Fail-Stop Processors*

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

A *fail-stop processor* never performs an erroneous state transformation due to a failure. Instead, the processor halts and its state is irretrievably lost. The contents of *stable storage* are unaffected by any failure. Fail-stop processors and stable storage can simplify construction of fault-tolerant computing systems. The problem of designing approximations to fail-stop processors and stable storage is addressed. Their use is compared with the *state machine approach*, another general paradigm for constructing fault-tolerant systems.

Introduction

Designing and programming a fault-tolerant computing system is a difficult task. As a result of a failure, a processor might exhibit arbitrary and malicious behavior thereby destroying critical information. An obvious solution to this is to employ multiple processors. However, a malfunctioning processor might still destroy state information and thus affect the operation of working processors. Clearly, use of processors that halt in response to any failure would avoid this difficulty. We call such processors *fail-stop*.

A *fail-stop processor* never performs an erroneous state transformation due to a failure. Instead, the processor simply halts and its internal state (registers, etc.) and the contents of its memory are irretrievably lost. Similarly, the contents of *stable storage* are unaffected by any failure. We have found these abstractions to be useful for building fault-tolerant computing systems and have developed a methodology based on them, called the *fail-stop processor approach*. The development of programs for fail-stop processors and stable storage is treated in detail in [11]. This paper addresses the problem of constructing fail-stop processors and stable storage. We also contrast our approach with the *state machine approach* [3,4,5,12], another paradigm for constructing fault-tolerant computing systems.

Programming Fail-Stop Processors with Stable Storage

When using a computing system constructed from fail-stop processors and stable storage, a failure never contaminates values in stable storage. However, a failure does destroy state information in other storage attached to malfunctioning fail-stop processors. This suggests the following structure for a fault-tolerant program. The program is divided into logical units of computation, called *actions*. Concurrently executed actions communicate and synchronize by using shared variables in stable storage.

For each action, a *recovery protocol* is also programmed. The recovery protocol R for action A is designed so that:

RP1: Only values stored in stable storage are required for R to run correctly.

RP2: R will yield the same results as A, and will do so when started in any intermediate state that can be visible should execution of A or R be halted (due to a failure).

A fault-tolerant computing system is then constructed from a collection of fail-stop processors and some shared, stable storage. (The number of fail-stop processors and amount of stable storage required will depend on the number of failures the system must be able to tolerate and any response-time constraints imposed by the application.) Whenever a fail-stop processor *fsp* fails, a *reconfiguration rule* is used to assign programs that were running on *fsp* to operating fail-stop processors. Once assigned, the corresponding recovery protocol is invoked for each action that is moved and the computation continues. Failures during execution of the recovery protocol cause the program to be moved to another fail-stop processor and the recovery protocol to be reinvoked. Thus, processor failures are invisible, except for possibly increased execution times.

Structuring programs as actions and recovery protocols is reasonably straightforward. An action is designed to save enough of its state in stable storage so that its recovery protocol, should it be invoked, can always complete the state transformation that was in progress. Although it is tempting to store all program variables in stable storage, thus simplifying the task of designing the recovery protocol, accessing stable storage is likely to be expensive compared with accessing other storage. Therefore, the number of accesses to stable storage that an action makes should be minimized. On the other hand, recovery protocols are executed infrequently, so they can make heavy use of stable storage and even do extra computation when invoked.

One simple way to structure an action is for it to read some information from stable storage, do a computation based on that information, and then write the results of that computation back to stable storage. This approach is often used in database management systems. Items are stored on disks, which are assumed to implement stable storage. Transactions can update one or more items by moving the items to proces-

*This work is supported in part by NSF Grant MCS-81-03605.

sor storage, performing the update, and then issuing an instruction to cause the items to be written back onto the disk. A *log* is used to save the old and new values for each item in the database that is changed; the log is stored in stable storage. A failure during processing may leave the database in a state that reflects partial execution of one or more transactions, depending on which new values had actually been written to the database prior to the time of the failure. However, a recovery protocol can make use of the log to restore the database to a state that reflects complete execution of some of the transactions.

A methodology for developing actions and recovery protocols is described in [11]. It is based on axiomatic program verification and can be viewed as extending Dijkstra's programming calculus [1] for programming fault-tolerant systems. The methodology has been successfully applied to a number of small examples, including the two-phase commit protocol [10] and a process control application [13].

