STABILIZING COMMUNICATION PROTOCOLS

Mohamed G. Gouda and Nicholas J. Multari

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-90-20 June 1990
October 1990 (Revision)

ABSTRACT

A communication protocol is *stabilizing* iff starting from any unsafe state (i.e., one that violates the intended invariant of the protocol), the protocol is guaranteed to converge to a safe state within a finite number of state transitions. Stabilization allows the processes in a protocol to re-establish coordination between one another, whenever coordination is lost due to some failure. In this paper, we identify some important characteristics of stabilizing protocols; we show in particular that a stabilizing protocol is nonterminating, has an infinite number of safe states, and has timeout actions. We also propose a formal method for proving protocol stabilization: in order to prove that a given protocol is stabilizing, it is sufficient (and necessary) to exhibit and verify what we call a "convergence stair" for the protocol. Finally, we discuss how to redesign a number of well-known protocols to make them stabilizing; these include the sliding-window protocol and the two-way handshake.

Keywords: communication protocols, convergence, formal verification, self-stabilization, sliding-window protocol, two-way handshake.

1 Introduction

A communication protocol can be described by a collection of processes that exchange messages over connecting channels in a computer network. One failure that can severely disrupt the proper execution of a protocol is *coordination loss*. Informally, coordination is said to be lost at a given global state of a protocol iff the local states of different processes in the protocol, though each of them may be correct in its own right, are inconsistent with one another in the given global state.

Loss of coordination in a protocol has many causes; we list here some of them.

i. *Inconsistent Initialization:* The different processes in the protocol may be initialized, by mistake, to local states that are inconsistent with one another.

ii. *Mode Change:* In order to change the mode of operation in the protocol, all processes are informed about the desired change and are requested to effect it. However, because of the distributed nature of the network, not all processes get the request and effect the change at the same time. Thus, the protocol is bound to reach a global state in which some processes have changed while the others have not.

iii. *Transmission Errors:* If some sent messages are lost, corrupted, or reordered before being received, the local state of the sending process may no longer be consistent with that of the receiving process.

iv. *Process Failure and Recovery:* If a process "goes down" then "comes up" again, its local state may become inconsistent with the local states of other processes.

v. *Memory Crash:* The local memory of a process may crash causing its local state to be inconsistent with the local states of other processes.

So far, researchers have concentrated on dealing with the individual causes that may lead to loss of coordination, rather than investigating how to counter coordination loss irrespective of its cause. The net result is that most existing protocols have different procedures that perform the same basic function of restoring coordination whenever it is lost. For example, the initialization procedure in most protocols is

different from the procedure for mode change, which in turn is different from the procedures for dealing with transmission errors and process failure and recovery, and so on.

This traditional approach is not satisfying both from a scientific and an engineering point of view. First, this approach allows one phenomenon, coordination loss, to be investigated as a collage of seemingly different phenomena. Second, it allows many protocols to have different procedures that perform the same basic function.

In this paper, we propose a new method, stabilization, that can be used to counter coordination loss regardless of its cause. In order to explain this method, we need first to introduce the concepts of global states and safe states of a protocol.

Each process and each channel in a communication protocol has a (possibly infinite) set of local states. Every combination of the local states, one from each process and each channel, constitutes a global state of the protocol. Each global state is identified by the protocol designer as being either "safe" or "unsafe". Once this identification has been made, the protocol designer proceeds to make the safe states reachable from the initial global state of the protocol, and the unsafe ones unreachable. This guarantees that the initial global state of the protocol is safe and that once the protocol is in a safe state, then any further execution of the protocol leads to a safe state. There are no guarantees on what will happen if the protocol starts its execution at an unsafe state. This last observation suggests the following, rather informal, definition.

A communication protocol is stabilizing iff starting from any unsafe state, the protocol is guaranteed to reach a safe state in finite time (or more precisely, after executing a finite number of state transitions).

Example: Consider a communication protocol between two processes: "sender" and "receiver". The sender has a local variable "sent" that counts the number of messages sent by the sender, and the receiver has a local variable "rcvd" that counts the number of messages received by the receiver. The sent messages are stored in a channel until they are received (by the receiver). The natural coordination between the two processes is defined by the global predicate:

$$\text{sent} = \text{number of messages in the channel} + \text{rcvd} \qquad (1)$$

Thus, every global state that satisfies this predicate is safe, and every global state that satisfies the complementing predicate:

$$\text{sent} \neq \text{number of messages in the channel} + \text{rcvd} \qquad (2)$$

is unsafe.

Therefore, the protocol is stabilizing iff starting from any global state satisfying (2), it is guaranteed to reach a global state satisfying (1).

We are not going to discuss here how to make this particular protocol stabilizing (more elaborate examples are discussed later); rather, we will try to explain what benefits are achieved by making this protocol stabilizing.

There are many possible causes for this protocol to ever reach an unsafe state satisfying (2). One possibility is a wrong initialization procedure that causes variable "sent" to be initialized to 0 and variable "rcvd" to be initialized to 10. Another possibility is a message loss from the channel that causes:

$$\text{sent} > \text{number of messages in the channel} + \text{rcvd}.$$

A third possibility is a temporary failure of the sender process that resets variable "sent" to 0 mid-way during the protocol execution, and so on.

By making the protocol stabilizing, the protocol can counter all these different types of failures, and regain coordination, i.e., return to (1), in finite and possibly short time after coordination is lost (for whatever cause). Clearly, stabilization provides the protocol with a high degree of fault-tolerance. □

The issue of stabilization is not entirely new. In fact, it has been studied before in mathematics and in control theory. However, it arises in these disciplines not as a fault-tolerance issue, but rather as a platform for studying convergence. Consider for example the following well-known Newton-Raphson algorithm for iteratively computing the root of a given function f:

while $f(x) \neq 0$ **do** $x := x - (f(x)/f'(x))$.

This algorithm can be regarded as self-stabilizing in the following sense. Each state where $f(x) = 0$ is regarded as safe, and each state where $f(x) \neq 0$ is regarded as unsafe. Given that function f is well-behaved, this algorithm is guaranteed to converge from any unsafe state to a safe state; hence, it is a stabilizing algorithm.

The application of stabilization to distributed computing systems is relatively new. It has started in 1973–1974 by two classical papers of Dijkstra [6, 7]. More recently, a number of researchers including Lamport [12, 13], Bastani, Yen, and Chen [2], Brown, Gouda, and Wu [3], Burns and Pachl [4], Gouda and Evangelist [8], Gouda, Howell, and Rosier [9], and Katz and Perry [11], have started to work more diligently in this area. Our objective in the current paper is to study the application of stabilization to communication protocols [14].

