Computer Science
Technical Report

Colorado State University

Self-Stabilization: A New Paradigm for Fault Tolerance in Distributed Algorithm Design*

Gheorghe Antonoiu and Pradip K. Srimani
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

November 10, 1997

Technical Report CS-97-120

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

*To appear in the Proceedings of **International Conference on Computers and Devices (CODEC)**, Calcutta, India, January 14-17, 1998.

Self-Stabilization: A New Paradigm for Fault Tolerance in Distributed Algorithm Design*

Gheorghe Antonoiu and Pradip K. Srimani
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

Abstract

Our purpose in the present paper is to present a brief overview of the relatively new paradigm of self-stabilization to provide fault tolerance in distributed systems. Stabilizing algorithms are optimistic *in the sense that the distributed system may temporarily behave inconsistently but a return to correct system behavior is guranteed in finite time while traditional robust distributed algorithms follow a* pessimistic *approach in that it protects against the worst possible scenario which demands an assumption of the upper bound on the number of faults.*

1 Introduction

Robustness is one of the most important requirements of modern distributed systems. Different types of faults are likely to occur at various parts of the system. These systems go through the transient states because they are exposed to constant change of their environment. In a distributed system the computing elements or nodes exchange information only by message passing. One of the goals of a distributed system is that the system should function correctly in spite of intermittent faults. In other words, the global state of the system should ideally remain in the legitimate state. Often, malfunctions or perturbations bring the system to some illegitimate state, and it is desirable that the system be automatically brought back to the legitimate state without the interference of an external agent. Systems that reach the legitimate state starting from any illegitimate state in a finite number of steps are called *self-stabilizing systems* [1, 2]. This kind of property is highly desirable for any distributed system, since without having a global memory global synchronization is achieved in finite time and thus the system can correct itself automatically from spurious perturbation or failures. Our purpose in the present paper is to present a brief high level overview of the recently emerging area of self-stabilizing distributed algorithm design; a somewhat dated survey can be found in [3].

*To appear in the Proceedings of **International Conference on Computers and Devices (CODEC)**, Calcutta, India, January 14-17, 1998.

2 Distributed Systems and Algorithms

Distributed computing systems have experienced a massive growth in the last few years. Many new frontiers of applications have opened up due to the availability of these distributed and parallel systems. On site credit card validation, integrated airline reservation system, world-wide automatic teller machine network, Internet email system, world wide web, national and international information superhighways are, among many others, all examples of successful application of distributed systems in everyday real life applications. More and more organizations with diverse objectives are moving to employ computing as well as other systems in newer and more effective ways and thus we are observing a growth of more and more large scale distributed applications. Today, the distributed computing systems offer highly reliable, highly integrated computing to users who may be physically separated and interacting through diverse medium of communication via different hardware and software platforms. Yet, the objective is to provide a seamless whole and consistent service to different people by combining large number of mostly independent programs running at places geographically scattered all over the world on a variety of heterogeneous platforms.

A distributed system essentially consists of a number of *autonomous* computers (with their own hardware and software) who are connected to each other via a communication network. Despite this *interconnectivity* the participating computing site can run its applications independent of the others while the networking software (and hardware) provides the functionalities like database, shared storage (data and code) etc. Typically, there is no shared memory and communication between two nodes (sites) is implemented

through explicit message passing. The most important characteristics of the system is that it presents to the user (at any of the participating sites) a view of a *single* highly reliable program thus making the decentralized nature of the hardware and the software transparent to the user. The challenge of developing software, hardware and network interface becomes more formidable because of the need to dynamically respond to failures and subsequent recoveries.

In the last five or so years we have seen excellent growth of client server computing, and networked personal computing systems. The next phase of distributed computing will probably see a growth in the area of enterprise computing systems where a very close coupling (coordination) is necessary between a large number of applications running on different platforms across the globe connected by network hardware and software. Mobile computing, electronic stock and bond market, combining far away resources to develop a complex application like weather modeling or aircraft designing or nuclear explosion simulation could all be example settings where distributed computing will be indispensable. To quote Birman in his forward to the book [4], "an awesome resource has been created by the computing and telecommunication technologies, but it is still largely untapped".

Interconnection networks are integral in designing distributed systems; there are a large variety of interconnection networks that differ in transmission speed, topology used (point to point or shared medium), area they serve (LAN, WAN) data package format, switching mechanisms, network architecture, the error rate etc. The performance of the distributed systems and the applications running thereon depend heavily on the implementation of the underlying network architecture. For our purpose we simply assume that the network nodes are connected by point to point links according to some given topology (since this seems to capture the basic requirement of node connectivity in designing distributed algorithms).

