The State Machine Approach:

A Tutorial[*]

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract. The state machine approach is a general method for achieving fault tolerance and implementing decentralized control in distributed systems. This paper reviews the approach and identifies abstractions needed for coordinating ensembles of state machines. Implementations of these abstractions for two different failure models—Byzantine and fail-stop—are discussed. The state machine approach is illustrated by programming several examples. Optimization and system reconfiguration techniques are explained.

1. Introduction

The state machine approach is a general method for managing replication. It has broad applicability for implementing distributed and fault-tolerant systems. In fact, every protocol we know of that employs replication—be it for masking failures or simply to facilitate cooperation without centralized control—can be derived using the state machine approach. Although few of these protocols actually were obtained in this manner, viewing them in terms of state machines helps in understanding how and why they work.

When the state machine approach is used for implementing fault tolerance, a computation is replicated on processors that are physically and electrically isolated. This permits the effects of failures to be masked by voting on the outputs produced by these independent replicas. Triple-modular redundancy (TMR) is a familiar example of this scheme. Although when the approach is used additional

coordination is necessary to distribute inputs and collect outputs from replicas, failures cannot increase task completion times. This makes the approach ideally suited for real-time systems, where deadlines must be met and timing is critical. Other approaches, such as failure detection and retry, are ill suited for real-time applications because unpredictable delays can be observed in response to a failure.

The state machine approach permits separation of fault-tolerance from other aspects of system design. The programmer is not forced to think in terms of a particular programming abstraction, such as transactions [Liskov 85] [Spector 85], fault-tolerant actions [Schlichting & Schneider 83], reliable objects [Birman 85], replicated remote procedure calls [Cooper 84] or the multitude of other proposals that have appeared in the literature. Instead, a programming abstraction suited for the application at hand can be defined and used; the state machine approach is employed to realize a fault-tolerant implementation of that abstraction.

This paper is a tutorial on the state machine approach. It describes the approach and its implementation for two representative environments. Small examples suffice to illustrate the points; however, the approach has been successfully applied to larger examples. Section 2 describes how a system can be viewed in terms of a state machine, clients, and output devices. Measures of fault-tolerance are discussed in section 3. Achieving fault-tolerance is the subject of the following three sections: Section 4 discusses implementing fault-tolerant state machines; section 5 discusses tolerating faulty output devices; and section 6 discusses coping with faulty clients. An important class of optimizations—based on the use of time—is discussed in section 7. Optimizations possible by making assumptions about failures are discussed in section 8. Section 9 describes dynamic reconfiguration. Related work is discussed in section 10.

2. State Machines

A *state machine* consists of *state variables*, which implement its state, and *commands*, which transform its state. Each command is implemented by a deterministic program; execution of the command is atomic with respect to other commands and modifies the state variables and/or produces some output. A *client* of the state machine makes a *request* to specify execution of a command. The request names a state machine, names the command to be performed, and contains any information needed by the command. Output from request processing can be to an actuator (e.g. in a process-control system), to some other peripheral device (e.g. a disk or terminal), or to clients awaiting responses from prior requests.

The name "state machine" is a poor one, since it is suggestive of a finite-state automata. Our state machines are more powerful than finite-state automata because they contain program variables and, therefore, need not be finite state. However, state machines intended for execution on real machines should be finite state because real computers have finite memories. Our state machines are also easier to specify than finite-state automata because any programming notation can be used. "State machine" is used in this paper for historical reasons—it is the term used in the literature.

State machines can be described by explicitly listing state variables and commands. As an example, state machine *memory* of Figure 2.1 implements a mapping from locations to values. A *read* command permits a client to determine the value associated with a location, and a *write* command associates a new value with a location. Observe that there is little difference between our state machine description of *memory* and an abstract datatype or (software) module for such an object. This is deliberate—it makes it clear that state machines are a general programming construct. In fact, a state machine can be implemented in a variety of ways. It can be implemented as a collection of procedures that share data,

```
memory : state_machine
            var store : array [0..n] of word

read :  command(loc : 0..n)
            send store [loc] to client
            end read;

write :  command(loc : 0..n , value : word)
            store [loc] := value
            end write
         end memory
```

Figure 2.1. A memory

as in a module; it can be implemented as a process that awaits messages containing requests and performs the actions they specify; and, it can be implemented as a collection of interrupt handlers, in which case a request is made by causing an interrupt. (Disabling interrupts permits each command to be executed to completion before the next is started.) For example, the state machine of Figure 2.2 implements commands to ensure that at all times at most one client has been granted access to some resource.[1] It would likely be implemented as a collection of interrupt handlers as part of the kernel of an operating system.