The rest of the paper is organized as follows. In Section 2, we formally introduce the concept of stabilizing protocols. Then in Section 3, we discuss three important characteristics of these protocols; these three characteristics can be regarded as necessary conditions that should be satisfied by every stabilizing protocol. In Section 4, we introduce the concept of a "convergence stair" and show that the existence of a convergence stair for a protocol is both necessary and sufficient for proving that the protocol is stabilizing. Then in the next two sections, we discuss how to redesign two well-known protocols to make them stabilizing: the sliding-window protocol is discussed in Section 5, and the two-way handshake is discussed in Section 6. Concluding remarks are given in Section 7.

2 Stabilizing Protocols

In this section, we present a simple notation that can be used in defining communication protocols and their properties; we then use this notation to formally introduce the concept of stabilizing protocols.

2.1 Syntax

A *communication protocol* is a set of two or more processes; each process has the syntax:

process <process name>
  var <var name>:<var type> ; ... ; <var name>:<var type>
  begin <action> [] ... [] <action>
  end

The variables declared in a process are called *local* to that process. Each action has the syntax:

<guard> → <sequence of local or send statements>

where a local statement is one that reads and writes only the local variables of its process, and a send statement has the syntax **send** <message> **to** <process name>. The guard of an action is in anyone of the following three forms:

<local guard>
<local guard> **and** <receive statement>
timeout <global guard>

where a local guard is a predicate that involves only the local variables of its process, a receive statement has the syntax **rcv** <message> **from** <process name>, and a global guard is a predicate that involves all local variables (of all processes) and all channels in the protocol. The concept of a channel is defined next.

Associated with each pair (p,q) of distinct processes in a protocol is a shared variable C_{pq} whose value is a sequence of messages taken from those messages that can be sent from p to q. Variable C_{pq} is called the *channel* from p to q.

Notice that timeout actions have global guards where all local variables and all channels in the protocol can be tested. Because of these additional capabilities, timeout actions are expensive to implement (using realtime clocks), and good protocol designers try to avoid them as much as possible. Unfortunately, as we show in Section 3, stabilizing protocols cannot be designed without timeout actions.

2.2 Semantics

A *state* of a protocol is defined by one value for each local variable in each process in the protocol, and one value (i.e. a sequence of messages) for each channel in the protocol.

An action c in some process p is said to be *enabled* at some protocol state s iff the guard of c, whether local or global, is true at s, and moreover if c has a receive statement **rcv** m **from** q then the head message in channel C_{pq} is m at state s.

A protocol state s' is said to *follow* a protocol state s over some action c in a process p iff c is enabled at s, and executing the (local, send, or receive) statements of c starting from s yields s'. As expected, executing a local statement updates the local variables of p, executing a send statement "**send** m **to** q" adds message m to the *tail* of the message sequence C_{pq}, and executing a receive statement "**rcv** m **from** q" removes the *head* message m from C_{qp}.

A protocol *computation* is a sequence of protocol states $s_1, \ldots, s_n$ such that each state s_{i+1} follows state s_i (over some action). A protocol computation is *maximal* iff it is infinite, or it is finite and no action is enabled at its last state.

Properties of a protocol are defined using *global predicates*, i.e. predicates that involve all local variables (of all processes) and all channels in the protocol. In particular, safety properties are defined using a special type of global predicates called closed predicates, while progress properties are defined using a convergence relation over closed predicates. The two concepts of closed predicates and convergence relations are defined next.

Let R be a global predicate of a protocol. A protocol state s is called an *R-state* iff the values of all local variables and all channels at state s satisfy R. Predicate R is called *closed* in the protocol iff each protocol state that follows an R-state (over some action) is an R-state.

Let R and S be two closed predicates of a protocol. R is said to *converge to* S in the protocol iff each maximal protocol computation that starts in an R-state has an S-state.

As an example, consider the two global predicates *true* and *false*. Each protocol state is a *true*-state and no protocol state is a *false*-state; thus both *true* and *false* are closed in every protocol. Moreover, for any closed predicate R of some protocol, R converges to *true*, and does not converge to *false* in the protocol.

2.3 Stabilization

Let R be a closed predicate of a protocol. The protocol is called R-stabilizing iff *true* converges to R in the protocol; in other words every maximal protocol computation has an R-state.

The next theorem follows directly from this definition and from the facts that *true* converges to *true*, but does not converge to *false* in every protocol.

Theorem 1: Every protocol is *true*-stabilizing, and no protocol is *false*-stabilizing. □

Our objective in the light of this theorem is to investigate protocols that are R-stabilizing where R is a stronger global predicate than mere *true*.

3 Characteristics of Stabilizing Protocols

We discuss in this section three important characteristics of stabilizing protocols. These characteristics can be viewed as necessary (but not necessarily sufficient) conditions for achieving stabilization in communication protocols. In Section 3.1, we show that stabilizing protocols cannot terminate properly. Thus a protocol should be designed to be nonterminating if its stabilization is desired. In Section 3.2, we show that finite-state protocols such as the "Alternating-bit protocol" cannot be stabilizing, and so a protocol has to have an infinite number of "safe states" if it is to be stabilizing. Finally, we show in Section 3.3, that a stabilizing protocol has to have timeout actions in one or more of its processes.

3.1 Termination versus Liveness

A protocol state s is *properly terminating* iff all channels are empty at s and the values of all variables at s guarantee, by themselves, that the guards of all actions are false at s.

The second condition of proper termination needs some explanation. At a properly terminating state, each process in the protocol should be able to "recognize" that termination has occurred. In other words, the occurrence of termination should be completely deducible from the current values of all variables. Because the guards of all actions are false at termination, the current values of all variables should, by themselves, guarantee the falsehood of all guards at termination.

A protocol is *R-terminating* for some global predicate R iff each R-state of the protocol is properly terminating.

Theorem 2: If a protocol is R-stabilizing for some closed predicate R, then it is not R-terminating.

