Stabilization

Anish Arora
Department of Computer Science
The Ohio State University, Columbus

The concept of system stabilization has enjoyed growing attention from computer scientists over the past 25 years. Intuitively speaking, stabilization refers to the ability of a system to converge within finitely —and possibly boundedly— many computation steps, from arbitrary system states to states from where the system exhibits desired behavior.

Historically speaking, this concept was introduced to computer science by Edsger W. Dijkstra [1]. He coined the term "self-stabilization", although our understanding is that he later concluded that the emphasis on "self" was less than helpful. Since the concept characterizes just one of several interesting system properties, the less anthropomorphic term "stabilization" will be used instead throughout this chapter.

The most frequently cited motivation for designing stabilization is to provide a sort of fault-tolerance, whereby the system automatically recovers from fault occurrences by eventually resuming its desired behavior [2]. The faults considered most often along these lines are "transients", which corrupt the system state (be it control or data) in an arbitrary manner. Empirical evidence exists that transient faults are common in practice and, since it is not always possible for the system to exhibit desired behavior if they occur, the alternative that the system eventually exhibits desired behavior is well motivated. That said, it is important to point out that stabilization can be considered for other types of faults, e.g. process crashes and restarts, Byzantine failures, etc. If, for instance, process crashes and restarts are coupled with message losses then network systems can be driven into arbitrary system states, in which case designing stabilization is reasonable. More generally, stabilization ideas are recurrent in the literature on physics, control theory, mathematical analysis, and systems science, so it comes as no surprise that several other motivations exist for designing stabilization.

In this chapter, we focus our attention on various facets of stabilization. But we cannot hope to do justice to all lines of stabilization research in these few pages. Fortunately, we can refer the reader to several resources which will compensate for our omissions. A current and on-line bibliography [3] accompanied by an access guide to the literature is available thanks to Ted Herman, and fine surveys of the area have been written by Marco Schneider [4], Mitchell Flatebo, Ajoy Datta, and Sukumar Ghosh [5], and Mohamed Gouda [6].

1 Understanding stabilization

A number of definitions have been proposed to capture the spirit of system stabilization. We begin with the one that is used most frequently–but first we must address, albeit abstractly, what we mean by systems. A system S is defined with respect to a set of states. S is a set of (possibly infinite) sequences over these states, that is suffix closed, i.e., every suffix of a sequence in S is also in S. A computation of S refers to one of its state sequences. A state predicate of S characterizes some subset of its states.

Definition 0. Let P be a state predicate of system S. S is stabilizing to P iff it satisfies the following two conditions:

- *Closure:* P is closed in S, i.e., in every computation of S that starts at a state in P, all states are in P.
- *Convergence:* every computation of S has a finite prefix such that the following state in P. □

In Definition 0, from Convergence it follows that regardless of initial state, every computation of S eventually reaches a state where P holds, and from Closure it follows that all subsequent states of the computation are in P. The definition may now be interpreted as follows: Assuming P characterizes a set of states from where all computations of S are "desirable", S eventually recovers to states starting from where its computations are desirable. Thus, S tolerates arbitrary initialization; it tolerates any finite number of fault occurrences that perturb the state arbitrarily; and it withstands loss of coordination between different system components, for instance due to memory loss, communication errors, component restarts, or mode changes.

The assumption that there always exists a set of states closed in S such that every computation starting from these states is desirable is not always valid. A more general assumption is that the set of desirable behaviors is some subset of the computations of S. We may therefore generalize Definition 0 as follows.

Definition 1. Let S' be a subset of the computations of S. S is stabilizing to S' iff it satisfies the following two conditions:

- *Closure:* S' is suffix closed, i.e. in any computation of S' every suffix is also in S'.
- *Convergence:* every computation of S has a finite prefix such that the following sequence is in S'. □

For some situations, the requirement that the system converge regardless of the initial state may be too severe. For instance, in the presence of fault occurrences system computations may be perturbed only up to some set of states (which we call the fault-span) that is larger that the set of states reached in the absence of faults but not necessarily as large as the set of all system states. This leads us to a different sort of generalization [2].