Requests are processed by a state machine one at a time, in an order consistent with causality. Therefore, clients of a state machine can be programmed under the assumptions that

O1:   requests issued by a single client to a given state machine *sm* are processed by *sm* in the order they were issued, and

O2:   if the fact that request *r* was made to a state machine *sm* by client *c* could have caused a request *r'* to be made by a client *c'* to *sm*, then *sm* processes *r* before *r'*.

In this paper, for expository simplicity, client requests are specified as tuples of the form

$\langle state\_machine.command, arguments \rangle$

and the return of results is done using message passing. For example, a client could execute

$\langle memory.write, 100, 16.2 \rangle$;
$\langle memory.read, 100 \rangle$;
receive *v* from *memory*

to set the value of location 100 to 16.2, request the value of location 100, and await that value, setting *v* to it upon receipt.

---

[1]We use *x∘y* to append *y* to the end of list *x*.

```
mutex: state_machine
          var user : client_id init Φ;
                waiting : list of client_id init Φ

acquire: command
          if user=Φ → send OK to client;
                          user := client

          [] user≠Φ → waiting := waiting o client
          fi
          end acquire

release: command
          if waiting =Φ → user := Φ

          [] waiting ≠Φ → send OK to head(waiting);
                              user := head(waiting);
                              waiting := tail(waiting)
          fi
          end release
      end mutex
```

Figure 2.2. A resource allocator

The defining characteristic of a state machine is not its syntax, but that it specifies a deterministic computation that reads a stream of requests and processes each, occasionally producing output:

Semantic Characterization of State Machine. Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time and any other activity in a system.

Any program satisfying this definition will be considered a state machine for the purposes of this paper.

For example, the following program solves a simple process-control problem in which an actuator is adjusted repeatedly based on the value of a sensor. Periodically, a client reads a sensor and communicates the value read to state machine pc:

```
monitor: process
              do true → val := sensor;
                          ⟨pc.adjust, val⟩;
                          delay D
          od
          end monitor
```

State machine pc adjusts an actuator based on past adjustments saved in state variable q, the sensor reading, and a control function F.

```
pc:  state_machine
    var q : real;

adjust:  command(sensor_val : real)
         q := F(q, sensor_val);
         send q to actuator
         end adjust
    end pc
```

Although it is tempting to structure *pc* as a single command that loops—reading from the sensor, evaluating *F*, and writing to *actuator*—if the value of the sensor is time-varying, then the result would not satisfy the semantic characterization given above and therefore would not be a state machine. This is because values sent to *actuator* (the output of the state machine) would not depend solely on the requests made to the state machine but would, in addition, depend on the execution speed of the loop. In the structure used above, this problem has been avoided by moving the loop into *monitor*.

Having to structure a system in terms of state machines and clients does not constitute a restriction. Anything that can be structured in terms of procedures and procedure calls can also be structured using state machines and clients—a state machine implements the procedure, and requests implement the procedure calls. In fact, state machines permit more flexibility in system structure than is usually available with procedure calls. With state machines, a client making a request is not delayed until that request is processed, and the output of a request can be sent someplace other than to the client making the request. We have not yet encountered an application that could not be programmed cleanly in terms of state machines and clients.

3. Fault-Tolerance

A component is *faulty* once its behavior is no longer consistent with its specification. In this paper, we consider two representative classes of faulty behavior from a spectrum of possible ones:

Byzantine Failures. The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components [Lamport et al 82].

Fail-stop Failures. In response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and then stops [Schneider 84].

Byzantine failures can be the most disruptive, and there is anecdotal evidence that such failures do occur in practice. Allowing Byzantine failures is the weakest possible assumption that could be made about the effects of a failure. Since a design based on assumptions about the behavior of faulty components runs the risk of failing if these assumptions are not satisfied, it is prudent that life-critical systems tolerate Byzantine failures. However, for most applications, it suffices to assume fail-stop failures.

A system consisting of a set of distinct components is *f fault-tolerant* if it satisfies its specification provided that no more than *f* of those components become faulty during some interval of interest.[2] Fault-tolerance traditionally has been specified in terms of MTBF (mean-time-between-failures), probability of failure over a given interval, and other statistical measures [Siewiorek & Swarz 82]. While it is clear that such characterizations are important to the users of a system, there are advantages in

---

[2]An *f* fault-tolerant system might continue to operate correctly if more than *f* failures occur, but correct operation cannot be guaranteed.

describing fault tolerance of a system in terms of the maximum number of component failures that can be tolerated over some interval of interest. Asserting that a system is f fault-tolerant makes explicit the assumptions required for correct operation; MTBF and other statistical measures do not. Moreover, f fault-tolerance is unrelated to the reliability of the components that make up the system and therefore is a measure of the fault tolerance supported by the system architecture, in contrast to fault tolerance achieved simply by using reliable components. Of course, MTBF and other statistical reliability measures of an f fault-tolerant system will depend on the reliability of the components used in constructing that system—in particular, the probability that there will be f or more failures during the operating interval of interest. Thus, f should be chosen based on statistical measures of component reliability. Once f has been chosen, it is possible to derive MTBF and other statistical measures of reliability by computing the probabilities of various configurations of 0 through f failures and their consequences [Babaoglu 86].

