting and reset process alternately to handle the transient faults.

B. The Algorithm

After the informal description in the previous section, we are now ready to give the details of the data structures and program at each node. The state of each node i consists of some primary and auxiliary variables. The auxiliary variables are used to temporarily store the sent/received messages. The primary (respectively, auxiliary) variables of node i are defined in Figure III.1 (respectively, Figure III.2). The program executed at node i is given in Figure III.3, while the functions used in the algorithm are presented in Figure V.1.

Figure III.1 Primary Variables for the Algorithm.

Parent	Parent of i in T which is set to *null* for the root node r of T.
Children	Set of children of i in T.
Ack_Expected[j]	A boolean flag for each child j of i; *true* indicates i is expecting an acknowledgment from j.
Counter	A counter used to achieve the synchronization among the nodes.
Id	Hard-coded identifier Id of node i.
Val	Input value Val at node i.
Min_Id	Minimum Id in the tree rooted at i.
Min_Val	Minimum Val in the tree rooted at i.
Final_Val	The final Val at node i after sorting.
R_Id_Ch[j]	Last received Id from the child j of i.
R_Val_Ch[j]	Last received Val from the child j of i.
Id_C	Boolean flag; *true* indicates Id is chosen and removed.
Id_C_T	Boolean flag; *true* indicates Id is chosen at all nodes in T_i.
Val_C	Boolean flag; *true* indicates Val is chosen and removed.
Val_C_T	Boolean flag; *true* indicates Val is chosen at all nodes in T_i.
R_Id_C_Ch[j]	Boolean flag for each child j of i; *true* indicates the child j of i has informed i that Id at each node in T_j is chosen.
R_Val_C_Ch[j]	Boolean flag for each child j of i; *true* indicates the child j of i has informed i that Val at each node in T_j is chosen.

Figure III.2 Auxiliary Variables for the Algorithm.

s	Identifier of the sender (only used for received messages).
$counter$	Value of $Counter$ of the sender (only used for received messages).
r_id, b_id	Id received, Id to be broadcast, respectively.
r_val, b_val	Val received, Val to be broadcast, respectively.
r_id_c, b_id_c,	Id_C received, Id_C to be broadcast, respectively.
r_val_c, b_val_c	Val_C received, Val_C to be broadcast, respectively.

IV. Correctness Reasoning

We assume that the computation cost to be negligible with respect to communication cost, and we assign a unit of time to send a message from node i to a neighbor j.

Definition 2: A distributed system $G = (V, E)$ is *synchronized* if and only if $\forall (i,j) \in E$, $\forall m \in M_{(i,j)}$, $Counter_i = Counter_j = Counter_m$.

Definition 3: A distributed system $G = (V, E)$ is *acknowledged* if and only if G is synchronized and $\forall i \in V$, $DONE_i$ holds.

Informally, when the system is acknowledged, all nodes have received the acknowledgment of the messages they sent in the previous cycle, meaning that the previous cycle completed.

The following claim shows the bound of the time complexity to reach a synchronized state, and we refer the interested reader to [8] for its proof:

Claim 1: A distributed system G will reach a synchronized state in $O(h)$ time starting from any arbitrary state, where h is the height of the underlying spanning tree T.

Let us now bound the time to reach an acknowledged state from a synchronized state.

Lemma 1: Starting from a synchronized state, the system reaches an acknowledged state in $O(h)$ time.

Proof: By the definition of $DONE$ function, all leaf nodes are always in $DONE$ state. We will consider two cases depending on whether the $DONE$ function holds or not for the other nodes:

Case 1: $\forall i \in V, |Children_i| \neq 0 \Rightarrow DONE_i = false$.

From such a state, only leaf nodes send acknowledgment messages to their parents (I3); the parent nodes receive these acknowledgment messages (I1), go to $DONE$ state, and send acknowledgment messages to their parents (I3). Using induction on the height of the tree, it is easy to show that in time h, the system will reach the acknowledged state.

