JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 1, June 2004

Self Stabilization

Shlomi Dolev[*]
Ben-Gurion University of the Negev, Beer-Sheva, Israel 84105

FAULT tolerance and reliability are important issues for flight vehicles such as aircraft, space-shuttles, and satellites. A self-stabilizing system recovers automatically following disturbances that force the system to an arbitrary state. The self-stabilization concept is an essential property of any autonomous control/computing system.

Important branches of distributed computing theory were initiated because of the need for fault-tolerance of aircraft computing devices. The Byzantine fault model, for example, was a creation of the NASA SIFT project of more than a couple of decades ago.[9] The Byzantine fault model[8] is an elegant abstraction of faults where it is assumed that faulty parts behave as adversaries. The idea is to use redundancy—for instance, in the number of processors—in order to overcome the behavior of faulty processors. This line of thinking fits the common practice in engineering in which the design of critical parts is done independently by several design teams to make sure that there is no mistake in the calculations. Analogously, in the Byzantine fault model, some processors are simultaneously computing the same calculations for implementing a robust flight controller; thus the flight controller can function well in spite of the faulty behavior of several of the processors. Faulty behavior cannot be anticipated, the most severe behavior is therefore assumed—one that is reminiscent of the behavior of an adversary in the Byzantine court, in which backstabbing was common.

A formal impossibility proof shows that, in order to ensure the correct behavior of the system, less than one-third of the processors may be of the Byzantine type. The task examined is the agreement, or consensus, for which processors need to decide on the same output, which is the input of one of the processors. The decision can be viewed as choosing the common result among the results of the design teams in the engineering example. The intuition behind the impossibility result is as follows: assume you have met two persons, Alice and Bob, one of which is honest while the other is not. You may try to decide what to do by speaking with each of them. Because you do not know which of the two is honest, you have to find this out. You may try a direct question to Alice asking who among them is not honest; Alice will answer Bob, and Bob, if asked, will obviously answer Alice. Each of them may also describe the conversations he/she had with the other, knowing that no one listened to the communication between them. This symmetry in the weights of the answers of Alice and Bob make it impossible to decide. It is possible to formally prove that agreement can be achieved if, and only if, less than one-third of the processors are Byzantine (e.g. Ref. 7).

Systems that tolerate Byzantine faults are designed for flight devices, which need to be extremely robust. In such a device, the assumptions made for reaching agreement can be violated: Is it reasonable to assume that, during *any* period of the execution, less than one-third of the processors are faulty? What happens if, for a short period, more than a third are faulty, or perhaps experience a weaker fault than a Byzantine fault (say, caused by a transient electric spark)? What happens if messages sent by non-faulty processors are lost in one instant of time?

Seven years prior to the introduction of the Byzantine fault model, Edsger W. Dijkstra suggested an important fundamental fault tolerance property called *self-stabilization*;[2, 3] that is, to design the system as if there were no (yesterday) past history—a system that can be started in any possible state of its state space. It would therefore not be assumed that consistency was maintained from the fixed initial state by always executing steps according to the program of the processors. Self-stabilizing systems thus overcome transient faults. Temporary violations of the assumptions made by the algorithm designer can be viewed as leaving the system in an arbitrary initial state from which the system resumes. Self-stabilizing systems work correctly when started in any initial system state. Thus, even if the system loses its consistency due to an unexpected temporary violation of the assumptions made, it converges to legal behavior once the assumptions start to hold again.

Self-stabilization is a strong fault tolerance property for systems; it ensures automatic recovery once faults stop occurring. A self-stabilizing system is designed to start in any possible state where its components—e.g., processors, processes, communication links, communication buffers—are in an arbitrary state; i.e., arbitrary variable values,

[*]Associate Professor, Department of Computer Science; JACIC Associate Editor, AIAA Member.

arbitrary program counter. The designer assumes that at least some of the programs are executed and proves that, under his/her assumptions of program execution, the system converges. Hence, there is no need to anticipate every fault scenario and handle each scenario explicitly and independently. Designing a self-stabilizing system is therefore safer (there are no forgotten scenarios) and simpler (there is no detailed recovery procedure for every scenario). Once it is proven that the system stabilizes, the designer is standing on a solid ground: there is no need to worry whether unexpected fault combinations will occur (see e.g. Ref. 10).

The research activity for designing self-stabilizing systems is very active e.g. Refs. 2, 11, 12. Other research projects advocate the need for automatic recovery, and self-* properties such as: self-healing, self-adaptive, self-organized.[1,6] The focus of computer industry leaders is in autonomic computing.[*] These are indications for the importance of knowing the self-stabilization paradigm and the techniques used for achieving the self-stabilization property.