Approximating Fail-Stop Processors and Stable Storage

It is impossible to implement a completely fault-tolerant computing system using a finite amount of hardware. With only a finite amount of hardware, a finite number of failures could disable all the error detection hardware and thereby allow arbitrary behavior. Thus, we must be satisfied with constructing systems that behave as desired unless too many failures occur within some specified time interval. A *k-fail-stop processor* is a computing system that behaves like a fail-stop processor unless $k+1$ or more failures occur. Similarly, an *s-stable storage unit* behaves like stable storage unless $s+1$ or more failures occur. Obviously, as k and s approach infinity, these approximations become closer to the ideals they approximate. In order to construct a computing system that can tolerate up to t failures, t-fail-stop processors and t-stable storage units are used.

Byzantine Agreement

Our approximations are based on establishing *Byzantine Agreement* [7] (also known as *interactive consistency* [9]) among a collection of processors. Given a collection of potentially faulty processors, where one is designated the *transmitter*, a Byzantine Agreement protocol allows the transmitter to distribute a value at time T according to its clock so that:

Byz1: each non-faulty processor will agree at time $T+\Delta$ (on its clock) on the value sent by the transmitter, and

Byz2: if the transmitter is non-faulty, the non-faulty processors agree on the value sent by the transmitter

Δ depends on the upper bound on the time it takes the network to deliver a message once it has been sent and on the maximum rate by which clocks at non-faulty processors can differ. Thus, one way the transmitter might fail is by executing very slowly. In this case, all non-faulty processors will agree on a distinguished value "timeout".

Protocols that establish Byzantine Agreement are described in [2,7,8]. In addition, lower bounds have been obtained for various aspects of establishing Byzantine Agreement among N processors when at most t can be faulty [8]:

- the protocol must run for at least $O(t+1)$ rounds, and
- the number of messages exchanged in each round must be at least $O(N^2)$.

These quantify the cost of any protocol to establish Byzantine Agreement; it is not cheap.

k-Fail-Stop Processors

A *k-fail-stop processor* is constructed from $k+1$ processors and a stable storage unit. Failures are detected by having processors run the same program and compare results when they access stable storage.

Initially, we assume the existence of perfectly reliable stable storage; that assumption is relaxed in the next section. We also assume that

A1: The clocks at all non-faulty processors and the stable storage unit are synchronized and run at the same rates.

Algorithms exist to synchronize processor clocks despite arbitrary and malicious failures [6]. These algorithms require at least $2t+1$ processors in order to tolerate up to t faulty ones. Thus, provided our implementation of stable storage involves at least k processors, clocks can be kept synchronized, since our implementation of a k-fail-stop processor involves $k+1$ processors. The implementation of an s-stable storage unit given in the next section uses $2s+1$ processors, $s \geq k$.

Finally, we assume

A2: The origin of a message can be authenticated by its receiver.

This is a reasonable assumption; digital signatures can be used for this purpose.

Each of the $k+1$ processors that make up a k-fail-stop processor *fsp* executes the same program. A processor accesses stable storage by sending a message. In order for processor p to write value *val* to location *var* in stable storage, it sends a timestamped message m:

$$m = <T\text{:}\mathbf{time},\ p\text{:}\mathbf{origin},\ \text{write } val \text{ to } var\text{:}\mathbf{request}>;$$

to read the value of location *var* in stable storage, p sends

$$m = <T\text{:}\mathbf{time},\ p\text{:}\mathbf{origin},\ \text{read } var\text{:}\mathbf{request}>;$$

Since all processors execute the same program and have synchronized clocks, all non-faulty processors should send messages containing identical **request** and **time** fields to a stable storage unit. Moreover, a stable storage unit should receive those messages by time $T+\delta$ on its clock, where δ is a function of the upper bound on the time it takes the network to deliver a message once it has been sent and the maximum rate by which clocks at non-faulty processors and the stable storage unit can differ. The failure of a k-fail-stop processor is detected by checking all messages received as of time T with timestamp $T-\delta$. If

(i) exactly one message from each of the $k+1$ processors has been received and

(ii) the **request** field in each of these messages is identical,

then (either all or) none of the $k+1$ processors that make up the k-fail-stop processor are faulty. (The case where all $k+1$ processors 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.) Otherwise, at least one processor is behaving differently from the others and so the k-fail-stop processor is deemed faulty. Stable storage is updated only if the k-fail-stop processor is not faulty; that is all processors send the same request.

