THE DISTRIBUTED COMPUTING COLUMN

BY

MARIO MAVRONICOLAS

Department of Computer Science, University of Cyprus
75 Kallipoleos St., CY-1678 Nicosia, Cyprus
mavronic@cs.ucy.ac.cy

SELF-STABILIZATION FROM THEORY TO PRACTICE *

Olga Brukman Shlomi Dolev† Yinnon Haviv
Limor Lahiani Ronen Kat Elad M. Schiller
Nir Tzachar Reuven Yagel

Abstract

This paper advocates the use of self-stabilization as a provable property to achieve the goals of the self-* paradigms for systems, including availability, reliability, serviceability, disaster recovery and autonomic computing. Several recent results starting from hardware concerns, through the operating system, and ending in the applications are integrated: the self-stabilizing microprocessor [10, 11], with a self-stabilizing operating system [27, 28, 29, 30], the self-stabilizing file system [18], the self-stabilizing middleware [24, 25, 26, 19], the self-stabilization preserving compiler [12], the self-stabilizing autonomic recoverer for applications [2], and at last the combination of the local recovery and the self-stabilization properties to obtain self-organizing systems [21, 22].

Keywords: self-stabilization, availability reliability serviceability, disaster recovery, autonomic computing, safety critical systems.

*Partially supported by IBM faculty award, NSF grant 0098305, Lynne and William Frankel Center for Computer Sciences, and the Rita Altura trust chair in computer sciences. This paper summarizes recent works of the distributed computing and computer networks group in the Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel. Aspects of practical self-stabilizing infrastructure were studied and investigated together with co-authors.

† Contact author: Shlomi Dolev dolev@cs.bgu.ac.il.
1 Introduction

The interest in robust systems has increased dramatically during the last years. New terms such as self-healing, self-repair, autonomic computing, and automatic recovery are used to describe the current computing challenges, see e.g., [6, 7, 36, 52]. This is no coincidence, the design of (1) critical systems such as computer aircraft control, or (2) systems that should always be available such as web servers that support e-commerce, or (3) on-going core system components such as (possibly distributed) operating systems, must be verified to recover automatically, or in fact be self-stabilizing [7]. Otherwise, once the assumptions concerning the type and amount of failures are violated, the system may enter a state from which it will never recover. The self-stabilization property captures the desire to automatically recover no matter what happened in the past.

Fault tolerance using space and/or time redundancy have been extensively studied. These approaches differ from the one we propose since they do not cope with the recovery from an arbitrary state. Our approach compliments the traditional approaches: we suggest designing systems that can be started in an arbitrary state, so that even if the assumptions concerning its operation do not hold for a while, the system will recover.

Self-Stabilization. Self-stabilization is an elegant approach for designing fault tolerant systems [7]. A self-stabilizing system is designed to start in any possible configuration where processors, processes, communication links, communication buffers, etc. are in an arbitrary state. The idea is to explore the state space of the system, simply by considering any possible memory content and proving that from every such state the system eventually converges to the desired behavior. In other words, given a specification, we will say that an algorithm is a self-stabilizing algorithm for the given specification if: There exists a set of configurations, called safe configurations, that being in one of them ensures executions according to the specification, and this set of configurations is closed; being in a safe configuration ensures transition to a safe configuration. Starting from any configuration, a self-stabilizing algorithm must reach, in finite time, a safe configuration. Self-stabilizing algorithms can be combined in a way that the output of the first stabilizing algorithm serves the second stabilizing algorithm as an input, to form a single self-stabilizing algorithm. For example, self-stabilizing algorithms may assume correct behavior of the microprocessor and will stabilize after the (self-stabilizing) microprocessor converges to its desired behavior.

Arbitrary state is not due to self-stabilization. There is a common misunderstanding, attributing the fact that the system reached an arbitrary configuration (due to faults), to the ability of a system to recover from such an arbitrary configuration, the ability to stabilize. The assumption that the system should be able to be started in an arbitrary configuration may be regarded as too severe. Still
soft-error may essentially leave the system in an arbitrary configuration. Moreover, a design that copes with each fault scenario separately risks the possibility of one forgotten scenario that was not-considered. So in fact, it is simpler and more reliable to design a system to be self-stabilizing.