Proof: Consider an R-terminating protocol, for some closed predicate R, where $R \neq false$; we show that this protocol is not R-stabilizing. Because $R \neq false$, the protocol has at least one R-state. Let s be an R-state of the protocol. Because s is an R-state and the protocol is R-terminating, s is properly terminating, i.e., the guards of all actions in all processes in the protocol are *false* at s, and the channels are all empty at s. Let s' be another protocol state that satisfies the following two conditions. First, the value of each variable (in each process) in the protocol at s' is the same as its value at s. Second, at least one channel in the protocol has some message(s) at s'. Because of the first condition, the guards of all actions are *false* at s' as they are *false* at s. Because of the second condition, s' is not a properly terminating state. From this fact and the fact that the protocol is R-terminating, s' is not an R-state. Consider the computation that consists of the single state s'. This computation is maximal, because the guards of all actions are *false* at s'. Moreover, this computation does not have an R-state since s' is not an R-state. Therefore, the protocol is not R-stabilizing. □

Theorem 2 suggests that stabilizing protocols should be designed to be nonterminating or live. The concept of liveness is defined next.

A protocol is *R-live* for some global predicate R iff each maximal protocol computation that starts at an R-state is infinite, and along each such computation at least one process in the protocol sends an infinite number of messages, and no timeout action is executed an infinite number of times.

Theorem 3: There are protocols that are both R-stabilizing and R-live for the same closed predicate R.

Proof: In the next section we give a protocol example that is both R-stabilizing and R-live for the same closed predicate R. □

3.2 Finiteness versus Restrain

A protocol is *R-finite* for some global predicate R iff the number of R-states of the protocol is finite.

Theorem 4: If a protocol is R-stabilizing and R-live for the same closed predicate R, then it is not R-finite.

Proof: Consider an R-live and R-finite protocol for some closed predicate R, where $R \neq false$; we show that this protocol is not R-stabilizing. Because the protocol is R-finite, it has a finite number of R-states. Thus, there is a positive integer K such that the number of messages in each channel is at most K at every R-state. Because the protocol is R-live and $R \neq false$, there is an infinite computation c that starts with an R-state. Because the initial state of c is an R-state and R is closed, every state in c is an R-state. But because there is a finite number of R-states at least one state s is repeated (infinitely often) along c. Consider an infinite computation d that first goes from s to s without executing any timeout actions as in c then repeats itself over and over. Because the protocol is R-live and s is an R-state, at least one process p sends at least one message to some process q in going from state s to state s along d. This implies that q receives at least one message from p in going from s to s along d. Assume that the sequence of messages sent from p to q in going from s to s along d is x, and the sequence of messages received by q from p in going from s to s along d is y. Also assume that $C_{pq} = z$ at s. There are two cases to consider.

Case 1. z = *the empty sequence of messages*: Consider a state s' that is identical to s except that $C_{pq} = (x)^{K+1}$. Also consider the infinite computation d' that starts at s' and executes the same actions as in computation d. It is straightforward to show (from the fact that z = empty sequence and so $x = y$) that state s' is repeated infinitely often along d' (as s is repeated infinitely often along d). But because C_{pq} has more than K messages at s', state s' is not an R-state. From the three facts that R is closed, s' is repeated infinitely often along d', and s' is not an R-state, we conclude that the maximal computation d' does not have an R-state and the protocol is not R-stabilizing.

Case 2. $z \neq$ *the empty sequence of messages*: Consider a state s' that is identical to s except that $C_{pq} = z; (x)^K$ and repeat the same proof as that of Case 1 to show that the protocol is also not R-stabilizing in this case. □

Theorem 4 states in effect that a stabilizing protocol has to have an infinite number of safe states. The next theorem suggests that this infinite number of safe states can be achieved by using unbounded counters in the processes of the protocol,

while keeping the number of messages in each channel bounded. In order to state this next theorem, we need the following definition.

A protocol is *R-restrained* for some global predicate R iff there is an integer K such that each channel has at most K messages in each R-state of the protocol.

Theorem 5: There are protocols that are R-stabilizing, R-live, and R-restrained for the same closed predicate R.

Proof: In the next section we give a protocol example that is R-stabilizing, R-live, and R-restrained for the same closed predicate R. □

3.3 Timeouts

The next theorem states that timeout actions are necessary for achieving stabilization in communication protocols.

Theorem 6: If a protocol is R-stabilizing, R-live, and R-restrained for the same closed predicate R, then there is at least one timeout action in one (or more) of its processes.

Proof: The proof is by contradiction. Assume that there is an R-stabilizing, R-live, and R-restrained protocol where the guard of each action is either of the form <local guard> or of the form <local guard> $\wedge$ <receive statement>. We refer to the local guard in the first form as a *send guard*. There are two cases to consider.

Case 1. There is a process p in the protocol such that the disjunction of all send guards in p = true: Consider an infinite computation in which only the actions with send guards in p are executed. Because the protocol is R-stabilizing, this computation has an R-state. Consider the suffix c of this computation that starts with the first R-state. Clearly, c itself is an infinite computation whose states are all R-states. Because the protocol is R-live, one of its processes (in this case process p) sends an infinite number of messages along c. From this fact and the fact that no process receives messages along c, the number of messages in at least one channel in the protocol grows beyond any bound along c. The two facts that every state in c is an R-state, and that the number of messages in some channel grows beyond any bound in the states of c contradict the assumption that the protocol is R-restrained.

Case 2. For each process p in the protocol, the disjunction of all send guards in p ≠ true: For each process p there is an assignment of values to the local variables of p such that the conjunction of all send guards in $p = false$. Consider the global state s of the protocol where all channels are empty and the local variables of every process are assigned values that falsify all the send guards in the process. Therefore, every action in each process in the protocol is disabled at state s. Now, consider the maximal computation that consists of the single state s. This computation, being maximal, has an R-state because the protocol is R-stabilizing; thus, s is an R-state. This fact along with the fact that the computation s is maximal contradict the assumption that the protocol is R-live. □

In summary, in order to design a stabilizing protocol, it is necessary to observe the following three conditions. First, the protocol should be live rather than terminating. Second, the protocol should have unbounded local variables in some of its processes. Third, the protocol should have timeout action(s) in some of its processes.

4 Verification of Stabilization

We present in this section a method for verifying that a given protocol is R-stabilizing for some closed predicate R. The method is suggested by the following theorem.

Theorem 7: A protocol is R-stabilizing iff it has a sequence $(R_1, \ldots, R_n)$ of global predicates that satisfies the following three conditions.

i. *Boundary:* $R_1 = true$ and $R_n = R$.
ii. *Closure:* Each R_i is closed.
iii. *Convergence:* Each R_i converges to R_{i+1}, for $i < n$.

Proof: *If part*: Because R is closed, it is sufficient to prove that each maximal protocol computation has an R-state. Consider an arbitrary maximal protocol computation. The first state in this computation is an R_1-state because $R_1 = true$. Also, the computation has an R_2-state because R_1 converges to R_2. Similarly, the computation has an R_3-state because R_2 converges to R_3, and so on. Thus the computation has an R_n-state, which is also an R-state.

