The Fail-Stop Processor Approach*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

Jan. 1984

Chapter appearing in:

Reliability in Distributed Software and Database Systems

edited by Bharat Bhargava.

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

1. Introduction

Programming a computer system that is subject to failures is a difficult task. A malfunctioning processor might perform arbitrary and spontaneous state transformations, instead of the transformations specified by the programs it executes. Thus, even a correct program cannot be counted on to implement a desired input-output relation when executed on a malfunctioning processor. On the other hand, it is impossible to build a computer system that always operates correctly in spite of failures in its components by using (only) a finite amount of hardware[1]. Fortunately, most applications do not require complete fault-tolerance; it is sufficient that the system work correctly provided no more than some predefined number of failures occur within some time interval, or provided certain types of failures do not occur. This more modest goal is attainable.

In this chapter we present an approach to designing fault-tolerant computing systems based on the notion of a *fail-stop processor*, a processor with well-defined failure-mode operating characteristics. Briefly, our approach is as follows. First, software is designed assuming the existence of a computing system composed of one or more fail-stop processors; the number of processors required is dictated by the desired degree of fault-tolerance and any response-time constraints that must be satisfied by the system. Then, a computing system is designed that implements the requisite fail-stop processors.

We proceed as follows. Section 2 defines the operating characteristics of a fail-stop processor. Section 3 concerns the problem of programming fail-stop processors, including axiomatic proof rules for a new programming construct we describe. The methodology described in section 3 is illustrated by two examples in section 4. The implementation of fail-

[1] *Sed quis custodiet ipsos Custodes*? (Who shall guard the guards themselves?) [Juvenal 130].

stop processor approximations is discussed in section 5. Section 6 contains a discussion of related work. Some of the insight that led to the development of fail-stop processors is discussed in section 7. Section 8 contains some conclusions.

2. Fail-Stop Processors

A *fail-stop processor* satisfies the following properties:

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

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

Stable Storage Property. Processor storage 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 fail-stop processor. The contents of volatile storage are not accessible to other fail-stop processors and are lost as a result of a failure.

A fault-tolerant computing system is constructed from a collection of fail-stop processors, interconnected so that each can read the stable storage of the others. The Failure Status Property allows one fail-stop processor to determine that another has halted (due to a failure). This allows tasks that were being run by the halted processor to be continued, so that real-time constraints can be met. The Halt on Failure Property prevents a failure from causing erroneous state information to be visible to other fail-stop processors. Finally, the Stable Storage Property ensures that the state of a task that was running on a halted processor is available to the fail-stop processor that is to continue the task. 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.

3. Programming a Fail-Stop Processor

3.1. Recovery Protocols

A program executing on a fail-stop processor is halted when a failure occurs. Execution may then be restarted on a correctly functioning fail-stop processor. When a program is restarted, the internal processor state and the contents of volatile storage on the fail-stop processor on which it was running are unavailable. Thus, some routine is needed that can complete the state transformation that was in progress at the time of the failure and restore storage to a well-defined state. Such a routine is called a *recovery protocol*. Note that because the code for a recovery protocol must be available after a failure, it must be kept in stable storage.

Clearly, a recovery protocol (i) must execute correctly when started in any intermediate state that could be visible after a failure and (ii) can only use information that is in stable storage. We associate a recovery protocol R with a sequence of statements A, called the *action statement*, to form a *fault-tolerant action* FTA as follows:

FTA: **action**
    A
recovery
    R
end

Execution of FTA consists of establishing R as the recovery protocol to be in effect when A is executed and then executing A. If execution of FTA is interrupted by a failure, upon restart execution continues with the recovery protocol in effect. Subsequent failures cause execution of FTA to be halted and execution of the recovery protocol in effect to begin anew when the program is restarted. Execution of FTA terminates when execution of either A or R is performed in its entirety without interruption. At that time, either the recovery protocol in effect when FTA started is reestablished, or if another fault-tolerant action FTA' follows FTA then the recovery protocol for FTA' is established.

The following syntactic abbreviation will be used to denote that A serves as its own recovery protocol:

FTA: **action** , **recovery**
  A
end

Such a fault-tolerant action is called a *restartable action*[2].

A program running on a fail-stop processor must at all times have a recovery protocol in effect. This will be the case if the program itself is a single fault-tolerant action. Alternatively, a program can be structured as a sequence of fault-tolerant actions, assuming that establishment of a recovery protocol can be done in such a way that at all times either the old recovery protocol or the new one is in effect. This is achieved by storing the identity of the recovery protocol in effect in stable storage.