**Self-stabilization and fault masking live happily ever after.** Fault masking is a desired property since it guarantees some properties in the presence of on-going faults. Several well known examples for fault masking techniques, including the way to mask Byzantine faults, fail-stop faults and communication faults have been studied for decades. Does self-stabilization give up the fault masking property assuming an arbitrary configuration? There is a long history of research concerning self-stabilization in the presence of Byzantine faults, in the presence of fail-stops and in the presence of communication failures. The algorithms that self-stabilize in the presence of faults eventually reach a safe configuration due to their stabilization properties, and mask faults, as other masking algorithms do.

**Self-stabilizing systems do not have an initial state and therefore start arbitrarily.** Roughly speaking, any safe configuration can be used as an initial configuration avoiding any convergence when no convergence is needed. In some cases, the safe configuration encodes the result of the distributed algorithm, say, the spanning tree of the communication graph. It is good practice to define an initial configuration for such algorithms, say a state in which each node knows only its neighbors. Thus, the system will compute the spanning tree in the desired fashion and will mask faults as much as possible.

When the number of faults exceeds the fault containment capabilities, the system will not be lost, it will recover to function well in the presence of the faults, and again mask them.

When the number of faults exceeds the fault containment capabilities, the system will not be lost, it will recover to function well in the presence of the faults, and again mask them.

**Practical experience fairy tale.** During a demonstration of a research project supported by IBM to implement a self-stabilizing distributed file system [18], Julian Satran (best known for the iSCSI technology) easily managed to crash the “self-stabilizing” distributed system. We had to explain that the operating system that we used was the one that did not recover. Therefore, the need for all-level stabilization from the hardware to the applications is a must. The stabilization of one level is the base for the stabilization of the next level [8, 7]. This paper is an overview of the research on the ways to make different system levels stabilize, given the stabilization of lower levels.

**Paper organization.** We start with hardware, describing in the next section a way to design a self-stabilizing microprocessor. We continue in Section 3 with a description of techniques to achieve self-stabilizing operating systems. Then we describe a compiler that preserves the stabilization property of a self-stabilizing pro-
The Bulletin of the EATCS

gram (Section 4). Self-stabilizing distributed file systems and middleware (group communication and replicated state machines) are described in Section 5 and Section 6, respectively. The way to ensure recovery of applications is described in Section 7. Issues concerning locality of recovery are addressed in Section 8. Conclusions appear in Section 9.

2 Self-Stabilizing Microprocessor

The basic assumption that the self-stabilizing algorithm designer uses is that the program that implements the algorithm is executed by the processor(s). This assumption should be examined. In fact, it is clear that current designs of microprocessors do not have the automatic recovery property, and hence the self-stabilizing program that the microprocessor should execute, will not be executed, and the system will obviously, not stabilize. For example, consider a space-shuttle controller that was transferred to the halt state due to a soft error, the microprocessor will not stabilize to start fetching commands and obviously, will not execute its control programs. Another example is the possibility that unexpected input output occurs, causing the processor to enter the halt state, as described in the processor manual [37]: “In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is executed, the process shuts down due to lack of stack space”. A microprocessor is self-stabilizing if and only if every behavior that starts in any configuration, reaches in a finite number of pulses a safe configuration after which its behavior is in the set of legal-behaviors.

The hardware designers are coping with soft errors, also called single event upset (seu). Soft errors are voltage changes caused by cosmic rays (or other disturbances); they can change the output value of a logical gate in the digital circuit. This is a bold example of the need for self-stabilization, where any combination of bit values is considered, in particular combinations reached due to soft errors.

We require that a safe configuration is reached, and then the microprocessor will repeatedly execute fetch-decode-execute, where each machine code command is executed according to the manufacturer specifications. The manufacturer manual defines the specification of each machine code command.

One can ensure that a fetch-decode-execute sequence is eventually executed by using an upper bound on the number of clock pulses that may pass in between every two successive executions of the fetch-decode-execute sequence. We assume that every processor repeatedly executes the fetch-decode-execute sequence when it is started in a predefined state (e.g., the initial state defined by the manufacturer). Thus, one can use a watchdog circuit that will detect the situation in which the processor has not executed the fetch-decode-execute sequence for a pe-
The watchdog circuit itself may experience soft-errors. Fortunately, it is possible to ensure that the watchdog circuit is self-stabilizing. One can implement the watchdog as a counter that is decremented in every clock pulse, using the exact number of bits needed to count the upper bound on the number of pulses in between two successive fetches. We assume that the watchdog counter can be initialized in any possible state (due to a soft error), causing in the worst case, an immature reset of the microprocessor. A self-stabilizing microprocessor recovers following the occurrence of faults that derive its state to an arbitrary state. During its automatic recovery the processor converges to a legal behavior (for example, a behavior that can be achieved from its predefined initial configuration). Naturally, the influence of soft-errors are not masked (immediately after an arbitrary state is reached, and) during the convergence period. Thus, (self-stabilizing) programs that the microprocessor executes may lose their consistency and will have to start a convergence period themselves.

