1-1-1995

Self-stabilizing sorting algorithms

Joseph Chacko
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/495

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeib Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
SELF-STABILIZING SORTING

ALGORITHMS

by

Joseph Chacko

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
August 1995
The thesis of Joseph Chacko for the degree of Master of Science in Computer Science is approved.

Chairperson. Ajoy Kumar Datta. Ph.D

Examinining Committee Member. Kazem Taghva. Ph.D

Examinining Committee Member, Laxmi Gewali. Ph.D

Graduate Faculty Representative. Ashok Iyer. Ph.D

Interim Graduate Dean. Cheryl L. Bowles, Ed.D

University of Nevada, Las Vegas
August 1995
ABSTRACT

A distributed system consists of a set of machines which do not share a global memory. Depending on the connectivity of the network, each machine gets a partial view of the global state. Transient failures in one area of the network may go unnoticed in other areas and may cause the system to go to an illegal global state. However, if the system were self-stabilizing, it would be guaranteed that regardless of the current state, the system would recover to a legal configuration in a finite number of moves.

The traditional way of creating reliable systems is to make redundant components. Self-stabilization allows systems to be fault tolerant through software as well. This is an evolving paradigm in the design of robust distributed systems. The ability to recover spontaneously from an arbitrary state makes self-stabilizing systems immune to transient failures or perturbations in the system state such as changes in network topology.

This thesis presents an $O(nh)$ fault-tolerant distributed sorting algorithm for a tree network, where $n$ is the number of nodes in the system, and $h$ is the height of the tree. Fault-tolerance is achieved using Dijkstra’s paradigm of self-stabilization which is a method of non-masking fault-tolerance embedding the fault-tolerance within the algorithm. Varghese’s counter flushing method is used in order to achieve synchronization among processes in the system. In the distributed sorting problem each node is given a value and an id which are non-corruptible. The idea is to have each node take a specific value based on its id. The algorithm handles transient faults by weeding out false information in the system. Nodes can start with completely false information concerning the values and ids of the system yet the intended behavior is still achieved. Also, nodes are allowed to crash and re-enter the system later as well as allowing new nodes to enter the system.
# Contents

Abstract ............................................................... iii
Acknowledgments .................................................. v

1 INTRODUCTION ................................................. 1
  1.1 Concepts ......................................................... 4
    1.1.1 Execution models ......................................... 5
    Demons .......................................................... 5
    Message Passing Verses Shared Memory .................. 6
  1.1.2 Methods of Self-stabilization ......................... 7
    Closure and Convergence ..................................... 7
    Local Checking and Correction ............................ 8
    Distributed Checking ......................................... 10
    Distributed Resets ............................................ 12
    Counter Flushing .............................................. 14

2 SORTING ALGORITHM .................................... 18
  2.1 Counter Flushing on Tree-Based Algorithms ............ 19
  2.2 Sorting Algorithm .......................................... 21
  2.3 Correctness Reasoning ...................................... 24

3 CONCLUSIONS ............................................... 32

BIBLIOGRAPHY .................................................. 33
ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Ajoy Kumar Datta for helping and encouraging me throughout the course of this thesis. As a student coming back to school after twelve years of being in a work environment, I needed extra attention to keep the momentum of my studies. He was always there as my advisor and, more importantly, as a friend. Special thanks to Dr. Kazem Taghva, Dr. Laxmi Gewali, and Dr. Ashok Iyer for serving on my committee. This thesis is dedicated to my parents, my wife Liby, and my son Christopher who were always the drive and moral support for completing my research work.

J.C.
A fundamental criterion in the design of robust distributed systems is to embed the capability of recovery from unforeseen perturbances. While most of the existing systems cater to permanent failures by introducing redundant components, the issue of transient failures is often ignored or inadequately addressed. Considering the computation in a distributed system to be a totally or partially ordered sequence of states in the state space, it is conceivable to encounter a transient malfunction due to message corruption, sensor malfunction or incorrect read/write memory operations, that transforms the global state of the system into an illegal state, from which recovery is not guaranteed. Examples are token-ring networks in which the token is lost or duplicate tokens are generated, or sliding window protocols in which the window alignment is lost due to transient errors. The essence of these examples is that if the set of possible global states of a distributed system is partitioned into legal and illegal states, then transient failures can potentially put the system into an illegal state, which may continue indefinitely unless it is externally detected and suitable corrective measures are taken. A self-stabilizing system guarantees that regardless of the current state, the system recovers to a legal configuration in a finite number of steps and remains in the legal configuration thereafter, until a subsequent malfunction occurs. This property makes the system more robust. No startup or initialization procedure needs
to be used because the system stabilizes by itself. If one machine fails and restarts, its local state may cause an illegal global state, but the system will correct itself in a finite amount of time. The ability of the system to correct certain errors without outside intervention makes a self-stabilizing system more reliable and more desirable than systems that are not self-stabilizing.

Ghosh in [21] defined self-stabilization as an exercise for achieving global convergence through local actions. Consider computation as a journey in state space from some initial state to a final state satisfying a condition. A privilege is a local measure of the distance of the current state from the final state. In arbitrary initial state, any number of machines may enjoy privileges but in the final state, no machine may enjoy a privilege.

It seems often times, that self-stabilization is an easier model than others in fault-tolerance. For instance, every process is guaranteed to participate in the algorithm and to execute only according to its code under all circumstances. This differs from, for example, Byzantine failures, where some processes can actually ignore the code taking arbitrary and even malicious steps in the system [24]. However, self-stabilization is a more complete model in fault-tolerance. Other models only allow some specific subset of processes to fail with correctness only guaranteed for processes which have never failed. In self-stabilization, all processes may have arbitrary initial state, but will stabilize such that a global legal state is reached.

Self-stabilization is not an easy task since processes have no way to distinguish when the system has stabilized. Also, since there is no initialization of variables, no process can rely on its local variables and counters since processes may be started with arbitrary values in the domain of their variables. Thus, self-stabilizing algorithms can never turn over control to other non-stabilizing protocols since that would require that a process be able to know when the system is stabilized. Another interesting result of this property is noted by Lamport and Lynch in [26] where they note:

Simply bounding the number of instance identifiers is of little practical significance, since practical bounds on an unbounded number of identifiers are easy to find. For example, with 64-bit identifiers, a system that
chooses ten per second and was started at the beginning of the universe would not run out of identifiers for several billion more years. However, through a transient error, a node might choose to large an identifier, causing the system to run out of identifiers billions of years too soon — perhaps within a few seconds. A self-stabilizing algorithm using a finite number of identifiers would be quite useful, but we know of no such algorithm.

The notion of self-stabilization has been prevalent in the field of mathematics and control theory for many years. Consider for example the Newton-Raphson method of finding the square root of a number where, regardless of what estimate is made about the initial value of the square root, the solution converges to the desired value in a finite number of steps. Similar notions have been used in feedback control systems for many decades. In the field of distributed systems, the study of self-stabilization was pioneered by Dijkstra [16] who solved the mutual exclusion problem for a ring of processors in a self-stabilizing manner. However, Dijkstra did not address the significance of the property of self-stabilization [31]. This fact was belabored by Lamport who said at his invited address in 1983 to the 3rd ACM Symposium on Principles of Distributed Computing [25]:

I regard this as Dijkstra’s most brilliant work - at least, his most brilliant published paper. It’s almost completely unknown. I regard it to be a milestone in work on fault tolerance.