4. Fault-tolerant State Machines

An f fault-tolerant state machine can be implemented by replicating it and running a copy on each of the processors in a distributed system. Provided each copy being run by a non-faulty processor starts in the same initial state and executes the same requests in the same order, then each will do the same thing and produce the same output. If we assume that each failure can affect at most one processor, hence one state machine copy, then by combining the output of the state machine copies in this *ensemble*, the output for a t fault-tolerant state machine can be obtained.

When processors can experience Byzantine failures, an ensemble implementing a f fault-tolerant state machine must have at least $2f+1$ copies, and the output of the ensemble is the output produced by the majority of the state machine copies. This is because with $2f+1$ copies, the majority of the outputs remain correct even after as many as f failures. If processors experience only fail-stop failures, then an ensemble containing $f+1$ copies suffices, and the output of the ensemble can be the output produced by any of its members. This is because only correct outputs are produced by fail-stop processors, and after f failures one non-faulty copy will remain among the $f+1$ copies.

Our scheme for implementing an f fault-tolerant state machine is based on fault-tolerant implementations of two abstractions.

Agreement. Every non-faulty copy of the state machine receives every request.

Order. Requests are processed in the same order by every non-faulty copy of the state machine.

However, knowledge of command semantics sometimes permit weaker (i.e., cheaper to implement) abstractions to be used. For example, when fail-stop processors are used, a request whose processing does not modify state variables need only be sent to a single non-faulty state machine copy, thus permitting relaxation of Agreement. It also possible to exploit request semantics to relax Order. Two requests r and r' *commute* in a state machine if the sequence of outputs that would result from processing r followed by r' is the same as would result from processing r' followed by r. Not surprisingly, the schemes outlined above for combining outputs of the members of an ensemble work even when two requests that commute are processed in different orders by different state machine copies in an ensemble, thus permitting relaxation of Order.

An example of a state machine where Order is not necessary appears in Figure 4.1. State machine *tally* determines the first from among a set of alternatives to receive at least *MAJ* votes and sends this choice to *SYSTEM*. If clients cannot vote more than once and the number of clients *Cno* satisfies $2MAJ > Cno$, then every request commutes with every other. Thus, implementing Order would be

```
tally:  state_machine
           var votes : array[candidate] of integer init 0

   cast_vote:  command(choice : candidate)
                  votes [choice] := votes [choice]+1;
                  if votes [choice]≥MAJ → send choice to SYSTEM;
                                          halt
                  [] votes [choice] < MAJ → skip
                  fi
                  end cast_vote
           end tally
```

Figure 4.1. Election

unnecessary—different copies of the state machine will produce the same outputs even if they process requests in different orders. On the other hand, if clients can vote more than once or $2MAJ \leq Cno$, then reordering requests beyond the first $MAJ-1$ might change the outcome of the election.

Various implementations of Agreement and Order are possible. Two are discussed below.

4.1. Agreement

The Agreement abstraction is implemented by any protocol that allows a designated processor, called the *transmitter*, to disseminate a value to other processors in such a way that:

   IC1:   All non-faulty processors agree on the same value.

   IC2:   If the transmitter is non-faulty then all non-faulty processors use its value as the one they agree on.

If whenever a client makes a request, it employs a protocol satisfying IC1 and IC2 to disseminate that request to all copies of the state machine, then Agreement is achieved. Thus, we have:

   Agreement Implementation. Clients disseminate requests using a protocol that establishes IC1 and IC2.

Notice that the Agreement Implementation does not require a client to be the transmitter. A client might send its request to a single copy of the state machine and let that copy serve as the transmitter in further distributing the request to the other members of the ensemble. This permits clients to be simpler. However, a request can be lost or corrupted if the client sends it directly to only a single copy of the state machine and that copy is being executed by a faulty processor.

Protocols to establish IC1 and IC2 have received much attention in the literature. If digital signatures are available and processors can exhibit Byzantine failures or if processors are restricted to fail-stop failures, then $f+1$ processors are sufficient in order to tolerate f failures; otherwise $3f+1$ processors are necessary to tolerate f failures. See [Strong & Dolev 83] for a survey of protocols that can tolerate

Byzantine processor failures and [Schneider et al 84] for a (significantly cheaper) protocol that can tolerate (only) fail-stop processor failures.

4.2. Order

The Order abstraction can be implemented by having clients assign unique identifiers to requests and having state machine copies process requests according to a total ordering relation on these unique request identifiers. However, simply having each state machine copy process in ascending order the requests it has received does not imply that every state machine will processes requests in the same order. Two requests could be delivered to one state machine in one order and to another state machine in the other order. We must devise a way to avoid this problem.