Only if part: consider an R-stabilizing protocol. It is straightforward to show that the sequence of two predicates $(true, R)$ satisfies the above three conditions of boundary, closure, and convergence. □

We call a sequence of predicates that satisfies the above three conditions of boundary, closure, and convergence a *convergence stair* to R. Theorem 7 suggests that in order to prove that a given protocol is R-stabilizing, it is sufficient (and necessary) to exhibit and verify a convergence stair to R for the given protocol.

Example: Consider a communication protocol that consists of two processes p and q. Process p sends request messages to q which answers by reply messages. Each request message is of the form rqst(i) where i is a unique sequence number, and each reply message of the form rply(i) where i is the sequence number of the request message being replied to.

Initially the two channels C_{pq} and C_{qp} are empty; thus process p times out and sends its first request message. After sending each rqst(i) message, process p waits for a rply(i) message. Only after p receives rply(i), does it send the next rqst(i+1) message, and the cycle repeats. Process p can be defined as follows.

```
process p;
    var  pi : integer                    {* sequence number of last sent rqst *}
         , i : integer                   {* sequence number of last rcvd rply *}
    begin
         rcv rply(i) from q → if i ≠ pi  → skip
                              [] i = pi  → pi := pi + 1
                                         ;  send rqst(pi) to q
                              fi
      [] timeout C_pq = < > ∧ C_qp = < > → send rqst(pi) to q
    end
```

On the other side, when q receives a rqst(j) message from p, it answers back by sending rply(j) to p. Process q can be defined as follows.

```
process q;
    var  j : integer                     {* sequence number of last rcvd rqst *}
    begin
         rcv rqst(j) from p → send rply(j) to p
    end
```

Consider the following global predicate of the protocol:

$$\begin{aligned} R = \quad & (\text{for each rqst}(i) \text{ in } C_{pq}\text{: } i \leq pi) \\ \wedge \quad & (\text{for each rply}(i) \text{ in } C_{qp}\text{: } i \leq pi) \\ \wedge \quad & (\mid C_{pq} \mid + \mid C_{qp} \mid \leq 1) \end{aligned}$$

where $\mid C_{pq} \mid$ and $\mid C_{qp} \mid$ denote the number of messages in the two channels C_{pq} and C_{qp}, respectively. It is straightforward to show that the protocol is both R-live and R-restrained. First, all maximal protocol computations (including those that start at R-states) are infinite, and along each of them process p (and also q) sends an infinite number of messages; this proves the protocol R-live. Second, each of the two channels has at most one message at each R-state; this proves the protocol R-restrained.

In order to prove that the protocol is R-stabilizing, we need to exhibit and verify a convergence stair to R for the protocol. (Coming up with a convergence stair is an "art" in principle, but checking it is methodological as we demonstrate shortly.) Consider the following sequence of global predicates:

$$\begin{aligned} R_1 &= \mathit{true} \\ R_2 &= R_1 \wedge (\text{for each rqst}(i) \text{ in } C_{pq}\text{: } i \leq pi) \\ R_3 &= R_2 \wedge (\text{for each rply}(i) \text{ in } C_{qp}\text{: } i \leq pi) \\ R_4 &= R_3 \wedge (\mid C_{pq} \mid + \mid C_{qp} \mid \leq 1) \end{aligned}$$

To prove that these predicates constitute a convergence stair to R, we need to to prove that they satisfy the the three conditions of boundary, closure, and convergence. Proving the boundary condition is immediate since indeed $R_1 = \mathit{true}$ and $R_4 = R$.

Proving the closure condition consists of showing that if the protocol starts at an R_i-state, then executing any of the two actions in process p or the single action in process q yields an R_i-state, for $i = 2$, 3, and 4. For instance, if the protocol is at an R_2-state, then executing any of the two actions in process p will result in one of the following three cases:

- The values of variable pi and channel C_{pq} remain unchanged.
- The value of pi is incremented by 1, then a message rqst(pi) is added to C_{pq}.
- The value of pi remains unchanged, while a message rqst(pi) is added to C_{pq}.

In each of these cases, the resulting state is also an R_2-state. Similarly, executing the action of process q keeps the values of variable pi and channel C_{pq} unchanged and the resulting state is also an R_2-state.

Proving the convergence condition consists of proving three parts: R_1 converges to R_2, R_2 converges to R_3, and R_3 converges to R_4. We present here a proof for the first part; proofs for the other two parts are similar. Consider an arbitrary maximal protocol computation; we show that this computation has an R_2-state. Notice that each message that process p sends is of the form rqst(pi), and that the value of pi never decreases along the computation. Thus, all messages that are added to C_{pq} during the computation satisfy R_2. Therefore, the protocol state that results after the initial messages in C_{pq} are all received by q is an R_2-state. □

Two final comments concerning convergence stairs are in order. First, the number of predicates in a convergence stair significantly impacts the complexity of its verification. In particular, by choosing this number to be "large", verification of the convergence stair is divided into a large number of simple proofs. For instance, if we have chosen the convergence stair in the above example to be the sequence (R_1, R_4) only, then the verification would have been dominated by a complex proof of "R_1 converges to R_4". Second, one can always write the convergence stair of a stabilizing protocol as a "strengthening sequence" of predicates. In fact, it is straightforward to show that if (R_1, ... , R_n) is a convergence stair to R, then the strengthening sequence (S_1, ... , S_n) where $S_1 = R_1$ and for each i, $i > 1$, $S_i = S_{i-1} \wedge R_i$ is also a convergence stair to R. Note that the convergence stair in the above example is strengthening.

In the next two sections, we discuss how to redesign two well-known protocols to make them stabilizing; the sliding-window protocol is discussed in Section 5, and the two-way handshake is discussed in Section 6.

5 Stabilization of the Window Protocol

In this section, we describe how to make the sliding-window protocol stabilizing. We start in Section 5.1 by describing the original window protocol, and show that it is not stabilizing; this means that if the protocol ever reaches an unsafe state, due to some fault, it may stay within the unsafe states indefinitely and never return to its safe states. In Section 5.2, we present a stabilizing version of this protocol. Proving the stabilization of this version is sketched in Section 5.3.

5.1 Original Protocol

The window protocol consists of two processes p and q. Process p sends data messages of the form data(i) to q, where i is the message sequence number. Process q replies by sending acknowledgement messages of the form ack(j) to acknowledge receiving all the data messages data(0), data(1), ... , data(j–1).