Dijkstra’s notion of self-stabilization, which originally had a very narrow scope of application is proving to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed systems. The application of self-stabilization has since expanded to many areas of study related to distributed systems: message passing protocols, leader election, network routing, graph algorithms, atomic-commit, etc.

Self-stabilization provides a formal and unified approach to fault tolerance [31]. No treatment of each of the issues separately is necessary. Coordination loss is an example of a transient failure. Other methods attempt to mask the occurrence of
errors and thus prevent failure. Self-stabilization guarantees recovery in case transient faults occur. Thus self-stabilization provides a complementary approach to other methods of fault tolerance. Inherent is the assumption that while the abstract state of the program or system may be corrupted, the program itself is inviolable. A self-stabilizing protocol can recover from corruption of volatile memory. This property has not existed in previous fault-tolerant models (e.g., fail-stop, omission) [8]. "The self-stabilization model is especially appropriate for the case of infrequent catastrophes: every once in a while the system may crash yielding an arbitrary and possibly illegal state." [8]

1.1 Concepts

A distributed system consists of a set of processors, $P_1, P_2, P_3, \ldots, P_n$, that are interconnected with communication channels, $(P_i, P_j)$. Such a system will be represented by a graph $G = (V, E)$ where $V$ is the set of processors or nodes with $|V| = n$ and $E$ is set of connecting channels between any two neighboring nodes. We will use the terms process, processor, and node interchangeably throughout the sequel. Likewise, we will use the terms channel and link interchangeably in the sequel.

Each process, $P_i$ in a distributed system owns and maintains a set of variables and executes a program. The variables are read/write for the owner, but may only be read by neighboring processes to the owner. The program at each process takes on the following form:

\[ < \text{rule} > \| \cdots \| < \text{rule} > \]

With each rule taking the form:

\[ < \text{guard} > \rightarrow < \text{assignment statement} > \]

A guard in a process is a Boolean expression over its own variables and the variables of its neighbors. An assignment statement updates the values of the variables of a process. A rule whose guard is true at some system state is said to be enabled, and a process with an enabled rule is said to be privileged.

A state of a process is defined by a value for each variable in the node. A system
state or global state is the Cartesian product of the states of each process in the system.

When a node is privileged, it will within a finite amount of time make a move, which corresponds to executing its assignment statement and changing its local state, and thus changing the global state. Eventually, a series of privileges and moves will lead to a legal global state where the behavior of the system coincides with the specification of the system.

1.1.1 Execution models

There are number of execution models defined in the literature that range in discussion from scheduling demons to message passing versus shared memory [11, 12, 13, 19, 22].

Demons

One of the fundamental pieces in designing self-stabilizing systems is deciding which node or nodes will make a move at time \( t \) when several nodes are privileged. In Dijkstra's original paper, he mentioned two types of scheduling demons, a central demon and a distributed demon [16]. Other demons refining atomicity have also been introduced later [12, 19, 22]. Huang, Wu, and Tsai [22] have capsulized scheduling demons into four categories:

1. serial model
2. synchronous model
3. synchronized distributed model
4. distributed model

The serial model is equivalent to Dijkstra's central demon, and assumes that when multiple nodes are privileged, only one node will take an atomic step. In the serial model, an atomic step consists of reading the states of a node's neighbors, and also of itself, and then making a move to change its state. This is the model assumed in Dijkstra's original paper, and is the model he proved in [17].

The second model, the synchronous model, allows all nodes to make moves simul-
temaneously. Atomicity in this model is the same as in the serial model. The synchronized distributed model is similar to the synchronous model, however, instead of all nodes making simultaneous moves, an arbitrary subset of privileged nodes makes a move. This third model corresponds to the distributed demon of Dijkstra, with the second being a more refined version thereof. These two models are discussed in length in [12], and are called the synchronous step and the parallel step, consecutively. The advantage in these first three models is that a node knows the exact state of each of its neighbors.

In the fourth model, the distributed model (also called fully distributed demon or distributed demon assuming only read/write atomicity [19]), is like the synchronized distributed model in that a subset of nodes makes moves simultaneously, but now a weaker assumption is made about atomicity. In this model, an atomic step consists of a read or a write step, but not both. By making this refinement, a node only knows a recorded state of its neighbors rather than the actual state of its neighbors. This model is preferable, since its assumptions are the weakest, and thus the easiest to implement.

Flatebo and Datta have proposed one additional demon called the randomized central demon [20]. This demon is similar to the regular central demon in that each privileged node is selected one at a time to make a move atomically. However, with this demon, every node has an equal probability to be chosen. This is a stronger assumption than the standard centralized demon. This type of demon is used for a mutual exclusion algorithm in [20].

Message Passing Verses Shared Memory

There are two models that can be assumed in self-stabilizing systems, shared memory and message passing. In the shared memory model, two neighboring processes, \( P_i \) and \( P_j \), communicate through two shared communications registers, \( r_{ij} \) and \( r_{ji} \), where the first letter in the subscript indicates the process which writes its local variables so that the process indicated in the second letter of the subscript can read it [19]. In the message passing protocols, a process \( P_i \) sends a read message to a neighbor, \( P_j \),
and then $P_i$ waits until it receives a message containing the requested value(s) from $P_j$.

### 1.1.2 Methods of Self-stabilization

In the early literature for self-stabilization, problems were solved on a problem by problem basis, trying to modify specific algorithms to be self-stabilizing without using any unified methods for reaching the self-stabilizing property. This is a problem that has plagued much of the area of fault tolerant computing [3]. In recent literature however, such unified methods have seen an explosion of attention [1, 2, 4, 3, 5, 7, 8, 10, 24, 34].

**Closure and Convergence**

Arora and Gouda developed a system for designing fault-tolerant systems in a unified manner in [3]. While their paradigm is universal to all fault-tolerant systems, it is of particular use in self-stabilizing systems in providing a paradigm for creating such systems.

Arora and Gouda break fault-tolerant programs into four groups [3]:

1) Masking and global stabilizing.
2) Masking and local stabilizing.
3) Nonmasking and global stabilizing.
4) Nonmasking and local stabilizing.

A masking fault tolerant program is one in which the occurrence of faults is invisible external to the system. That is, faults have no effect on the system output. Nonmasking fault tolerance on the other hand refers to programs in which faults affect the output of the system, but for only a finite amount of time. Global stabilizing programs are ones in which any initial state converges to a legal global state. Local stabilizing programs are those that have a tighter fault span than that of global stabilizing ones. Clearly, self-stabilizing programs fall into class number three, nonmasking and global stabilizing.
The way that the four categories of fault tolerance above are motivated is through the definition of the system. Any fault tolerant system will have an invariant predicate, $S'$, in which program execution falls into the set of legal global states. There will also be an invariant predicate, $T$, in which program execution and faults will remain. It is evident that $S$ is thus a subset of $T$.

The formal definition of a $T$-tolerant fault tolerant system is thus a system in which system execution and faults are closed in a fault span $T$, and normal program execution without faults starting in $T$ converges to $S$. This is to say that any execution of a fault tolerant program starting from a state where $T$ holds will result, eventually, in a state where $S$ holds.

The main design method under these definitions involves the use of what are called closure actions and convergence actions [3, 5]. The former are rules that perform the intended actions of the program. The latter are rules that force the system from an illegal global state into a legal global state without preventing closure actions from reaching their intended goal.