3.2. Axioms for Fault-Tolerant Actions

Following the Floyd-Hoare axiomatic approach [Hoare 69], an *assertion* is a Boolean-valued expression involving program and logical variables. The syntactic object

$$\{P\}\ S\ \{Q\},$$

where P and Q are assertions and S is a programming language statement, is called a *triple*. The triple $\{P\}\ S\ \{Q\}$ is a *theorem* if there exists a proof of it in a specified formal deductive system, usually called a *programming logic*. A programming logic consists of a set of axioms and rules of inference that relate assertions, programming language statements, and triples. Of particular interest are those logics that are sound with respect to execution of programming

[2] As we shall see, any fault-tolerant action can be converted to such a restartable action simply by omitting the action statement.

language statements on the program state—i.e., deductive systems that are consistent with the operation of a "real" machine. Then, the notation $\{P\}\ S\ \{Q\}$ is usually taken to mean:

> If execution of S begins in a state in which P is true, and terminates, then Q will be true in the resulting state.

It is often more convenient to write a *proof outline* than a formal proof. A *proof outline* is a sequence of programming language statements interleaved with assertions. Each statement S in a proof outline is preceded directly by one assertion, called its *precondition* and denoted $pre(S)$, and is directly followed by an assertion, called its *postcondition* and denoted $post(S)$. A proof outline is an abbreviation for a proof if:

PO1: For every statement S, the triple $\{pre(S)\}\ S\ \{post(S)\}$ is a theorem in the programming logic.

PO2: Whenever $\{P\}$ and $\{Q\}$ are adjacent in the proof outline Q is provable from P.

Let FTA be a fault-tolerant action formed from action statement A and recovery protocol R. We wish to develop an inference rule that will allow derivation of

$$\{P\}\ FTA\ \{Q\}$$

as a theorem, while preserving the soundness of our programming logic with respect to execution on a fail-stop processor.

First, assume

F1: $\{P'\}\ A\ \{Q'\}$ and $\{P''\}\ R\ \{Q''\}$

have been proved. Then, for execution of A to establish Q, we will need

F2: $P\ \ P'$ and $Q'\ Q$.

Similarly, for the recovery protocol R to establish Q, the following (at least) must hold:

F3: $Q'' \; Q$.

Recall that R is invoked only following a failure. By definition, the contents of volatile storage are undefined at that time. Therefore, any program variables needed for execution of R must be in stable storage. Thus, we require

F4: All program variables named in P'' must be in stable storage[3].

We must also ensure that whenever the recovery protocol receives control, stable storage is in a state that satisfies P''. This will be facilitated by constructing a *replete proof outline*, a proof outline that contains assertions describing (only) those states that could be visible after a failure. Then, we will require that the precondition of the recovery protocol be satisfied in those states.

A *replete proof outline* is a proof outline in which certain assertions have been deleted so that:

RPO1: No assertion appears between adjacent fault-tolerant actions.

RPO2: Every triple $\{P\}\ S\ \{Q\}$ in the replete proof outline satisfies either

(a) S is a sequence of fault-tolerant actions, or

(b) $P \ OR \ Q$ is invariant over execution of S .

RPO1 and RPO2(a) capture the fact that the program state between the execution of two fault-tolerant actions FTA_1 and FTA_2 is never visible to the recovery protocol of an enclosing fault-tolerant action—either the recovery protocol for FTA_1 or the recovery protocol for FTA_2 will receive control. RPO2(b) follows because if $P \ OR \ Q$ remains true while S is being executed, then either P or Q will be true of the state visible to the recovery protocol should a failure

[3]If P'' is stronger than $wp(R,Q'')$ [Dijkstra 76] then variables may appear in P'' that need not be stored in stable storage. Thus, in the interest of minimizing the amount of stable storage used, proofs should be in terms of the weakest assertions possible.

occur, and both $\{P\}$ and $\{Q\}$ already appear as assertions in the replete proof outline.

