Byzantine Generals in Action: Implementing Fail-Stop Processors

FRED B. SCHNEIDER
Cornell University

A *fail-stop processor* halts instead of performing an erroneous state transformation that might be visible to other processors, can detect whether another fail-stop processor has halted (due to a failure), and has a predefined portion of its storage that will remain unaffected by failures and accessible to any other fail-stop processor. Fail-stop processors can simplify the construction of fault-tolerant computing systems. In this paper, the problem of approximating fail-stop processors is discussed. Use of fail-stop processors is compared with the state machine approach, another general paradigm for constructing fault-tolerant systems.

Categories and Subject Descriptors: B.1.3 [**Control Structures and Microprogramming**]: Control Structure Reliability, Testing and Fault-Tolerance—*redundant design*; B.3.4 [**Memory Structures**]: Reliability, Testing and Fault-Tolerance—*redundant design*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and serviceability*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart, fault-tolerance*

General Terms: Reliability

Additional Key Words and Phrases: Byzantine Generals, fail-stop, fail-fast

1. INTRODUCTION

Designing and programming a fault-tolerant computing system is a difficult task. Due to a failure, a processor might exhibit arbitrary behavior, resulting in erroneous outputs or in the destruction of critical state information. Even when multiple processors are used, a malfunctioning processor can cause problems by causing erroneous state information to be visible to other processors. This could have disastrous consequences if these processors take actions based on such information. Clearly, using processors that take into account the following property avoids these difficulties.

Halt on Failure Property. A processor will halt instead of performing an erroneous state transformation that will be visible to other processors.

Processors that merely halt in response to failures, however, are not sufficient for implementing systems whose correctness criteria involve generating outputs

This work was supported in part by NSF Grant MCS-8103605.
Author's address: Department of Computer Science, Cornell University, Ithaca, NY 14853.

in a timely manner. Tasks that were being run on a halted (malfunctioning) processor must be continued, if real-time constraints are to be met. This means that processors must also satisfy a second property:

Failure Status Property. Any processor can detect when any other processor has failed, and therefore has halted.

This permits other processors to assume the tasks of a failed processor. Of course, there are obvious limitations to this strategy—there must be sufficient processing capacity in the smaller system for it to be able to continue performing all of its tasks in a timely manner.

Finally, in order to continue a task that was running on a failed processor, the state of that task must be available to the processor that is to continue it. This can be accomplished by using *stable storage*—storage that is unaffected by any failure and is accessible to every processor. Thus, we require processors to satisfy a third property:

Stable Storage Property. The storage of a processor is partitioned into stable storage and volatile storage. The contents of stable storage are unaffected by any failure and can always be read by any processor. The contents of volatile storage are not accessible to other processors and are lost as a result of a failure.

A *fail-stop processor* is a processor that satisfies the Halt on Failure Property, the Failure Status Property, and the Stable Storage Property. To construct a fault-tolerant computing system that can tolerate up to f failures for an application requiring N processors (assuming there are no failures), $N + f$ fail-stop processors are employed. Whenever a fail-stop processor in this system halts, the other fail-stop processors detect this and partition its work among themselves by reading from its stable storage.

Fail-stop processors simplify, but do not completely solve, the problem of building fault-tolerant computing systems. The problem is simplified because it is unnecessary to cope with arbitrary behavior and corrupted state information. However, it is still necessary to design programs that make infrequent references to stable storage, which is likely to be expensive and slow, while saving enough state information there so that a task can be continued only by accessing stable storage.

Perhaps the strongest argument for investigating the implementation of fail-stop processors is that most protocols for implementing fault-tolerant systems assume models where processors are either fail-stop processors or their equivalent.[1] In some models, instead of the Failure Status Property, "timeouts" are used to detect failures. However, use of timeouts requires the further assumption that processor clocks are synchronized. Otherwise, two processors might not agree that a third has halted, which can have disastrous consequences if the third processor has not. In other models, the Stable Storage Property is not assumed; instead, state information is replicated at other processors. However, this turns out to be just an approximation of the Stable Storage Property.

[1] The only work we know of that does not involve fail-stop or stronger assumptions about processor failures is described in [6], [7], and [9].