To the end of formal verification of these systems, closure actions are proven to meet the program specification, while convergence actions are shown to converge to $S$ through a constraint graph. A constraint graph is a graph that draws a relationship between variables and the actions that address those variables. In general, a series of steps must lead towards $S$ within the constraint graph so that $S$ is eventually reached. Fault intolerance and impossibility results can also be proven under this general, uniform model [3].

The fault tolerant problems solved by these methods include atomic commitment, data-transfer, byzantine agreement, delay-insensitive circuits, diffusing computation, spanning tree maintenance, and token ring mutual exclusion.

Local Checking and Correction

Another paradigm that is very useful for a large number of problems in distributed systems is the idea of local checking [8, 1, 33]. This method can be used as a backbone to other paradigms such as distributed reset subsystems [7, 8]. A network protocol is
said to be *locally checkable* if its set of legal states can be expressed as a conjunction of link predicates. A protocol (or a system) is called *locally correctable* if the global state eventually becomes legal even if each subsystem is corrected independently. If every link predicate is *stable*, i.e., remains true regardless of whether other link predicates are true, a locally checkable protocol is locally correctable. The basic idea is that each node periodically takes snapshots of each of its incident links. When the system is not in a global legal state, some node will have a local condition which is violated, and thus be able to make a move initiating the self-stabilizing global correction protocol. In this way, the program avoids having to rely on costly global snapshots. Additionally, no a priori knowledge of network size is necessary, and the protocols will work even if network partitions exist.

This idea comes in two flavors: global correcting and local correcting. The method used in [24] is *centralized checking and correction at a leader*. The idea for local checking is first seen in Afek, Kutten, and Yung [1] with the global correcting method used. The authors use this paradigm to construct a spanning tree algorithm that works with a distributed demon assuming only read/write atomicity. The stabilizing time of this protocol is $O(n^2)$.

Another work addressing this important and widely applicable paradigm is due to [8, 33] which allows local correction to stabilize the system rather than having the local detection merely initiate a global correction as in [1].

Varghese [33] divides the network into a number of overlapping *link subsystems*. A link subsystem consists of a pair of neighboring nodes and the channels between them. This work describes sufficient conditions under which these methods can be applied. Intuitively, a network protocol is *locally checkable* if whenever the protocol is in a bad state, some link subsystem is also in a bad state. Thus if the protocol is in a bad state, some link subsystem will be able to detect this fact locally. As in [24], one can correct a locally checkable protocol by doing *global correction* of the network. However, in some cases one can do better if the protocol is also *locally correctable*. Intuitively, a network protocol is locally correctable if the network can be corrected to a good state by each link subsystem independently correcting itself to a good state.
The method is not confined to acyclic graphs only; both the end-to-end and reset protocols work on arbitrary topologies.

The paper [33] gives a formal basis for the method of local checking and correction in message passing systems. These definitions are used to state a very important result, the *local correction theorem*. This theorem states that any locally checkable and correctable protocol can be transformed into an equivalent stabilizing protocol. This thesis applies the method of local checking to a simple mutual exclusion protocol. This research also contains another important result, the *tree correction theorem*. This theorem states that any locally checkable protocol on a tree can be efficiently stabilized in time proportional to the height of the tree. In other words, if the underlying topology is a tree, we can dispense with the need for local correctability.

The work of [33] proves another major result, the *global correction theorem*. This theorem states that any locally checkable protocol can be stabilized in time proportional to the number of network nodes. This theorem shows that we can dispense with the need for local correctability and the need for the underlying topology to be a tree as long as we are willing to pay a higher price in stabilization time. He presents stabilizing protocols for computing a spanning tree and solving the topology update problem as examples of global correction.

The first self-stabilizing end-to-end communication protocol in fail-stop networks is described in [8, 33]. The concept of local checking and local correction makes it possible to design the self-stabilizing protocols without the use of unbounded counters.

**Distributed Checking**

One of the most evident paradigms for making a system stabilize is by periodically taking a global snapshot of the system, and if no global legal predicate is satisfied, then start a protocol that returns the system to such a legal state. This type of design method is formalized in the works [24, 10].

Katz and Perry provide one of the earliest published paradigm based papers in the literature [24]. This paper provides a method by which a non-self-stabilizing program can be augmented into a self-stabilizing program. In general, a series of global
snapshots are taken in order to extend the program to be stabilizing. Specifically, a program, $Q$, is said to be a self-stabilizing extension of another program, $P$, if $Q$ is self-stabilizing and each global legal state in $Q$ has a projection of variables and messages onto a legal state of $P$. The main idea here is one called super-imposition in which repeated snapshots are taken insuring all along no interference with the underlying program. The results of the snapshot are interpreted at a distinguished leader called an initiator process. Then, a reset can be initiated (resets are covered later in this section).

The snapshot algorithm is an extension of Chandy and Lamport's work. The extension here works iteratively in waves of messages initiated by a leader, $P_0$, which also collects the final results of the snapshot. They prevent deadlock in this procedure by using a periodic sending of snapshot messages out. Iteration numbers are used to prevent two separate snapshots from conflicting.

Katz and Perry [24] show how to stabilize distributed algorithms by doing centralized checking at a leader. Also [1] described a self-stabilizing algorithm for leader election that took time $O(n)$. The combination of centralized checking and the need to elect a leader reduce the performance of the compiler.

Afek, et. al., [1] suggest replacing global checking, by doing local checking of neighboring nodes followed by global correction; they apply this idea to the problem of constructing a spanning tree. [8] takes the next natural step and shows how, in certain important cases, they can replace global correction by doing local correction of the state of a node and its neighbors. They apply their technique to some important interactive tasks such as end-to-end message delivery and network resets. By contrast, the distributed program checking [10] concentrates on general techniques for non-interactive tasks, for many of which (e.g., minimal spanning tree, etc.) no locally-correctable implementation is known.

In the work of Awerbuch and Varghese, two compilers are presented to yield self-stabilizing protocols, a rollback compiler and a re-synchronizer compiler [10]. Their method works specifically for programs which are non-interacting. This is to say this paradigm works with programs in which correctness is specified by an input/output
reaction. Similar to Katz and Perry, a periodic check is run. If a problem is revealed in this check, a recovery sequence results. The major difference here comes from no need for a leader node to interpret the information.

The first technique, rollback, works by all nodes in the system keeping logs of every move it has taken to get to its current state. Then, each node sends its log to all neighbors so that all nodes can check and subsequently correct every improper move it has made in the past. Obviously, these logs can grow to be quite unwieldy. If, however, some type of underlying synchronizer is implemented, the size of these logs can be reduced down to be proportional with the time complexity of the program. Using the periodic log checking with such a synchronizer yields the rollback method.

The rollback method, however, has the disadvantage of wasting space and bandwidth in situations where time complexity is not small. Therefore, another more optimized method must be developed with more general usage possibilities. The re-synchronizer method achieves this. This paradigm is essentially a self-stabilizing extension of Awerbuch’s ground breaking synchronizer protocol in [6]. Periodically, a check is made to see if all nodes are in synch. If not, a broadcast-convergecast is used to correct the system to be synchronized. Termination detection is used to determine when the system is finally synchronized again. The re-synchronizer uses the concepts of local checking and correction discussed in Section 1.1.2 [8].

