Practical Self-Stabilization for Tolerating Unanticipated Faults in Networked Systems *

Anish Arora
Ohio State University
Columbus, OH
anish@cis.ohio-state.edu

Yi-Min Wang
Microsoft Research
Redmond, WA
ymwang@microsoft.com

Abstract

It is our position that the property of stabilization is desirable for distributed, networked systems to deal with unanticipated faults. In this article, we provide a gentle introduction to the concept of stabilization, respond to various criticisms and misconceptions about its use, and suggest practical approaches for its design.

1 Motivation for Stabilization

Stabilization is the branch of computing that focuses on how systems deal with arbitrary state corruption [14, 20, 12]. Strictly speaking, a stabilizing system is such that every computation of the system, upon starting from an arbitrary state, eventually reaches a state from which the computation is "correct", i.e., satisfies its specification.

Why arbitrary state corruption? More often than not, the motivation for considering arbitrary states is to account for the effects of faults. This is especially true when the faults are transient, originating in environmental effects, in Heisenbugs in hardware or software, in incorrect interactions between components, etc. Characterizing the effects of transient faults specifically is hard, hence the assumption that the resulting state is arbitrary.

That said, other faults–be they transient or permanent faults; detectable or undetectable faults; communication or computation faults, crash or omission or timing or Byzantine faults–can each be accounted for in terms of their effect on the system state and dealt with in a stabilizing manner [3]. Indeed, we observe that for any given fault class there exists a computing problem that admits a stabilizing solution that tolerates that fault class. Still, one might argue that for particular, well known fault classes the net effect on the system state of executing the system in the presence of faults can be characterized precisely, so why consider arbitrary states? In some cases, this is done simply because the process of characterizing the corrupted states is complex. A stronger argument for networks is that even humble faults, such as combinations of node failures, node repairs (into, say, prespecified start states) and message loss, can drive networks into arbitrary states [15]. It is not surprising, then, that many stabilizing protocols have been developed for *dynamic* and/or *ad hoc* networking problems.

Why eventual satisfaction? The motivation for considering "eventual" satisfaction of the specification after an arbitrary state corruption occurs lies in the difficulty of dealing with faults in a stricter manner. The alternative of "always" satisfying the specification in the presence of faults–in other words, to mask the faults–is sometimes impossible, impractical, or unnecessary. Another alternative of always preserving the safety properties of specification, but not necessarily the liveness properties–in other words, to failsafe the system–sometimes compromises the availability requirements of systems. Thus, allowing faults to violate the specification, albeit temporarily, in the presence of faults is worthy of consideration.

Example of a stabilizing network service. Aladdin [24, 23] is a system for dependable, extensible control of heterogeneous devices via an in-home PC cluster and heterogeneous network. A key to the extensibility of Aladdin is its *Lookup service*. This service re-

*This work was partially sponsored by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and two grants from Microsoft Research.

sponds to two types of queries: (i) an attribute based query which returns a list of unique names that match the attributes, and (ii) a name based query which returns the address of an object given the unique name. The lookup service maintains information regarding addressing, location, and status of the various types of objects in the home network, including sensors, devices, and controllers.

Objects typically join and leave the network spontaneously. To automate the discovery process, objects periodically "refresh" their status and location in the lookup service, at a frequency of their choice. This frequency is chosen based on a number of factors, e.g., how often their status changes, how much network bandwidth is available, and how much power conservation is of concern. Depending upon the frequency chosen, refreshes are classified as being either *low-frequency* or *high-frequency*.

The specification of the lookup service is stated informally as: "each query to the lookup service returns a unique up-to-date response". For dependability, we replicate the service on an in-home PC cluster. We assume that refresh messages and queries are broadcast to all the replicas.

We implemented a stabilizing lookup service, as follows [6]. In all correct lookup behaviors, we postulate there always exists a unique server that responds to queries, and this server–which we refer to as the leader–has up-to-date status of the objects. Let i range over the replicas; the boolean model variable $alive.i$ denotes that node i is running a functioning replica; $leader.i$ denotes that replica i believes it is the leader; and $uptodate.i$ denotes that replica i has the most recent information. Then, the following logical conditions –Activity, Unicity, and Recency–are invariantly true of all states in correct lookup behaviors.