In order to control the flow of data messages from p to q, process p is restricted to sending at most w data messages without receiving acknowledgements for any of them; the constant w, whose value is at least 1, is called the *window size*. To implement this control mechanism, process p is provided with two local variables:

na to store the sequence number of the next ack message to receive,
and ns to store the sequence number of the next data message to send.

The actions of process p guarantee that the following local predicate is satisfied at all times:

$$na \leq ns \leq na + w$$

Process p is defined as follows.

```
process p;
    const w : integer                          {* w ≥ 1 *}
    var   na, ns, i : integer                  {* na ≥ 0, ns ≥ 0, i ≥ 0 *}

    begin ns < na + w          → send data(ns) to q; ns := ns + 1
        [] rcv ack(i) from q   → if i > na  → na := i
                                 [] i ≤ na → skip
                                 fi
        [] timeout (ns > na) ∧ (¬ready) ∧ C_pq = <> ∧ C_qp = <>
                               → for i = na ,..., ns
                                 do send data(i) to q od

    end
```

Process p has three actions. In the first action, p sends a new data message after it checks that sending this message will not falsify the local predicate $ns \leq na + w$. In the second action, p receives an ack(i) message and updates na. In the third action, p detects that there are data messages that have been sent but not yet acknowledged and that no ack messages are forthcoming; it then timesout and resends all the unacknowledged data messages. (The boolean variable "ready" that appears in the global guard of the timeout action is a local variable of process q.)

On the other side, process q has a local variable

nr to store the sequence number of the next data message to receive,
or the sequence number of the next ack message to send.

Process q is defined as follows.

```
process q;
    var  nr, j: integer                          {* nr ≥ 0, j ≥ 0 *}
         ; ready : boolean

    begin rcv data(j) from p   → if j = nr → nr := nr + 1
                                 [] j ≠ nr → skip
                                 fi; ready := true
          [] ready             → send ack(nr) to p; ready := false

    end
```

Process q has two actions. In the first action, q receives a data message and checks whether it is the one it expects: if so, it increments nr by 1. In either case, q gets ready to reply by an ack message that is sent in the second action.

Correctness of this protocol can be established by showing that it satisfies the next two properties.

Safety: The following predicate R is closed in the protocol:

$$\begin{aligned} R = \quad & (na \leq nr) \wedge (nr \leq ns) \wedge (ns \leq na + w) \\ & \wedge (\text{for each data}(i) \text{ in } C_{pq} : i < ns) \\ & \wedge (\text{for each ack}(i) \text{ in } C_{qp} : i \leq nr) \end{aligned}$$

Progress: Along every maximal computation that starts with an R-state, the first action in process q is executed infinitely often when $j = nr$.

The safety property guarantees that if the protocol starts at an R-state, then it will continue to execute within the R-states indefinitely. An R-state from which the protocol can start executing is as follows:

$$\begin{aligned}&(na = 0) \wedge (nr = 0) \wedge (ns = 0) \wedge (\neg\text{ready})\\ &\wedge (C_{pq} = <>)\\ &\wedge (C_{qp} = <>).\end{aligned}$$

The progress property guarantees that as the protocol executes within the R-states, process q will receive "new" data messages infinitely often.

Unfortunately, this protocol is not R-stabilizing which means that if the protocol ever reaches a $(\neg R)$-state due to some fault, it may never return back to the R-states. This can be demonstrated by an example. Assume that an instance of the protocol where $w = 1$, reaches a $(\neg R)$-state that satisfies the following predicate

$$\begin{aligned}R_1 = \quad &(na = 2) \wedge (nr = 1) \wedge (ns = 3) \wedge (\neg\text{ready})\\ &\wedge (C_{pq} = <>)\\ &\wedge (C_{qp} = <>)\end{aligned}$$

The only action that is enabled at this R_1-state is the timeout action in process p. Executing this action leads to the state:

$$\begin{aligned}R_2 = \quad &(na = 2) \wedge (nr = 1) \wedge (ns = 3) \wedge (\neg\text{ready})\\ &\wedge (C_{pq} = <\text{data}(2)>)\\ &\wedge (C_{qp} = <>)\end{aligned}$$

The only action that is enabled at this state is the first action in process q; executing this action leads to the state

$$\begin{aligned}R_3 = \quad &(na = 2) \wedge (nr = 1) \wedge (ns = 3) \wedge (\text{ready})\\ &\wedge (C_{pq} = <>)\\ &\wedge (C_{qp} = <>)\end{aligned}$$

The only action that is enabled at this state is the second action in process q; executing this action leads to the state

$$\begin{aligned} R_4 = \quad & (na = 2) \wedge (nr = 1) \wedge (ns = 3) \wedge (\neg \text{ready}) \\ & \wedge (C_{pq} = <>) \\ & \wedge (C_{qp} = <\text{ack}(1)>) \end{aligned}$$

The only action that is enabled at this state is the second action in process p; executing this action leads back to the first state R_1, and the cycle repeats. Each of the states R_1 to R_4 is a $(\neg R)$-state because each of them violates the conjunct $(na \leq nr)$ in R. Therefore, the infinite computation $(R_1, R_2, R_3, R_4, R_1, R_2, \ldots)$ does not converge to an R-state and so the protocol is not R-stabilizing. Note that the protocol will "livelock" along this infinite computation as each process will send and receive infinitely often but no "real" progress will be made along the computation.

5.2 Stabilizing Protocol

In order to make the sliding-window protocol stabilizing, we adopt the following strategy: whenever a process discovers that it is "lagging behind" the other process, it tries to "catch up" with it. This strategy is accomplished by modifying the protocol as follows.

i. Whenever process p receives ack(i) where $i > ns$, p updates both na and ns: $na, ns := i, i$

ii. Each data message is provided with two sequence numbers instead of one:
data(j, k)
where j is the sequence number of the message as before,
and k is the value of na when the data message is sent.

iii. Whenever process q receives data(j, k) where $k > nr$, q updates nr :
$nr := k$.