3 Self-Stabilizing Operating System

The robustness of software systems is, in some cases, more important than its performance [5, 9, 31, 34, 39, 40, 45, 52]. The experience with existing operating systems, and in fact with every large on-going software package, is that the system almost has its own independent behavior. The behavior is tuned up and modified by system administrators who constantly and continuously monitor it. The system is usually too complicated to monitor. The system administrators use human behavior and character terms, as if the system is an entity with its own will, to refer to its input output scenarios. The importance of a design that is based on well understood theoretical paradigms, and that can give us control over the resulting system, cannot be exaggerated. In particular in the case of the operating system, robustness is a must, as the operating system forms a basic infrastructure in almost every computing system.

Designing a robust operating system is a complicated and challenging task. The system designer makes several probabilistic assumptions that may not hold if an execution is long enough. For example, soft errors [41, 48, 49] may cause an arbitrary change in memory bits that the error correcting schemes used will not identify. Another example, is that the communication between the system components can be made reliable, say by the use of error correcting codes. However, this assumption is also based on probability (where the life length of the system is a parameter). Once the probabilistic assumptions do not hold, the designer can
no longer guarantee the desired behavior. In [27], we propose several approaches for designing automatic recovering operating systems that are based on the self-stabilization paradigm. A self-stabilizing algorithm/system makes the obvious assumption that it is executed. This assumption is not simple to achieve, since both the microprocessor [10] and the operating system should be self-stabilizing ensuring that eventually the (self-stabilizing) application programs are executed. An elegant composition technique of self-stabilizing algorithms [7] is used to show that once the underlying microprocessor stabilizes, the self-stabilizing operating system (which can be started in an arbitrary state) stabilizes, and then the self-stabilizing algorithms that implement the application, stabilize.

Operating systems are essential parts of most computer systems [54]. The operating system manages the hardware resources, and forms an abstract (virtual) machine that is convenient to program by higher level applications developers.

**Operating systems are not self-stabilizing.** The operating system is typically the largest software constantly executed by a processor. Fault free software is a hard task to achieve (see, e.g., [2]). When the operating system is designed for a specialized restricted task, such as the TinyOS [35], formal methods of verification may assist in achieving fault free software. Still, the resulting system may fail due to a transient fault (e.g., a soft-error). Therefore, a self-stabilizing approach is a must in such basic and on-going components such as operating systems. Apparently the current design of operating systems does not take into account the automatic recovery property of the system as a basic requirement. For example, there are processors (e.g., Intel’s Pentium) which cannot support an implementation of a self-stabilizing operating system. These processors are designed to support external interrupts. One class of these interrupts is the Non-Maskable Interrupts (NMI). While the operating system is handling an NMI, the processor is not reacting to additional interrupts. To enable additional interrupts the iret machine command must be executed [38]. Self-stabilizing systems must be able to start from any initial state, in particular a state in which interrupts are masked, and therefore should either repeatedly execute irets or should not use interrupts. In fact it is possible to mask the Non-Maskable Interrupt (NMI). The NMI may be masked by i/o memory instructions [50]. Also, when in a system management mode (say, due to transient faults) the NMI is disabled [38] (Vol. 3, Sec.13.7), the processor has masked NMI states. Note that the NMI example is not based on corrupting the code of the algorithm but only on changing its (soft) state, namely altering the content of the variables.