Real processors do not satisfy the Halt on Failure, Failure Status, or Stable Storage properties. In fact, most real processors are not even good approximations of fail-stop processors. This is disappointing in light of the number of protocols written that assume processors are fail-stop. In this paper, we develop an implementation of a fail-stop processor approximation. This serves two purposes: First, it gives a feel for the cost and complexity of implementing fail-stop processors. Comparison of protocols that assume fail-stop processors with protocols that make weaker assumptions (e.g., Byzantine Agreement protocols [1, 8, 10, 11]) is then possible. Secondly, our fail-stop processor approximation is a first step toward a practical realization of fail-stop processors.

We must be content with only an approximation of a fail-stop processor because it is impossible to implement a completely fault-tolerant computing system using a finite amount of hardware. With a finite amount of hardware, a finite number of failures could disable all error detection facilities and thereby allow behavior that violates the properties that define a fail-stop processor. Our approximation is for a *k-fail-stop processor*—a collection of processors and memories that behaves like a fail-stop processor unless $k + 1$ or more failures occur within its components. Obviously, as k approaches infinity, a k-fail-stop processor becomes closer to the ideal it approximates.

We proceed as follows: Section 2 contains the design and correctness argument for a k-fail-stop processor. Section 3 concerns techniques to combine a collection of k-fail-stop processors into a fault-tolerant computing system. In Section 4, the fail-stop processor approach is contrasted with the state machine approach, another general technique for constructing fault-tolerant computing systems. Finally, Section 5 contains a discussion of other ways to approximate fail-stop processors and considers some open problems.

2. APPROXIMATING FAIL-STOP PROCESSORS

A k-fail-stop processor FSP is implemented by a collection of real processors, each with its own storage, that are interconnected by a communications network. Failures that could result in another fail-stop processor reading the results of an erroneous state transformation are detected by voting; the effects of other failures are masked. The implementation consists of:

(1) $k + 1$ *p-processes* (p for program), each running on its own processor; let $p(FSP) = \{p_1, p_2, \ldots, p_{k+1}\}$ be this set of processes;
(2) $2k + 1$ *s-processes* (s for storage), each running on a different processor; let $s(FSP) = \{s_1, s_2, \ldots, s_{2k+1}\}$ be this set of processes.

The question of allocating processors to processes is discussed in Section 3.

A program running on FSP is run by each of the $k + 1$ p-processes in $p(FSP)$. Failures that should cause FSP to halt are detected by comparing results when each p-process in $p(FSP)$ writes to stable storage in FSP, since reading stable storage is the only way the effects of a failure can be made visible. Because p-processes run on different processors, they fail independently. Provided fewer than $k + 1$ failures occur in the processors running p-processes, if any failure that should cause FSP to be halted occurs then there will be a disagreement in the write requests made by its p-processes. This disagreement will be detected by its s-processes.

A copy of the contents of the stable storage of *FSP* is stored by each of the s-processes in $s(FSP)$. Since there are $2k + 1$ s-processes, each running on a different processor, after as many as k failures in these processors a majority of them will still be able to access correct values. Of course, this presupposes that each correctly functioning s-process updates it state whenever a write is performed to stable storage; a protocol for this is described below.

The only way a p-process can access stable storage is by sending messages to s-processes. These messages m contain the following information:

m.time	the time at which this request was made according to the local clock on the processor running the requesting p-process;
m.rectime	the time this request was received according to the local clock on the processor running the s-process receiving the request;
m.type	depending on the request, either "read" or "write";
m.var	the variable in stable storage to be written if *m.type* = write; the variable in stable storage to be read if *m.type* = read;
m.val	the value to be written if *m.type* = write.

We make the following assumption about the communications network.

Network Reliability Assumption. Messages are delivered uncorrupted and the process $orig(m)$, originating a message m, can be authenticated by its receiver.

In theory, satisfying this assumption requires that there be $2k + 1$ independent and direct communication links between each p-process and s-process. Independent channels allow the majority value to be taken as the value of the message—this value will be correct provided fewer than $k + 1$ failures occur; direct channels allow authentication of message origin. In practice, a packet-switching network can be made to approximate the Network Reliability Assumption. Checksums and message retransmission are used to ensure that messages are delivered uncorrupted with high probability; digital signatures implement authentication (with high probability).