The re-synchronizer can be improved using a single pulse checking method such as a distance variable. For example, if a node marks its distance to itself as 0, and a neighbor to that node is at distance 1 from that node, a node can check if it is in synch with a neighbor of smallest distance to the node in question. If not, the node has noticed an error in the system in a single pulse.

**Distributed Resets**

One of the first proposed and most logical paradigms for achieving self-stabilization is the idea of a reset subsystem placed within the program that upon detection of erroneous behavior resets the system to some predefined legal global state [4, 7].

The first paper on this subject is by Arora and Gouda [4]. The reset subsystem
suggested augments existing processes with three new disjoint protocols that can all work simultaneously yet still achieve a common goal of resetting the system. One protocol is a spanning tree creation built around a root. The root is determined by using a leader election protocol that simply chooses the highest node ID in the system as the leader. The second protocol is a diffusing computation which: (i) works the request up to the root, (ii) sends the reset down the spanning tree, and (iii) acknowledges back up the tree to the root where the reset completes. This reset works by including a rule that maintains the proper relationships between nodes so that no initialization is needed. A session number is used to distinguish between resets initiated independently at different nodes.

Two versions of this subsystem are proposed. one with a distributed demon and unbounded session numbers, and a second with a fully distributed demon and modulo arithmetic bounded session numbers. Arora and Gouda used the idea of round to compute the time complexity. A round in an asynchronous system is said to be that time during which every node in the system is allowed to make a move if privileged. Using this idea, the spanning tree/leader election portion stabilizes in $O(K + (\text{deg} \times \text{dia}))$ rounds where $K$ is the maximum number of up processes in the system, $\text{deg}$ is the maximum degree of any node, and $\text{dia}$ is the diameter of the network. The diffusing computation takes $O(\min(\text{ht} \times \text{deg}, K))$ rounds where $\text{ht}$ is the height of the spanning tree.

Other appearances of reset subsystems are present in the papers of Awerbuch et. al., in [7, 8, 9] and in [33]. They define a network reset protocol as a protocol that can be used by some other protocol $P$ in order to restore $P$ to a good state. Protocol $P$ is given interfaces to make reset requests; the network reset protocol responds by providing reset signals at each network node. If each node (that implements $P$) locally initializes its state at the instant it receives a signal, then $P$ will be restored to a good state. In order to use such a network reset protocol as a tool for building other stabilizing protocols, the network reset must itself be stabilizing. The method of local checking and correction is applied to create a stabilizing network reset protocol [8, 33].
The subsystem in [9] is considerably quicker than that of [4], first $O(n)$ then $O(dia)$. In [8], the reset works by using an ad hoc tree that does not outlive the system reset rather than creating a spanning tree. If it is assumed that no node makes an infinite number of reset requests, then the final reset signal sent from a node will offer a reference time for the system. In [7], the authors first present a series of synchronization protocols that eventually reach one that is optimal when using unbounded registers. Then, the registers are shown to be bounded using the technique of [9] which is a reset subsystem. This reset subsystem works by using local checking to detect when a maximum counter is in the system. This is then treated as a fault, and a reset is begun. This algorithm is made optimal by running a shortest path tree subgraph algorithm which is essentially Bellman-Ford's algorithm. By maintaining this subgraph, no path is larger than $dia$ and thus a reset is propagated in $O(dia)$ time [7].

Counter Flushing

One other method for designing self-stabilizing systems is counter flushing [34]. This technique is applicable to a number of distributed algorithms. This paradigm is most useful in stabilizing systems where a total algorithm [32] is needed. A total algorithm is one in which all nodes need to cooperate to achieve a common goal. This technique is applicable to token passing [16], propagation of information with feedback [30], deadlock detection [28], network resets [4], and non-blocking network snapshots [14]. Some problems solved by counter flushing can also be provided by local checking and correction. However, the method of local checking requires a fairly tedious enumeration of the protocol invariants which need to be checked; the addition of local checking also has a fair amount of complexity [33]. Also, taking correct snapshots of local state requires some careful synchronization which makes actual implementations somewhat tricky. By contrast, the modifications required by counter flushing are extremely simple.

Traditional models of a FIFO Data Link have used Unbounded Capacity Data Links that can store an unbounded number of packets. Now, real physical links do
have bound on the number of stored packets. However, the unbounded capacity model is a useful abstraction in a non-stabilizing context. Unfortunately, this is no longer true in a stabilizing setting. If the link can store an unbounded number of packets, it can have an unbounded number of "bad" packets in the initial state. It has been shown [18] that almost any non-trivial task is impossible in such a setting. Thus the original simplification of considering only unbounded links is no longer valid. Since real links are bounded and bounded links can be modeled elegantly, one can restrict to bounded link models.

A Unit Capacity Data Link (UDL) can store at most one packet at any instant. It can be shown [33] that a UDL can be implemented over real physical channels and can easily be generalized to bounded capacity data links. Roughly, a UDL can be thought of as a model of a reliable Data Link protocol that only delivers one message at a time (i.e., it uses a window size of 1). A UDL can be implemented [33] by an underlying stabilizing Data Link that sends and receives acknowledgments.

Modeling the synchrony between the transmitter and receiver is possible but is somewhat involved and also tends to imply that our basic idea is confined to such synchronous systems. Instead in [34], Varghese models a bounded link as a queue such that packet send events add elements asynchronously to the head of the queue and packet receive events remove elements asynchronously from the tail of the queue. The assumptions made are:

- For self-stabilization in the initial state all links queues are bounded. However, it allows the queue to grow unboundedly after that.
- For time complexity purposes, any message stored in a link queue will be delivered 1 unit of time later, regardless of the size of the queue.

To apply the counter flushing paradigm, we also need the assumption that there is a leader node in the network. There are many stabilizing protocols to construct a leader, especially the protocol of [7] that calculates this leader in $O(D)$ time, where $D$ is the network diameter. We assume therefore that a fixed node is designated as the leader. For the case of a general network, we also assume that there is also a pre-computed spanning tree of the network rooted at the leader. The spanning tree
also can be computed in $O(D)$ time as shown in [18].

Suppose in a network a leader node wishes to periodically send a Request packet to a set of network nodes. The responders must each send back a Respond packet before the sender sends its next request. In order to properly match responses to requests, the sender numbers each request with a counter. Responders only accept request packets with a number different from the last Request accepted. After accepting Request the responder sends back a Response with the same number as the Request. The sender retransmits the current Request till it receives each matching Response packets arrive, the sender chooses a new counter value and starts a new phase of sending Request.

The leader node ($r$) changes its counter value using a function \textit{CHOOSE} which returns an arbitrary counter value that is different from node $r$'s stored value. There are three specific realizations of the \textit{CHOOSE} function that guarantees self-stabilization: the \textit{Increment}, \textit{Random}, and \textit{Random-Increment}.

The size of the counter and the \textit{CHOOSE} function ensure that within bounded time, the sender will reach what is called a “fresh” counter value - i.e., a counter value that is not currently stored in either the links or the responders. The method is called \textit{counter flushing} because the request-response protocol must guarantee the following “flushing” property: Suppose the sender sends a request numbered $c$, where $c$ is a fresh value. Then after all matching responses to this request arrive, there must be no counter values other than $c$ that are stored in the links or at the responders. In other words, the sending of a freshly numbered request and the receipt of all matching responses, should “flush” the links and responders of “old” counter values.