In the Byzantine model, faulty processors may exhibit malicious behavior (representing a worst case change in the program the processor executes). The fault model used for self-stabilizing system assumes that (at least a portion of) the processors in the system execute a correct code (e.g., [23]). The requirement for code correctness (of at least two thirds, or so, of the processes) is obvious, even
in the case of Byzantine faults [44]. If all the processors are Byzantine then the system can exhibit arbitrary behavior. The requirement concerning the fault-free programs can be achieved by designing the system to repeatedly access a fixed read only memory device that reloads the executable code from, say, a compact disk. An additional prominent example of the lack of stabilization with regard to the processor/operating system interaction design, is related to the interrupt descriptor table (iDT). The (iDT) contains pointers to the different operating system routines called when interrupts occur, such as the timer/clock interrupt. The Pentium processor has a register pointing to this table (iDR). A transient fault that causes a value change of this register may disable the entire interrupt capability, and even cause the processor to execute an infinite loop. A similar scenario is possible when the interrupt table itself is corrupted.

**Approaches for self-stabilizing operating systems.** One approach in designing a self-stabilizing operating system is to consider an existing operating system (e.g., Microsoft Windows, Linux) as a black-box and add components to monitor its activity and take actions accordingly, so that automatic recovery is achieved. We call this approach the black-box based approach. The other extreme approach is to write a self-stabilizing operating system from scratch. We call this approach the tailored solution approach. In [27] we present three design solutions in the scale of the black-box to the tailored solutions. The first, simplest technique we propose for the automatic recovery of an operating system, is based on repeatedly reinstalling the operating system and then re-executing it. The second technique, is to repeatedly reinstall only the executable portion, monitoring the state of the operating system, and assigning a legitimate state whenever required. Then we present a tailored, very tiny, self-stabilizing design for components of an operating system. Memory management, device drivers, and virtualization aspects of the operating system are addressed in [28, 29, 30].

4 **Stabilization Preserving Compiler**

In order to prove that a program is self-stabilizing, one needs to show that the program converges to its legal behavior from an arbitrary state. Therefore, programmers of stabilizing systems typically assume two properties from the languages they use for describing their algorithms in: (a) The language enables the programmer to explicitly define the state space for the program. (b) The language semantics is expressed in big steps. Roughly speaking, the granularity of steps in the scope of program semantics [51] resembles the granularity issue of atomic steps in distributed computing [42, 8]. This requirement provides the programmer with a higher level of abstraction and enables the programmer to abstract the states that are used for implementing a step.
Languages that are based on state machines are natural candidates for writing and arguing about self-stabilizing programs. The reason for such a choice is that the stabilization arguments consider any possible state, namely any content of variables, and prove convergence to a subset of safe states from which the run behaves as desired. Guarded commands [6], Input Output Automata [43], and Abstract State Machine (ASM) [32] are three bold examples for state based programming languages. We note that a pseudo code description of self-stabilizing algorithms stated in a different format can be transferred to the above representation, integrating a mechanism that mimics a program counter into the guards. We choose a variant of ASM as the high-level programming language, that is inspired by the guarded commands language. We note that our results are applicable to other choices of programming languages as well.

Motivating example. We now present a simple example for motivating the self-stabilization preserving compiler. Figure 1 consists of an example of JVM code produced for the statement for $i=0$ to 9 do $f(i)$. The commands used are analogous to the ones presented in [53]. Line 2 initiates the local variable $i$ to zero. Lines 4 – 12 contain the code for one iteration. Line 4 – 7 call $f(i)$. Line 8 increases $i$ by one. The end condition for the loop is checked in lines 9 – 11. Line 9 makes a copy of $i$, line 10 pushes the value 10 to the stack and line 11 pops the two recently pushed values from the stack and branches out of the loop code if they are equal. Finally, if $i$ is not equal to 10, then the loop execution is repeated due to the unconditional jump in line 12.

The programmer assumes that $f(i)$ will be executed ten times at most (even when started at an arbitrary state). However, in case the value of $i$ is equal to 11 (or, e.g., $-2^{30}$) and the program counter is inside the loop (e.g., line 4), the
execution is practically similar to an infinite loop. The reason for the corruption of $i$ may be a soft error in the memory or other transient fault, see e.g., [41].

Moreover, even when a range check of the loop variable is added to the code of the loop, an existing optimizer may consider the additional check as a redundant code and will remove it. Thus, we have to examine and define the requirements for the compiler and the optimizer.