We shall say that a request is *stable at p* once no request from a correct client and bearing a lower unique identifier can be subsequently delivered to the state machine copy at processor p. Given an implementation of stability, the following is an implementation for the Order abstraction:

> **Order Implementation.** Stable requests are processed by a state machine in ascending order by unique identifier.

The choice of request identifiers is further constrained when clients of a state machine are programmed under the assumption that requests are processed in an order consistent with potential causality (i.e., O1 and O2 of section 2). Now, processing requests in ascending order by unique identifier must be in an order consistent with O1 and O2. One way to produce unique request identifiers satisfying O1 and O2 is to use logical clocks; a second way is to use approximately synchronized real-time clocks.

Using Logical Clocks

A *logical clock* [Lamport 78a] is a mapping Λ from events to the integers. $\Lambda(E)$, the "time" assigned to an event E by logical clock Λ, is such that for any two distinct events E and F, either $\Lambda(E) < \Lambda(F)$ or $\Lambda(F) < \Lambda(E)$, and if E might be responsible for causing F then $\Lambda(E) < \Lambda(F)$. It is a simple matter to implement logical clocks in a distributed system. Associated with each process p is a counter λ_p. A *timestamp* is included in each message sent by p. This timestamp is the value of λ_p when that message is sent. In addition, λ_p is changed according to:

CU1: λ_p is incremented after each event at p.

CU2: Upon receipt of a message with timestamp τ, process p resets λ_p:
$$\lambda_p := \max(\lambda_p, \tau) + 1.$$

The value of $\Lambda(E)$ for an event E that occurs at processor p is constructed by appending a fixed-length bit string that uniquely identifies p to the value of λ_p when E occurs.

A logical clock can be used to implement a mapping from requests to unique identifiers with a total ordering that satisfies O1 and O2. $\Lambda(E)$ is used as the unique identifier associated with a request whose issuance corresponds to event E. Therefore, all that remains for an implementation of the Order abstraction is to formulate a test for stability.

It is pointless to implement a stability test in an asynchronous system[3] where Byzantine failures are

---

[3] A system is *asynchronous* if message delivery delay or the relative speeds of processors is unbounded; it is *synchronous* if delivery delay and relative processor speeds are bounded.

possible. This is because no deterministic protocol can achieve IC1 and IC2 under these conditions [Fischer et al 85], so it is impossible to implement Agreement.[4] Since it is impossible to implement Agreement, there is no point in implementing Order. The case where processors are synchronous is equivalent to assuming that they have synchronized real-time clocks and will be considered shortly. This leaves the case where processors are asynchronous and can exhibit fail-stop failures. We now turn to that.

By attaching sequence numbers to the messages between every pair of processors, it is trivial to ensure that the following property holds of communications channels.

FIFO Channels. Messages between a pair of processors are delivered in the order sent.

We also assume:

Failure Detection Assumption. A processor p detects that a fail-stop processor q has failed only after p has received the last message sent to p by q.

The Failure Detection Assumption is consistent with FIFO Channels, since the failure event for a fail-stop processor necessarily happens after the last message sent by that processor and, therefore, should be received after all other messages.

Using logical clocks to generate request identifiers implies that a request made by a client must have a larger unique identifier than was assigned to any previous request made by that client. Thus, assuming FIFO Channels, once a request from a client c is received by a copy sm_i of the state machine, no request from c with smaller unique identifier can be received by sm_i. Moreover, if sm_i detects that c has failed then no request from c with smaller unique identifier can be received by sm_i, due to the nature of fail-stop failures and the Failure Detection Assumption. Combining these restrictions, we deduce that if c_1, c_2, ... c_n are all the clients that have not failed and request r_k, with unique request identifier $uid(r_k)$, denotes the last request received by sm_i from c_k, then any request subsequently received by sm_i must have a unique identifier that is larger than min_uid, where

$$min\_uid = \min_{1 \leq k \leq n} uid(r_k).$$

This means that every request with unique identifier at most min_uid must be stable. There is one remaining flaw in this scheme, however. A non-faulty client that does not make requests—for whatever reason—will prevent requests from becoming stable. This problem can be avoided by requiring that otherwise inactive clients periodically make null requests. Summarizing, we get:

Logical Clock Stability Test Tolerating Fail-stop Failures. Every client periodically makes some request to the state machine. A request is stable at sm_i if a request with larger timestamp has been received by sm_i from every client running on a non-faulty fail-stop processor.