Varghese [34] has shown that Dijkstra’s N-state example [16] can be understood very simply using counter flushing. This paper shows that this protocol can be easily extended to a message passing version which appears to be simpler than the token passing protocols used in today’s Local Area Networks. This technique is used to provide stabilizing deadlock detection by transforming a protocol due to [28]. The counter flushing is extended to trees as exemplified by the well-known Propagation of Information with Feedback (PIF) protocol due to Segall [30]. The
paper [34] also describes how to use counter flushing to produce a stabilizing reset for a general network. The reset protocol in turn can be used to stabilize certain diffusing computations. as shown by a stabilizing version of the Chandy-Lamport protocol [14] that stabilizes in $O(D)$. Varghese also conjectured that counter flushing techniques are applicable to virtual circuit problems as well.

There are some problems for which counter flushing is applicable but local checking is not (like token passing on a ring). The problems like routing protocols and leader election are examples where local checking is applicable but counter flushing is not. But, both paradigms are applicable for the reset problems and token passing on a tree.
Chapter 2

SORTING ALGORITHM

The problem of distributed sorting has been solved previously [35, 23, 27, 29], but the approaches used in these papers are not fault-tolerant. It is trivial to see that the algorithms are not stabilizing since they are correct only if the variables are properly initialized. This chapter presents a simple software fault-tolerant algorithm for this problem using self-stabilization. An instance of this problem has a set of connected nodes \( n \) in a tree (if the network topology is not a tree, then any of a number of spanning tree algorithms can be used to achieve this topology [4, 15]), each node has an \( id \) and \( value \). The goal is to sort the \( values \) of the system to correspond with \( ids \).

The pairing relationship works as follows: The node with the lowest \( id \) in the system takes the lowest \( value \) in the system, the second lowest \( id \) in the system takes the second lowest \( value \) in the system. This pattern continues until the highest \( id \) in the system takes the highest \( value \). A variable is used to store the sorted \( value \) at any particular node. We call the sorted \( value \) the \textit{final value} for a particular node.

Since the algorithm presented in this thesis is self-stabilizing, transient errors are handled automatically without any initialization or intervention. Thus the algorithm inherently allows nodes to have arbitrary values in their variables. This type of fault tolerance is highly robust. Not only is no initialization needed, but nodes can fail during the algorithm and new nodes can enter the system. Even in these circum-
stances the algorithm will converge to the intended behavior within finite time (in particular $O(nh)$ time where $n$ is the number of nodes in the system, and $h$ is the height of the tree). In this algorithm, this is achieved through a continuous reset process. Nodes communicate through a series of messages which are synchronized using counters. Each counter value initiated by the root represents a wave. After each wave, a new counter is initiated allowing nodes to reset themselves. In this way, new information can be processed as well as allowing nodes not properly initialized to have an opportunity to correct their bad states.

The remainder of the chapter is organized as follows: Section 2.1 discusses how to apply the paradigm of counter flushing [34] to tree based algorithms. Section 2.2 covers the sorting algorithm itself, Section 2.3 provides arguments as to the correctness of the given algorithm.

2.1 Counter Flushing on Tree-Based Algorithms

In Propagation of Information with Feedback (PIF), a single leader wishes to broadcast some information value to all nodes in the network and wishes to know when the information has reached all the nodes. In the stabilizing setting, we assume that the leader has a stream of values it wishes to broadcast to all neighbors; only after the $i$-th value is broadcast to a function $f$ that computes the next value to be sent as a function of the previous value sent.

We will assume as usual that we have a leader node (say $r$) and a spanning tree rooted at node $r$, such that each node $i$ has a parent variable $parent(i)$ that points to its parent in the tree. Without stabilization, it is easy to solve this problem using the protocols due to Segall [30]. When the root finishes broadcasting a previous value and receives acknowledgments from its children, it chooses a new value using the function $f$. It then sends a token message containing the new value to all its children; other nodes accept new values only from their parents, upon which they send the value to their children. When a leaf of the tree gets a new value, it simply sends an ack up to its parent. Nodes other than the root send an ack up to their parents, when they
have received acks from all children. When the root (i.e., the leader) receives an ack from all children, the root starts a new wave by choosing a new value.

To make the protocol stabilizing, we will tag each message sent and each value stored, with a counter. When sending a new value, the root chooses a new counter value. Nodes accept a new value only when it is tagged with a different counter value from the counter stored at the node. Nodes accept acks only when the counter in the ack matches with their current counter value.

Another fairly general method for constructing stabilizing protocols is the method of local checking [8]. In fact in [33] there is a theorem that states that any locally checkable protocol on a tree can be stabilized using local checking. Thus it is natural to ask whether we can solve the stabilizing PIF problem with local checking instead of counter flushing. However, it is easy to show that the PIF protocol is not locally checkable. A protocol is locally checkable only if whenever every pair of neighbors is in a good state, then the system is in a good state. Suppose we find a bad global state of a protocol such that every pair of neighbor is in a state that appears in some other good global state. Then every pair of neighbors appears to be in a good state locally but the system is in a bad state, and hence the protocol is not locally checkable.

In a good state of the PIF protocol there can be at most two values present in the tree, the value currently being propagated and the old value that is still present in the lower limbs of the tree. Thus in a good local state it is possible that a parent has counter $c$ and the child has counter $c' \neq c$. But in that case we can construct a bad global state in which each child of the root has a different counter value but each pair of neighbors appears to be in a good state locally. Thus the protocol is not locally checkable.

Propagation of Information with Feedback is a specific example of a centralized total algorithm [32].
2.2 Sorting Algorithm

The algorithm works as follows. Initially, all nodes in the network have arbitrary final values. These final values may not even be values ($val_i$) at any node in the network since no degree of initialization is used for software variables. Likewise, a variety of values exist for the counters at each node. It can be assumed without the loss of generality that an underlying spanning tree protocol exists. This is because all variables between the sorting algorithm and the spanning tree algorithm would have disjoint write sets, and thus non-interfering. By disjoint write-set we mean that the set of variables written to by the two protocols will be completely disjoint. This allows for the two algorithms to stabilize independently (the sorting algorithm still relies on information from the creation of the spanning tree). This underlying protocol will stabilize the $N_i$ (neighbor set) and $parent_i$ (parent node pointer) variables, and we assume these to be correct for the validity of this algorithm.

In the sorting algorithm, the predetermined leader $r$ (root node of the spanning tree) will, upon receiving information from all of its children (or at least believing it has) initiates a new counter value and passes a token message down through the network with that counter value, as well as its perceived minimums for the value and id in the network initiating a new wave. A wave can be defined as the set of Token messages starting from the root with a specific counter value, broadcast down the tree to the leaves, and finally broadcast back up the tree to the root.

Every node $i$ in the network checks these values and ids to see if they match their own. If indeed the value is its own $val_i$, $i$ will remove its $val_i$ from further consideration, since some other node $j$ will be choosing this value as its final value. If the id is that of $i$, then $i$ will take the value it has received as its final values, and will also remove its own id from further consideration. Node $i$ then forwards the information on to its children, and waits to hear the next response from its children. Upon receiving responses from all of its children, $i$ will recalculate the minimum value and id in the subtree rooted at $i$. This information is forwarded to the parent. In this manner, $r$ will eventually again pick another new counter and send its new perceived
minimums down through the network starting another wave. This guarantees that in finite time, all nodes will receive their final values and reset (actually this takes $O(nh)$ time where $n$ is the number of nodes in the system, and $h$ is the height of the spanning tree). Once correct synchronization results, each wave will begin with a fresh counter. A fresh counter is a counter value that does not exist anywhere in the system, either at the nodes or in the links.