For example, if

$$\{P\} \;\; FTA_1 \;\; \{P_1\} \;\; FTA_2 \;\; \{P_2\} \;\; \ldots \;\; FTA_n \;\; \{P_n\}$$

is a proof outline, then

$$\{P\} \;\; FTA_1; \quad FTA_2; \quad \ldots \;\; FTA_n \;\; \{P_n\}$$

is a replete proof outline. As another example, if assignment of an integer value to a variable is performed by executing a single, indivisible, (store) instruction—as it is on most machines—then

$$\{x = 3\} \; x := 6 \; \{x = 6\}$$

is also a replete proof outline. This is because either the precondition or the postcondition of $x := 6$ is true of every state that occurs during execution of the assignment. Even if assignment is not implemented by execution of a single instruction, the proof outline

$$\{val = 3\} \; x := val \;\; \{x = 3 \quad val = 3\}$$

is replete because the assertion $val = 3$ is not destroyed by assignment to x; it is true before, during and after execution of $x := val$.

In addition to F1 - F4, correct operation of a recovery protocol requires:

F5: Given a fault-tolerant action with action statement A and recovery protocol R satisfying F1, let $a_1, a_2, \ldots, a_n$ be the assertions that appear in a replete proof outline of $\{P'\} \; A \; \{Q'\}$, and $r_1, r_2, \ldots, r_m$ be the assertions that appear in a replete proof outline of $\{P''\} \; R \; \{Q''\}$. Then:

(i) $(\forall i : 1 \le i \le n : a_i \quad P'')$

(ii) $(\forall i : 1 \le i \le m : r_i \quad P'')$

F5 states that P'' is an invariant of A and R. Thus, P'' will be true of any state visible after a failure.

Lastly, it must be guaranteed that failures at processors other than the one executing *FTA* do not interfere with (i.e., invalidate) assertions in the proof outline of *FTA*. Suppose an assertion *a* in *FTA* names variables stored in the volatile storage of another fail-stop processor *FSP*.[4] Then, should *FSP* fail, *a* would no longer be true since the contents of volatile storage would have been lost. Hence, we require that:

F6: Variables stored in volatile storage may not be named in assertions appearing in programs executing on other processors.

Given a fault-tolerant action, a restartable action that implements the same state transformation can always be constructed from the recovery protocol alone. (The proof of this follows from F3 and F5.) Thus, in theory, the action statement is unnecessary. In practice, the additional flexibility that results from having an action statement different from the recovery protocol is quite helpful. Presumably, failures are infrequent enough so that a recovery protocol can do considerable extra work in order to minimize the amount of (expensive) stable storage used. Use of such algorithms for normal processing would be unacceptable.

It is natural to ask whether F1 - F6 are too restrictive. In that case there would exist fault-tolerant actions that would behave correctly, but for which no proof would be possible. While we have not proved the relative completeness of the proof rule, the success we have had with its application and the way in which it was derived, suggest F1 - F6 are not too restrictive to allow proof of any ''correct'' fault-tolerant action.

---

[4]This is often necessary when the actions of concurrently executing processes are synchronized. For example, if it is necessary to assert that a collection of processes are all executing in the same "phase" at the same time, then each would include assertions about the state of the others. See [Schlichting & Schneider 82] for an example of such reasoning.

3.3. Termination and Response Time

Most statements in our programming notation are guaranteed to terminate, once started. However, loops and fault-tolerant actions are not. Techniques based on the use of variant functions or well-founded sets can be used for proving that a loop will terminate [Dijkstra 76]. Unfortunately, without knowledge about the frequency of failures and statement execution times, termination of a program written in terms of fault-tolerant actions cannot be proved. This is because if failures occur with sufficiently high frequency then there is no guarantee that the component fault-tolerant actions will terminate; neither the action statement nor the recovery protocol of a fault-tolerant action can be guaranteed to run uninterrupted, and so the recovery protocol could continually restart. Moreover, such properties cannot even be expressed in a programming logic like the one above. Thus, we must resort to informal means to argue that a program will terminate in a timely manner when fail-stop processors and fault-tolerant actions are used. Presumably, at some point in the future it will be possible to formalize such arguments.

For a given execution of a program S on a fault-free processor, let $t(s)$ be the maximum length of time that elapses once execution of a component statement s is begun until execution of the next fault-tolerant action in S is started. Define

$$T_{\max} = \max_{s \text{ member } S} t(s).$$

For an execution of S to terminate at all, it is sufficient that there be (enough) intervals of length $T_{\max}$ during which there are no failures. Then, no fault-tolerant action will be forever restarted due to the (high) frequency of failures.