Figure III.3 Sorting Algorithm.

```
/* For root */
(R1) receive(s, r_id, r_val, r_id_c, r_val_c, counter) →
/* Values from a child s. */
     if (counter = Counter) then R_MIN_CHILD;
/* Record values. */
(R2) true → BCHILDREN; /* Broadcast to children. */
(R3) DONE →
     COMPUTE_MIN;
     /* Update Min_Id, Id_C_T, Min_Val, Val_C_T */
     PROCESS; /* Process new values */
     Counter := (Counter + 1) mod (Cap_Max * |E'| + |V|);
     /* New broadcast phase. */
     /* Update broadcast variables */
     b_id := Min_Id; b_val := Min_Val;
     b_id_c := Id_C_T; b_val_c := Val_C_T;

/* For other nodes */
(I1) receive(s, r_id, r_val, r_id_c, r_val_c, counter) →
     if ((s ∈ Children) ∧ (counter = Counter))
     then R_MIN_CHILD; /* Record values. */
     if ((s = Parent) ∧ (counter ≠ Counter)) then
        /* Process values, synchronize with Parent and
        update broadcast variables. */
        PROCESS; Counter := counter;
        b_id := r_id; b_val := r_val; b_id_c := r_id_c;
        b_val_c := r_val_c;
     fi;
(I2) true → BCHILDREN; /* Broadcast to children. */
(I3) DONE →
     COMPUTE_MIN;
     /* Update Min_Id, Id_C_T, Min_Val, Val_C_T */
     send(Parent, Min_Id, Min_Val, Id_C_T, Val_C_T,
          Counter); /* Send values to parent. */
```

Case 2: $\exists i \in V$ *s.t.* $|Children_i| \neq 0 \wedge DONE_i = true$.

It is obvious from the result of *Case 1*, all nodes which satisfy $DONE = false$, will receive the acknowledgment messages from all their children in $O(h)$ time. Two situations may occur depending on the value of $DONE_r$:

Case 2.a: $DONE_r = true$. r starts a new counter (R3); r sets (in the procedure $PROCESS$) $Ack\_Expected_r[] = true$. So, in the current state, $DONE_r = false$ while $\forall i \in V, i \neq r$, $Counter_r \neq Counter_i$. Again starting from this state, the system will reach a synchronized state in $O(h)$ time (Claim 1) and an acknowledged state in another $O(h)$ time (from the result of *Case 1*).

Case 2.b: $DONE_r = false$. This is the same as *Case 1*, except if at least one of the nodes where $DONE$ doesn't hold, receives its last acknowledgment message after r receives the last acknowledgment message. In this case, r executes rule (R1) and the same reasoning as in *Case 2.b* can be applied.

Summing up, it takes $O(h)$ time to reach an acknowledged state from a synchronized state.

□

The next three lemmas show that in $O(nh)$ time, starting from an acknowledged state, all Id's and Val's will be chosen and removed from further consideration. The first lemma shows the correctness of the broadcast implementation.

Lemma 2: Starting from an acknowledged state, the messages broadcast by the root are delivered to every node in the system in $O(h)$ time.

Proof: $DONE$ holds at the root since the root is in an acknowledged state (Definition 3). Using (R2) and (R3), r starts a new counter by incrementing $Counter_r$ and broadcasts the new values to all children by calling $BCHILDREN$. Since the starting state was a synchronized state, the following condition is true after the root executes the action of rules (R2) and (R3):

$\forall i, j \in V \setminus \{r\}, Counter_i = Counter_j \wedge Counter_r \neq Counter_i$

Then, all other nodes receive and relay the new broadcast values (originated by r) by executing rules (I1) and (I2). By Claim 1, the new values will reach all nodes of the system in $O(h)$ time since h is the maximum distance between r and any node in the underlying tree. When all nodes receive the new counter value, the broadcast is complete. □

Note 1: Due to the change of input values and transient faults, in an acknowledged state, the root r may broadcast a pair $\langle Id, Val \rangle$ of values (Lemma 2) where this particular Id or Val may not exist in the system at all.

Definition 4: The *current minimum Id* is $\min_{i \in V} Id_i$ *s.t.* $Id\_C_i = false$.

Definition 5: The *current minimum Val* is $\min_{i \in V} Val_i$ *s.t.* $Val\_C_i = false$.