The two processes of the modified protocol can be defined as follows.

```
process p
    const w : integer                          {* w ≥ 1 *}
    var na, ns, i : integer                    {* na ≥ 1, ns ≥ 0, i ≥ 0 *}

    begin na ≤ ns ∧ ns < na + w   → send data(ns, na) to q; ns := ns + 1
        [] rcv ack(i) from q      → if i > na ∧ i ≤ ns  → na := i
                                    [] i > na ∧ i > ns  → na, ns := i, i
                                    [] i ≤ na           → skip
                                    fi
        [] timeout (na ≠ ns) ∧ (¬ready) ∧ C_pq = <> ∧ C_qp = <>
                                  → if na ≤ ns ∧ ns ≤ na + w  → skip
                                    [] na > ns                → ns := na
                                    [] ns > na + w            → na := ns
                                    fi
                                    ; for i = na ,..., ns
                                    do send data(i, na) to q od
    end

process q
    var nr, j, k : integer                     {* nr ≥ 0, j ≥ 0, k ≥ 0 *}
        ; ready : boolean

    begin rcv data(j, k) from p   → if j = nr ∧ k ≤ nr  → nr := nr + 1
                                    [] j ≠ nr ∧ k ≤ nr  → skip
                                    []          k > nr  → nr := k
                                    fi; ready := true
          [] ready                → send ack(nr) to p; ready := false
    end
```

Correctness of this protocol can be established by showing that it satisfies the next two properties.

Safety: The following predicate S is closed in the protocol:

$$
\begin{aligned}
S = \quad & (na \le nr) \wedge (nr \le ns) \wedge (ns \le na + w) \\
& \wedge (\text{for each data}(i, j) \text{ in } C_{pq} : i \le ns \wedge j \le nr \wedge j < ns) \\
& \wedge (\text{for each ack}(i) \text{ in } C_{qp} : i \le nr)
\end{aligned}
$$

(Notice the strong similarities between this S of the modified protocol and the corresponding R of the original protocol.)

Progress: Along every maximal computation that starts with an S-state, the first action in process q is executed infinitely often when $(j = nr \wedge k \leq nr)$.

5.3 Proof of Stabilization

In order to show that the modified window protocol is S-stabilizing, we need, according to Theorem 7, to exhibit and verify a convergence stair to S. Consider the following strengthening sequence (S_1, S_2, S_3, S_4) of global predicates of the protocol:

$$
\begin{aligned}
S_1 = \;& \mathit{true} \\
S_2 = \;& S_1 \wedge (na \leq ns\,) \wedge (ns \leq na + w) \\
S_3 = \;& S_2 \wedge (nr \leq ns\,) \\
& \wedge (\text{for each data}(i, j) \text{ in } C_{pq} : i < ns \wedge j < ns) \\
& \wedge (\text{for each ack}(i) \text{ in } C_{qp} : i \leq ns) \\
S_4 = \;& S_3 \wedge (na \leq nr) \\
& \wedge (\text{for each data}(i, j) \text{ in } C_{pq} : j \leq nr) \\
& \wedge (\text{for each ack}(i) \text{ in } C_{qp} : i \leq nr)
\end{aligned}
$$

It is straightforward to show that (S_1, S_2, S_3, S_4) is a convergence stair to S for the protocol. A detailed proof of this fact appears in [14].

6 Stabilization of the Two-Way Handshake

We describe in this section how to make the two-way handshake protocol stabilizing. Our presentation follows the same steps of the previous section. In Section 6.1, we present the original protocol and show that it is not stabilizing. A stabilizing version of the protocol is presented in Section 6.2. Finally, we sketch a stabilization proof (i.e., a convergence stair) for the stabilizing version in Section 6.3.

6.1 Original Protocol

In order that process p "opens a connection" with a process q, it sends a *set* message to q. If the reply from q is a *rjct* message, the request for opening the connection has been rejected. If the reply from q is an *acpt* message, the connection has been opened. To close down the open connection, p sends a *rset* message to q which replies by an ack message. On receiving the *ack* message, p recognizes that the "connection is closed".

Each sent message from p to q is of the form msg(u), where u indicates the message type either *set* or *rset*. Each sent message from q to p is of the form msg(t), where t indicates the message type: *rjct, acpt,* or *ack.*

Process p has a local variable *sp* whose value is closed, wait, open, or done. These four values have the following informal meanings:

sp = closed means that the connection from p to q is closed.

sp = wait means that p is waiting for a reply for its request to open the connection.

sp = open means that the connection from p to q is open.

sp = done means that p is waiting for an ack for its request to close the connection.

Similarly, process q has a local variable "*sq*" whose value is either closed or open. These two values have the following informal meanings:

sq = closed means that either the connection from p to q is closed or soon will be closed.

sq = open means that either the connection from p to q is open or soon will be open.

The two processes p and q can be defined as follows.

```
process p
    var  sp : (closed, wait, open, done)
       ; t : (acpt, rjct, ack)

    begin     sp = closed          → send msg(set) to q; sp := wait
            [] sp = open           → send msg(rset) to q; sp := done
            [] rcv msg(t) from q   → if t = acpt    → sp := open
                                    [] t = rjct     → sp := closed
                                    [] t = ack      → sp := closed
                                    fi
            [] timeout (sp = wait ∨ sp = done) ∧ (C_pq = <> ∧ C_qp = <>
                                   → if sp = wait   → send msg(set) to q
                                    [] sp = done    → send msg(rset) to q
                                    fi
    end

process q
    var  sq : (closed, open)
       ; u : (set, rset)

    begin rcv msg(u) from p  → if  u = set   → send msg(acpt) to p; sq := open
                               []  u = set   → send msg(rjct) to p; sq := closed
                               []  u = rset  → send msg(ack) to p; sq := closed
                               fi
    end
```

Correctness of this protocol can be established by showing that it satisfies the following two properties.

Safety: The following predicate R is closed in the protocol.

$$
\begin{aligned}
R = \quad & (sp = \text{closed} \wedge sq = \text{closed} \wedge |C_{pq}| + |C_{qp}| = 0) \\
\vee \quad & (sp = \text{open} \wedge sq = \text{open} \wedge |C_{pq}| + |C_{qp}| = 0) \\
\vee \quad & (sp = \text{wait} \\
& \quad \wedge (\quad |C_{pq}| + |C_{qp}| = 0 \\
& \qquad \vee \quad (sq = \text{closed} \wedge C_{pq} = \langle \text{msg(set)} \rangle \wedge C_{qp} = \langle\rangle)
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad \vee \quad (sq = \text{closed} \quad \wedge\ C_{pq} = <> \wedge C_{qp} = <\text{msg(rjct)}>) \\
\qquad\qquad \vee \quad (sq = \text{open} \quad\ \wedge\ C_{pq} = <> \wedge C_{qp} = <\text{msg(acpt)}>) \\
\qquad\qquad) \\
\qquad) \\
\vee \quad (sp = \text{done} \\
\qquad\quad \wedge\ (\qquad |C_{pq}| + |C_{qp}| = 0 \\
\qquad\qquad\quad \vee \quad (sq = \text{open} \quad\ \wedge\ C_{pq} = <\text{msg(rset)}> \wedge C_{qp} = <>) \\
\qquad\qquad\quad \vee \quad (sq = \text{closed} \quad \wedge\ C_{pq} = <> \wedge C_{qp} = <\text{msg(ack)}>) \\
\qquad\qquad) \\
\qquad)
\end{array}
$$

Progress: Along every maximal computation that starts with an R-state, the first action in process p is executed infinitely often.