Synchronized Real-Time Clocks

A second way to produce unique request identifiers satisfying O1 and O2 is with approximately synchronized real-time clocks. Define $T_p(E)$ to be the value of the real-time clock at processor p when

---

[4]The result of [Fischer et al 85] is actually stronger than this. It states that IC1 and IC2 cannot be achieved by a deterministic protocol in an asynchronous system with a single processor that fails in an even less restrictive manner—by simply halting.

event E occurs. We use $T_p(E)$ followed by a fixed-length bit string that uniquely identifies p as the unique identifier associated with a request made as event E by a client running on a processor p. To ensure that O1 and O2 (of section 2) hold for unique identifiers generated in this manner, two restrictions are required. O1 follows provided no client makes two or more requests between successive clock ticks. If processor clocks have resolution ρ, then each client can make at most one request every ρ seconds. O2 follows provided the degree of clock synchronization is better than the minimum message delivery time. If clocks on different processors are synchronized to within δ seconds, then it must take more than δ seconds for a message from one client to reach another; otherwise, O2 would be violated because a request r made by one client could have a unique identifier that was smaller than a request r' made by another, even though r was caused by a message sent after r' was made.

A number of protocols to achieve clock synchronization while tolerating Byzantine failures have been proposed. They are surveyed in [Schneider 86]. The protocols require that known bounds exist for the execution speed and clock rates of non-faulty processors and for message delivery delays along non-faulty communications links. These requirements do not constitute a restriction in practice. Clock synchronization achieved by the protocols is proportional to the variance in message delivery delay, making it possible to satisfy the restriction—necessary to ensure O2—that message delivery delay exceeds clock synchronization.

A stability test can be implemented by exploiting synchronized real-time processor clocks and the bounds on delivery delays. If requests are disseminated using a protocol employing a fixed number of rounds, like the ones cited above for establishing IC1 and IC2, then there will exist a constant Δ such that a request r with unique identifier $uid(r)$ will be received by every correct processor no later than time $uid(r)+\Delta$ according to the local clock at the receiving processor.[5] Thus, once the clock on a processor p reaches τ, p cannot subsequently receive a request r such that $uid(r)<\tau-\Delta$. Therefore, a stability test is:

> **Real-time Clock Stability Test Tolerating Byzantine Failures I.** A request r is stable at a state machine copy sm_i being executed by processor p if the local clock at p is τ and $uid(r)<\tau-\Delta$.

One disadvantage of this stability test is that it forces the state machine to lag behind its clients by Δ, where Δ is proportional to the worst-case message delivery delay. This disadvantage can be avoided. Due to property O1 of the total ordering on request identifiers, if communications channels satisfy FIFO Channels, then a state machine copy that has received a request r from a client c can subsequently receive from c only requests with unique identifiers greater than $uid(r)$. Thus, a request r is also stable at a state machine copy provided a request with larger unique identifier has been received from every client.

> **Real-time Clock Stability Test Tolerating Byzantine Failures II.** A request r is stable at a state machine copy sm_i being executed by processor p if a request with larger unique identifier has been received from every client.

This second stability test is foiled by a single faulty processor that refuses to make requests. However, by combining the first and second test, so that a request is considered stable when it satisfies either test,

---

[5]In general, Δ will be a function of the variance in message delivery delay, the maximum message delivery delay, and the degree of clock synchronization. See [Cristian et al 85] for a detailed derivation for Δ in a variety of environments.

a stability test results that lags clients by Δ only when faulty processors or network delays force it.

5. Tolerating Faulty Output Devices

When implementing an f fault-tolerant system, there are problems with using a single voter to combine the outputs of an ensemble of state machine copies into a single output. In particular, a single failure—of the voter—can prevent the system from producing the correct output. The solution to this problem depends on how the output of the state machine implemented by the ensemble is used.

Outputs Used Outside the System

If the output of the state machine is sent to an output device, then that device is already a single component whose failure cannot be tolerated. Thus, being able to tolerate a faulty voter is not sufficient—the system must also be able to tolerate a faulty output device. The usual solution to this problem is to replicate the output device and voter. Each voter combines the output of each state machine copy, producing a signal that drives one output device. Whatever reads the outputs of the system is assumed to combine the outputs of the replicated devices. This reader, which is not considered part of the computing system, implements the critical voter.

If output devices can exhibit Byzantine failures, then by taking the output produced by the majority of the devices, $2f+1$-fold replication permits up to f faulty output devices to be tolerated. For examp a flap on an airplane wing might be designed so that when the $2f+1$ actuators that control it do agree, the flap always moves in the direction of the majority (rather than twisting, which would be a voter failure). If output devices exhibit only fail-stop failures, then only $f+1$-fold replication is necessary to tolerate f failures because any output produced by a fail-stop output device can be assumed correct. For example, terminals usually present information with enough redundancy so that they can be treated as fail-stop—failure detection is implemented by the viewer. With such an output device, a human user can look at a one of $f+1$ devices, decide whether the output is faulty, and only if it is faulty, look at another, and so on.