3 Fault Tolerance in Distributed Systems

There are different existing approaches towards designing fault tolerant software like N-version programming, recovery blocks, consensus recovery blocks, etc. But design of fault tolerant software or algorithms has been traditionally investigated in the context of particular applications, system architectures as well as specific technologies. As a result we have different models and techniques for different applications and there is no simple way to verify these fault tolerant systems. Also, the basic approach in designing fault tolerant software has traditionally been to mask or tolerate design faults in the software itself.

The common approach to design the fault tolerant systems is to mask the effects of the fault; but, fault masking is not free; it requires additional hardware or software and it considerably increases the cost of the system. This additional cost may not be an economic option, especially when most faults are transient in nature and a temporary unavailability of a system service is acceptable. *Self-stabilization* is a relatively new way of looking at system fault tolerance, especially it provides a "built-in-safeguard" against "transient failures" that might corrupt the data in a distributed system.

4 Self-Stabilization – A Paradigm for Distributed Fault Tolerance

The objective of self-stabilization is (as opposed to mask faults) to recover from failure in a reasonable time and without intervention by any external agency. Self-stabilization is based on two basic ideas: first, the code executed by a node is re-entrant and incorruptible (as if written in a fault resilient memory) and transient faults affect only data locations; second, a fault free system behavior is usually checked by evaluating some predicate of the system state variables. Every node has a set of local variables whose contents specify the local state of the node. The state of the entire system, called the *global state*, is the union of the local states of all the nodes in the system. Each node is allowed to have only a partial view of the global state, and this depends on the connectivity of the system and the propagation delay of different messages. Yet, the objective in a distributed system is to arrive at a desirable global final state (legitimate state).

The set of all possible global states is divided into two disjoint sets: the error free or the *legitimate* states, and the erroneous states or the *illegitimate* states. The self-stabilization paradigm assumes that each node computes a predicate of its own local state and its neighbors' states (a predicate that would use other node states than the neighbor node states requires an underlying routing protocol to be implemented). When an inconsistent state is detected, a common approach in centralized systems is to force the system to a well defined state by a hardware reset or a power-cycle. This is often not an option in distributed systems that may cover a large geographical area. In a self-stabilizing system, when a node detects a local inconsistency, it takes a local action (a node can modify only its own states) in an attempt to correct the error. This node becomes locally consistent, but some of its neighbor nodes may become inconsistent (which were locally consistent before the action) and this ripple effect may propagate the entire system. A system state is globally legitimate when each node is locally legitimate or consistent. An algorithm is self-stabilizing if for any initial illegitimate state it reaches a consistent state after a finite number of node moves. A distributed system running a self-stabilizing algorithm is called a *self-stabilizing system*.

In general, a node is triggered into action when a lo-

cal inconsistency is detected; hence, in a legitimate system state no node may move. However, there are many services (provided by distributed systems) that require the system to change its state continually. A classical example is the token circulation for a distributed mutual exclusion algorithm. In a legitimate state, a node with a token selects one of its neighbors to pass on the token. If the system is in a illegitimate state, at least one node detects the error and takes corrective action. Thus, the error recovery procedure is integrated in the normal algorithm function. An algorithm is then self-stabilizing if (i) for any initial illegitimate state it reaches a legitimate state after a finite number of node moves, and (ii) for any legitimate state and for any move allowed by that state, the next state is a legitimate state.

A self-stabilizing system does not guarantee that the system is able to operate properly when a node continuously injects faults in the system (Byzantine fault) or when the communication errors occur so frequently that the new legitimate state cannot be reached until the new communication error. While the system services are unavailable when the self-stabilizing system is in an illegitimate state, the repair of a self stabilizing system is simple; once the offending equipment is removed or repaired the system provides its service after a reasonable time.

4.1 Requirements of Self-Stabilization

There exist several models for self-stabilization; we present here only the basic common concepts of these models. The *state* of a node is specified by its local variables. The system state is a vector of all local states of the participating nodes. We use $\mathcal{T}$ to denote the set of all possible system states. A system state is either *legitimate* or *illegitimate*. The precise specification of a legitimate state depends on the algorithm, but as a general rule, when the system is in a legitimate it has the property required by that application. To allow system recovery after transient faults, each node executes repeatedly a piece of code. This code consists of a set of rules:

begin
rule
⋮
rule
end

Each rule has the form:

(label) [guard]: <program>;

A guard is a boolean expression of the variables that the processor can read: its own variables and the variables of its neighbors. The program part of a rule is the description of the algorithm used to compute the new values for local variables. If the guard of a rule is true, that rule is called *enabled*. When at least one rule is enabled the node is *privileged*. An *execution* of a enabled rule is the determination of the new node state value using the algorithm described by the program part of the rule. A *move* of a node is the execution of a nondeterministically chosen enabled rule.