Dijkstra's first self-stabilizing algorithm is the best example of the paradigm. Consider a ring with n processors $p_1, p_2, \cdots, p_n$ such that every processor can read the state of the previous processor; in particular, p_1 reads the state of p_n. A desired behavior of the system is one in which there is always exactly one processor that can change its state. One may propose an algorithm in which each processor has $k = 2$ states denoted by 0 and 1, and each processor except p_1 copies the state of the previous processor, p_1 will change a state whenever the state of p_n is identical to its state; p_1 will increment its state value by 1 modulo k.

When the system is started in a configuration in which the states of the processors are all zeroes, denoted $\{0, 0, 0, \cdots, 0\}$, then p_1 is the only processor that can change a state and a system configuration $\{1, 0, 0, \cdots, 0\}$ is reached. Then p_2 is the only processor that can change a state to reach $\{1, 1, 0, \cdots, 0\}$. Similarly the next processors are privileged to change state one by one until the configuration $\{1, 1, 1, \cdots, 1\}$ is reached where p_1 is the one to change its state to 0. Unfortunately, this simple algorithm is not self-stabilizing. Consider a ring with four processors that is started in the following configuration $\{0, 1, 0, 0\}$ where p_1, p_2, and p_3 can change a state. An execution in which p_3, p_2, p_1, p_4 take steps in this order results in $\{1, 0, 1, 1\}$ which is the same as the first configuration up to replacing 0 by 1. Thus, there is an infinite execution that never converges to the desired behavior.

On the other hand, if each processor had $k = n + 1$ states, then the algorithm will stabilize. Stabilization is proved by first observing that in any arbitrary configuration at least one state is missing; for example, in the configuration $\{2, 0, 3, 4\}$ the state 1 is missing. We then convince ourselves that in any infinite execution p_1 changes state infinitely often (assuming that, from some stage p_1 does not change a state and showing that eventually every processor including p_n holds the state of p_1 and that thereafter p_1 must change a state). And last, we see that p_1 changes states in a round robin fashion that ensures that p_1 reaches the state that is missing in the arbitrary initial configuration, thus enforcing a subsequent copy of this state by all the processors to reach a configuration in which all the processors hold the same state. Once all the processors hold the same state, the convergence phase is over and the closure of legal states is guaranteed analogously to the way the first non-stabilizing algorithm preserves consistency.

The above may be viewed as a toy example, but the paradigm is far from being a toy. Self-stabilization is a very important fundamental concept. Self-Stabilization is an essential property for microprocessors, operating systems, and other basic and complicated components of computer systems.[3, 4, 5]

The implicit requirement for a formal proof is part of the definition of self-stabilization. In some cases, the design is a proof-oriented design, where possible optimizations are eliminated for the sake of streamlining the proof.

References

[1]Brukman, O., Dolev, S., and Kolodner, E., "Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software," *IEEE International Conference on Software-Science, Technology & Engineering*, (SwSTE03), Herzelia, 2003, pp. 20-29. Also in the Workshop on Adaptive Distributed Systems (WADiS03), Sorrento, Italy, 2003.

[2]Dijkstra, E. W. D., "Self-stabilization in spite of distributed control, EWD391," *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, Berlin, 1982, pp. 41-46. Also in *Communication of the ACM*, Vol. 17, 1974, pp. 643-644.

[3]Dolev, S., *Self-stabilization*, MIT Press, 2000.

[4]Dolev, S., and Haviv, Y., "Self-Stabilizing Microprocessor, Analyzing and Overcoming Soft-Errors" *17th International Conference on Architecture of Computing Systems*, LNCS:2981, Springer-Verlag, 2004, pp. 31-46.

[5]Dolev, S., and Yagel, R., "Toward Self-Stabilizing Operating Systems," IEEE International Conference on Software-Science, Technology & Engineering, Industrial Track, Doctoral Symposium, (SwSTE03), Herzelia, Nov. 2003. Also Technical report, 2004-01, Dept. of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel, March 2004.

[6]Fox, A., and Patterson, D., "Self-Repairing Computers," *Scientific American*, June 2003.

[7]Lynch, N. A., *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[*]Data available online at http://www.research.ibm.com/autonomic (cited May 2004).

[8]Pease, M., Shostak, R., and Lamport, L., “Reaching Agreement in the Presence of Faults,” *Journal of the Association for Computing Machinery*, Vol. 27, No. 2, April 1980.

[9]Wensley, J. H., Lamport, L., Goldberg, J., Green, M. W., Levitt, K. N., Melliar-Smith, P. M., Shostak, R. E., and Weinstock, C. B., “SIFT: Design and Analysis of Fault-Tolerant Computer for Aircraft Control,” *Proceedings of IEEE*, Vol. 66, No. 10, 1978, pp. 1240-1255.

[10]Wilson, R., “The trouble with rover is revealed,” *EE Times*, Feb. 2004.

[11]*Proceedings of the 5th Workshop on Self-Stabilizing Systems*, LNCS:2194, Springer-Verlag, 2001.

[12]*Proceedings of the 6th Symposium on Self-Stabilizing Systems*, LNCS:2704, Springer-Verlag, 2003.