Outputs Used Inside the System

If the output of the state machine is to a client, then the client itself can combine the outputs of state machine copies in the ensemble. Here, the voter—a part of the client—is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant. When Byzantine failures are possible, the client waits until it has received $f+1$ identical responses, each from a different member of the ensemble, and takes that as the response from the f fault-tolerant state machine. When only fail-stop failures are possible, the client waits until it has received the first response from any member of the ensemble and takes that as the response from the f fault-tolerant state machine.

When the client is executed on the same processor as one of the state machine copies and Byzantine failures can occur, optimization of client-implemented voting is possible.[6] Now, the local state machine copy is correct exactly when the client is. Therefore, the response produced by the state

---

[6]Care must be exercised when analyzing the fault-tolerance of such a system because a single processor failure can now cause two system components to fail. Implicit in most of our discussions is that system components fail independently. It is not always possible to transform a f fault-tolerant system in which clients and state machine copies have independent failures to one in which they share processors.

machine copy running locally can be used as that client's response from the f fault-tolerant state machine and we have:

> **Dependent-Failures Output Optimization.** If a client and a state machine copy run on the same processor, then even when Byzantine failures are possible, the client need not gather a majority of responses to its requests to the state machine. It can use the single response produced locally.

6. Tolerating Faulty Clients

Implementing an f fault-tolerant state machine is not sufficient for implementing an f fault-tolerant system. Faulty clients must not be able to make requests that cause the state machine to produce erroneous output or that corrupt the state machine so that subsequent requests from non-faulty clients are incorrectly processed. When a client is itself structured as a state machine or can be restructured as a state machine, the approach of section 4 can be used to implement an f fault-tolerant client. Unfortunately, such restructuring is not always possible and other, application-dependent, techniques must sometimes be employed.

6.1. Sensor Replication

A client c that obtains input from a time-varying input source can be restructured as a state machine $sm(c)$ by isolating that time-varying input source and making it a client c' of $sm(c)$. While $sm(c)$ can be made f fault-tolerant (using the approach of section 4), this appears to bring us no closer to solving the original problem—the new client c' is still not fault tolerant and still obtains input from a time-varying input source. One solution to this dilemma is to restructure $sm(c)$, obtaining a state machine $sm'(c)$ that reads its input from multiple sensors, and therefore does not depend on the correctness of any single sensor. To accomplish this, client c' and its sensor are each replicated; every copy of c' reads from a different sensor. Where $sm(c)$ obtained a single value from the sensor, $sm'(c)$ obtains values from copies of c' and combines them. To summarize:

> **Fault-tolerant Sensor.** Given a client c that reads from a time-varying input source, the input source is replicated and c is restructured as a state machine $sm'(c)$ and a collection of clients. Each client reads from a different copy of the time-varying input source; $sm'(c)$ reads from all the clients.

If a sensor can exhibit Byzantine failures, then it must be replicated $2f+1$ fold; $sm'(c)$ chooses the median value. This works because even when as many as f copies of the client (c') or sensor are faulty, the median of these $2f+1$ values is guaranteed to be either a value from a correct sensor or bounded by values from correct sensors. If sensors exhibit only fail-stop failures, then $f+1$ fold replication suffices, and $sm'(c)$ can chooses any sensor value that is known not to be faulty.

It is possible to optimize the implementation of a fault-tolerant sensor when the same processors are being used both to run copies of the state machine of which c is a client and to run copies of $sm'(c)$. Since the output of $sm'(c)$ is destined to another state machine—say m—we could use the output produced by the single copy of $sm'(c)$ as input to m instead of combining the outputs produced by copies of $sm'(c)$, as described in the Dependent-Failures Output Optimization of section 5. Moreover, we can merge the copy of $sm'(c)$ and m, obtaining a single state machine. For example, when this scheme is applied to the process control system of section 2, *monitor* is c' and is replicated, and pc is the result of combining m and $sm'(c)$. This is shown for the case where Byzantine failures are possible in Figure 6.1.

```
monitor [i ]:  process
              do true → val := sensor[i ];
                        ⟨pc.adjust, i, val⟩;
                        delay D
              od
          end monitor [i ]


pc :  state_machine
      var q : real;
          resp : set of client_id init Φ;
          val_rcd : array[1..2t+1] of real;

    adjust :  command(cid, sensor_val)
              resp := resp ∪ cid;
              val_rcd [cid] := sensor_val;
              if ⏐resp⏐ <⌈(2t+1)/2⌉ → skip
              [] ⏐resp⏐ ≥⌈(2t+1)/2⌉ → q := F(q, median(val_rcd [c ]));
                                                      c ∈ resp
                                        resp := Φ;
                                        send q to actuator
              fi
          end adjust
      end pc
```

Figure 6.1. Revised process control system

6.2. Defensive Programming

Sometimes a client cannot be restructured as a state machine, and thus cannot be made f fault-tolerant using the approach just described. If it were possible for state machine copies to agree on the identities of faulty clients, then tolerating faulty clients would be simple—ignore requests from them. Unfortunately, this is not always possible. When Byzantine failures can occur, not all failures will produce identifiable symptoms. Without restricting possible failure modes, there is no way for a state machine to be able to identify faulty clients. However, careful design of a state machine can limit the effects of faulty requests. For example, *memory* (Figure 2.1) permits any client to write to any location. Therefore, a faulty client can overwrite all locations, destroying valuable information in state variables. This problem could be prevented by restricting write client access to only certain memory locations— the state machine can enforce this.

