BYZANTINE GENERALS IN ACTION: IMPLEMENTING FAIL-STOP PROCESSORS*

Fred B. Schneider

TR 83-569
August 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work is supported, in part, by NSF Grant MCS-8103605.
*This report supercedes an earlier report TR 82-533, entitled "Comparison of the Fail-Stop Processor and State Machine Approaches to Fault-Tolerance".

Byzantine Generals in Action: Implementing Fail-Stop Processors*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

August 11, 1983

ABSTRACT

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 is unaffected by failures and accessible to any other fail-stop processor. Fail-stop processors can simplify 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.

*This work is supported, in part, by NSF Grant MCS-8103605.

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 destruction of critical state information. Even when multiple processors are used, a malfunctioning processor can cause problems because it might cause erroneous state information to be visible to other processors. This could have disastrous consequences if these processors take actions based on bogus information they read. Clearly, use of processors that satisfy the following avoids these difficulties.

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

However, processors that merely halt in response to failures are not sufficient for implementing systems whose correctness criteria involve generating outputs 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 satisfy the

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

because then other processors can assume the workload of a processor that has failed. Of course, there are obvious limitations to this strategy—there must be sufficient processing capacity in the smaller system to continue performing all of its tasks in a timely manner.

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

> *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. 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 by only accessing stable storage. Then, to construct a fault-tolerant computing system that can tolerate up to k failures for an application requiring t processors assuming there are no failures, $t + k$ 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.

Perhaps the strongest argument for investigating the implementation of fail-stop processors is that most protocols for implementing fault-tolerant systems assume that processors are fail-stop or equivalent to fail-stop processors.[1] In some models, instead of the Failure Status Property, "timeouts" are used to detect failures. However, use of timeouts requires another assumption: that processor clocks run at the same rate. Otherwise, two processors might not agree that a third has halted, which can have disastrous consequences if the third processor has not halted but merely was running slowly.

[1]The only work we know of that *does not* involve fail-stop or stronger assumptions about processor failures is [Lb81, Ll78b, Ll81].

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.

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 [Do82, LSP82, NFF82, PSL79]) is then possible. Secondly, our fail-stop processor approximation is a first step towards a practical realization of fail-stop processors.

We must be content with only an approximation of fail-stop processors 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 the 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 computing system that behaves like a fail-stop processor unless $k+1$ or more failures occur within its components. Choice of a value for k depends on the reliability desired. 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. 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 is a discussion of other ways to approximate fail-stop processors and some open problems.

2. Approximating Fail-Stop Processors

A k-fail-stop processor FSP_i is implemented by a collection of real processors, each with its own storage, that are interconnected by a communications network. The implementation consists of:

- $k+1$ *p-processes* (p for program), each running on its own processor. Let $p(FSP_i)=\{p_1, p_2, ...p_{k+1}\}$ be this set of processes.

- $2k+1$ *s-processes* (s for storage), each running on a different processor. Let $s(FSP_i)=\{s_1, s_2, ...s_{2k+1}\}$ be this set of processes.

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

In our implementation of FSP_i, 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. A program running on FSP_i is run by each of the $k+1$ p-processes in $p(FSP_i)$. Failures that should cause FSP_i to halt are detected by comparing results when each p-process in $p(FSP_i)$ writes to stable storage in FSP_i, since reading stable storage is the only way another fail-stop processor can obtain information about the state of FSP_i. Because p-processes run on different processors, they will have independent failure modes. Provided fewer than $k+1$ failures occur in processors running p-processes, if any failure that should cause FSP_i 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_i is stored by each of the s-processes in $s(FSP_i)$. 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 correct. Of course, this presupposes that each correctly functioning s-process updates its state whenever a write is performed to stable storage; protocols for this are described below.

An s-process receives a message whenever a p-process accesses stable storage. Each of these messages m contains the following information:

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

We assume the

Network Reliability Assumption: Messages are delivered uncorrupted and $orig(m)$ the process 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 processors. Independent channels allows 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 with high probability messages are delivered uncorrupted; 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 $FSP_1, FSP_2, ..., FSP_N$ 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 local processor's clock.