Operation of a stable storage unit is summarized by the following program.

```
At each clock tick do;
   $T$:= current time;
   $R$:=  bag of received messages $m$ where:
                        $m$.time $= T - \delta$
   if $|R| = 0 \rightarrow$ skip
    [] $|R| = k+1$  $\wedge$
      ($\forall m, m' : m, m' \in R$:
            $m$.request $= m'$.request  $\wedge$
            $m$.origin $\neq m'$.origin )) $\rightarrow$
                 $m$:= elementof($R$);
                 if $m = <$..., write $val$ to $var > \rightarrow$
                                        $var := val$
                  [] $m = <$..., read $var > \rightarrow$
                                        forall $p \in fsp$ :
                                         send $var$ to $p$
                 fi
    [] otherwise  $\rightarrow$ $fsp$:= reconfiguration_rule($fsp$);
                 forall $p \in fsp$:
                   start recovery protocol at $p$
   fi
```

It should be clear that this implementation of stable storage changes the value of a variable only if all $k+1$ processors that make up a k-fail-stop processor request it to.

When the processors that make up a k-fail-stop processor read a variable from stable storage, they are all sent the same value. Unfortunately, this does not ensure that this value will be the one received by all. However, processors that receive the wrong value and subsequently behave differently as a consequence will be deemed faulty at that time; this is entirely reasonable, since a failure has indeed occurred.

If the failure of a fail-stop processor is detected, the computation that was in progress is moved to another fail-stop processor. To do this, a new k-fail-stop processor is configured, according to a *reconfiguration rule* and the recovery protocol for the action in progress at the time of the failure is started on each of these processors. However, note that a computing system that must be able to tolerate k failures need not involve $k+1$ k-fail-stop processors, or a total of $(k+1)^2$ processors. After a new fail-stop processor has been configured and the computation has been moved there, diagnostics can be started on the processors that made up the malfunctioning fail-stop processor. Processors that pass these tests can be returned to the pool of processors available for use in subsequent reconfigurations. Thus, a single faulty processor will not cause k working processors to become unavailable.

s-Stable Storage

We now relax the assumption, made in the previous section, that stable storage is completely reliable. An s-stable storage unit is constructed from $2s+1$ processors. Each of these keeps a copy of each variable contained in stable storage. Provided each processor in the approximation receives every request to read or write stable storage in the same order and in a timely fashion, each non-faulty processor will independently perform the same updates to its copy of stable storage. All the copies will therefore agree. By assumption, at most s of the processors used to implement s-stable storage can be faulty. Thus, the value of each variable in stable storage can be determined by taking the majority value of the $2s+1$ copies.

To ensure that all of the $2s+1$ processors that make up s-stable storage receive the same messages in the same order and in a timely fashion, a Byzantine Agreement protocol is used—each processor in the k-fail-stop processor serves as transmitter to the processors that make up the stable storage unit. Thus, as of time T, the processors that make up a stable storage unit can be certain that no request with timestamp smaller than $T-\Delta$ will ever be received, due to Byz1. Therefore, each can run the program described above with δ changed to Δ and messages placed in the bag R only if they result from a Byzantine Agreement, and again the implementation will change the value of a variable in stable storage only if all $k+1$ of the processors that make up a k-fail-stop processor request it to.

The operation of the processors making up a fail-stop processor must also be changed slightly for them to interact with an s-stable storage unit instead of with the fully reliable stable storage postulated in the previous section. After requesting the value of a variable from an s-stable storage unit, a processor will receive a value from every non-faulty processor in the s-stable storage approximation, whereas previously it would receive exactly one response from the fully reliable stable storage. Above, we assumed that the origin of a messages could be authenticated by a receiver. Thus, if (a faulty) one of these processors sends more than one message, the additional messages can be discarded. Consequently, at most $2s+1$ messages will be considered in response to a request to read a value from stable storage. Provided at most s processors are faulty, $s+1$, a majority of the values received should agree.

Similarly, since as many as s of the processors in an s-stable storage may be faulty, a recovery protocol should be started only after at least $s+1$ requests to do so have been received. Recall that each of the non-faulty processors in an s-stable storage unit will independently determine that there has been a failure, then compute the same reconfiguration, and finally attempt to start the recovery protocol running on the new fail-stop processor.

Implementation Notes

An s-stable storage unit can service a number of fail-stop processors, the only limitation being the aggregate rate of accesses that must be supported. Moreover, it is also possible for each physical processor to be multiprogrammed and thus to implement more than one virtual processor. If this is done, then it is important that the virtual processors that make up a given k-fail-stop processor and an s-stable storage unit be implemented by different physical processors. Otherwise, failures would not be independent—a single failure could result in the failure of multiple virtual processors.

It is easy to imagine situations in which various tasks require different degrees of fault-tolerance. In that case, a k_i-fail stop processor would be used for each task τ_i, where k_i is based on the degree of fault-tolerance required. A single s-stable storage unit can service such a heterogeneous collection of fail-stop processors, provided $s \geq k_i$, for all k_i.