Lemma 3: Starting from an acknowledged state where $\exists i, j \in V$ *s.t.* $Id\_C_i = false \wedge Val\_C_i = false$, the current minimum Id and current minimum Val of the system are computed by the root in $O(h)$ time.

Proof: Every leaf node k sends Id_k or a chosen indicator to its parent $p = Parent_k$ (I3) depending on the value of Id_C_k. Every node p which is a parent of some leaf nodes of T receives the minimum Id of every subtree T_k from the corresponding child k of p (I1), or the indicators meaning that all Ids have been chosen in all subtrees. p (locally) computes the minimum Id of the subtree T_p, and sets $Id\_C\_T_p$ accordingly. p then sends this minimum Id (or an indicator meaning that all Ids have been chosen in the subtree rooted at p) to its parent $Parent_p$ (I3). So, in one time unit from the synchronized and acknowledged state, the nodes at distance one from the leaf nodes of T compute the minimum Id of the trees rooted at them, while checking for already chosen values. By induction on h it can be shown that, in time $O(h)$, the root computes the minimum Id of the tree T.

The proof is the same for computing the minimum Val.

□

Lemma 4: From a configuration where the root computed the current minimum Id and minimum Val of the system, in $O(h)$ time, (i) the minimum Val is assigned as a final value to the appropriate node, and (ii) the minimum Id and minimum Val are chosen and removed.

Proof: Once the root r computes the current minimum Id and minimum Val, r starts a new counter by broadcasting this pair using rules (R2) and (R3). Other nodes receive and relay this pair using rules (I1) and (I2). By Lemma 2, all nodes will receive this minimum pair in $O(h)$ time. Assume Id_i = minimum Id and Val_j = minimum Val, where $i, j \in V$. Node i on receiving the minimum pair (rule (I1)), calls $PROCESS$ which in turn calls $CHOSEN$. In the $CHOSEN$ procedure, the minimum Val is assigned as the final value of i and the identifier Id_i (= minimum Id) is removed by i. Similarly, node j removes Val_j (= minimum Val). □

Lemma 5: Starting from an acknowledged state, all Id's and Val's are chosen and removed in $O(nh)$ time.

Proof: By Lemmas 3 and 4, in $O(h)$ time, the current lowest Id and lowest Val of the system will be removed. The leaf nodes then initiate the process of computing the second lowest Id and second lowest Val; in another $O(h)$ time, this pair of second lowest is computed and removed. It is obvious that the leaf nodes cannot initiate the computation of the second lowest pair before the first lowest pair is computed and removed from the system. So, if the leaf nodes initiate this process n times for n nodes in the system, in $O(nh)$ time, nth lowest pair will be computed, and also, all Id's and Val's will be chosen and removed. □

By Note 1, the wrong Id's and Val's may be chosen and thus the system needs to be periodically reset to insure the self-stabilization.

Definition 6: A node i is *reset* if and only if $Id\_C_i = false \wedge Val\_C_i = false$. A system $G = (V, E)$ is *reset* if and only if $\forall i \in V$, i is reset.

Lemma 6: Starting from a configuration where all Id's and Val's are chosen and removed from the

system, the root initiates the reset process in $O(h)$ time.

Proof: The leaf nodes keep sending the Id and Val (I3) to continuously initiate the process of computing the next pair of minimums. Other nodes also continue to receive these pairs (I1), compute the pair of minimums, and send the pair to their parents (I3). Once all Id's and Val's are removed (Lemma 5), the leaf nodes send the chosen indicators for Id and Val as the minimum pair (I3); all other nodes receive the chosen indicators as the minimum pair, and also compute the same in the procedure $COMPUTE\_MIN$ (I3) ((R3) for the root). So, by Lemma 3, in $O(h)$ time, the root computes the chosen indicators as the minimum Id and minimum Val, executes $PROCESS$ (R3), resets itself, and initiates the reset process by broadcasting (calling $BCHILDREN$ in (R2)) the chosen indicators as the new values. □

Theorem 1: Starting from any arbitrary state, the system is reset in $O(nh)$ time.