Of course, this gives no bound on how much time will elapse before S completes. Rather, we have argued that S is guaranteed to terminate if the elapsed time between successive failures is long enough, often enough. This should not be surprising. However, it does provide some insight into how to structure a program in terms of fault-tolerant actions if frequent failures are expected: one should endeavor to minimize $T_{\max}$. This can be achieved by making entry into a fault-tolerant action a frequent event—either by nesting fault-tolerant actions, or composing them in sequence.

4. Examples

4.1. Updating Two Variables

In addition to allowing partial-correctness proofs of programs written in terms of fault-tolerant actions, F1 - F6 permit a programmer to develop a fault-tolerant program and its proof hand-in-hand, with the proof leading the way, as advocated in [Dijkstra 76] [Gries 81]. F4 allows variables that must be stored in stable storage to be identified in a mechanical way from the proof; construction of a replete proof outline provides a mechanical way to determine the intermediate states that could be visible following a failure.

To illustrate the use of rules F1 - F6 as an aid in developing a recovery protocol, we consider the following (artificial) problem.

> Periodically, variables x and y are updated based on their previous values. Thus, given a function G, a routine called *update* is desired that runs on a fail-stop processor and satisfies the following specification:
>
> $$\{P: x = X \quad y = Y\} \quad \textit{update} \quad \{Q: x = G(X) \quad y = G(Y)\}.$$

Logical variables X and Y represent the initial values of x and y, respectively.

If the possibility of failure is ignored, the following program will suffice:

S1: $\{P: \; x = X \quad y = Y\}$
  S1a: $x := G(x)$; $\{P1a: x = G(X) \quad y = Y\}$
  S1b: $y := G(y)$; $\{P1b: x = G(X) \quad y = G(Y)\}$
  $\{Q: \; x = G(X) \quad y = G(Y)\}$

Note that this is a replete proof outline, provided assignment is implemented as an atomic operation: *P OR P1a* is invariant over execution of *S1a* and *P1a OR P1b* is invariant over execution of *S1b*.

Things become more complicated when the possibility of failure is considered. In particular, *S1* could not be the action statement of a restartable action because F5 is violated (assuming G is not the identity function): both *P1a* $\quad P$ and *P1b* $\quad P$ are false. In order to construct a restartable action, we must find a way to make progress—compute $G(X)$ and $G(Y)$—but without destroying the initial values of x and y until both values have been updated. One way to do this is to modify *S1* so that the new values are computed and stored in some temporary variables, giving the following restartable action:

U1: **action**, **recovery**
  $\{P: \; x = X \quad y = Y\}$
  U1a: $xnew := G(x)$; $\{x = X \quad xnew = G(X) \quad y = Y\}$
  U1b: $ynew := G(y)$; $\{x = X \quad xnew = G(X) \quad y = Y \quad ynew = G(Y)\}$
  end
  $\{Q': \; x = X \quad xnew = G(X) \quad y = Y \quad ynew = G(Y)\}$

Note that in order to satisfy F4, x and y must be stored in stable storage; variables used in computing G need not be. Having established Q', it is a simple matter to establish Q:

S2: $\{Q'': \; xnew = G(X) \quad ynew = G(Y)\}$
  S2a: $x := xnew$; $\{x = xnew = G(X) \quad ynew = G(Y)\}$
  S2b: $y := ynew$; $\{x = xnew = G(X) \quad y = ynew = G(Y)\}$
  $\{Q: \; x = G(X) \quad y = G(Y)\}$

This is a replete proof outline, and provided *xnew* and *ynew* are stored in stable storage, F1 - F6 are satisfied. So

$U2$: **action** , **recovery**
  $\{Q'': \ xnew = G(X) \quad ynew = G(Y)\}$
  $U2a$: $x := xnew;\quad \{x = xnew = G(X) \quad ynew = G(Y)\}$
  $U2b$: $y := ynew;\quad \{x = xnew = G(X) \quad y = ynew = G(Y)\}$
  end
  $\{Q: \ x = G(X) \quad y = G(Y)\}$

is a restartable action. Since $Q' \ Q''$, the desired program is:

$$update: \ U1; \ \ U2$$

4.2. A Process-Control Program

We now turn to a more substantial illustration of the application of the methodology: development of a fault-tolerant process control program. First, a correct program for a fault-free computing system is developed. The program is then extended to run on a system of fail-stop processors by defining the requisite fault-tolerant actions.