The safety property guarantees that if the protocol starts executing at an R-state (e.g., at the state sp = closed $\wedge$ sq = closed $\wedge$ $C_{pq} = <> \wedge C_{qp} = <>$), then it will continue to execute within the R-states indefinitely. Note that within the R-states, (sp = closed $\Rightarrow$ sq = closed) and (sp = open $\Rightarrow$ sq = open); in other words, the protocol can never reach a state in which $sp \neq sq$ while sp is either closed or open. The progress property guarantees that as the protocol executes within the R-states, process p will request the connection to be opened infinitely often.

This protocol is not R-stabilizing. For example, consider the infinite computation

$$(R_1, R_2, R_3, R_4, R_5, R_6, R_1, R_2, \ldots)$$

where

$$
\begin{array}{l}
R_1 = \quad (sp = \text{closed}) \wedge (sq = \text{open}) \\
\qquad \wedge\ (C_{pq} = <\text{msg(rset)}>) \wedge (C_{qp} = <\text{msg(acpt)}>)
\end{array}
$$

$$
\begin{array}{l}
R_2 = \quad (sp = \text{wait}) \wedge (sq = \text{open}) \\
\qquad \wedge\ (C_{pq} = <\text{msg(rset); msg(set)}>) \wedge (C_{qp} = <\text{msg(acpt)}>)
\end{array}
$$

$$R_3 = (sp = \text{wait}) \wedge (sq = \text{closed}) \wedge (C_{pq} = \langle \text{msg(set)} \rangle) \wedge (C_{qp} = \langle \text{msg(acpt); msg(ack)} \rangle)$$

$$R_4 = (sp = \text{open}) \wedge (sq = \text{closed}) \wedge (C_{pq} = \langle \text{msg(set)} \rangle) \wedge (C_{qp} = \langle \text{msg(ack)} \rangle)$$

$$R_5 = (sp = \text{done}) \wedge (sq = \text{closed}) \wedge (C_{pq} = \langle \text{msg(set); msg(rset)} \rangle) \wedge (C_{qp} = \langle \text{msg(ack)} \rangle)$$

$$R_6 = (sp = \text{done}) \wedge (sq = \text{open}) \wedge (C_{pq} = \langle \text{msg(rset)} \rangle) \wedge (C_{qp} = \langle \text{msg(ack); msg(acpt)} \rangle)$$

Because each of the states R_1 to R_6 is a $(\neg R)$-state, the protocol does not converge to R along this computation. Thus, the protocol is not R-stabilizing.

6.2 Stabilizing Protocol

The above protocol is both R-finite and R-live; thus it cannot be R-stabilizing according to Theorem 4. This observation suggests that in order to make this protocol stabilizing, we should get rid of its "finiteness". This can be accomplished by adding an unbounded sequence number to each message. Process p keeps track of the current sequence number in a local variable cp whose value is incremented by 1 whenever p receives a rjct or an ack message. Process q replies to each message from p with a message that has the same sequence number as the received message. Whenever process p receives an unexpected message or a message with an unexpected sequence number, it discards the message.

The two processes p and q for this protocol can be defined as follows.

process p
  var sp : (closed, wait, open, done)
    ; t : (acpt, rjct, ack)
    ; cp, c : **integer**

  begin sp = closed $\rightarrow$ **send** msg(set, cp) **to** q; sp := wait
    [] sp = open $\rightarrow$ **send** msg(rset, cp) **to** q; sp := done
    [] **rcv** msg(t, c) **from** q

```
                    → if  cp ≠ c                          → skip
                       [] cp = c ∧ t = acpt ∧ sp = wait → sp := open
                       [] cp = c ∧ t = acpt ∧ sp ≠ wait → skip
                       [] cp = c ∧ t = rjct ∧ sp = wait →
                                                sp, cp := closed, c + 1
                       [] cp = c ∧ t = rjct ∧ sp ≠ wait → skip
                       [] cp = c ∧ t = ack ∧ sp = done →
                                                sp, cp := closed, c + 1
                       [] cp = c ∧ t = ack ∧ sp ≠ done → skip
                       fi
          [] timeout (sp = wait ∨ sp = done) ∧ C_pq = < > ∧ C_qp = < >
                    → if  sp = wait → send msg(set, cp) to q
                       [] sp = done → send msg(rset, cp) to q
                       fi
    end

process q
    var  sq : (closed, open)
       ; u : (set, rset)
       ; d : integer

    begin rcv msg(u, d)  from p
                    → if u = set →  send msg(acpt, d) to p; sq := open
                       [] u = set →  send msg(rjct, d) to p; sq := closed
                       [] u = rset→  send msg(ack, d) to p; sq := closed
                       fi
    end
```

Correctness of this protocol can be established by showing that it satisfies the next two properties.

Safety: The following predicate S is closed in the protocol:

$$
\begin{aligned}
S = \quad & (sp = \text{closed} \ \wedge \ sq = \text{closed} \ \wedge \ |C_{pq}| + |C_{qp}| = 0) \\
\vee \quad & (sp = \text{open} \quad \wedge \ sq = \text{open} \quad \wedge \ |C_{pq}| + |C_{qp}| = 0) \\
\vee \quad & (sp = \text{wait}
\end{aligned}
$$

$$
\begin{array}{l}
\quad \wedge \ (\quad |C_{pq}| + |C_{qp}| = 0 \\
\qquad \vee \ (sq = \text{closed} \quad \wedge \ C_{pq} = \langle \text{msg(set, } cp) \rangle \wedge C_{qp} = \langle\rangle) \\
\qquad \vee \ (sq = \text{closed} \quad \wedge \ C_{pq} = \langle\rangle \wedge C_{qp} = \langle \text{msg(rjct, } cp) \rangle) \\
\qquad \vee \ (sq = \text{open} \quad \wedge \ C_{pq} = \langle\rangle \wedge C_{qp} = \langle \text{msg(acpt, } cp) \rangle) \\
\quad \ ) \\
) \\
\vee \ (sp = \text{done} \\
\quad \wedge \ (\quad |C_{pq}| + |C_{qp}| = 0 \\
\qquad \vee \ (sq = \text{open} \quad \wedge \ C_{pq} = \langle \text{msg(rset, } cp) \rangle \wedge C_{qp} = \langle\rangle) \\
\qquad \vee \ (sq = \text{closed} \quad \wedge \ C_{pq} = \langle\rangle \wedge C_{qp} = \langle \text{msg(ack, } cp) \rangle) \\
\quad \ ) \\
)
\end{array}
$$

Progress: Along every maximal computation that starts with an S-state, the first action in process p is executed infinitely often.