Definition 2. Let P and Q be state predicates of system S. S is Q-stabilizing to P iff it satisfies the following two conditions:

- *Closure:* P and Q are respectively closed in S.
- *Convergence:* every computation of S that starts at a state in Q has a finite prefix such that the following state in P. □

Observe that Definition 0 is the special case, S is *true*-stabilizing to P, where *true* is the state predicate characterizing all states of S.

Still other sorts of generalizations of stabilization have been proposed, focusing on different issues. One of these requires that convergence be satisfied only if some assumptions are made about the relative speed at which (or the fairness with which) different components of the system are scheduled to execute. Another relaxes the Convergence requirement so that it is satisfied with probability 1; in other words, the measure of the computations that do not satisfy Convergence is 0. Yet another discards the Closure requirement of Definition 1; this generalization has been named pseudo-stabilization by James Burns, Mohamed Gouda, and Raymond Miller.

Remark. The definitions above have intentionally left underspecified several dimensions of the model of system computation, for example, the decomposition of the system into processes, the atomicity (granularity) of the actions of processes, the communication mechanisms between processes, the type of processes (identical or different), the type of the communication network connecting processes (its topology and directionality), the fairness/timing criteria governing the execution speed of processes, the input/output behavior associated with each computation step, etc. The ability of a system to stabilize often depends on the specific choices of these dimension values, hence it is important to precisely characterize them.

Also, the definitions above are biased in favor of system states as opposed to system actions, and they favor such concurrency semantics as interleaving / central demon semantics and power-set / distributed demon semantics. It is however possible to define stabilization in terms of system actions or to account for such concurrency semantics as partial orders. (*End of remark.*)

2 Examples

As an illustration, let us consider a stabilizing program for performing "diffusing" computations on a finite, rooted tree. In this program, starting from the desirable initial state where all tree nodes are colored green, the root node initiates a diffusing computation. The diffusing computation then propagates from the root to the leaves, coloring the tree nodes red. Upon reaching the leaves, the diffusing computation completes from the leaves to the root, coloring the nodes green again. And the same cycle repeats. If however the program state is corrupted arbitrarily its subsequent execution recovers to the desirable initial state.

Our program consists of three actions at each node j. Let $c.j$ be the color of node j, and $sn.j$ be a boolean session number that is used to distinguish "j has not started participating in the current diffusing computation" from "j has completed participating in the current diffusing computation". Also, let $p.j$ be the parent node of j in the tree (hence if j is the root then $p.j$ is j, else $p.j$ is the unique node from which there is an edge to j in the tree).

The first action is executed by the root if no diffusing computation is in progress, i.e., the root color is green; it initiates a new diffusing computation by changing both the color and the session number. The second action is executed by non-root nodes if their parent is propagating a new diffusing computation, i.e., node color is green and the parent has a different sequence number; it propagates this computation by copying the parent's color and sequence number. It is also executed to recover from states reached only in the presence of state corruption. The last action is executed by a node if all children of the node have completed the diffusing computation being propagated, i.e. their color is green and they have the node's session number; it completes the node's role in the diffusing computation by reverting node color to green.

program *Diffusing-computation*
process $j : 1..N$;
var $c.j : \{green, red\}$;
$sn.j$: **boolean** ;
begin

$$c.j = green \;\wedge\; p.j = j \longrightarrow c.j \,,\, sn.j := red \,,\, \neg sn.j$$

▯

$$sn.j \not\equiv sn.(p.j) \;\vee\; (c.j = red \;\wedge\; c.(p.j) = green) \longrightarrow c.j \,,\, sn.j := c.(p.j) \,,\, sn.(p.j)$$