Proof: By Claim 1, Lemmas 1, 5, and 6, starting from any arbitrary state, in $O(nh)$ time, the root resets itself and initiates the reset process. The children of the root receive the reset signal (I1), call $PROCESS$, reset themselves, and relay the reset signal down the tree. By Lemma 2, the reset signal initiated by the root will reach all nodes in $O(h)$ time. So, once the root resets itself (which takes $O(nh)$ time from any arbitrary state), all nodes will receive the reset signal and reset themselves in another $O(h)$ time. □

Finally, we show that once the system is reset, in a stable configuration, i.e., when input transitions, faults and exceptional events cease, the Val's are sorted correctly.

Lemma 7: Starting from a configuration where the system is reset, the set of local identifiers and the final values satisfy both the sort and value conditions in $O(nh)$ time.

Proof: We will first prove the sort condition. Once the system is reset, using the technique used in the proof of Lemmas 3 and 4, we can show that the lowest Id and lowest Val of the system will reach the root r in $O(h)$ time; in another $O(h)$ time, the lowest Val will be assigned as the final value of the node with the lowest Id, and the lowest Id and Val are removed. If this process is repeated, then in another $O(h)$ time, the second lowest Val will be assigned as the final value of the node with the second lowest Id, and so on. So, after n cycles, which takes $O(nh)$ time after the system reset, the sort condition is satisfied.

To prove the value condition, we will first show that every node computes a valid Min_Id and a valid Min_Val whenever the node calls the procedure $COMPUTE\_MIN$ after the system is reset; a Min_Id (Min_Val) is valid if it is one of the system Id's (Val's). Consider the leaf nodes first. After the reset process, the leaf nodes validate Min_Id, Id_C_T, Min_Val, and Val_C_T in $COMPUTE\_MIN$ in (I3); they send their own hard-coded Id, their input Val as the Min_Id, Min_Val, respectively. So, the leaf nodes send valid Min_Id and Min_Val to their parents. Next consider parents of the leaf nodes. It is clear from rules (I1) and (R1), and the procedure $PROCESS$ that, during the reset process, the non-leaf nodes validate Id_C and Val_C (set them to *false*). After the system is reset, upon receiving the Min_Id and Min_Val from the leaf nodes (in rule (I1)), their parents call $COMPUTE\_MIN$ in (I3). Since all inputs to $COMPUTE\_MIN$ are valid, the returned Min_Id and Min_Val from this procedure must be valid. Following the similar reasoning, we can prove that all non-leaf nodes including the root compute valid Min_Id and Min_Val. □

Theorem 2: The algorithm in Section III-B is self-stabilizing and solves the distributed sorting problem with time complexity $O(nh)$ and memory requirement of $O(log(n)deg)$ where deg is the degree of the tree.

Proof: Assume that the input transitions, faults, and exceptional events cease, and that $Final\_Val$'s and Id's do not satisfy the sort and value conditions. By Theorem 1, the system will be reset in $O(nh)$ starting from this current (illegal) state, and by Lemma 7, in another $O(nh)$ time after the reset, $Final\_Val$'s and Id's will satisfy the sort and value conditions, and maintain the system in a legitimate state.

The information stored at each node has fields which may handle at most a node identifier. Since all identifiers are different, these fields are of size $O(\log(n))$. Since the nodes collect information only from their parent and children, it is obvious that the memory requirement is $O(\log(n)deg)$. □

V. Conclusions

This paper has provided a simple input-adaptive, global, non-masking, fault-tolerant algorithm for the distributed sorting problem. Its significance lies in the fact that no initialization is needed, a number of transient and/or permanent faults are handled in software without need for human intervention, and it accommodates input transitions in a transparent way.

Distributed sorting algorithms can be used as a basis for other higher level algorithms, such as tasks and process migration over networks. They can also be used to extend some fundamental distributed algorithms. For example, we may consider the k-leader election, where the processors with the first k maximum identifiers need to be elected. One solution is to input k zero values at several nodes, and random non-zero values at all other nodes, so that after the sorting process, the k maximum-id processors will get the zero values (thus will be elected) while the others will not get the zero value (thus will not be elected). Another approach is to design a ranking algorithm. This problem can be trivially solved using n different and consecutive values starting from zero as input. At the end of the execution, every node in the network will have its rank starting from the lowest bound (0).