Lastly, determination of k and s will be based on the reliability characteristics of the processors used. These parameters define the number of concurrent failures that can be tolerated, where failures are considered ***concurrent*** if they occur between successive accesses to stable storage. Given that the cost for processors with different reliabilities is likely to differ, it is desirable to use a few highly reliable processors in the implementation of stable storage, and less reliable, cheaper processors in the implementation of the fail-stop processor approximations. This reduces the cost of achieving Byzantine Agreement, because the total number of processors involved is smaller, without affecting reliability.

Fail-Stop Processors and the State Machine Approach

The implementation of the s-stable storage unit described above is an application of the ***state machine approach***, a general approach for constructing distributed programs first described in [3] and later extended for environments in which failures could occur in [4,5,12]. Given any program S, a distributed version that can tolerate up to t faults can be constructed by running S on $2t+1$ processors. Byzantine agreement is used to ensure that each version of the program reads the same inputs; majority voting is used to determine the output of the computation. The state machine approach is an attractive alternative to building and programming fail-stop processors because there is no need to partition a program into actions and to construct recovery protocols for those actions.

The fail-stop processor approach has its advantages, however. Consider an application that, if run on P fault-free processors, would meet its response-time constraints. If the state machine approach is used, in order for this program to be able to tolerate up to t faults,

$$N_{sm} = P(2t + 1)$$

processors are required. Suppose the fail-stop processor approach is used instead. Let us assume that $1/a$ of the instructions executed by the application involve accesses to stable storage. P t-fail-stop processors and an s-stable storage unit with sufficient bandwidth are required. The s-stable storage unit must respond to access requests from all $P(t+1)$ processors in the fail-stop processor approximations. This is a total of:

$$N_{fs} = P(t+1) + \frac{P(t+1)}{a}(2t+1)$$

processors. Thus, the fail-stop processor approach requires fewer processors provided:

$$\frac{1}{a} \leq \frac{t}{2t^2 + 3t + 1}$$

If accesses to stable storage are not frequent, then using fail-stop processors is cheaper then the state machine approach because $P(t+1)$ processors do most of the processing and detect up to t failures, while in the state machine approach $P(2t+1)$ processors are required to do that processing. However, when the fail-stop processor approach is used, a few additional processors are required to implement stable storage.

Other significant differences between the two approaches are, unfortunately, more difficult to quantify. They include:

- The fail-stop processor approach may incur additional response time when a task is moved from one fail-stop processor to another because its recovery protocol must be executed. Such delays are not incurred when the state machine approach is used.
- We have only limited experience designing and programming applications in terms of actions and recovery protocols. It is likely to be more difficult to write a program in such a stylized manner, as required in the fail-stop processor approach. The state machine approach imposes no such constraints. Also, the running time of programs that are structured in terms of actions and recovery protocols may be higher because information must be moved periodically to stable storage.

Discussion

We believe the fail-stop processor approach to be a viable way to construct fault-tolerant computing systems. It does not involve appreciably more processors than required by the state machine approach, and in some cases requires fewer processors.

Another popular scheme for constructing reliable computing systems involves majority voting and "warm stand-bys": a program is run on t processors and if ever there is a disagreement among these processors, the dissenting one is taken off-line and replaced by one of the stand-bys. Such a scheme is clearly less fault-tolerant than ours, since it is not immune to concurrent failures. Moreover, a careful analysis shows that such a scheme requires a comparable number of processors as the fail-stop and state machine approaches do, although it can tolerate only a limited class of failures.

A major cost incurred when using the fail-stop approach is that of achieving Byzantine Agreement. This cost need not be significant, however, since actions can be designed to make infrequent references to stable storage. In effect, updates to stable storage are batched by the processors that make up a fail-stop processor. When this is done, in the worst case a big chunk of the computation will have to be repeated after a failure has been detected by the stable storage unit.

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.

There are undoubtedly other ways to approximate fail-stop processors. By using conservative design practices and introducing redundancy at lower levels, it should be possible to implement a computing system that, with high probability, behaves like a fail-stop processor. The fact that programmers have long treated disk drives as acceptable approximations of stable storage and real processors as acceptable approximations of fail-stop processors strengthens this argument.

Acknowledgments

B. Alpern, K. Birman, D. Gries, L. Lamport, R. Schlichting, and D. Skeen provided helpful comments on an earlier draft of this paper.

References

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

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

[3] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM 21,* 7 (July 1975), 558-565.

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] Schlichting, R.D., F.B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. Submitted to *TOPLAS.* Also available as TR 81-479, Dept. of Computer Science, Cornell University, Ithaca, New York.

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

[13] 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