Note that both sending information to the parent and children as well as processing information from the parent and children are all going on essentially concurrently within each node $i$. We say essentially because actually, only one action is performed within each node in each atomic step, but no strict ordering is enforced on which action is performed next. Only the assumption of a fair scheduler is used so that each action that has its preconditions satisfied will eventually be executed. In this way, the information passed around the network may be initially wrong. But, because of the nature of the synchronization achieved by counter flushing, we achieve a point where eventually only correct information is being passed around the network, and in finite time, all nodes pass around the correct information. Thus, all nodes will in finite time have their correct final value.

To be more specific, we now describe each support function for the actions. These functions can be seen in Figure 1 along with a description of the variables and data structures used by the algorithm.

The one function not shown in Figure 1 is the \textbf{CHOOSE}(Max, c) function. This function chooses a new counter value $c$ that is a positive integer less than or equal to \textit{Max}. This function can be implemented in three separate ways as discussed in [34]. We are assuming that the \textbf{CHOOSE} is implemented by a simple increment. If current value for the counter at $r$ is $c_r$, then \textbf{CHOOSE} will take the new value $c_{nr} = c_r + 1 \mod Max$. This implementation is assumed because it does not decrease the time complexity of the overall algorithm, and simplifies the proofs.

\textbf{FINISHED}(i) is a simple Boolean function that each node $i$ uses to determine whether or not it is expecting any input from its children. Node $i$ will expect input from its children when it has forwarded new information from the root to its children.
FINISHED\((i)\) is true when and only when node \(i\) has received a response from all of its children during the current wave.

**COMPUTE\_MINIMUM** is a function which accepts a set of numbers and special *deletion indications*. The function determines the minimum of the set and returns that minimum. Deletion indications are true entries into a special array received at node \(i\) indicating whether or not some node \(j\) has deleted its id or value from further consideration. If all values of the set are deletion indications, then a special designation is returned indicating all values have been deleted.

**DELETE\((i)\)** is a function to determine if the minimum *value* or *id* being passed through the network is the *value* or *id* of the process \(i\) running the function. If the *id* received is that of \(i\), \(i\) takes the minimum *value* as its final value in the sorting, and removes its *id* from further consideration by “deleting” it. If the *value* received is that of \(i\), \(i\) removes its *value* from further consideration by “deleting” it. “Deleting” of a *value* or *id* is done by marking a special boolean variable at \(i\) to true.

The actions of the algorithm are formally described in Figure 2, but we elaborate a bit on their functionality here.

**ROOT\_START**, is an internal action which is used only by the predetermined leader \(r\). Node \(r\) upon receiving information from all of its children first determines whether or not to reset the algorithm. This is done by determining if any deletion indications were received from its children. If there were, then \(r\) initiates a reset by resetting its perceived minimum *value* and *id* to its own *value* and *id* (the reset is propagated by \(r\) by sending the deletion down the network using the **SEND**\(_{i,j}\) action described below). Also, \(r\) removes any deletion indication concerning its own *id* and *val*\. If no reset is needed, then \(r\) will prepare to send the minimum *value* and *id* down through the network with the **SEND**\(_{i,j}\) action (shown below). Then \(r\) will check if it needs to delete its *id* or *value* by using the **DELETE**\((i)\) function. Finally, \(r\) picks a new counter value to start a new wave. Node \(r\) also sets itself so that it is expecting information from its children before sending.

**SEND**\(_{i,j}\) is an external action which relays information up and down the spanning tree. When relaying down the tree, \(i\) simply forwards the last received information
from a parent downward to its children. When relaying information up the tree, the minimum value and minimum id among those in i's subtree are forwarded. This is done in the leaves by having leaves send up their own id and value unless either of these have been deleted, in which case the deletion indication is forwarded up. In internal nodes, the minimums to be sent are determined by the RECEIVE\textsubscript{i,j} action described below. Whether information is being sent up or down, the counter value used in the message is equal to the counter value stored at node i.

RECEIVE\textsubscript{i,j} is an external action which takes information from neighboring nodes, and decides what information needs to be stored and what information needs to be relayed to other nodes. This action does the bulk of the work of the algorithm. Upon receiving information from the parent with a new counter value, the node i will set its stored values to those it has just received including the counter value. These will be the values forwarded to the children of i. Node i then sets itself to expect information from all of its children. Then, if the values received include a deletion indication, i resets its minimum value and id to its own. This is how the algorithm resets itself. Node i will then check whether or not to delete its id or value from further consideration, by using the DELETE\textsubscript{z} function. If i is receiving information from a child, then i will store that information. Once information is received from all its children, i will determine the minimum id and value in its sub-tree by using the COMPUTE\_MINIMUM function twice, once for value and once for id.

2.3 Correctness Reasoning

Lemma 2.1 Any counter value c\textsubscript{r} produced at the root will reach all nodes in the tree within O(h) time.

Proof: By induction on the distance in hops from the root.
Basis: Distance of 0. The proof for the root is trivial since the value is produced at the root.
Induction: The hypothesis is all nodes at distance δ from the root will have the
counter value $c_r$ (the counter value sent by the root). We will show that all nodes at distance $\delta + 1$ from the root will obtain the counter value $c_r$. A node with counter value of $c_r$ will send that same value continually to all of its children until a new value $c'_r$ is received at the node. Any child of these nodes will accept the value $c_r$ unless they already have the value $c_r$. Since, any node at distance $\delta + 1$ must be children of nodes at distance $\delta$, all nodes with distance $\delta + 1$ will either accept this value or already have it. Thus, all nodes with distance $\delta + 1$ will take the value $c_r$. Each passing of values from a node to its children takes $O(1)$ time. Therefore, all nodes will take the value in $O(h)$ time. □

**Lemma 2.2** A new counter will be produced at the root within $O(h)$ time from any arbitrary state.

**Proof:** A new counter is produced at the root whenever $\text{FINISHED}(r)$ is true.

Case 1: If the root is initialized with $\text{FINISHED}(r)$ to true, it produces a new counter in $O(1)$ time.

Case 2: If the root is initialized with $\text{FINISHED}(r)$ to false, then in the worst time instance, the root has just sent a token to all of its children. The token will reach all of its descendants in $O(h)$ time by Lemma 2.1. Once, all children have received the token, they will respond back up the tree with the same counter value after some local computation. Just as it takes $O(h)$ time to disseminate information down the tree, $O(h)$ will be required to return the information back up the tree. This can be seen by a simple inductive argument based on the maximum distance in hops of a node from a leaf. Information is guaranteed to flow up the tree, since any token produced by the root will reach all nodes, and then, all nodes will hold the same token value. Thus, when the leaves send information to their parents, it will be accepted because the counters are equal. Therefore, the total time needed before a new counter will be produced is $O(h)$ □

**Lemma 2.3** A fresh counter will be produced at the root within $O(nh)$ time from any arbitrary state.
Proof: By Lemma 2.2, a new counter is produced at the root every $O(h)$ time. Since links are initially bounded, and at most $n$ counter values exist at the nodes, there is a maximum number of counter values in the network $c_{max}$. We take the counter value at the root to be $c$. Because there are at most $n-1$ links in a tree, $c_{max} = L_{max}(n-1) + n$ where $L_{max}$ is the bound on the links. Therefore, if the maximum counter value is taken to be $Max > c_{max}$, a value $c'$ exists such that no node or outstanding link message has value $c'$ for its counter. Therefore, at most $c_{max}$ new counter values can be created at the root using the increment function before $c'$ is created at the root. Since $c'$ was not previously in the system, it is a fresh counter by definition. Since a new counter is produced in $O(h)$ time by Lemma 2.2, $c_{max} \times O(h)$ time is needed to create a fresh counter. Since $c_{max} = L_{max}(n-1) + n$, a fresh counter is produced at the root in $O(nh)$ time. □

Lemma 2.4 The minimum $id_j$ and $val_j$ of all processes $j$ in the subtree rooted at $i$ will reach node $i$ in $O(h)$ time from the time a new counter value is produced at the root.