> Given are *sensors* to determine the state of the environment and *actuators* to exert control over the environment. Correct operation of a process-control system requires that:
>
> PC: The values written to the actuators are related to the values read from the sensors according to a given application-specific function.[5]

Assuming No Failures

Our process-control system will be structured as a collection of cyclic processes that execute concurrently. Each process p_i is responsible for controlling some set of actuators act_i. To do so, it reads from some sensors and updates $state_i$—a vector of *state variables* that reflects the sensor values p_i has read and the actions it has taken. Interprocess communication is accomplished by the disciplined use of shared variables; a process can read and write its

[5]It is likely that correct operation also involves a liveness property, like "sensors are read and actuators are updated often enough". We will make no attempt to argue that our program satisfies such real-time response constraints, although informal arguments could be used if timing data were available.

state variables, but can only read state variables maintained by other processes. For the moment, we will ignore the problems that arise from concurrent access to state variables.

As is typical in process-control applications, each process will consist of a single loop. During execution of its loop body, process p_i: (1) reads from some sensors, (2) computes new values for the actuators it controls and state variables it maintains, (3) writes the relevant values to act_i and (4) updates $state_i$. Presumably, we are given application-dependent routines to compute values to be written to the actuators and values to be stored in the state variables.

Without loss of generality, assume that each state variable and sensor is read at most once in any execution of those routines[6]. Let $state_j[i,t]$ denote the value of $state_j$ read by p_i during the t^{th} execution of its loop body, $sensors[i,t]$ denote the values read by p_i from sensors during the t^{th} execution of its loop body, and $act_i[t]$ denote the values written to act_i by p_i during the t^{th} execution of the loop body.

Behavior satisfying PC is characterized by the following, for each process $p_1, p_2, ..., p_n$. First, the values in $state_i$ must correctly encode past actions performed by p_i. That encoding will be denoted here by the function E. Therefore, at the beginning of the $t+1^{\text{st}}$ execution of the loop body at p_i:[7]

$$Istate(i,t):\ \ t=0 \textbf{ cor } state_i = E(sensors[i,t], state_1[i,t] ..., \ state_n[i,t]).$$

Secondly, values written to actuators by p_i must be computed according to the application-specific function, here called A, based on the sensor values read and the past

[6]Code that satisfies this restriction can be written by using local variables to store state variables and sensor values: each state variable and sensor value is stored in a local variable when it is first read; subsequent references are then made to the local variable.

[7]We use the notation "A **cor** B" to mean "**if** A **then** true **else** B".

actions of processes. Therefore, after p_i updates act_i for the t^{th} time,

$$Iact(i,t):\;\; t = 0 \;\mathbf{cor}\; act_i[t] = A(E(sensors[i,t], state_1[i,t] \ldots,\; state_n[i,t])).$$

must be true.

Let T_i be an auxiliary variable[8] defined such that at any time $T_i - 1$ executions of the loop body have completed. Thus, T_i is initialized to 1 and (implicitly and automatically) incremented immediately after the loop body is executed. Then, the correctness criterion PC is satisfied if:

$$I(i):\;\; Istate(i,T_i{-}1) \quad Iact(i,T_i{-}1)$$

is true at the beginning of each execution of the loop body, for each process p_i.

In order to construct the loop, local variable *newstate* is introduced. This is necessary so that values used to update $state_i$ and the actuators are consistent with each other. Thus,

$$Vnewstate(i,t):\;\; newstate = E(sensors[i,t], state_1[i,t] \ldots state_n[i,t]).$$

The loop at process p_i, which has $I(i)$ as its loop invariant, is:

p_i: **process**
  do $true \;\rightarrow\; \{I(i)\}$
    $calc: newstate := E(sensors, state_i, \ldots, state_n);$
      $\{Vnewstate(i,T_i) \quad Istate(i,T_i{-}1) \quad Iact(i,T_i{-}1)\}$
    $up\_act: act_i := A(newstate);$
      $\{Vnewstate(i,T_i) \quad Istate(i,T_i{-}1) \quad Iact(i,T_i)\}$
    $up\_st: state_i := newstate;$
      $\{Vnewstate(i,T_i) \quad Istate(i,T_i) \quad Iact(i,T_i)\}$
  od
end

[8] An *auxiliary variable* is one that is included in a program solely for purposes of performing a correctness proof. The value of an auxiliary variable never influences execution.