An s-process for a k-fail-stop processor FSP_i in a system with up to N k-fail-stop processors $FSP_1, FSP_2, \ldots, FSP_N$ executes the program in Figure 1. There, "choose (m, M)" stores an arbitrary element from M into m, and "CLOCK" evaluates to the current time according to the processor's local clock. "*Stable* $[\cdots]$" is the copy of stable storage maintained by this s-process. In addition, we require that when a p-process p_j makes a request to stable storage of FSP_i, it disseminates the request in a way that satisfies IC1 and IC2:

IC1. If p_j is nonfaulty, then every nonfaulty s-process s_u in $s(FSP_i)$ receives the request within δ seconds (as measured on s_u's clock).

IC2. If s-processes s_u and s_v in the same k-fail-stop processor are nonfaulty, then both of them agree on every request from p_j.

Condition IC1 ensures that all s-processes receive a message within a bounded length of time δ whenever a request is made by a nonfaulty p-process. Condition IC2 ensures that all s-processes will agree on a request, even if the p-process making the request is faulty. IC2 is necessary because a faulty p-process might make different requests to two different s-processes. The copies of stable storage

```
owner := i;  failed := false;
do true → /* major loop */
  for s := 1 to N
    T := CLOCK;
    D := bag of requests m delivered such that:
        orig(m) ∈ p(FSP_s) ∧ (m.type = read ∨ m.type = write)
    do D ≠ Φ →
      minT := minimum value of m.time in D;
      minRecT := minimum value of m.rectime such that:
        m ∈ D ∧ m.time = minT
      if minRecT < T − δ →
        M := bag of requests m such that m ∈ D ∧ m.time = minT;
        D := D − M;
        if (∀m:m ∈ M: m.type = read) →
          do M ≠ Φ → choose (m, M); M := M − {m};
                     send Stable [m.var] to orig(m)
          od
        ▯ (∀m, m':distinct m, m' ∈ M:
          m = m' ∧ m.type = write ∧ orig(m) ≠ orig(m')) ∧
          |M| = k + 1 ∧ s = owner ∧ ¬ failed) →
                              choose (m, M);
                              Stable [m.var] := m.val
          ▯ otherwise → if s = owner ∧ ¬ failed →
                              failed := true;
                              forall d ∈ p(FSP_i) send "halt" to d
                            ▯ otherwise → skip
                            fi
            fi
          ▯ otherwise → skip
      fi
    od
  rof
od
```

Fig. 1. Program for s-process in FSP_i.

maintained by these s-processes could then become inconsistent if one s-process performed an update and another didn't.

Finally, we require:

IC3. For each k-fail-stop processor FSP, the clocks of all processors running p-processes in $p(FSP)$ are synchronized.

IC3 ensures that if a request is made by one nonfaulty p-process in $p(FSP)$ at time T on its clock, then, since all processes in $p(FSP)$ are running the same program, the same request is made by each other nonfaulty p-process at time T on its local clock.

A number of protocols for establishing IC1 and IC2—called *interactive consistency* or *Byzantine Agreement*—have been developed [1, 8, 10, 11]. In these protocols, δ is based on message delivery time and on maximum difference in the clock speeds of any two correctly functioning processors running s-processes. At least $f + 1$ processors are required to handle up to f faults when messages can be authenticated [2]. Since our implementation of a k-fail-stop processor need

tolerate at most k failures and involves $2k + 1$ processors for running s-processes, IC1 and IC2 can be achieved.

A protocol for achieving clock synchronization, as required by IC3, is described in [3]. The protocol also requires at least $f + 1$ processors to handle up to f faults when messages can be authenticated. As described above, for a single k-fail-stop processor IC3 requires the $k + 1$ processors running p-processes to have synchronized clocks. Thus, IC3 can be achieved.

2.1 Stable Storage Property

To show that the Stable Storage Property holds for our implementation, we must show three things:

(1) that a majority of the copies of stable storage are correct and identical as long as k or fewer failures occur;
(2) that a nonfaulty fail-stop processor can write to its stable storage; and
(3) that any fail-stop processor can read from the stable storage of any fail-stop processor *FSP* (including its own) regardless of whether *FSP* has halted in response to a failure.