Activity : $((\exists i : alive.i) \wedge (\forall i : leader.i \Rightarrow alive.i))$
Unicity : $((\exists i : leader.i) \wedge \neg(\exists i, j : leader.i \wedge leader.j))$
Recency: $(\forall i : leader.i \Rightarrow uptodate.i)$

Stabilization was achieved by enforcement of each of these three state predicates. Our implementation essentially (re)satisfies each predicate upon its violation. The corrector for *Activity* runs a protocol that diagnoses and repairs failed replicas. The corrector for *Uniqueness* implements a "weak leader election" protocol; in this protocol, only a unique replica knows it is the leader (the term 'weak' is used in the sense that the other nodes need not know the identity of the leader). Only the leader responds to queries. *Recency* is automatically corrected in part by the periodic refreshes that are received from the various objects. The refreshes serve to repopulate the lookup database at the replicas once they have been repaired. While this is adequate for high-frequency refreshes, for the case of low-frequency refreshes the replica might respond to queries with out-of-date information for a long interval. Therefore, as part of the diagnosis and repair protocol, the information corresponding to the low frequency refreshes is streamed to the newly repaired servers to ensure *Recency* for low-frequency objects.

2 How Stabilization Deals with Unanticipated Faults

As the complexity of networked systems increases, their likelihood of experiencing unanticipated faults sooner or later also grows. By unanticipated network faults, we mean not only (a) occurrence of new types of faults that were not explicitly planned for in network design, but also (b) occurrence of well known types of faults but at frequencies or in combinations that are abnormal. Standard network protocol designs are optimized for common-case of well-known faults (for instance, a bounded numbers of node crashes or link crashes or message failures occurrences over some period of time). Exception handlers are used to deal with the common-case fault scenarios by detecting specific logical conditions that arise in these scenarios. In the rare-case, however, a new type of fault or a burst of well known faults may occur that violates those specific conditions.

In some traditional fault-tolerance techniques, unanticipated faults of sort (b) are overcome by calculating the weaker conditions that hold in the presence of multiple fault occurrences. But, as mentioned before, such a calculation can be complex; routing is a classic example where calculating the illegitimate states that are reachable in the presence of multiple fault occurrences quickly becomes complex. Also, many exception handlers may result, yielding designs that are subject to software engineering concerns about the lack of quality testing of exception handlers, as well as the potential for the handlers to interfere with one another or with the underlying protocol methods, as evidenced by the AT&T network disaster of 1990 [1].

Lazy conformance to correct network behavior. Stabilization provides one alternative that avoids case-by-case handling of unanticipated faults of sort (b) and can deal with faults of sort (a). We illustrate this point with an abstract explanation of stabilization: Stabilization involves defining "correct" behavior of the network and by continually checking whether the network is presently conforming to this behavior. In other words, instead of defining specific patterns of incorrect behavior, stabilization checks for occurrence of any anomalous behavior. Should incorrect behavior be detected, stabilization promises to restore the network behavior so that eventually it is correct.

Preventing state corruption from spreading. There are, of course, many different ways to restore network behavior, the simplest of which is to reset the network to a prespecified (or checkpointed or otherwise legitimate) state. Resets alone do provide tolerance to unanticipated faults whose net effect is transient, but not necessarily for other types of unanticipated faults. The key intuition, then, in designing stabilization to deal with a rich class of unanticipated faults is to partition the protocol into components that are each capable of (i) correcting their own state, when non-conformance to correct behavior is locally detected, and (ii) detecting upon demand (directly or otherwise) whether the communications they receive contain corrupted data, and avoiding/compensating for the corrupted ones [7]. The latter prevents persistent spreading of corruption from permanently fault-affected components.

Lookup service example (continued) During a deployment period of 14 days without any restart, on a cluster of four machines–Jasmine, MagicCarpet, Abu, and Genie–an unidentified 'glitch' caused unanticipated network partitioning on a number of occasions. We briefly describe one such occurrence (which in fact recurred several times): MagicCarpet became isolated from the rest of the nodes; Jasmine lost contact with Genie; and, Abu lost contact with both Jasmine and Genie. As a result of the partitioning, more than one node assumed leader status. In all cases, stabilization restored the system to having a unique leader and subsequently the lookup specification was resatisfied.

Incidentally, our implementation not only tolerated certain unanticipated faults via stabilization, it also provided stricter guarantees for anticipated faults. We identified the following faults as being common-case: the crash of a single replica and the loss of a single refresh message. To handle the crash of a single replica efficiently, the stabilization for *Activity* ensures that there are always two or more alive replicas with up-to-date information. In case the leader fails, the leader election protocol allows one of the remaining replicas with up-to-date information to automatically assume the role of leader. To handle the single refresh loss efficiently, we add an acknowledgment mechanism in which the leader is required to acknowledge the low-frequency refreshes. Low-frequency objects are expected to re-transmit their refreshes until an acknowledgment is received. In case of high-frequency refreshes, high data-quality is achieved per se and so we do not require the use of acknowledgments. Stabilization thus did not come at the expense of (or interfere with) other desirable tolerances.