**The distributed case.** In order to enable the compilation of existing distributed self-stabilizing algorithms into a machine code, we have also examined the Distributed Abstract State Machine (DASM) language. Part of our contribution is a design that extends the DASM primitives in a way that supports refined operations, in the level of read or write operations to safe, regular or atomic registers. The new primitives are mapped directly onto the underlined hardware (the exact communication register used). The programmer is responsible for the correct operation (and proof) of the program using the refined communication devices.

We note that in [46] a language and a compiler for protocols are described where some stabilization properties are preserved. Our work complements the work of [46] in the sense that we present a compiler that ensures convergence of each processor (either in a stand-alone system or as part of a distributed system) when started in an arbitrary state of the compiled program. We take into account refined states produced by the compiler; i.e., states used for implementing the transitions defined by the high level language.

We show that the usage of an off-the-shelf (ASM) compiler with a consistency check, in which there is no given correspondence function between the original ASM state and the refined machine state, is a hard task. In fact, we show that given a refined machine state it is NP-hard to find whether a corresponding state of the ASM program exists. We then turn to the positive side.

Given a self-stabilizing program written in ASM we would like to produce a machine code that eventually has the same input output relation as the original ASM program. Roughly speaking, we use the fact that ASM execution is composed of moves, each corresponding to the evaluation of a rule and the updating of the current state. The syntax of ASM allows only bounded loops within a rule, which implies that each move is executed in a finite number of steps. This enables our compiler to create a code that efficiently checks the validity of the state between consecutive moves. We prove that the machine code portion produced by our compiler for an ASM rule can be executed from any point (program counter and arbitrary microprocessor state) and still terminate. The machine code produced is a sequence of portions, each of which reflects a rule. Hence, no matter

---

1 Roughly speaking, a read operation from a safe register may return an arbitrary value when read and write operations overlap. A regular register may arbitrarily return a value of an overlapping write or the value that was in the register prior to the read. Lastly, an atomic register yields a total order of reads and writes, [42].
where we start the execution of the machine code, the execution of a rule terminates and an additional portion which checks the consistency of the current state is executed. The consistency check verifies that the variables that correspond to the original ASM program are in their value range and that the data structures used by the execution environment e.g., the scheduler of the ASM rules are also consistent. In [12] we present detailed techniques for (a) Examination of programming languages for writing self-stabilizing algorithms and a choice of automata like description, namely ASM. (b) The extension of ASM to refined ASM where each rule of the ASM includes one communication operation (e.g., read, write, send, receive) at most and a full correspondence of the communication operation to the given hardware; for example, a read that overlaps a write from/to a safe register returns an arbitrary value. (c) Definitions of the requirements for a self-stabilization preserving compiler. (d) Design and implementation of a self-stabilization preserving compiler.

Figure 2(a) describes the way the code generated from our compiler is structured. The code for evaluating the conditions of each of the transition rules is placed following the scheduling code. The code compiled from the body of each transition rule is placed just below the scheduling code. Executing the code compiled from the body of a transition results in an update set, which is applied in the \texttt{apply-update-set} section. The \texttt{enforce-invariants} section contains a code for ensuring that a state \(s_l\) of the produced code corresponding to a state \(s_h\) of the original ASM program is reached.

In particular, the enforce invariant section enforces the variables to be in the required range. In the example of Figure 2 the loop index is forced to be less than 11. Executing this code section from its first line ensures that \(s_l\) is a valid state - namely, that there is a state \(s_h\) for which the set of traces of the executions starting from \(s_l\) is equal to the set of traces of the runs that start in \(s_h\). Thus, from this point on, the loop behavior is the same as the behavior of the original ASM.

The basic idea for designing the compiled code is to make sure that regardless of the current state, the program counter will reach the beginning of the \texttt{enforce-invariants} section, after which the state satisfies a validity condition ensuring that the execution corresponds to the ASM execution. Moreover, we design our compiler so that the \texttt{enforce-invariants} section is reached within \(2 \cdot R_{\text{max}}\) steps, where \(R_{\text{max}}\) is the maximal number of steps used for implementing a move of the original ASM. We state two conditions on the transformation made by the compiler which together ensure that our compiler does preserve the self-stabilization property: 

\textbf{C1}. Starting in an arbitrary state of \(PR_l\) the system reaches within at most \(2 \cdot R_{\text{max}}\) steps a state in which the first line of \(PR_l\) is executed, namely the first line of the enforce-invariant section. 