▯

$$c.j = red \;\wedge\; (\forall k :: \; p.k = j \;\Rightarrow\; (c.k = green \;\wedge\; sn.j \equiv sn.k)) \longrightarrow c.j := green$$

end

Why is this program stabilizing? Informally, this is because upon starting from any state, the program eventually reaches the desirable initial state (where all nodes are green and have the same session number), and henceforth diffusing computations are performed correctly. To make this precise, let's characterize the program computations as the set of all infinite state sequences where each successor state is obtained from its predecessor state by executing an "enabled" action of some node, i.e. one whose condition is true in the predecessor. Next, per Definition 0, let's characterize state predicate P so that it captures all states where no diffusing computation is in progress or where a diffusing computation has propagated to some or all nodes, and completed by some leaves and their ancestors. A succinct formulation is:

$$P \;=\; (\forall j :: \; (c.j = c.(p.j) \;\wedge\; sn.j \equiv sn.(p.j)) \;\vee\; (c.j = green \;\wedge\; c.(p.j) = red))$$

Now, the Closure and Convergence requirements of Definition 0 can be verified (hint: if P holds with respect to all nodes in any sub-tree that includes the root, this fact remains true upon further execution, and eventually the largest sub-tree for which P holds grows). In fact, Definition 0 is satisfied even if the set of computations is enriched so that in each step any number of nodes simultaneously execute one of their enabled actions. Also, with minor modification to the size of the session number, the program can be transformed while preserving its stabilization so that nodes communicate by either reading or writing

shared state (but not both simultaneously) or by message passing.

In addition to diffusing computations, a plethora of computational problems have stabilizing solutions. The literature contains several stabilizing *distributed algorithms* for *graph problems*, such as embedding a spanning tree, finding a center, biconnectivity determination, maximal matching, searching, and graph coloring. Likewise, for *synchronization problems* such as leader election, mutual exclusion, dining philosophers, distributed reset, global state snapshot, termination detection, diffusing computations, barrier synchronization, and clock synchronization. Also, for *distributed data structures*, *digital and analog circuits*, *genetic algorithms*, and *cellular automata*.

Stabilizing *network protocols* have been discovered for routing, maximum network flow determination, heartbeat synchronization, information exchange, window management, rate-based congestion control, network management, and reconfiguration. Scalable web service and load balancing in networks of workstations also benefit from stabilization. Many of these solutions stabilize with respect to a rich class of faults, including communication faults such as message loss and duplication, and in the presence of changing topologies, i.e. where nodes and channels fail-stop and/or repair.

3 Verification and Design

Reasoning about stabilization requires care since system behavior from all states —even those that are not reached during normal execution— has to be accounted for. Errors have been found in algorithms purported to be stabilizing, not only in Convergence reasoning but just as frequently in identifying proper predicates and in establishing their Closure. This has motivated formulation of several reasoning methods, including some mechanical ones.

Verifying convergence involves showing that there are no undesirable states where system behaviors deadlock, cycle, or diverge infinitely. At its heart, therefore, is reasoning about well-founded sets. To simplify such reasoning, proofs are decomposed into chains/trees of convergences, called convergence stairs. Also, proofs of a whole system are deduced from relatively simpler proofs of its components using, for instance, the shape of the dependency relationship between its components or its composition structure (e.g., sequential, parallel, phase, or layered composition).

Turning to stabilization design, one approach is to add components to non-stabilizing systems, that correct system behavior whenever the need arises. "Corrector" components may be monolithic —taking global shapshots of the system state and performing global reset if need be— or piecemeal — each making some part (and together the whole system) stabilizing. Several building blocks that commonly appear in correctors have been identified, namely, *counter flushing, window washing, reset timers, information-exchange* and *power supply*. Of course, in order to make the system stabilizing, correctors have to be themselves stabilizing; moreover, they must not interfere with desirable system behaviors.

4 Refinements of Stabilization

Recent research on stabilization has identified several stronger notions of interest.

Multitolerance. Stabilizing systems may be made to exhibit safe or secure behavior during their convergence to desirable states. By the same token, a system that is stabilizing with respect to a fault-class may be made to also mask that or another fault-class. In particular, we find that if a system can be made to mask a fault-class F, it is possible to first make the system stabilizing with respect to F (in the sense of Definition 2) by adding corrector components, and to then further augment it to mask F, by adding so-called detector components.