3 A Primer on Designing Stabilization

As explained above, designing stabilization involves characterization of correct network behavior. Abstractly speaking, this characterization involves:

- identifying the specification that the network protocol is satisfying,
- calculating the fault-free computations of the network protocol, and
- showing that the fault-free computations satisfy the specification.

Likewise, showing that the network behavior in the presence of faults is eventually correct implicitly involves showing that from some point onwards, the network behavior is the same as that of some fault-free computation.

Clearly, this abstract characterization is not amenable for designing stabilization. We now discuss a standard two-step approach for stabilization design.

Step 1: Design an invariant. In this step, correct network behavior is characterized in the form of a set of legitimate states, called an *invariant* predicate. As the name suggests, an invariant is a Boolean condition that is always true (in all states of all fault-free computations of the network protocols). That is, the set of states denoted by an invariant is closed under fault-free network execution. It is important to appreciate that many state predicates (i.e. sets of states) that are closed under fault-free network execution, e.g. *true*, *false*, etc., are not invariants by themselves. An invariant must contain enough information so that one can show that "starting from *any* state in the invariant, *every* fault-free computation of the network protocol satisfies the specification". The set of all reachable states in all fault-free computations of a network protocol often comprises an invariant. Or, designers can guess predicates weaker than the set of reachable states that suffice for correctness reasoning.

(We remark that the demand that the invariant be "complete"–in the sense that it contains all facts needed to prove correctness with respect to the specification–is nonstandard as well as nontrivial. That it is nonstandard has caused confusion, but is not serious. That it is nontrivial is more serious, and we will further discuss this point later on. For now, we note that fortunately invariant-based proofs are often temporally local: safety and liveness properties of the specification are verified by reasoning about individual state transitions only, thus avoiding the treacherously long chains of state transitions contained in operational proofs of network behavior.)

Step 2: Design convergence to invariant. In this step, protocol computations starting from arbitrary states are shown to converge to some invariant state. Since the invariant is closed, once an invariant state is reached, protocol computations behave as fault-free computations. And since any computation starting from an invariant state satisfies the specification, it follows that correct network behavior is eventually restored in all computations starting from arbitrary states.

Convergence is typically achieved by separately designing *detectors* (that establish whether the network is in a state where its invariant is violated) and *correctors* (that restore the state so as to resatisfy invariant) [8]. As noted in the previous section, during the convergence process spreading of corrupted data from one component to the other needs to be circumvented; detectors and correctors suffice for this purpose.

4 Major Criticisms/Misconceptions About Stabilization

Computer networking already has several protocols that we regard as being stabilizing. Soon after Dijkstra reported his stabilizing token ring protocol in 1973 [10], for instance, he became aware of its implementation in a Dutch network. As other examples, stateless protocols such as IP are stabilizing by definition. RIP and OSPF are based on stabilizing protocols; the former relies on Bellman-Ford which is per se stabilizing and the latter simply discards its routes periodically and recomputes based on fresh state. Also, several soft-state protocols such as ARP/RARP are inherently stabilizing.

A seasoned critic might reasonably argue that stabilization is merely a serendipitous property of these protocols, since many networking experts are not explicitly aware that these protocols are stabilizing. In fact, our critic will raise a variety of objections to question whether stabilization is, practically speaking, of any value to networking.

1. What good is eventual satisfaction? Stabilization guarantees only that after a fault occurs–and assuming no further faults occur–the network will *eventually* reach a state from where correct operation will resume. Given that in any significantly sized network, at any time, some part or the other of the network is subject to a fault, our critic would object that the "eventually" guarantee is too weak to be practical.

2. Stabilization is not an "easy" property. The stabilization literature is peppered with enough sophisticated, complex algorithms that it is now a knee-jerk reaction to relegate judgment of correctness of a freshly-minted stabilizing algorithm to a stabilization expert. Indeed, many incorrect solutions have been submitted for public view. As can be expected, many of these contained incorrect proofs of convergence. What is interesting, then, is that in our experience just as many erroneous proofs were due to incorrectly identifying the invariant or in incorrectly showing that the invariant itself was closed under protocol execution. Our critic would surmise that all this suggests that designing sound and complete invariants of protocols is hard, and most protocol designers would not find this task appealing, *especially when protocols become large*. Note that the designer's task increases linearly in the code size, at the very least, e.g. for checking the closure of the invariant.