Proof: There are two minimums that need to reach the node, the minimum $id_i$ and the minimum $val_i$ for all nodes $i$ in the network. The argument for one of these reaching the node is exactly the same as for the other. Therefore, we will only present the argument once and say that the "value" reaches the node in $O(h)$ time. The term "value" can be substituted with $val_i$ or $id_i$.

Once the root creates a new counter value, then all of its descendents will receive this information in $O(h)$ time by Lemma 2.1.

We now proceed with an inductive argument based on the maximum distance in hops of a node from a leaf.

Basis: The minimum "value" in a leaf node will simply be its own "value", since the sub-tree rooted at $i$ contains only $i$.

Induction: The induction hypothesis will be that any node at a maximum distance $\delta$ from a leaf node will have its minimum "value". We will show that any node at a maximum distance $\delta + 1$ will receive the minimum of its sub-tree. Node $i$ will upon
receiving information from all of its children, take the minimum of its own "value" and those "values" received from its children. Therefore, since each send and consequent receive takes $O(1)$ time, the overall time needed is $O(h)$. □

**Corollary 1** The minimum $id_i$ and $val_i$ of all processes $i$ in the system will reach the root in $O(h)$ time from the time a new counter value is produced at the root.

**Proof:** Follows directly from Lemma 2.4.

**Lemma 2.5** Once a fresh counter is produced at the root $r$, node $r$ will initiate a reset in $O(nh)$ time.

**Proof:** Since a fresh counter will not occur anywhere else in the system by definition, and all descendents of the root accept packets with counter values different from their own stored values, all nodes will accept the packets initiated at the root. Thus, all nodes are receiving information as passed from the root.

Two cases are needed: one assuming that no false information appears in the system, and one assuming that some false information does exist.

**Case 1:** It is assumed that no false information appears in the network. The minimum $id_i$ and $val_i$ in the system will reach the root by Corollary 1 in $O(h)$ time. Therefore, since all nodes are receiving the information from the root, that node $i$ with the lowest $id_i$ will receive back its $id_i$ and the lowest value for its final value. Node $i$ will then "delete" its $id_i$ from further consideration. Likewise, the node $i$ with the minimum $val_i$ will receive this information back from the root, and "delete" its $val_i$ from further consideration. These values are clearly deleted in $O(h)$ time by Lemma 2.1. Once these values are deleted from consideration, the second minimums will then become the minimums. A simple inductive argument is used to show that every $O(h)$ time a new set of $value$ and $id$ are deleted. Once these are deleted, the deletion indication is forwarded to the root in $O(h)$ time. Thus, since $n$ values and $ids$ are in the network, in $O(nh)$ time a reset occurs.

**Case 2:** If correct behavior cannot be assumed, then some node(s) will have a false "value(s)". In this case, three sub-cases can occur. Either a false $id(s)$ exist or
a false value(s) exists or both. If it is a false id(s) that exists, then all values will be deleted before all ids have taken a value. This is clearly less than the $O(nh)$ time indicated above. If it is a false value(s) that exists, then all ids will be deleted before all values are used, again clearly in less than $O(nh)$ time. If both false ids and values exits, then whichever there are more false variables for will be all deleted first. Since, as many as $n$ false ids or false values could exist, $O(nh)$ time could be required. In all three sub-cases, once all "values" are deleted the root will learn within $O(h)$ time and a reset occurs.

Thus, at most $O(nd)$ is required to achieve a reset initiation in the system at the root. □

**Lemma 2.6** The true minimum id and value in the system will reach the root in $O(h)$ time from a reset.

**Proof:** Once the reset is initiated, the root will set its minimum id and minimum value to be its own $id_r$ and $val_r$ respectively. The root will then pass the deletion indication down to its children who will then reset their perceived minimums to be their own $id_i$ and $val_i$ just as the root did. The reset reaches all nodes in $O(h)$ time by Lemma 2.1. By Corollary 1, the minimums in the network are now passed up to the root within another $O(h)$ time, i.e., a total of $O(h)$ time. No false minimum values are sent because all nodes have reset their perceived minimums to be their uncorruptable true $id_i$ and $val_i$. □

**Theorem 2.1** The given algorithm is a correct distributed sorting algorithm, and stabilizes within $O(nh)$ time.

**Proof:** By Lemma 2.3, it can be seen that a fresh counter is produced in $O(nh)$ time from any initial state. Once a fresh counter is produced, all nodes will be synchronized with each other. Thus, all nodes are participating in the same wave.

At the time of a fresh counter, a reset will be initiated within $O(nh)$ time by Lemma 2.5. Once this occurs, by Lemma 2.6 the true minimum id, and val, in the network will reach the root in $O(h)$ time. Then, correct behavior is achieved meaning
no false values remain in the system. By the arguments concerning no false values leading to a reset in Lemma 2.5, the proper final values are given to the proper nodes in $O(nh)$ time. Thus, the algorithm is both correct and self stabilizing. Therefore, the total time complexity is $O(nh) + O(nh) + O(h) + O(nh) = O(nh)$. □
The state of each node $i$ consists of:

- $N_i$ the set of neighbors of the node $i$ in the spanning tree.
- $\text{parent}_i$ the id of the parent node of node $i$.
- $c_i$ a counter.
- $\text{token}\_\text{expected}_i[j]$ a boolean flag for each child $j$ of $i$; true indicates $i$ is expecting a token from $j$.
- $\text{val}_i$ initial value at node $i$.
- $id_i$ id of node $i$.
- $\text{sr}\_\text{val}_i$ value being sent/received.
- $\text{sr}\_\text{id}_i$ id being sent/received.
- $\text{min}\_\text{val}_i$ the minimum value in the tree rooted at $i$.
- $\text{min}\_id_i$ the minimum id in the tree rooted at $i$.
- $\text{final}\_\text{val}_i$ the final value at node $i$ after sorting.
- $\text{r}\_\text{val}_i[j]$ value last received from the child $j$ of $i$.
- $\text{r}\_\text{id}_i[j]$ id last received from the child $j$ of $i$.
- $\text{val}\_d_i$ a boolean flag; true indicates value at $i$ is deleted.
- $\text{id}\_d_i$ a boolean flag; true indicates id at $i$ is deleted.
- $\text{sr}\_\text{val}\_d_i$ a boolean flag; true indicates value at each node in the tree rooted at $i$ is deleted.
- $\text{sr}\_\text{id}\_d_i$ a boolean flag; true indicates id at each node in the tree rooted at $i$ is deleted.
- $\text{r}\_\text{val}\_d_i[j]$ a boolean flag for each child $j$ of $i$; true indicates the child $j$ of $i$ has informed $i$ that value at each node in the tree rooted at $j$ is deleted.
- $\text{r}\_\text{id}\_d_i[j]$ a boolean flag for each child $j$ of $i$; true indicates the child $j$ of $i$ has informed $i$ that id at each node in the tree rooted at $j$ is deleted.