Locality of stabilization. As defined, stabilization does not require that the cost of correcting small errors be less than that of correcting big ones. Hence, stabilizing systems may be refined to be fault-containing, so that errors are corrected with cost proportional to their extent. This implies only local correction of the

system state, as opposed to global correction. Alternatively, system "output" is corrected quickly, albeit correcting the system state takes longer.

Minimizing stabilization time. Likewise, stabilization does not require that convergence time be bounded. Yet, it is often the case that convergence time is bounded by some function on system size or error extent. Sometimes, there is a tradeoff between the convergence time and the system response time in the absence of faults. In such cases, designs are tuned to favor convergence time or response time as desired.

Minimizing stabilization state. Analogously, stabilization does not require that the additional state added to achieve it be bounded. Minimizing the additional state improves stabilization in that it often decreases the number of undesirable states / behaviors added and correspondingly increases system robustness.

5 Concluding Remarks

State corruption provides a clean abstraction of how many systems are perturbed by their diverse environments, and stabilization provides a viable alternative for dealing with state corruption. As distributed and networked systems become more complex and demand for reliability grows, stabilization will increasingly impact practice.

6 References

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

[2] A. Arora and M. G. Gouda, *Closure and convergence: A foundation of fault-tolerant computing*, IEEE Transactions on Software Engineering, 19(11), 1015–1027, 1993.

[3] T. Herman, *Self-stabilization bibliography: Access guide*, Chicago Journal of Theoretical Computer Science, Working Paper WP-1, initiated November 1996. http://www.cs.uchicago.edu/cgi-bin/cjtcs/get/working-papers/1.html

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

[5] M. Flatebo, A. K. Datta, and S. Ghosh, *Self-stabilization in distributed systems*, Chapter 2, *Readings in Distributed Computer Systems*, 100-114, IEEE Computer Society Press, 1994.

[6] M. G. Gouda, *The triumph and tribulation of system stabilization*, Invited Lecture, Proceedings of 9th International Workshop on Distributed Algorithms, Springer-Verlag:972, 1–18, 1995.

7 Cross References

Data Structures, Distributed see Stabilization.
Distributed Algorithms see Stabilization.
Failure Detection see Stabilization.
Fault-Tolerant Systems see Stabilization.
Fault-Tolerant Clusters see Stabilization.
Granularity, Computation see Stabilization.
Graph Algorithms see Stabilization.
High Assurance Systems see Stabilization.
High Confidence Systems see Stabilization.
Load Balancing see Stabilization.
Monitoring, Distributed see Stabilization.
Mutual Exclusion see Stabilization.

Network Protocols see Stabilization.
Network Topologies see Stabilization.
Probabilistic Algorithms see Stabilization.
Process Management, Distributed see Stabilization.
Program Verification, Distributed see Stabilization.
Programming, Concurrent see Stabilization.
Proof Techniques see Stabilization.
Reconfiguration see Stabilization.
Recovery see Stabilization.
Semantic Models of Concurrency see Stabilization.
Synthesis of Concurrent Programs see Stabilization.
Synchronization see Stabilization.
Temporal Logic see Stabilization.
Time Synchronization see Stabilization.
Token Ring see Stabilization.
UNITY see Stabilization.
Workstation Clusters see Stabilization.

8 Dictionary Terms

Closure In closed sets of states, system computations that start from a state within the set always remain within the set.

Convergence In convergence from one closed set of states to another, system computations that start from the first set eventually behave as computations that start from the second.

Diffusing Computation Computation used to perform a task that accesses or modifies the collective system state. Typically, a distinguished system process periodically initiates the computation, which propagates across the system to perform some subtask at each process. Having completely spanned the system, the computation then collapses back to the distinguished process. Examples include global state snapshot, termination detection, deadlock detection, and distributed reset.

Error-detection Determining whether the system has deviated from desirable states (or behaviors).

Error-correction Restoring the system to desirable states (or behaviors) upon error-detection.

Fault-containment Limiting the spatial extent to which system state is corrupted in the presence of faults.