3. Stabilization is not a local property. While stabilization deals well with arbitrary state perturbations, it is not always clear how well it deals with only "minor" state perturbations. Consider the problem of routing (or, more abstractly, spanning tree maintenance) for example. If a root node of a (sub)tree fails, the following scenarios are conceivable:

- Instead of just fixing the root from among its nearby nodes, a stabilizing algorithm may choose a replacement root that is far away, and substantially tear down the tree so as to rebuild it with respect to the new, distant root.
- Unless a child of the failed root informs all of its downstream nodes before it changes its route, a (potentially long) cycle in the route can be formed. This is because if a distant descendant of the child is not aware of the failure of the root, it may offer to the child an alternative (but incorrect) route to the root. If the descendant is not made aware of the failed root, the child may point to the descendant which (indirectly) points to the child, thereby yielding a cycle.

In each of these scenarios, a spatially local perturbation causes spatially non-local checking or non-local healing. Similar arguments could have been made in temporal or energy domains. In other words, even for a minor fault, which if anticipated could have been dealt with via a low-cost exception handler, stabilization yield a high-cost response. Our critic would therefore argue that stabilization is an "expensive" property, since it tends to avoid distinguishing particular perturbed states.

4. Stabilization is not a composable property. In contrast to stabilization, masking of faults offers the attractive feature of being composable. That is, if a component is wrapped to mask certain faults, then the wrapped version of the component can be (hot/cold) swapped with the unwrapped version, without affecting the correctness of the overall system. In other words, when a component is masking fault-tolerant, other components do not have to deal with any unanticipated communications with it, and hence the original reasoning that applied to the system components in the absence of faults also applies in the presence of faults. In general, this is not true of stabilizing components. While there is existing research on composing stabilizing components in layers, in serial manner, and in parallel, the composition techniques are arguably limited. Components in general have to account for "arbitrary" communications with other components, and indefinite propagation of corruption from the one component to another has to be avoided. Our critic would therefore argue that stabilization is not readily useful for COTS based systems.

5. Stabilization is not a refinable property. Most design of stabilization is performed using linguistic notations suitable for abstract protocols, such as guarded commands, communicating finite state machines, high-atomicity message-passing/shared memory languages. Do the proofs of stabilization continue to remain valid when the abstract protocol is refined into a C program that uses TCP/UDP sockets? Worse, can standard compilers take a stabilizing C-program as input and produce object code that violates the property of stabilization? Our critic would therefore argue that it is unclear that the confidence gained by formal reasoning about invariants and convergences at an abstract level remains once the abstract protocols are refined/compiled into their final concrete form.

5 Redressing The Criticisms

In this section, we respond to each of the objections of our seasoned critic from the vantage point of the state-of-the-art in stabilization research.

Response to Criticism 1. We claim that "eventually" is merely a convenient generalization of "not immediately". Stabilization necessarily relies on "not immediately" since otherwise systems must check all the time for unanticipated state/behavior. In several cases, stabilization is indeed achieved in constant time, e.g. as a constant function of the time periods used in soft-state protocols. In most other cases, it is achieved in time that is bounded by a function of certain network parameters, such as degree, diameter, current number of nodes, etc. Indeed, a lot of research in stabilization has focused on achieving stabilization in optimal time and state [11].

That the definition of stabilization has not been amended to demand bounded convergence is simply a matter of taste. For one, stabilization researchers prefer not to overspecify the models in which they show stabilization; as a result, the property of stabilization is more readily portable from one model to another. By way of contrast, convergence bounds are not as readily ported from one model to another, e.g. from a shared memory model to a bounded-time message passing model. As another reason, even when stabilization time is bounded, simply by waiting for that bounded amount of time, a network process cannot deduce that the network has resumed correct operation. An undetectable fault may have occurred during the wait itself, and all that the process knows is that correct operation has resumed only if no new faults have occurred.

Response to Criticism 2. It has now been well recognized that verifying stabilization is a special case of verifying correctness. Indeed, closure and convergence are easily captured in various temporal logics, such as Chandy and Misra's UNITY and Lamport's TLA. Lamport and others have shown mechanical verifications of stabilizing protocols, and in the process also shown that no new techniques are involved. Thus, we can at least observe that verifying stabilization is no harder than verifying protocol correctness.