In other words, there is a relation $\mathcal{R} \subset \mathcal{T} \times \mathcal{T}$ such that if $(s_i, s_f) \in \mathcal{R}$, then (i) the states s_i, s_f differ by a single node x value, and (ii) if the system is in state s_i there is a enabled rule of node x such that after execution of the corresponding code, the system is in state s_f. A system evolution $\mathcal{E} = (s_i)_{i \in I}$, is a finite or infinite sequence of moves such that (i) if (s_i, s_{i+1}) is a consecutive pair of states in $\mathcal{E}$ then $(s_i, s_{i+1}) \in \mathcal{R}$, and (ii) if the system evolution has a finite number of states, s_f being the last one, then there is no state $s \in \mathcal{T}$ such that $(s_f, s) \in \mathcal{R}$.

To prove the correctness of a self-stabilizing algorithm, the conditions of closure and the convergence must be shown. The closure property means that when the system is in a legitimate state the next state is always a legitimate state. The convergence property means that for any state and for any sequence of possible moves, after a finite number of moves the system is in a legitimate state. As Gouda observes in his paper [5], self-stabilization can be in principle defined by a set, $\mathcal{T}$, a relation $\mathcal{R} \subset \mathcal{T} \times \mathcal{T}$ and a specification of the legitimate state set $\mathcal{L}$. Different classes of closure and convergence can be defined and general methods of proving the self-stabilization can be sketched.

One useful and elegant strategy to prove the correctness of self stabilizing algorithms is to use bounded monotonically decreasing functions defined on global system states [6]; some existing self-stabilizing algorithms are proved to be correct by defining a bounded function that is shown to decrease monotonically at every step [7]. Many self-stabilizing algorithms do not use this bounded function method since it is usually very very difficult to design such a function. In stead, they develop a different proof technique using induction on the number of nodes in the tree [8, 9].

4.2 Implementation Issues

The stabilizing algorithms achieve fault tolerance in a manner that is radically different from traditional fault tolerance in distributed systems. The paradigm allows us to abandon failure models and a bound on the number of failures. The theory is elegant, but how practical is the concept for implementation with present day technology? Here are some issues:

- The concept of the global state of the system requires a *common* time for all nodes. The physical time may be used as a common time but it may not be explicitly used by the component processes (drifts in local clocks, relativity etc.) The partial order relation (among events) generated by message exchange can not be uniquely extended to a total order relation. If there is no global

clock, global states can not be defined and legitimate states must defined locally, i.e. based on the local state of a node and the states of its neighboring nodes and on the partial order relation associated to sending and receiving messages. This greatly complicates the correctness proof of any stabilizing algorithm.

- A *move* is a complex operation; the state of the neighbors must be read, the guards must be evaluated and the associated code segment must be executed. Some models assume that a new move may not start until the previous move is completed, i.e. the moves are atomic. In a real distributed system the reading of a neighbor's state can be implemented in two ways: one option is to request every node to send its state to his neighbors periodically or whenever it changes its state. Each node caches the state received from its neighbors and moves according to the state cached in its memory. The other option is to use a query message: when a node needs to read its neighbor's state, it sends a query message and waits for reply. In both cases, when a node moves, it uses the cached states of the neighbors instead the real states of its neighbors. Since the states of the neighbors may have already been changed, the moves are not really atomic.

- To enforce the atomicity and the serializability of the moves Dijkstra, [1], has introduced the concept of central daemon. When multiple nodes are privileged, the central daemon arbitrarily selects one node to be active next. The concept of a central daemon is very much against the concept of a distributed system in that it serializes the moves and does not allow concurrent node executions. Proving correctness is easier for a serial execution, but there are many parallel executions that might be "equivalent" with a serial execution [10]. These executions should be allowed by an efficient move scheduler.

Different models of self-stabilization offer different prospects of cost effective implementation of the concept and it is not clear at this point which would win.

5 Classifications of Self-Stabilizing Systems

A *model* can be viewed as an *interface definition* and a set of assumptions (the algorithm designer can make) that define the behavior of the system. Unfortunately, there is no unifying model for the distributed system concept [11]. If some specific features of a system are not introduced in the model, the algorithms may be less efficient than they could possibly be; on the other hand, those features may not general enough to be present in all systems.