In addition, we require that when a p-process p_i makes a request to stable storage in FSP_k at time T on its clock, it disseminates that request in a way that satisfies:

IC1: If p_i is non-faulty, then every non-faulty s-process s_j in $s(FSP_k)$ receives the request by time $T + \delta$ on s_j's clock.

IC2: If s-processes s_i and s_j in the same k-fail-stop processor are non-faulty, then either each of them receives the same request m (with $m.time = T$) from p_i by time $T + \delta$ on its clock or each of them receives no request m (with $m.time = T$) by time $T + \delta$ on its clock.

Condition IC1 ensures that all the s-processes receive a message by $T + \delta$ on their local clocks for any request made by a non-faulty p-process at time T on its local clock. Condition IC2 ensures that even if a p-process is faulty, all s-processes will agree on its request. IC2 is necessary because if a p-process fails, it might make different requests to two different s-processes. The copies of stable storage maintained by these s-processes could then become inconsistent if one s-process performs an update and another doesn't.

Finally, we require:

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

IC3 ensures that the if a request is made by one non-faulty p-process in $p(FSP_i)$ at time T on its clock then the same request is made by each other non-faulty p-process at time T on its local clock.

A number of protocols to establish IC1 and IC2—called *interactive consistency* or *Byzantine Agreement*—have been developed [Do82, LSP82, LFF82, PSL79]. In those

```
$owner := i$; $failed := false$;
do $true \rightarrow$ /* major loop */
  for $s := 1$ to $N$
      $T :=$ CLOCK;
      $D :=$ bag of requests $m$ delivered such that:
              $orig(m) \in p(FSP_s) \wedge (m.type=\text{read} \vee m.type=\text{write})$
      do $D \neq \Phi \rightarrow$
        $minT :=$ minimum value of $m.time$ in $D$;
        if $(\exists m: m \in D: m.time=minT \wedge m.time < T-\delta) \rightarrow$
            $M :=$ bag of requests $m$ such that $m \in D \wedge m.time=minT$;
            $D := D - M$;
            if $(\forall m: m \in M: m.type=\text{read}) \rightarrow$
                do $M \neq \Phi \rightarrow$ choose$(m,M)$; $M := M-\{m\}$;
                                send $m.var$ to $orig(m)$
                od
            [] $(\forall m,m'$ : distinct $m,m' \in M$:
                $m=m' \wedge m.type=\text{write} \wedge orig(m) \neq orig(m'))\ \wedge$
               $|M|=k+1 \wedge s=owner \wedge \neg failed \rightarrow$
                                choose$(m,M)$;
                                $m.var := m.val$
            [] otherwise $\rightarrow$ if $s=owner \wedge \neg failed \rightarrow$
                                    $failed := true$;
                                    forall $d \in p(FSP_i)$ send "halt" to $d$
                                [] otherwise $\rightarrow$ skip
                                fi
            fi
        [] otherwise $\rightarrow$ skip
        fi
      od
  rof
od
```

Figure 1 — Program for s-process in FSP_i

protocols, δ is based on message delivery time and maximum difference in the clock speeds of any two correctly functioning processors. For B processors, when at most t can be faulty and the origin of messages can be authenticated it has been shown [PSL79, FL82]:

- $B \geq 2t+1$

- the protocol must involve at least $O(t+1)$ rounds of message exchange.

Since our implementation of a k-fail-stop processor need tolerate at most k failures and involves at least $2k+1$ processors for running s-processes, IC1 and IC2 can be achieved.

Protocols to achieve clock synchronization, as required by IC3, are described in [LM82]. These protocols require at least $2t+1$ processors in order to tolerate at most t faulty ones, if the origin of messages can be authenticated. For a single k-fail-stop processor, IC3 requires the k+ 1 processors running p-processes to have synchronized clocks. However, it is impossible to use only these $k+1$ processors and still keep clocks synchronized in spite of up to k failures. Fortunately, participation of processors that are used to run s-processes in clock synchronization was not precluded. Since at least $2k+1$ processors are required to run s-processes, IC3 can be achieved if they participate.