The proof that our implementation satisfies part (1) of the Stable Storage Property is as follows. All p-processes run the same program, so all nonfaulty p-processes make the same requests to stable storage. Since by IC3 the clocks of all nonfaulty p-processes are synchronized, the nonfaulty p-processes will all make requests at the same time according to their local clocks. By IC1 and IC2, if a nonfaulty s-process s_u receives the first such request by time T_r on its clock, it will receive all such requests by time $T_r + \delta$ on its clock.

Thus, no request made at time T and received by an s-process at time T_r will be added to D after $T_r + \delta$, and all s-processes will have the same request (of time T) in their respective D bags by time $T_r + \delta$. No request made at time T will be copied from D to M by an s-process before $T_r + \delta$ (on its clock) because of the way the s-process program is coded. Thus, the contents of M at each nonfaulty s-process will be the same as at every other nonfaulty s-process. Execution of the s-process program in Figure 1 is completely determined by the contents of M. Consequently, each nonfaulty s-process executes identically, so the nonfaulty s-processes will update their copies of stable storage in the same way. Since there are $2k + 1$ s-processes, at least $k + 1$ will be nonfaulty. Therefore, a majority of the s-processes will update their copies of stable storage.

We now turn to part (2) of the Stable Storage Property. Above, we argued that all nonfaulty s-processes perform the same changes to stable storage and that therefore a majority of the copies of stable storage are correct and identical. From the program in Figure 1 it is clear that a write operation attempted by fail-stop processor FSP_i is not performed by an s-process unless all $k + 1$ p-processes in $p(FSP_i)$ request it. Moreover, write operations requested by other fail-stop processors are ignored because of the $s = owner$ conjunct in the guard. Clearly, if all $k + 1$ p-processes request an operation, then either none or all have failed in a way that makes erroneous state information—the value being written—visible to other processes. If all have failed then arbitrary behavior is permitted because there have been $k + 1$ failures. If none have failed then the write will be performed by the nonfaulty s-processes.

Table I. Interface Between s-process and p-process

For p-process p_j in FSP_i to write to stable storage in FSP_i: initiate a Byzantine Agreement for the write request with all the s-processes in $s(FSP_i)$.
For p-process p_j in FSP_i to read from stable storage in FSP: (1) broadcast the read request to all the s-processes in $s(FSP)$, (2) use the value received from at least $k + 1$ different s-processes.
For p-process p_j in FSP_i to determine if FSP has halted due to a failure: read the variable *failed* from the stable storage in FSP.

Finally, for part (3) it suffices to note that a read operation attempted by FSP_i should result in identical responses being sent by nonfaulty s-processes to each p-process in $p(FSP_i)$. If fewer than $k + 1$ failures occur then at least $k + 1$ correct values (from a total of $2k + 1$) will be received. Thus, by taking the majority value of the responses, a p-process can obtain the correct value for the variable being read.

2.2 Halt on Failure Property

To detect a failure, during each (major) loop iteration it suffices for each s-process to check the write requests it has received, since spurious writes are the only means by which the effects of a failure can be made visible to another process. If

(a) exactly one write request from each of the $k + 1$ p-processes has been received, and
(b) all the requests are identical,

then either all or none of the $k + 1$ p-processes that make up FSP are malfunctioning. (Again, the case where all $k + 1$ p-processes are faulty need not concern us here because the definition of a k-fail-stop processor allows it to display arbitrary behavior under these circumstances.) If write requests from only some of the $k + 1$ p-processes in $p(FSP)$ are received, then the p-processes in that fail-stop processor are all sent a "halt" message, and the stable storage variable *failed* is set true. (Correctly functioning p-processes will halt upon receiving a "halt" message from at least $k + 1$ s-processes.) Once *failed* is true, the values of the variables in the nonfaulty s-processes don't change since the conjunct "$\neg$ *failed*" guards the assignment statement.