\textbf{C2}. Executing the \texttt{enforce-invariants} section from its first line results in a valid state. Moreover, if the state is already valid, executing the section will not change the state.
5 Self-Stabilizing Distributed Storage

Distributed data storage has become a key component in today’s computer systems. Large local system storage is replaced by distributed and online storage systems. One type of such a storage system is the distributed file system, which resolves issues of (data size) volume, performance, availability, and reliability. File systems are required to operate continuously, thus required to be self-stabilizing.

In [18] we proposed the first self-stabilizing distributed file system with a scalable non-centralized distributed storage access system design, where storage, cache, and control is distributed over cooperating workstations. We base our design on a self-stabilizing maintenance of a distributed replica tree for each volume. The tree relates servers that maintain volumes, in order to achieve volume replicas consistency. We view the choice of a distributed spanning tree as the most efficient distributed structure (and a key feature in our design) which relates and connects replicas.

We present novel and practical techniques for TCP/IP networks that achieve (1) a self-stabilizing leader election, (2) a self-stabilizing tree construction (with as short as possible edges), and (3) a scheme to combine a backbone tree with a peripheral tree. The communication is local; each message is forwarded through only a limited number of routers. In addition we study the aspects that should be considered during the design of a file system in the perspective of a designer...
of self-stabilizing systems. Namely, we do not assume initial replica/caches consistency. We believe that the new proposed approaches that include, event driven checking (read/write of a location triggers a consistency check of this location) and automatic conflict resolution (by using the largest supporting set of replicas or another function), enrich the techniques that are used by file system designers. Conflict resolutions, file locking and updates are provided by the use of a self-stabilizing synchronizer. Our prototype implementation srFS of the self-stabilizing file system [18, 55] is important evidence, demonstrating the usefulness of the self-stabilization paradigm for real systems, and the need for all-level stabilization.

6 Self-Stabilizing Middleware

Middleware services have become a cornerstone for developing complex network and distributed applications. The design of a self-stabilizing middleware is a key component toward reliable large scale applications. One such example is a group communication service; another is a distributed replicated state machine that provides computing redundancy and reliability.

The group communication infrastructure is useful for numerous applications, starting in video, audio, multimedia virtual conferences, and including safety critical tasks that require at most one resource transaction in every given time. Online on-going systems are prone to unexpected state transitions due to transient faults, thus, it is important to design such systems to recover automatically from any possible state.

The key features of a group communication facility are (1) indicating to each node of the distributed system which “group” it belongs to, that is, with which other nodes it can currently communicate, and (2) letting nodes within a group communicate with each other in an ordered and reliable manner. The first feature is called a group membership facility, while the second feature consists of various kinds of broadcasts and multicasts.

Many fault-tolerant algorithms do not consider the case of faults that cause a temporary violation of the failure assumptions made by the algorithm designer. For example, most of the algorithms that are designed to cope with Byzantine faults do not recover if more than one-third of the processors temporarily experience a fault and then continue to execute their program starting from the state following the fault. Since a group communication layer is a “middleware” for a distributed system, it is designed to execute forever, like an operating system. Thus it is highly unlikely that it will never experience a transient failure, especially in highly dynamic wireless mobile networks.

We have proposed self-stabilizing group communication services for dynamic
undirected/directed networks in [24, 26, 25]. In particular, the group communication service for ad hoc networks, presented in [26], uses random walk to cope with the chaotic environment and achieves the powerful primitives of group communication.

**Self-Stabilizing asynchronous replicated state machine.** One of the most well known methods for improving the reliability of a computing system is adding redundancy. In such an application design, the same application step is executed on multiple processors, and for each step the processors perform a ballot in order to decide on the result of the application step. In such a system, a fault in less than the majority of the processors is easily overcome by using the results of the other processors. An example for such an implementation is Triple Modular Redundancy (TMR) in which every computing step is performed by three machines and the output of each step is decided by a majority vote. One drawback of such methods is the requirement for tight synchronization between the machines, another is the possibility for the majority mechanism or its output to experience a fault. In [19] we show a design for self-stabilizing asynchronous N Modular Redundancy – a self-stabilizing replicated state machine that overcomes the need for tight synchronization and can execute for a practically infinite time.