Because processes execute asynchronously, access to state variables must be synchronized. Otherwise, a process might read state variables while they are in the midst of being updated, which could cause the process to perform the wrong actions. To avoid this problem, the state variables maintained by each process p_i are assumed to be characterized by CC_i, called the *consistency constraint* for $state_i$. CC_i is kept true of $state_i$ except while p_i is updating those variables—i.e. performing *up_st* above. We assume that the code to compute the application dependent functions A and E works correctly as long as values that satisfy the consistency constraints are read. To ensure that only values satisfying the consistency constraints are read, read/write locks [Gray 78] can be used to implement reader-writer exclusion on the state variables maintained by each process. A process trying to read variables in $state_i$ must first acquire a read lock for $state_i$. Such a lock will not be granted if a write lock is already held for those state variables, hence that process will be delayed if $state_i$ is being updated. A process about to update $state_i$ will be delayed if other processes are reading those values. Such lock operations are not explicitly included in our programs to simplify the exposition; they are part of the routine to compute E in *calc* and *up_st*.

Similarly, we assume that the code to compute A and E requires that sensor values used be consistent. The natural laws that govern our physical world ensure that the values of the sensors are consistent at all times. Thus, if a process read all the sensors simultaneously, consistent values would be obtained. Such a simultaneous read operation is not implementable, however. We therefore assume that sensors change values slowly enough and that processes execute quickly enough so that a consistent set of values is obtained by reading each of the sensors, in sequence, at normal execution speed.

Allowing Failures

We shall deal with failures by attempting to mask their effects. Thus, we shall endeavor to preserve:

PC′: At no time do state variables or actuators have values they could not have had if the failure had not occurred.

Recall that $I(i)$ characterizes values of the state variables and actuators that satisfy PC. Consequently, if it is possible to modify the loop body so that $I(i)$ is true of every state that could be visible after a failure then PC′ will be satisfied, as well. Our task, therefore, is to modify the loop body so that it constitutes a restartable action.

$I(i)$ is true except between the time execution of statement *up_act* begins and when *up_st* completes. Thus, we must either mask intermediate states during execution of *up_st* and *up_act*, or devise a way to execute *up_st* and *up_act* together as an atomic action. This latter option is precluded by most hardware. Thus, to implement the former, we construct a single fault-tolerant action that updates the actuators and state variables based on the value of *newstate*:

$$
\begin{array}{l}
\{Vnewstate(i,T_i)\} \\
upall \\
\{Vnewstate(i,T_i) \quad Istate(i,T_i) \quad Iact(i,T_i)\}
\end{array}
$$

As long as *newstate* is saved in stable storage, the following replete proof outline satisfies F1 - F6 and accomplishes the desired transformation.

$$
\begin{array}{ll}
upall: & \textbf{action}, \textbf{recovery} \\
& \{Vnewstate(i,T_i)\} \\
& up\_act: act_i := A(newstate) \\
& \{Vnewstate(i,T_i) \quad Iact(i,T_i)\} \\
& up\_st: state_i := newstate; \\
& \{Vnewstate(i,T_i) \quad Istate(i,T_i) \quad Iact(i,T_i)\} \\
& \textbf{end}
\end{array}
$$

A replete proof outline for the code executed at p_i is:

p_i: **process**
action, **recovery**
  do *true* $\rightarrow$ $\{I(i)\}$
    calc: $newstate := E(sensors, state_i, \ldots state_n)$;
    $\{Vnewstate(i, T_i) \quad Istate(i, T_i - 1) \quad Iact(i, T_i - 1)\}$
    upall: **action**, **recovery**
      up_act: $act_i := A(newstate)$;
      up_st: $state_i := newstate$;
      end
  od
end

Notice that following a failure, a process might attempt to acquire a given read/write lock that had already been granted to it. For example, if a failure occurred while *up_st* were being executed, the recovery protocol would attempt to acquire the write lock on $state_i$, which might already be owned by p_i. Clearly, repeated requests by a given process for the same lock, without intervening release operations, should not delay the invoker. Implementation of read/write locks with this property (binary semaphores do not suffice) is possible and is described in [Schlichting 82].

5. Implementing Fail-stop Processor Approximations

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. Consequently, we now address the problem of implementing a fail-stop processor approximation. We must be content with only an approximation because it is possible to tolerate only a finite number of failures with a finite amount of hardware.

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.

5.1. In Terms of Processes

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:

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

The question of allocating processors to processes is discussed in Section 5.2.

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 the s-processes in $s(FSP)$. This is done by comparing results when each p-process in $p(FSP)$ writes to stable storage in FSP, since subsequent reads to that stable storage would be the only way the effects of a failure could be 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 its state whenever a write is performed to stable storage; a protocol for this is described below.