A token message is encoded as a tuple $(\text{Token}, c, \text{sr}\_\text{val}, \text{sr}\_\text{id}, \text{sr}\_\text{val}\_d, \text{sr}\_\text{id}\_d)$ where the variables $\text{Token}, c, \text{sr}\_\text{val}, \text{sr}\_\text{id}, \text{sr}\_\text{val}\_d,$ and $\text{sr}\_\text{id}\_d$ contain the values of a node being sent/received.

**FINISHED** (* boolean function; set to true when not expecting tokens from any children *)

Return true if for all children $k$ of $i$: $\text{token}\_\text{expected}_i[k] = false$

Return true if $i$ is a leaf node.

**COMPUTE_MINIMUM** (* determine the minimum value, if any, in the set of values passed; if all values have been deleted, return a deletion indication *)

If (for all values of $m$, 1 through $k + 1$: $\text{del}_i[m] = true$)

$\text{del}_0 = true$

Else

$\text{del}_0 = false$

$\text{min}\_dt = \text{minimum of } \{\text{dt}[m] \text{ such that } 1 \leq m \leq k + 1 \text{ and } \text{del}_i[m] = false\}$

**DELETE** (* minimum values broadcast by root are deleted *)

If $\text{sr}\_\text{id}_i = \text{id}_i$ (* broadcast value is destined for $i$; set final value at $i$; remove $\text{sr}\_\text{id}_i$ from further consideration *)

$\text{final}\_\text{val}_i = \text{sr}\_\text{val}_i$; $\text{id}\_d_i = true$

If $\text{sr}\_\text{val}_i = \text{val}_i$ (* val_i will be stored as final value at $\text{sr}\_\text{id}_i$; remove $\text{sr}\_\text{val}_i$ from further consideration *)

$\text{val}\_d_i = true$

Figure 1. The variables, functions, and procedure used by the sorting algorithm.
ROOT.START, (* Leader starts a new cycle of broadcasting values. *)

Preconditions:
FINISHED

Effects: (* compute new values of sr.val, and sr.id, to be broadcast *)
If (sr.val.d = true or sr.id.d = true) (* reset min.val, and min.id, *)
val.d = false; id.d = false
Else sr.val = min.val; sr.id = min.id,
DELETE(r) (* delete val, or id, if no longer needed *)
cr = CHOOSE(Max, c) (* choose new counter value *)
For all children k of r
"token.expected,"[k] = true (* set to true when expecting a token *)

SENDi,j(Token, c, sr.val, sr.id, sr.val.d, sr.id.d) (* Node i sends token to node j *)

Preconditions:
c = ci (* set counter in token message equal to local counter *)
If j ≠ parenti (* j is a child of i *) (* send values equal to stored values *)
sr.val = sr.vali; sr.id = sr.idi; sr.val.d = sr.val.d; sr.id.d = sr.id.d;
Else If (j = parenti and FINISHED) (* j is the parent of i *)
(* send values equal to the current minimum values *)
If (|ci| = 1 and i ≠ r) (* i is a leaf node *)
If not val.d;
"min.val"," = vali; sr.val.d; = val.d;
If not id.d;
"min.id"," = idi; sr.id.d; = id.d;
sr.val = min.val; sr.id = min.id; sr.val.d = sr.val.d; sr.id.d = sr.id.d;

RECEIVEj,i(Token, c, sr.val, sr.id, sr.val.d, sr.id.d) (* Node i receives token from node j *)

Effects:
If j = parenti and c ≠ ci (* new counter from parent *)
(* set stored values equal to values in token message *)
sr.val = sr.vali; sr.id = sr.idi; sr.val.d = sr.val.d; sr.id.d = sr.id.d;
ci = c (* set local counter equal to counter in token message *)
For all children k of i
"token.expected,"[j] = true (* set to true when expecting a token *)
If (sr.val.d = true or sr.id.d = true) (* reset min.val, and min.id, *)
val.d = false; id.d = false
Else DELETE(i) (* delete value, or id, if no longer needed *)
Else If (j ≠ parenti and c = ci) (* current minimum values from a child *)
"token.expected,"[j] = false
r.val.c,i[j] = sr.val; r.id.c,i[j] = sr.id; r.val.d.c,i[j] = sr.val.d; r.id.d.c,i[j] = sr.id.d;
If FINISHED (* update min.val, and min.id, *)
COMPUTE.MINIMUM(input : vali, r.val.c,i, val.d, r.val.d.c,i;
output : min.val, sr.val.d)
COMPUTE.MINIMUM(input : idi, r.id.c,i, id.d, r.id.d.c,i;
output : min.idi, sr.id.d)

Figure 2. Sorting algorithm.
Chapter 3

CONCLUSIONS

Self-stabilization is an evolving paradigm in the design of robust distributed systems. The ability to recover spontaneously from an arbitrary state makes these systems immune to transient failures or perturbations in the system state. The uses of self-stabilization have spread to many areas of distributed systems. Work has been done in areas such as mutual exclusion, leader election, routing and topology update, and k-exclusion. Other areas in which research has been done are as follows: communication protocols, other network algorithms, clock synchronization, Byzantine generals problem, consensus and commit, and other fault-tolerance problems.

Self-stabilization can be used in any area which has well defined global states. The perturbances of the system are any changes in the global state, and the legal state which the system converges to is the solution to the problem. Self-stabilization is used in many areas of computer science. It can even be used in areas such as machine learning and neural networks where the legal states in the system are a set of facts. It provides an effective way of dealing with machine failures and transient faults in the distributed system environment, and an effective way of dealing with continuously changing data in the algorithm’s environment. Self-stabilization is now an important concept in the design of fault-tolerant systems and algorithms.

The goal of research in self-stabilization is to design fault-tolerant systems. If a
system is self-stabilizing, it can automatically recover from transient errors and has
an inherent fault-tolerance. There are two main aspects of such research. The first
is the designing of fault-tolerant systems. The other is the analysis of self-stabilizing
systems. This analysis will lead to a better understanding of both self-stabilization
and fault-tolerance. Formalizing the ideas and properties of self-stabilization will
help researchers in the area of fault-tolerance. This analysis includes many areas:
How many moves before the system is stable? Given the frequency of errors, is a
self-stabilizing system useful? Which problems can and can not be solved by self-
stabilization? How can we easily verify that the system is self-stabilizing? What
problems can be solved with deterministic, uniform algorithms? What topologies
can algorithms be applied to? These are the questions about self-stabilization that
researchers are attempting to answer.

This thesis has provided a simple global non-masking fault tolerant algorithm for
the distributed sorting problem. The significance of such an algorithm lies in that no
initiation is needed, transient errors are handled in software, and no need for human
intervention in the protocol. This algorithm is guaranteed to resume correct behavior
in $O(nh)$ time where $h$ is the height of the spanning tree. This guarantee is achieved
by a continuous system reset being a part of the network. This type of fault-tolerance
allows a great deal of flexibility since nodes can fail and leave the system and others
join the system without the need for human intervention to stabilize the system.
Furthermore, the algorithm given is the first solution for distributed sorting using
self-stabilization.
Bibliography