The replicated state machine is an asynchronous distributed state machine, in which every process independently executes steps. The output of each step is associated with an epoch number. For each epoch, the processors execute a consensus instance and agree on the common output. The consensus instance is executed in asynchronous rounds until the system adopts a common value. The replicated state machine requires a bounded amount of memory and employs a self-stabilizing wait-free reset to maintain its consistency. We show in [19] a complete self-stabilizing implementation of a replicated state-machine, a consensus algorithm that is based on the rotating coordinator concept and a semi-synchronous failure detector.

**Self-Stabilizing virtual automata for ad-hoc networks.** Ad-hoc networks have chaotic nature entities (sensors, mobile computers, RFID tags, etc.) in the network may rapidly change location and spontaneously join and leave the network. To cope with this chaotic behavior we have suggested locating virtual automata in geographic regions, possibly by tilling the space and locating in each tile an automaton. An entity that enters a tile will participate in simulating the automata of the tile, receiving from a resident entity a replica of the automata state and the inputs to the automata thereafter. When the entity leaves a tile it erases the state of the automata of the tile. Thus, we are able to form a fixed virtual infrastructure in the ad-hoc network. One application for such automata is to form a virtual traffic light in a junction. Assuming cars may communicate, whenever a car arrives to a junction and there is no other car in the junction, the traffic light virtual automaton is initialized to be green in the direction of the arriving car. Upon an arrival of
another car the virtual traffic light counts time and then changes the green light in, say, a round robin manner. The virtual automata for an ad-hoc network has been introduced and investigated in [13, 14, 15, 16]. Self-stabilizing implementation of timed virtual automata is presented in [17]. The self-stabilization property is important to ensure that the entities in a tile eventually have the same replica, get the same inputs and send outputs as required.

7 Self-Stabilizing Autonomic Recoverer of Black Box Applications

Computers that manage critical systems make the issues of correctness and faultless flow of long-lived and continuously running programs extremely important. Complex systems cannot be fully verified since verification of large systems may require an unreasonable amount of time and space. Such systems usually contain flaws – software bugs. The software industry tests software products extensively to eliminate flaws as much as possible. Usually software is tested by executing a large, but bounded and non-exhaustive set of input/output scenarios starting from a predefined initial state. Faulty and undesired behavior may occur due to scenarios that were not tested and may be hard to reproduce.

Consumers of critical systems would like to have a warranty that the system will operate as it should. It is not enough to be reimbursed when the software does not operate properly. A software system for control of a nuclear reactor that malfunctions may cause damage that is not on the same scale with the software cost.

Our approach for the black-box case. A general yet practical framework and paradigm, based on a theoretical foundation, for the monitoring and recovery of systems is suggested in [2]. Using theory tools leads to the precise definition of desired properties and, hence, provable design and performance. We propose a generic distributed architecture for any type of system.

A software package is assumed to be eventually Byzantine: the software executes correctly for some time (due to quality testing), but eventually some faults are likely to occur. The package is a black box, we cannot access its variables, but do have access to its structure, i.e., a graph describing the package component dependencies. The dependencies graph is a directed acyclic graph (DAG). We do not allow cyclic dependencies, as they imply a cyclic infinite recovery scenario. Package requirements are categorized into safety properties (what should not be violated) and liveness properties (what should eventually happen). Each property has a respective recovery action – an action that should be executed if the property is not respected by the package. The properties describe a desired behavior of each
component of the package and a desired behavior of subsystems (a subsystem is a

group of related package components).

Monitoring design. We design a middleware on top of which an unreliable black
box package is executed. The middleware consists of a collection of monitors

that monitor properties of package components and subsystems. Each component
and subsystem has a dedicated monitoring process. A component or subsystem
monitors initiate recovery upon property violation in a hierarchical manner. The
hierarchical approach enables the recovery of only faulty parts, while other subsys-
tems are unaffected.

The monitors have to be self-stabilizing and continuously executing, other-
wise we cannot guarantee that the package will eventually execute correctly. We
assume existence of a self-stabilizing stack (a self-stabilizing kernel [10], a self-
stabilizing operating system [27], and a self-stabilization preserving compiler
[12]). In order to ensure the middleware existence and execution, we introduce
a kernel resident process in a self-stabilizing operating system ensuring that the
monitors exist and execute their correct code. The package is a black box, so we
monitor its state by recording the input-output sequence of each process and of
each subsystem by using the new monitoring techniques (e.g., runtime reflection),
or by means of recording the system calls to the operating system (e.g., using
Unix/Linux strace command).