Self-stabilizing algorithms have been designed for two interprocess communications paradigms: shared memory and message passing. Since we are interested in self-stabilizing algorithms for distributed systems, we assume the message passing paradigm. It should be observed that a lower level protocol that sends and receives messages may transform a message passing model into a shared memory one.

5.1 Anonymous vs Id-based Networks

The concept of *node identity* is important in designing distributed algorithms. If each node has a unique hardwired *id*, the network is *id*-based; otherwise the network is *anonymous*. The anonymous network is a weaker model than an *id*-based network. For some problems there are no deterministic algorithms in anonymous networks [12]. The impossibility stems from the lack of deterministic symmetry breaking mechanisms without unique ids. In general, it is far more difficult to design algorithms for anonymous networks than for *id*-based networks. The *id*-based network is a more realistic model to design self-stabilizing algorithms, but a database of the used *id*'s is necessary to be maintained by a central authority; the addition of a new node requires a database search and the assignment of a new distinct *id*. This concept has been used in practice for a while (Ethernet addresses and IP addresses) and it has proved to be convenient. In principle, each node in an *id*-based network may have a global information of the topology and the state of all other nodes. Hence a local algorithm may be used to solve the problem. This scheme may be unacceptable since the information needed to update each node state is large and takes a considerable bandwidth. Besides, the system may respond too slowly to dynamic configuration changing.

5.2 Deterministic vs Probabilistic Algorithms

The self-stabilizing algorithms can also be divided into two classes: *deterministic* and *probabilistic* (randomized) algorithms. This criterion has nothing to do with the underlying distributed system but concerns with the algorithm design strategy. Randomization is normally used to break the symmetry in anonymous networks. Many randomized algorithms succeed with probability $1 - \epsilon$, $\epsilon > 0$ (the success is not certain). Besides, the random number generators used are actually pseudo random number generators and some undesirable correlations may appear between neighboring nodes.

5.3 Atomicity of the Operations

The first model proposed by Dijkstra assumes the existence of a central daemon that serializes the moves. A move

is an atomic action composed of two steps, reading the states of neighbors and modifying the local state. We also have the *restricted parallelism model*, where we have specific restrictions on the set of processors that may execute at each step, and *maximal parallelism model*, where all enabled processes may execute at each step. However, the moves are still assumed to be atomic. The execution models usually fall into into four categories: *serial model*, *synchronous model*, *synchronized distributed model* and *distributed model*. In serial model, only one node executes an atomic step at a time and the atomic step consists of reading the states of the neighbors and modifying its state if necessary. In the synchronous model, all nodes simultaneously execute an atomic step and each node in this model also sees the current state of its neighbors. The synchronized distributed model is like the synchronous model except that only a subset of all nodes synchronously execute the two sub-steps of a move. The distributed model is closest to a real life distributed system. In this model each atomic step is either a reading or a writing operation. That is, each node may read and record a state of one of its neighbors at a time and when all states are recorded, it can modify its state if necessary. By that time, some neighbor states may have already been changed.

To correct this problem two approaches have been followed. One approach, taken for example in [13] is to design algorithms assuming only read/write atomicity. The paper shows that mutual exclusion in nonuniform networks is possible using only read/write atomicity (tree construction algorithm is also presented). However, the proof of correctness is difficult for this kind of approach. Since it's easier to prove correctness for a serial model, some others [14] use the following strategy. Each process at a node is split into several processes: one of them is called *central process*, and the others are called *peripheral processes*. Each peripheral process corresponds to a neighbor and maintains a recorded state of the neighbor via atomic read operations. The central process maintains the state of the node and modifies the state by atomic write operation that depends on the states of the peripheral processes. We can show that the computation in the extended DAG formed by all processes corresponds to a computation in the serial model. Although this transformation works for some protocols, it does not guarantee that the transformed protocol performs what the original serial protocol does. To implement a self-stabilizing algorithm developed for a serial model, a runtime support environment may be introduced. The original algorithm is transformed using functions provided by the runtime environment such that the algorithm can be run in a distributed environment.

5.4 Self Stabilizing Data Link Protocols

Self-stabilizing systems relies on message passing to read the states of the neighbors. Though the error rate for some communication media may be very low [15], there are no error free communication links. A data packet sent on a communication link may be changed due to a communication error or may be dropped due to a buffer overflow. For the first case, error detecting codes may be used to recognize and eliminate the erroneous packets; we assume the arrived packets are error free and in order but some packets may have been dropped by the communication link. The purpose of a self-stabilizing data link protocol is to enable a node to send a sequence of messages without duplication, loss or miss-ordering. The alternating bit protocol is a fundamental data link protocol for data communication across error prone transmission media; but, this protocol is not self-stabilizing.