2.3 Failure Status Property

The Failure Status Property is implemented by the variable *failed.* Any process can obtain the value of *failed* at any time by reading it in stable storage. Thus, FSP can determine if FSP_i has halted due to a failure, by reading *failed* from FSP_i's stable storage

This completes our implementation of a k-fail-stop processor approximation. The interface between the s-processes and the p-processes is summarized in Table I.

3. ASSIGNING PROCESSES TO PROCESSORS

Consider an application that requires N fail-stop processors to meet its response-time constraints, if no failures occur. For this implementation to be able to tolerate up to k failures, $N + k$ independent k-fail-stop processors are required. Use of independent fail-stop processors ensures that a single failure will cause at most one fail-stop processor to halt. Thus, provided k or fewer failures occur, there will always be at least N fail-stop processors available to run the application.

A naive implementation of such a computing system will use $3k + 1$ processors—$k + 1$ processors for p-processes and $2k + 1$ processors for s-processes—for each k-fail-stop processor, resulting in a total of $(N + k) \times (3k + 1)$ processors. Recall, however, that programs for fail-stop processors will be structured to make minimal use of stable storage. Therefore, it would be wasteful to dedicate an entire processor to running an s-process for a single k-fail-stop processor.

Suppose a single processor is able to run S s-processes without delaying any of the p-processes that interact with those s-processes. Now, we require only $\lceil (N + k)/S \rceil \times (2k + 1)$ processors to run the s-processes and $N \times (k + 1)$ processors for p-processes. Clearly, this is a decrease in the number of processors over that required for the naive implementation. However, now the $N + k$ k-fail-stop processors are not independent—s-processes of different fail-stop processors share processors. Fortunately, this is not a problem because s-processes are replicated $2k + 1$-fold. Given that we are prepared to tolerate at most k failures, even if $S = N + k$, so that there are only $2k + 1$ processors running the s-processes for all $N + k$ k-fail-stop processors and all of the failures occur in these processors, there will still be $k + 1$ s-processes running on nonfaulty processors for each of the $N + k$ k-fail-stop processors. Thus, the majority of the s-processes will be running on nonfaulty processors.

When a fail-stop processor halts, all of the nonfaulty processors running its p-processes—up to $k + 1$ processors—halt. It is unlikely that all of these processors are, in fact, faulty. In order to recover nonfaulty processors that were associated with a fail-stop processor in which there was a failure, the following scheme can be used.

Processor Recycling Scheme. Processors are partitioned into three groups: *active*, *unavailable* and *available*. All processors are initially assigned to the available group. As fail-stop processors are configured, processors are removed from the available group and placed in the active group. Whenever a fail-stop processor halts, those processors that were running its p-processes are assigned to the unavailable group. Processors in the unavailable group run diagnostics, and any processor that passes its diagnotics is reassigned to the available group.

The Processor Recycling Scheme reduces the cost of a failure. Without it, a failure causes loss of all of the processors running p-processes for the fail-stop processor in which the failure was detected. With the Processor Recycling Scheme, only processors that are unable to pass their diagnostic tests remain unavailable. The others are reconfigured into new fail-stop processors.

4. OTHER APPROACHES TO FAULT-TOLERANCE

Our implementation of a k-fail-stop processor is an application of the *state machine approach*, a general approach for constructing distributed programs first

described in [5], and later extended for environments in which failures could occur in [6, 7] and [13]. Given any program, a distributed version that can tolerate up to k failures can be constructed by running that program on $2k + 1$ processors connected by a communications network in which message origins can be authenticated.[2] Byzantine agreement is used to ensure that each instance of the program sees the same inputs; majority voting is used to determine the output of the computation.

Consider an application that requires N processors to run and meet its real-time constraints. Using the state machine approach directly, a total of $N \times (2k + 1)$ processors are required to implement a system that can tolerate up to k faults. Each additional "k-fault-tolerant processor" costs $2k + 1$ real processors. Contrast this with the cost when the fail-stop processor approach is used where S s-processes can share a single processor. A total of $(N + k) \times (k + 1) + \lceil (N + k)/S \rceil \times (2k + 1)$ real processors are required and each additional k-fail-stop processor costs (approximately) $(k + 1) + (2k + 1)/S$ processors. Thus, there are cases where, to achieve the same degree of fault-tolerance, the fail-stop processor approach requires fewer processors than the state machine approach.