The monitoring of safety property is quite simple, ensuring that the safety
property is never violated. Violation of a safety property causes an immediate call
to a recovery action. The monitoring of a liveness property requires some sophis-
tication. If the liveness property is not respected, this does not imply that it will
not be respected eventually. Thus, we accumulate component states into a history
log. If the component is in the same state twice during the execution, such that the
liveness property was not respected during that portion of the execution, then the
component is in a livelock (assuming deterministic behavior of the component)
and will not respect the liveness property. In this case the respective recovery ac-
tion should be executed. The recovery action arsenal includes restarting a process
or a collection of processes, rolling back to a safe state, rescheduling, waiting, etc.

We propose a generic distributed architecture for any type of system, as op-
posed to stand alone system research projects [3, 4]. Rich system hierarchy DAG
structure is proposed instead of a tree, or even no hierarchy at all [3, 4, 34]. We
include progress monitoring and recovery of subsystems by using safety and live-
ness predicates and recovery actions, as opposed to restarting only when the pro-
cess does not send a heartbeat [3, 4].

Recovery oriented computing. Writing a perfectly correct code is a challenging
and nearly impossible task. In [1] we suggest the recovery oriented programming
paradigm in order to cope with eventual Byzantine programs. The program speci-
fication composer enforces the program specifications (for both the safety and the
liveness properties) in run time using predicates over input and output variables. The component programmer will use these variables in the program implementation. We suggest using the “sand-box” approach in which every instruction of the program that changes a specification variable, is executed first with temporary variables in order to avoid execution of an instruction that violates the specifications. In addition, external monitoring is used for coping with transient faults and for ensuring convergence to a legal state. The implementation of these ideas includes the definition of new instructions in the programming language with the purpose of allowing the addition of predicates and recovery actions. We suggest a design for a tool that extends the Java programming language. In addition, we provide a correctness proof scheme for proving that the code combined with the predicates and the recovery actions are self-stabilizing, under the restartability assumption. The combined code eventually fulfills the original code specifications.

8 Self-Organization

Distributed systems must continue to operate in spite of local errors or corruptions; following a local fault in some nodes, other nodes may still need to provide service correctly, and the system cannot wait the entire duration of the stabilization time to guarantee correctness. To capture this property of distributed system, we introduce the concept of self-organization. We say that an algorithm is self-organizing if (1) the algorithm is self-stabilizing (2) the stabilization time of the algorithm, \( s(n) \) (where \( n \) is the size of the system, e.g., the number of nodes) is in \( o(n) \) and (3) after the algorithm stabilizes, the stabilization time following a local fault, \( d(n) \), is in \( o(s(n)) \).

Our definition of self-organization implies several interesting properties for self-organizing algorithms; such algorithms are required to be local, since no global operations involving the entire system may be performed. Moreover, self-organizing algorithms are robust to dynamic topology changes, when considering communication systems. Furthermore, since stabilizing after a single fault must not require global stabilization, the fault may only impact on a small, confined area of the system, leaving parts which are farther apart (\( o(s(n)) \) away) unaffected. After defining the notion of self-organization, we present several self-organizing algorithms. In [21] we present a self-organizing hierarchy construction algorithm for distributed systems, which utilizes a self-organizing maximal-dominating set algorithm, a self-organizing and snap-stabilizing snapshot algorithm and a local synchronizer.

In [22] we consider distributed constructions of expander graphs. Expander graphs are ideal as a basis for communication networks, as they are highly connected, sparse and robust to link failures. We present self-organizing algorithm
which, given an expander graph as a communication network, can reduce the number of edges by a constant factor, preserving the expansion of the graph (up to the same factor). A distributed monitoring protocol to ensure that the expansion of the graph is indeed satisfactory is presented.

9 Concluding Remarks

The usage and usefulness of self-stabilizing systems in critical and remote systems cannot be over emphasized. For example, entire years of work may be lost when the operating system of an expensive complicated device (e.g., spaceship) reaches an arbitrary state (say, due to soft errors) and be lost forever (say, on Mars). The controllers of a critical facility (e.g., nuclear reactor) may experience unexpected faults (e.g., electrical spikes) that will cause it to reach an unexpected state, which may lead to harmful results. The self-stabilization theory is mature enough to become an integral part of practice.

References


Intel Pentium manual.