The only way a p-process in can access stable storage is by sending messages to s-processes. 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.rectime	The time this request was received according to the local clock on the processor running the s-process receiving the request.
m.type	"read" or "write", depending on the request.
m.var	the variable in stable storage to be written if $m.type = \mathrm{write}$; the variable in stable storage to be read if $m.type = \mathrm{read}$.
m.val	the value to be written if $m.type = \mathrm{write}$.

We assume the following 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 with high probability messages are delivered uncorrupted; digital signatures implement authentication (with high probability).

Each s-process in $s(FSP_i)$ 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,

> $\mathrm{choose}(m, M)$ 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:

```
owner := FSP_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 OR 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    FSP_s = owner    ¬failed) →
                                     choose(m, M);
                                     Stable[m.var] := m.val
               [] otherwise → if FSP_s = owner   ¬failed →
                                       failed := true;
                                       forall d ∈ p(owner)  send "halt" to d
                                  [] otherwise → skip
                                  fi
               fi
          [] otherwise → skip
          fi
        od
  rof
od
```

Figure 1 — Program for s-process in FSP_i

IC1: If p_j is non-faulty, then every non-faulty 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 non-faulty, 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 non-faulty p-process. Condition IC2 ensures that even all s-processes will agree on a request, even if the p-process making the request is faulty. 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, the clocks of all processors running p-processes in $p(FSP)$ are synchronized.

IC3 ensures that if a request is made by one non-faulty 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 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 [Dolev 82] [Lamport *et al.* 82] [Lynch *et al.* 82] [Pease *et al.* 79]. In those protocols, δ is based on message delivery time and 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 [Fischer & Lynch 82]. 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 to achieve clock synchronization, as required by IC3, is described in [Halpern *et al.* 83]. 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.

Stable Storage Property

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

(1) A majority of the copies of stable storage are correct and identical as long as k or fewer failures occur.

(2) A non-faulty 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 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 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, if a non-faulty 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 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 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 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 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 (of 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.

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 $p(FSP)$ 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 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 the s-processes and the p-processes is summarized in Table 1.

5.2. 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

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 a 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.

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

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. 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 $\lceil (N+k)/S \rceil \times (2k+1)$ processors to run the s-processes and $N\times(k+1)$ processors for p-processes. Clearly, this is a substantial 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. Even if $S = N+k$, so 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 non-faulty processors for each of the $N+k$ k-fail-stop processors. Thus, a majority of the s-processes will be running on non-faulty processors.

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 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.

5.3. Other Ways to Approximate Fail-Stop Processors

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 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 usually fail; the approximation above 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 other ways to implement fail-stop processor approximations, both with and without assumptions about failure modes.

6. Related Work

Recovery Blocks

Despite the apparent similarity between the recovery block construct developed at the University of Newcastle-upon-Tyne [Randell *et al.* 78] and our fault-tolerant actions, the two constructs are intended for very different purposes. A *recovery block* consists of a *primary block*, an *acceptance test*, and one or more *alternate blocks*. Upon entry to a recovery block, the primary block is executed. After its completion, the acceptance test is executed to determine if the primary block has performed acceptably. If the test is passed, the recovery block terminates. Otherwise, an alternate block—generally a different implementation of the same algorithm—is attempted and the acceptance test is repeated. Execution of each alternate block is attempted in sequence until one produces a state in which the acceptance test succeeds. Execution of an alternate block is always begun in the recovery block's initial state.

Recovery blocks are used to mask programming errors; fault-tolerant actions are used in constructing programs that must cope with failures in the underlying hardware (and software). Not surprisingly, use of recovery blocks to cope with operational failures can only lead to difficulties. For example, a recovery block has only a finite number of alternate blocks associated with it, and therefore a large number of failures in the underlying system can cause the available alternatives to be exhausted. Also, the recovery block model does not admit the possibility of using stable storage for program variables.

State Machine Approach

Few general techniques have been developed to aid in the design of programs that must cope with operational failures in hardware. One approach, based on the use of state machines, was pioneered by Lamport [Lamport 78a] and later extended for environments in which failures could occur in [Lamport 78b] [Lamport 81] [Schneider 82]. The implementation of a k-fail-stop processor described in section 5 is an application of this technique.