While the distributed sorting algorithm we presented is not optimal in terms of the message complexity (see [9]), it is guaranteed to resume the correct behavior in $O(nh)$ time where h is the height of T, while using a memory space of $O(log(n)deg)$ bits, starting from *any* initial configuration.

References

[1] A. Arora and M. Gouda, "Distributed Reset," *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, December, 1990, pp. 316-331; also in *IEEE Transactions on Computers*, Vol. 43, No. 9, 1994, pp. 1026-1038.

[2] N. Chen, H. Yu, and S. Huang, "A Self-Stabilizing Algorithm for Constructing Spanning Trees," *Information Processing Letters*, Vol. 39, 1991, pp. 147-151.

[3] E. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM 17*, pp. 643-644, 1974.

[4] S. Dolev, A. Israeli, and S. Moran, "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, pp. 103-117, 1990; also *Distributed Computing* Vol. 7, 1993, pp. 3-16.

[5] M.C. Loui, "The complexity of sorting on distributed systems," *Information and Control*, Vol. 60, pp. 70-85, 1984.

[6] D. Rotem, N. Santoro, and J. Sidney, "Distributed Sorting," *IEEE Transactions on Computers*, Vol. c-34, No. 4, 1985, pp. 372-376.

[7] M. Schneider, "Self-Stabilization," *ACM Computing Surveys*, Vol. 25, No. 1, March 1993, pp. 45-67.

[8] G. Varghese, "Self-Stabilization by Counter Flushing," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, California, August 1994.

[9] S. Zaks, "Optimal Distributed Algorithms for Sorting and Ranking," *IEEE Transactions on Computers*, Vol. c-34, No. 4, 1985, pp. 376-379.

Figure V.1 Functions used in the Algorithm.

boolean **DONE** /* boolean function; returns *true* when not expecting acknowledgments from children, *false* otherwise; always returns *true* for the leaf nodes. */
begin return $(\sim (\bigvee_{j \in Children} Ack\_Expected[j]) \vee (|Children| = 0))$ **end**

COMPUTE_MIN /* Determine the minimum value, if any, in the set of values passed; if all values have been chosen, return a chosen indicator.*/
begin
if $((Id\_C) \wedge ((\bigwedge_{j \in Children} R\_Id\_C\_Ch[j]) \vee (|Children| = 0)))$ **then** $Id\_C\_T := true$
else $Id\_C\_T := false$; Min_Id :=minimum of unchosen Val's in T_i **fi**
if $((Val\_C) \wedge ((\bigwedge_{j \in Children} R\_Val\_C\_Ch[j]) \vee (|Children| = 0)))$ **then** $Val\_C\_T := true$
else $Val\_C\_T := false$; Min_Val :=minimum of unchosen Val's in T_i **fi**
end

CHOSEN /* Minimum values broadcast by root are chosen. */
begin
  if $Id = r\_id$ **then** $Final\_Val := r\_val$; $Id\_C := true$ **fi**;
  if $Val = r\_val$ **then** $Val\_C := true$ **fi**
end

R_MIN_CHILD /* Receive minimum values from a child. */
begin
  $Ack\_Expected[s] := false$;
  $R\_Id\_Ch[s] := r\_id$; $R\_Val\_Ch[s] := r\_val$;
  $R\_Id\_C\_Ch[s] := r\_id\_c$; $R\_Val\_C\_Ch[s] := r\_val\_c$
end

PROCESS /* PROCESS procedure for the root node */
begin
foreach $(j \in Children)$ **do** $Ack\_Expected[j] := true$ **done**
end

PROCESS /* PROCESS procedure for the other nodes */
begin
  if $(r\_id\_c \vee r\_val\_c)$ **then** $Id\_C := false$; $Val\_C := false$
  /*Reset wave */
  else CHOSEN **fi**;
  /* Remove Id and/or Val if no longer needed. */
  foreach $(j \in Children)$ **do** $Ack\_Expected[j] := true$
done;
end

BCHILDREN /* Broadcasts values to children. */
begin
  foreach $(j \in Children)$ **do**
  $send(j, b\_id, b\_val, b\_id\_c, b\_val\_c, Counter)$ **done**
end