However, direct use of the state machine approach on an application has other advantages over the fail-stop processor approach:

- When using the state machine approach, there is no need to divide the program state between volatile and stable storage. Also, there is no need to develop recovery protocols that reconstruct the state of the program based on the contents of stable storage.
- When using the fail-stop processor approach, additional response time is incurred when a task is moved from one fail-stop processor to another. Such delays are not incurred when the state machine approach is used, since all failures are masked. Thus, it might not be possible to use the fail-stop processor approach for applications with tight timing constraints.
- When using the fail-stop processor approach, an expensive Byzantine Agreement must be performed for every access to stable storage; with the state machine approach, Byzantine Agreement need only be performed for every input read. Thus, if reading input is a relatively infrequent event, the state machine approach will expend less resources in executing Byzantine Agreement protocols.

5. DISCUSSION

Our k-fail-stop processor approximation is based on the construction of a reliable kernel (using the s-processes) that supports stable storage and detects failures. The kernel is reliable because it is replicated $2k + 1$-fold so that the effects of up to k failures are masked. Applications to be run on a k-fail-stop processor approximation are replicated only $k + 1$-fold, which is cheaper but sufficient only to detect errors and not to mask them.

One way to approximate fail-stop processors is described in this paper; a more expensive approach was described in [12]. There are undoubtedly other ways to approximate fail-stop processors. For example, disks are sometimes considered acceptable approximations of stable storage; a triple-redundant bus can be used

[2] If authentication is not possible then $3k + 1$ processors are required.

to approximate IC1 and IC2 when disseminating requests to disks; and a voter can be used to detect failures among processors running p-processes. The Tandem system is reported to employ fail-stop (there, called fail-fast) modules implemented directly by hardware [4]. These approximations are based on engineering data about how components usually fail; our approximation made no assumption about the nature of failures. On the other hand, our approximation is quite expensive—perhaps too expensive for all but the most demanding applications. This suggests that it might be worthwhile to pursue investigations into how to cheaply implement fail-stop processor approximations, both with and without assumptions about failure modes.

ACKNOWLEDGMENTS

N. Lynch and J. Lundelius provided very helpful comments on earlier drafts of this paper. I am also grateful to K. Birman, D. Gries, L. Lamport, R. Schlichting, and D. Skeen for discussions and comments on this material. Finally, I am especially grateful to B. Alpern and O. Babaoglu, who discovered some subtle bugs in previous versions of the fail-stop implementation in this paper.

REFERENCES

1. DOLEV, D. The Byzantine Generals strike again. *J. Algorithms 3*, (1982), 14–30.
2. FISCHER, M., AND LYNCH, N. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett. 14*, 4, (1982), 182–186.
3. HALPERN, J., SIMONS, B., AND STRONG, R. An efficient fault-tolerant algorithm for clock synchronization. IBM Research Rep. RJ 4094, IBM, San Jose, Calif., Nov. 1983.
4. KATZMAN, J.A. A fault-tolerant computing system. In *Proceedings of the 11th Hawaii International Conference on System Sciences*, 1978.
5. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
6. LAMPORT, L. The implementation of reliable distributed multiprocess systems. *Comput. Networks 2* (1978), 95–114.
7. LAMPORT, L. Using time instead of timeout for fault-tolerant distributed systems. Op. 59, Computer Science Laboratory, SRI International, Menlo Park, California, June 1981. *ACM Trans. Program. Lang. Syst. 6*, 2 (April 1984), 254–280.
8. LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.
9. LAMPSON, B. Atomic transactions. In *Distributed Systems—Architecture and Implementation.* Lecture Notes in Computer Science, vol. 105, Springer-Verlag, New York (1981), pp. 246–265.
10. LYNCH, N.A., FISCHER, M.J., AND FOWLER, R. A simple and efficient Byzantine Generals algorithm. Tech. Rep. GIT-ICS-82/02, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, Feb. 1982.
11. PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.
12. SCHLICHTING, R.D., AND SCHNEIDER, F.B. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug. 1983), 222–238.
13. SCHNEIDER, F.B. Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 125–148.

Received December 1982; revised August 1983; accepted November 1983