When a system consists of u k-fail-stop processors, if $u>1$ then processors running s-processes may not have to participate in clock synchronization. With u k-fail-stop processors, there are $u \times (k+1)$ processors running p-processes. Provided the system is intended to tolerate no more that $(uk+u-1)/2$ failures, clock synchronization can be achieved by synchronizing all of the processors running p-processes (instead of just synchronizing clocks within each individual k-fail-stop processor).

Stable Storage Property

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

(1) The contents of stable storage is unaffected as long as k or fewer failures occur.

(2) A fail-stop processor can write to its stable storage.

(3) Any fail-stop processor can read from the stable storage of any fail-stop processor (including its own).

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 non-faulty p-processes make the same requests to stable storage. Since by IC3 the clocks of all the non-faulty p-processes are synchronized, the non-faulty p-processes will all make requests at the same time according to their local clocks. By IC1 and IC2, every s-process s_j receives by time $T+\delta$ (on its clock) any request that is issued by a p-process p_i at time T (on p_i's clock). Therefore, if any p-process makes a request to stable storage at time T (on its clock) this request will be received by each s-process s_i before $T+\delta$ (on s_i's clock). Moreover, if a non-faulty p-process makes a request at time T (on its clock), requests from all of the non-faulty p-processes will be received by each s-process s_i before $T+\delta$ (on s_i's clock).

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

We now turn to part (2) of the Stable Storage Property. Above, we argued that all non-faulty s-processes perform the same changes to stable storage and 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. 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 non-faulty s-processes.

Finally, for part (3) it suffices to note that a read operation attempted by FSP_i should result in identical responses being sent by non-faulty s-processes to each p-process in $p(FSP_i)$. If fewer than $k+1$ failures occur then at least $k+1$ correct values 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.

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 way that the effects of a failure could be made visible to another process. If

(i) exactly one write request from each of the $k+1$ p-processes has been received and

(ii) 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 $p(FSP_i)$ 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 non-faulty s-processes don't change since the conjunct "$\neg$*failed*" guards the assignment statement.

Failure Status Property

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

This completes our implementation of a k-fail-stop processor approximation. The interface between the the s-processes and the p-processes is summarized in Table 1.

3. Assigning Processes to Processors

Consider an application that requires t 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, $t+k$ 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 t 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

For p-process p_i 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_i in FSP_i to read from stable storage in FSP_j:

1. Initiate a Byzantine Agreement for the read request with all the s-processes in $s(FSP_j)$.
2. Use the value received from at least $k+1$ different s-processes.

For a p-process p_i in FSP_i to determine if FSP_j has halted due to a failure.

Read the variable *failed* from the stable storage in FSP_j.

Table 1 -- Interface between s-process and p-process

processor, resulting in a total of $(t+k)\times(3k+1)$ processors. However, recall 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 $\lceil (t+k)/S \rceil \times (2k+1)$ processors to run the s-processes and $t\times(k+1)$ processors for p-processes. Clearly, this is a substantial decrease in the number of processors over that required for the naive implementation. However, now the $t+k$ 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 $S=t+k$, so there are only $2k+1$ processors running the s-processes for all $t+k$ k-fail-stop processors and all of the failures occur in these processors, there will still be $k+1$ s-processes for each of the $t+k$ k-fail-stop processors running on non-faulty processors. Thus, a majority of the s-processes will be running on non-faulty processors, so each copy of stable storage will perform as required.

When a fail-stop processor halts, all of the non-faulty 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 non-faulty 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. These processors run diagnostics and any processor that passes its diagnostics is reassigned to the available group.