We now turn to the practicality of including correctness properties in the design task, in particular, the calculation of an invariant and the proof of closure of and convergence to the invariant. Recent work in stabilization has focused on simplifying this task in several dimensions: (i) Convergence of *every* computation from *every* state is weakened to convergence of *some* computations from *some* states [13]. The first weakening still suffices to show that convergence occurs with high probability and the second weakening avoids consideration of highly unlikely states. (ii) The invariant is calculated based not on the protocol but based on the specification that the protocol satisfies, under the assumption that the specification is significantly smaller than the protocol [6]. This weakening is sound in the sense that the specification invariant must be satisfied by the protocol, albeit the protocol might satisfy some additional state predicates as well in its legitimate states. (iii) For similar reasons, convergence to an invariant is weakened to convergence to a partial invariant, in which only some but not all aspects of network state that are necessary for correct behavior are guaranteed to be satisfied.

Response to Criticism 3. Recent research has carefully investigated the extent to which problems admit local "containment". By maximizing the degree of local containment, it is possible to (a) limit the extent to which a local perturbation affects the rest of the network and (b) limit the amount of work that has to be done to stabilize the network once a local perturbation occurs [19].

By way of example, consider fair resource allocation, a problem which can be solved using dining philosophers. This problem admits a stabilizing solution in which every node depends only on nodes in its neighborhood that are at a distance of at most 2 hops from it. Thus, it can be shown [19], in the spirit of (a), that a misbehaving node (even a Byzantine node) cannot affect nodes that are more than 2 hops away from it. Moreover, in the spirit of (b), that if a node is locally perturbed into an incorrect state, then as long as it has no misbehaving nodes at a distance of at most 2 hops from it, the node eventually converges to locally correct behavior. This example is remarkable because node stabilization is achieved in the presence of Byzantine faults on nodes. (We make this point to dispel the myth that masking tolerance–and not stabilization–is the only relevant tolerance with respect to Byzantine faults.)

For problems that are inherently non-local, other research [25] has alternatively considered (i) changing the network model so that local solutions are admitted and (ii) approximating the problems so that local solutions are admitted. By way of example of (i), by assuming that wireless communication is possible (whereby all neighbors of a node can receive a communication from it at about the same time) or that the density of neighbors in the network is high, it becomes possible to achieve local stabilization for routing tree maintenance, despite the difficulties mentioned in the previous section. As an example of (ii), in the distributed node cluster formation problem, where nodes locally group themselves into internally connected and mutually disjoint clusters, demanding that every cluster has some constant number of nodes or some constant radius yields only non-local solutions; however, if we weaken the number/radius to some appropriate interval, local solutions are achievable.

Response to Criticism 4. We note that recent research is extending composition methods beyond the principles of layering, sequential/phased convergence, and parallel convergence. The basic idea is to deal with spreading of corruptions from one component to another in each and every of the following ways: (i) through method invocation with corrupted arguments, (ii) through receiving corrupted result arguments in responses to method invocations, and (iii) through violation of component rely/guarantee contracts.

The technique for dealing with the spreading of corruption in between components is to a posteriori wrap components, with special purpose stabilization components called detectors and correctors [8]. Moreover, the speed of executing detectors and correctors is tuned with respect to the spare resources available in the system and in proportion to the frequency and locality of perturbations. Detectors and correctors themselves are more easily developed from a reusable library of pattern-specific component templates [4, 22, 21, 5, 16]. In some cases, the arguments with which to instantiate the detector and corrector templates can be automatically synthesized given component specifications.

Response to Criticism 5. Recent research has focused on developing refinement techniques and compilers that preserve stabilization properties [9, 2, 18, 17]. The Austin APC compiler, for example, compiles stabilizing code from the AP (Abstract Protocol) notation into C-code that use UDP socket communications while preserving certain stabilization properties. Initial steps are being made towards making standard compilers stabilization preserving, and towards adding stabilization properties to operating systems and virtual machines. In particular, this research direction has led to consideration of stabilization in (sequential or distributed) data types, in contrast to previous work which has focused more on distributed control.

6 Practical Stabilization

While research on stabilization is progressing well towards simplifying its use and increasing its applicability, networked system designers can already leverage most of the benefits of stabilization through practical approximations. We summarize here techniques that are inspired by stabilization, which the interested network system designer should consider.