Including tests in commands is another way to design a state machine that cannot be corrupted by requests from faulty clients. For example, *mutex* as specified in Figure 2.2, will execute a *release* command made by any client—even one that does not have access to the resource. Consequently, a faulty client could issue such a request and cause *mutex* to grant a second client access to the resource before

the first has relinquished access. A better formulation of *mutex* ignores *release* commands from all but the client to which exclusive access has been granted. This is implemented by changing the *release* in *mutex* to:

> *release*: **command**
> > **if** *user* ≠ *client* → **skip**
> > [] *waiting* = Φ ∧ *user* = *client* → *user* := Φ
> > [] *waiting* ≠ Φ ∧ *user* = *client* → **send** *OK* **to** *head*(*waiting*);
> > > *user* := *head*(*waiting*);
> > > *waiting* := *tail*(*waiting*)
> > **fi**
> > **end** *release*

Sometimes, a faulty client *not* making a request can be just as catastrophic as one making an erroneous request. For example, if a client of *mutex* failed and stopped while it had exclusive access to the resource, then no client could be granted access to the resource. Of course, unless we are prepared to bound the length of time that a correctly functioning process can retain exclusive access to the resource, there is little we can do about this problem. This is because there is no way for a state machine to distinguish between a client that has stopped executing because it has failed and one that is executing very slowly. However, given an upper bound B on the interval between an *acquire* and the following *release*, *mutex* can automatically schedule *release* on behalf of a client. This is done by having the *acquire* command automatically schedule the *release* request.

We introduce the notation

> **schedule** ⟨*REQUEST*⟩ **for** +τ

to specify scheduling ⟨*REQUEST*⟩ with a unique identifier τ greater than the identifier on the request being processed. Such a request is called a *timeout request* and becomes stable at some time in the future, according to the stability test being used for client-generated requests. Unlike requests from clients, requests that result from executing **schedule** need not be distributed to all state machine copies of the ensemble. This is because each state machine copy will independently **schedule** its own (identical) copy of the request.

We can now modify *acquire* so that a *release* operation is automatically scheduled:[7]

---

[7]This means that *mutex* might process two *release* commands on behalf of a client: one from the client itself and one generated by its *acquire* request. The new state variable *time_granted* permits such superfluous commands to be ignored.

acquire: **command**
    if *user* = Φ → **send** *OK* **to** *client*;
                user := *client*;
                time_granted := *NOW*;
                schedule ⟨*mutex.timeout*, *time_granted*⟩ **for** +*B*
    [] *user* ≠ Φ → *waiting* := *waiting* ∝ *client*
    fi
    end *acquire*

timeout: **command**(*when_granted* : **integer**)
    if *when_granted* ≠ *time_granted* → **skip**
    [] *waiting* = Φ ∧ *when_granted* = *time_granted* → *user* := Φ
    [] *waiting* ≠ Φ ∧ *when_granted* = *time_granted* →
                    send *OK* **to** *head*(*waiting*);
                    user := *head*(*waiting*);
                    time_granted := *NOW*;
                    waiting := *tail*(*waiting*)
    fi
    end *timeout*

7. Using Time to Make Requests

A client need not explicitly send a message to make a request. Not receiving a request can trigger execution of a command—in effect, allowing the passage of time to transmit a request from client to state machine [Lamport 84]. Transmitting a request using time instead of messages can be advantageous because protocols that implement IC1 and IC2 can be costly both in total number of messages exchanged and in delay. Unfortunately, using time to transmit requests has only limited applicability, since the client cannot specify parameter values.

The use of time to transmit a request was employed in section 6 when we revised the *acquire* command of *mutex* to foil clients that failed to release the resource. There, a *release* request was automatically scheduled by *acquire* on behalf of a client being granted the resource. A client transmits a *release* request to *mutex* simply by permitting *B* (logical clock or real-time clock) time units to pass. It is only to increase utilization of the shared resource that a client might use messages to transmit a *release* request to *mutex* before *B* time units have passed.

A more dramatic example of using time to transmit a request is illustrated in connection with *tally* of Figure 4.1. Assume that

- all clients and state machine copies have (logical or real time) clocks synchronized to within Γ and

- the election starts at time *Strt* and this is known to all clients and state machine copies.