The Processor Recycling Scheme reduces the cost of a failure. Without it, a failure causes the 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 [L178a] and later extended for environments in which failures could occur in [L178b, L181, S82]. 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 t processors to run and meet its real-time constraints. To implement a version of this application that can tolerate up to k faults, a total of $t \times (2k+1)$ processors are required and 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 $(t+k) \times (k+1) + \lceil (t+k)/S \rceil \times (2k+1)$ 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, the state machine approach has other advantages over the fail-stop processor approach. They include:

- 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

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

5. Discussion

The fail-stop processor approach can be viewed as a formalization of a well-known technique: checkpoints are taken during the course of a computation, and after a failure the computation is restarted from the last checkpoint. Actually, our formulation of the approach was not based on this, but followed from our attempts to extend assertional methods for use in understanding fault-tolerance. The basis of axiomatic program verification is that theorems of the programming logic are also true statements about what would happen if the program were run on a computer. That is, the logic is *sound* with respect to operation of a computer. If a failed processor can perform arbitrary state transformations, then the programming logic will no longer be sound with respect to the computer on which the program is being run. Thus, to ensure soundness in light of the possibility of failures, it is necessary to prohibit failures from causing arbitrary state transformations. Hence, fail-stop processors.

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. Other useful operations—synchronization [S82], for example—could have been included in the kernel, as well. The kernel is reliable because it is replicated $2k+1$-fold so that the effects of up to k failures can be 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.

Constructing a reliable computing system involves two things: (1) getting the programs correct assuming that the hardware does what it is suppose to do and (2) constructing hardware that, with high probability, is well-behaved. Using assertional reasoning to aid in the construction of programs and assuming the existence of fail-stop processors addresses (1); approximating fail-stop processors addresses (2).

A methodology for developing provable correct programs to run on fail-stop processors is described in [SS83]. That work can be viewed as extending Dijkstra's programming calculus [Di76] to fault-tolerant computing systems. The methodology has been successfully applied to a number of small examples, including the two-phase commit protocol [S182] and a process control application [SS81].

One way to approximate fail-stop processors was described in this paper. There are undoubtedly other ways. For example, disks are sometimes considered acceptable approximations of stable storage; a triple-redundant bus can be used to approximate IC1 and IC2 when disseminating requests to disks; and, a voter can be used to detect failures among processors running p-processes. These approximations are based on engineering data about how components are likely to 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

B. Alpern, N. Lynch and J. Lundelius provided very helpful comments on earler 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.

References

[Di76] Dijkstra, E.W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J. 1976.

[Do82] Dolev, D. The Byzantine Generals Strike Again. *Journal of Algorithms 3*, 14-30 (1982).

[FL82] Fischer, M., N. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *IPL 14*, No. 4, 1982, 182-186.

[L178a] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM 21*, 7 (July 1978), 558-565.

[L178b] Lamport, L. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks 2* (1978), 95-114.

[Ll81] Lamport, L. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. To appear, *TOPLAS.* Also available as Op. 59, Computer Science Laboratory, SRI International, Menlo Park, California, June 1981.

[LM82] Lamport, L., P.M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. Op. 60, Computer Science Laboratory, SRI International, Menlo Park, California, March 1982.

[LSP82] Lamport, L., R. Shostak, M. Pease. The Byzantine Generals Problem. *TOPLAS 4*, 3 (July 1982), 382-401.

[Lb81] Lampson, B. Atomic Transactions. *Distributed Systems—Architecture and Implementation. Lecture Notes in Computer Science*, vol. 105, Springer-Verlag, New York, N.Y. 1981, pp. 246-265.

[LFF82] Lynch N.A., M.J. Fischer, R. Fowler. A Simple and Efficient Byzantine Generals Algorithm. Technical Report GIT-ICS-82/02, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, Feb. 1982.

[PSL79] Pease, M., R. Shostak, L. Lamport. Reaching Agreement in the Presence of Faults. *JACM 27*, 2 (April 1979).

[Sl82] Schlichting, R.D. *Axiomatic Verification to Enhance Software Reliability.* Ph.D. Thesis, Dept. of Computer Science, Cornell University, Jan. 1982.

[SS83] Schlichting, R.D., F.B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *TOCS 1,* 3 (August 1983) 222-238.

[S82] Schneider, F.B. Synchronization in Distributed Programs. *TOPLAS 4*, 2 (April 1982) 179-195.

[SS81] Schneider, F.B., R.D. Schlichting. Towards Fault-Tolerant Process Control Software. *Proc. Eleventh Annual International Symposium on Fault-Tolerant Computing,* IEEE Computer Society, Portland, Maine, (June 1981) 48-55.