6.3 Proof of Stabilization

In order to show that the modified two-way handshake protocol is S-stabilizing, we need to exhibit and verify a convergence stair to S. Consider the following strengthening sequence $(S_1, S_2, S_3, S_4, S_5)$ of global predicates of the protocol:

$$
\begin{array}{l}
S_1 = \mathit{true} \\
S_2 = S_1 \wedge (\text{for each msg}(t, c) \text{ in } C_{pq} : c \leq cp) \\
S_3 = S_2 \wedge (\text{for each msg}(t, c) \text{ in } C_{qp} : c \leq cp) \\
S_4 = S_3 \wedge (0 \leq (|C_{pq}| + |C_{qp}| + f) \leq 1) \\
S_5 = S
\end{array}
$$

where f is a function that assigns to each global state s of the protocol a value 0 or 1 as follows:

$$
\begin{array}{lll}
f(s) = 0 & \text{if} & sp = \text{wait} \ \vee \ sp = \text{done} \\
\quad\ 1 & \text{if} & sp = \text{closed} \ \vee \ sp = \text{open}
\end{array}
$$

It is straightforward to show that this sequence of global predicates is a convergence stair to S for the protocol. A detailed proof of this fact appears in [14].

7. Concluding Remarks

The objective of this paper is two-fold: first, to motivate and promote the idea of stabilization in the area of protocol design, and second, to plant the seeds for a theory for designing and verifying stabilizing protocols. In order to achieve the first goal, we have argued (in Section 1) that stabilization is a general method for countering coordination loss regardless of its cause. We have also demonstrated that stabilization is not costly to achieve: the stabilizing versions of the sliding-window protocol (in Section 5) and of the two-way handshake (in Section 6) closely resemble their respective nonstabilizing versions (except for the fact that the stabilizing versions are much more robust in face of coordination loss).

In order to achieve our second goal, we have presented three important characteristics of stabilizing protocols (in Section 3), and outlined a method for proving protocol stabilization (in Section 4). We have demonstrated the applicability of this method by applying it to prove all the stabilizing protocol examples in the paper.

One of the characteristics of stabilizing protocols that we have identified in Section 3 needs further discussion. Theorem 4 states (roughly) that each stabilizing protocol must have one or more processes with unbounded local variables. Confronted with this result, one can adopt either of two legitimate positions. On one hand, one can argue that a variable with a large number of bits, 64 say, is unbounded for all practical purposes. Thus, it is possible, for example, to implement all the stabilizing protocols in this paper, but use in them large-size variables instead of the required unbounded ones. The resulting protocols, to be exact, are not stabilizing, but they are close enough to being so; hence they can counter most forms of coordination loss.

On the other hand, one can argue that exact stabilization is not achievable in practice, and so we should strive for notions that are not as demanding as stabilization but are still close enough that the resulting protocols can counter coordination loss effectively. After we communicated this result (Theorem 4) to others, two new notions have surfaced: probabilistic stabilization [1], and pseudo-stabilization [5]. Both these notions can be fully achieved in practice, and both of them provide communication protocols with a high degree of robustness in face of coordination loss.

Besides the examples in this paper, there are other examples of elegant stabilizing protocols; see for instance [1, 10, 14]. This confirms our belief that stabilization

is a very fundamental idea with wide applicability; it deserves thorough attention and detailed consideration.

Acknowledgements. We are thankful to Anish Arora, Edsger W. Dijkstra, Simon Lam, Ming Liu, and Jayadev Misra for listening to our ideas and their encouragement, and to K. F. Carbone for preparing the manuscript.

References

[1] Y. Afek and G. Brown, "Self-stabilization of the alternating-bit protocol," *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pp. 80–83, 1989.

[2] F. Bastani, I. Yen, and I. Chen, "Class of inherently fault-tolerant distributed programs," *IEEE Trans on Software Engineering*, Vol. 14, No. 10, pp. 1432–1442, 1988.

[3] G. Brown, M. Gouda, and C. Wu, "Token systems that self-stabilize," *IEEE Trans. on Computers*, Vol. 36, No. 6, pp. 845–852, 1989.

[4] J. Burns and J. Pachl, "Uniform self-stabilizing rings," the *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 2, pp. 330–344, 1989.

[5] J. Burns, M. Gouda, and R. Miller, "Stabilization and Pseudo-stabilization," Technical Report TR-90-13, Dept. of Computer Sciences, Univ. of Texas at Austin, May 1990. Also submitted for journal publication.

[6] E. Dijkstra, EWD 391, "Self-stabilization in spite of distributed control," 1973; reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer Verlag, pp. 41–46, 1982.

[7] E. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Comm. of the ACM*, Vol. 17, pp. 643–644, 1974.

[8] M. Gouda and M. Evangelist, "Convergence/response tradeoffs in concurrent systems," Technical Report TR-88-39, Dept. of Computer Sciences, Univ. of Texas at Austin, 1988; also submitted to the *ACM Trans. on Computer Systems and Languages*, 1989.

[9] M. Gouda, R. Howell, and L. Rosier, "The instability of self-stabilization," presented at the MCC workshop on self-stabilization, August 1989. Also, *Acta Informatica*, to appear, 1990.

[10] M. Gouda, N. Maxemchuk, U. Mukherji, and K. Sabnani, "Delivery and Discrimination: The Seine Protocol," *Proceedings of the SIGCOMM '88 Symposium*, 1988, pp. 292–302.

[11] S. Katz and K. Perry, "Self-stabilizing extensions for message passing systems," presented at the MCC workshop on Self-Stabilization, August 1988.

[12] L. Lamport, "Solved Problems, Unsolved Problems, and Non-Problems in Concurrency," Invited Address, *Proceedings of the Third ACM Symposium on Principles of Distributed Computing* 1984, pp. 1–11.

[13] L. Lamport, "The Mutual Exclusion Problem: Part II—statement and solutions," *JACM*, Vol. 33, No. 2, pp. 327–348, April 1986.

[14] N. Multari, *Towards a theory for self-stabilizing protocols*, Ph.D. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1989.