In the state machine approach, a program is viewed as a state machine that receives input, generates actions (output) and has an internal state. 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.[9] 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. To implement a version of this application that can tolerate up to k faults, a total of $N \times (2k+1)$ processors are required if the state machine approach is used, and each additional "k-fault-tolerant processor" costs $2k+1$ real processors. It is instructive to 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.

---

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

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

7. Whence Fail-Stop Processors

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. Our formulation of the approach was not based on this, but actually followed from our desire to extend Hoare-style programming logics for use in understanding fault-tolerant programs. In a fail-stop processor, all failures are detected and no incorrect state transformation due to a failure is ever visible. Thus, if execution of a statement terminates, by definition the transformation specified by that statement has occurred—the effect of execution is consistent with a partial-correctness programming logic. On the other hand, failure, by definition, prevents statements from terminating. Thus, the partial correctness (as opposed to total correctness) nature of the programming logic subsumes the consequences of failures.

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.

8. Conclusions

Constructing a reliable computing system involves two things: (1) writing programs that run correctly assuming 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 addresses (1); approximating fail-stop processors addresses (2).

A methodology for developing provable correct programs to run on fail-stop processors was described in section 3. The methodology has been successfully applied to a number of small examples, including the two-phase commit protocol [Schlichting 82] and a process control application [Schneider & Schlichting 81].

A way to approximate fail-stop processors was described in section 5. The 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.

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.

Another argument for studying fail-stop processors is that most protocols for implementing fault-tolerant systems assume processors are fail-stop or equivalent to fail-stop processors. 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. 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.

Acknowledgements

The proof system for fault-tolerant actions was developed in collaboration with R.D. Schlichting; discussions with B. Alpern, O. Babaoglu, L. Lamport, and R.D. Schlichting were helpful in the design of the fail-stop processor approximation. D. Gries and D. Skeen read drafts of this and earlier papers describing fail-stop processors and provided helpful comments.

References

[Dijkstra 76]
Dijkstra, E.W. *A Discipline of Programming*. Prentice Hall, 1976.

[Dolev 82]
Dolev, D. The Byzantine Generals Strike Again. *Journal of Algorithms 3*, 14-30 (1982).

[Fischer & Lynch 82]
Fischer, M., N. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *IPL 14*, No. 4, 1982, 182-186.

[Gray 78]
Gray, J. Notes on Data Base Operating Systems. *Operating Systems An Advanced Course*, Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978.

[Gries 81]
Gries, D. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Halpern *et al.* 83]
Halpern, J., B. Simons, R. Strong. An Efficient Fault-tolerant Algorithm for Clock Synchronization. IBM Research Report RJ 4094, Nov. 1983.

[Hoare 69]
Hoare, C.A.R. An Axiomatic Basis for Computer Programming. *CACM 12*, 10 (October 1969), 576-580.

[Juvenal 130]
Juvenal (Decimus Junius Juvenalis, c.50 -c.130). Satires VI, line 347.

[Lamport 78a]
Lamport, L. Time, Clock and the Ordering of Events in a Distributed System. *CACM 21*, 7 (July 1978), 558-565.

[Lamport 78b]
Lamport, L. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks 2* (1978), 95-114.

[Lamport 81]
Lamport, L. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. Technical Report 59, SRI International, June 1981.

[Lamport *et al.* 82]
Lamport, L., R. Shostak, M. Pease. The Byzantine Generals Problem. *TOPLAS 4*, 3 (July 1982) pp. 382-401.

[Lynch *et al.* 82]
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.

[Pease *et al.* 79]
Pease, M., R. Shostak, L. Lamport. Reaching Agreement in the Presence of Faults. *JACM 27*, 2 (April 1979).

[Randell *et al.* 78]
Randell, B., P.A. Lee, P.C. Treleaven. Reliability Issues in Computing System Design, *Computing Surveys 10*, 2 (June 1978), 123-165.

[Schlichting 82]
Schlichting, R.D. *Axiomatic Verification to Enhance Software Reliability*. Ph.D. Thesis, Dept. of Computer Science, Cornell University, Jan. 1982.

[Schneider 82]
Schneider, F.B. Synchronization in Distributed Programs. *TOPLAS 4* 2 (April 1982), 125-148.

[Schneider & Schlichting 81]
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.

[Schlichting & Schneider 82]
Schlichting, R.D, F.B. Schneider. Understanding and Using Asynchronous Message Passing. *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ACM, Ottawa, Canada, (August 1982) 141-147.