1. Ensure convergence to partial invariants. Not all parts of the invariant are equally important for achieving robustness of a network protocol. In dealing with unanticipated faults, it is therefore more important or more efficient to deal with just the central parts of the invariant. So, ensuring convergence just to these parts should be valuable. Design effort should be spent more on ensuring that the chosen parts are sound (i.e. they are closed under network execution) than on ensuring that the chosen invariant is complete.

2. Use model-based or specification-based invariants. As a heuristic, the central parts of the invariant are likely to be predicates that can be deduced from abstract models of the protocol or from its specification. In other words, these parts may not be specific to the details of the protocol implementation. Since models and specifications can be significantly smaller than the protocol code, deducing partial invariants from them should be used when possible.

3. Ensure weak forms of convergence. Just as not all states (predicates) are equally likely to occur under corruption, not all protocol computations from a given state are equally likely either. Thus stabilization should be enforced first with respect to "reasonable" computations. For example, in computations which are taking too long to convergence, it may be appropriate to abandon the attempt to stabilize. Often, this may be justified by taking into account the probability of that computation.

4. Design low-cost stabilization. Since unanticipated faults should be the exception and not the norm in any networked system, stabilization should not involve high-cost support. Only non-critical resources should be allocated to stabilization, whereas critical resources may be allocated to deal with common-case anticipated faults. More generally, stabilization should not interfere with support for anticipated faults, as discussed in [6], which described a technique that stabilizes to a weaker invariant that includes the effects of executing the common-case faults.

5. Use pattern-specific templates for designing stabilization. To increase the confidence in the correctness of the detectors/correctors used for stabilization, standard patterns should be used where possible. Many such patterns have been discussed in the literature, e.g. for global snapshot, for distributed reset, for constraint-based satisfaction of invariants, for distributed cleaning, etc. Assuming that stabilization is attempted with low frequency, a slight loss of efficiency that may result as compared with hand-crafted detectors/correctors might still be worthwhile.

Acknowledgements. Many colleagues have influenced the views reflected in this position paper. In particular, we thank Marvin Theimer, Mohamed Gouda, and Ted Herman.

References

[1] AT&T failure of January 15, 1990. `http://www.cs.berkeley.edu/~nikitab/courses/cs294-8/hw1.html`.

[2] A. Arora, M. Demirbas, and S. S. Kulkarni. Graybox stabilization. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, July 2001.

[3] 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.

[4] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[5] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[6] A. Arora, R. Jagannathan, and Y. M.Wang. Model-based design of dependability in distributed systems. *Proceedings of the Workshop on Concurrency in Distributed Systems*, pages 61–68, 2001.

[7] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[8] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.

[9] M. Demirbas and A. Arora. Convergence refinement. *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

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

[11] S. Dolev. *Self-Stabilization*. MIT Press, March 2000. ISBN 0-262-04178-2.

[12] M. Flatebo, A. K. Datta, and S. Ghosh. Self-stabilization in distributed systems. In *Readings in Distributed Computing Systems*, chapter 2, pages 100–114. IEEE Computer Society Press, 1994. TL Casavant and M Singal, Editors.

[13] M.G. Gouda. The theory of weak stabilization. In *WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS:2194*, pages 114–123, 2001.

[14] T. Herman. Self-stabilization bibliography: Access guide. Chicago Journal of Theoretical Computer Science, Working Paper WP-1, initiated November 1996, last revised January 2001. http://www.cs.uiowa.edu/ftp/selfstab/bibliography/.

[15] M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 247–256, 1996.

[16] S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[17] T.M. McGuire and M.G. Gouda. Correct implementation of network protocols from abstract specifications. http://www.cs.utexas.edu/users/mcguire/research/.

[18] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer LNCS:1693*, pages 254–268, 1999.

[19] M. Nesterenko and A. Arora. Local tolerance to unbounded byzantine faults. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2002.

[20] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[21] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.

[22] G. Varghese. Self-stabilization by counter flushing. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

[23] Y. M. Wang, W. Russell, and A. Arora. A toolkit for building dependable and extensible home networking applications. *Proc. USENIX Windows Systems Symposium*, pages 101–112, 2000.

[24] Y.-M. Wang, W. Russell, A. Arora, J. Xu, and R. Jagannathan. Towards dependable home networking: An experience report. *International Conference on Dependable Systems and Networks (ICDSN)*, 2000.

[25] H. Zhang and A. Arora. Gs^3: Scalable self-configuration and self-healing in wireless networks. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.