Authors in [16] describe a window sliding protocol where message sequence numbers are taken from a finite domain and where both message disorder and loss can be tolerated. Most existing window protocols achieve only one of these two goals. The protocol is based on a new method of acknowledgment, called block acknowledgment, where each acknowledgment message has two numbers m and n to acknowledge the reception of all data messages with sequence numbers ranging from m to n. Using this method of acknowledgment, the proposed protocol achieves the two goals while maintaining the same data transmission capability of the traditional window protocol.

It is worth mentioning that without time-out or periodic re-sending it is impossible to design a self-stabilizing algorithm. A self-stabilizing system must tolerate a transient error that modifies one of its variables at any time. If a transient fault modifies a state variable such that a process "think" that it sent a message but in fact the message was not sent, the process might wait for an acknowledgment that will never come.

6 Limitations of Self-Stabilizing Algorithms

One important limitation of the self-stabilizing protocols is that the system can not provide the required service during the time when it is not in a legitimate state. Many applications can tolerate short periods of unavailable service but if uninterrupted service is critical the self-stabilization can not be used. Besides, the time needed for the system to correct itself when started in an illegal state or after an error may be too long for some applications.

Other difficulties lie in the model used for the distributed system. Non-uniform self-stabilizing algorithms relies on a special machine which is a major drawback both from theoretical and practical point of view. Many self-stabilizing algorithms uses the central daemon concept. This is only a theoretical concept that needs a run environment system to be implemented. The runtime environment may trade the degree of parallelism of the algorithm execution for a simpler implementation.

Perhaps the most critical disadvantage of a self stabilizing system especially from implementation poit of view is that it is not possible from within the system to observe that a legitimate configuration has been reached; hence the processes are never aware of when their behavior has become reliable. Another area is complexity analysis; time-complexity analysis for self-stabilizing distributed algorithms is much more difficult than for sequential algorithms. Formally verification of a distributed algorithm is in general more difficult; the self-stabilizing algorithms are not an exception from this rule and though some work have been done, there is still much more to do in this area. Some of the stabilizing algorithms known to date seem simpler than their classical counterparts; this is at least partly due to the fact that the stabilizing algorithms are not "optimized" with respect to their complexity; the so called "advantage" may vanish when the stabilizing algorithms are made efficient.

Another weakness of the self-stabilization is that it is a global property. A transient fault in a legitimate state at one node may lead to a long corrective action that spans the entire system. It is more natural and desirable that an error at a one node remains confined to a small number of nodes. Some work in this area has been done [17] but there is not yet a "stability" definition of a self-stabilizing algorithm and no systematic procedure to achieve it.

7 Conclusion

The concept of stabilization was known in mathematics for quite a while; an iterative method to solve a linear system can start from any value and converges to the unique solution of the system (of course, certain conditions must be met, but though the initial guess can increase the numbers of steps for the desired accuracy, the final solution will be the same). In computer science the concept was introduced by Dijkstra in 1974; but only since late 80's its connection to fault tolerance has been apparent and researchers have started working in this area.

References

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

[2] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[3] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[4] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1994.

[5] MG Gouda. The triumph and tribulation of system stabilization. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972*, pages 1–18, 1995.

[6] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Inf. Processing Letters*, 29(2):39–42, 1988.

[7] S. T. Huang. Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, 15(3):563–573, July 1993.

[8] G. Antonoiu and P. K. Srimani. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers Mathematics and Applications*, 30(9):1–7, September 1995.

[9] G. Antonoiu and P. K. Srimani. A self-stabilizing leader election algorithm for tree graphs. *Journal of Parallel and Distributed Computing*, 34:227–232, 1996.

[10] P.A Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems*. Addison-Wesley, 1987.

[11] Sartaj Sahni and Venkat Thanvantri. Parallel computin: Performance metrics and models. Technical Report TR-008, University of Florida, Gainesville, 1996.

[12] D. Angluin. Local and global properties in networks of processors. In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[13] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[14] S. T. Huang, L. C. Wuu, and M. S. Tsai. Distributed execution model for self-stabilizing systems. In *ICDCS94 Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 432–439, 1994.

[15] Craig Partridge. *Gigabit Networking*. Adison-Wesley, 1994.

[16] G.M. Brown, M. G. Gouda, and R. E. Miller. Block acknowledgement: Redisigning the window protocol. Technical Report TR-89-02, University of Texas at Austin, 1989.

[17] B Awerbuch and G Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991.