Using time, a client can cast a vote for a *default* by doing nothing; only when a client casts a vote different from its default do we require that it actually transmit a request message. Thus, we have:

    Transmitting a Default Vote. If client has not made a request by time *Strt* + Γ, then a request with that client's default vote has been made.

Notice that the default need not be fixed nor even known at the time the vote is cast. For example, the default vote could be "choose the first client that votes for itself". In that case, only one client—one that

votes for itself—need actually use message transmission to cast its vote. The result is a state machine to implement an election in which only the winner actually does something.

8. Exploiting Assumptions about Failures

Optimization of a state machine is frequently possible when assumptions can be made about the number and types of failures that can occur. The easiest assumption to make about failures is that they do not happen. Given a fault-free processor on which to execute a state machine, replication of the state machine becomes unnecessary and the Agreement and Order abstractions have trivial implementations: a client simply sends its request to the single state machine copy. Of course, the assumption that failures do not happen is not very realistic.

More realistic assumptions about failures permit somewhat less dramatic optimizations. To illustrate these, we consider various solutions to the database commit problem. A *commit protocol* permits an update to be performed on all or no copy of a replicated database according to a *commit rule* and information provided by the sites maintaining database copies. We can formulate a solution to the commit problem by using a state machine *commit*[*tid*] (see Figure 8.1) for each transaction *tid* and defining a client for each copy of the database. Each client involved in processing *tid* registers a suggested outcome—*commit* or *abort*—with *commit*[*tid*] and awaits a response. State machine *commit*[*tid*] runs forever, so that a client that has failed and restarted can ascertain the outcome of transactions it processed but neither committed nor aborted. In *commit*[*tid*], the commit rule is implemented by function

commit[*tid*]: **state_machine**
        var *sugs* : **array**[1..*maxclients*] **of** [*commit, abort, undecided*];
           wait_ans : **set of client_id**;
           outcome : [*commit, abort, undecided*] **init** *undecided*;

   status: **command**(*c_sug* : [*commit, abort*])
        sugs[*client*] := *c_sug*;
        if *outcome* = *undecided* → *outcome* := *Commit_Rule*(*sugs*)
        [] *outcome* ≠ *undecided* → **skip**
        fi;
        if *outcome* ≠ *undecided* → **send** *outcome* **to** *client*;
                            forall *pid* ∈ *wait_ans*:
                                  send *outcome* **to** *pid*
        [] *outcome* = *undecided* → *wait_ans* := *wait_ans* ∪ *client*
        fi
        end *status*
        end *commit*[*tid*]

Figure 8.1. Commit

Commit_Rule(sugs), which returns *commit*, *abort*, or *undecided*, based on the values in *sugs*. Note that it may also be necessary to employ timeout transitions in *commit*[*tid*] so that a faulty processor that does not register a suggested outcome cannot unconditionally delay the decision to commit or abort *tid*.

When *commit*[*tid*] is replicated and a copy is executed at each site running a client, the decentralized commit protocol of [Skeen 82] results. Other commit protocols that have appeared in the literature can be derived from *commit*[*tid*] by making assumptions about failures. For example, the 2-phase commit protocol described in [Gray 78] uses a single copy of the *commit*[*tid*] state machine and is based on two assumptions:

    (1)   The processor executing this state machine does not fail.

    (2)   Client failures are detectable (i.e., clients are fail-stop).

If the assumptions are violated, then the protocol may not work. In particular, if *commit*[*tid*] exhibits a fail-stop failure in the midst of sending *outcome* to clients in *wait_ans*, then (correct) clients might be unable to decide on an outcome, a phenomenon sometimes referred to as the "window-of-vulnerability" of this protocol; and if *commit*[*tid*] exhibits a Byzantine failure, then (correct) clients could receive conflicting information causing some to commit the transaction and others to abort it.

The 3-phase commit protocol of [Skeen 82] and 4-phase protocol of [Hammer and Shipman 80] result when more than a single copy of the *commit*[*tid*] state machine is run. These protocols can tolerate fail-stop failures of processors running copies of *commit*[*tid*] and of clients. However, they do not employ sufficient replication to tolerate Byzantine failures.

9. Reconfiguration

An ensemble of state machine copies can tolerate more than f faults if it is possible to remove state machine copies running on faulty processors from the ensemble and add copies running on repaired processors. (A similar argument can be made for being able to add and remove copies of clients and output devices.) Let $P(\tau)$ be the total number of processors at time τ that are executing copies of some state machine of interest, and let $F(\tau)$ be the number of them that are faulty. In order for the ensemble to produce the correct output, we must have

    Combining Condition: $P(\tau) - F(\tau) > Enuf$ for all $0 \leq \tau$.

$$\text{where } Enuf \equiv \left\{ \begin{array}{l} P(\tau)/2 \text{ if Byzantine failures are possible.} \\ 0 \text{ if only fail-stop failures are possible.} \end{array} \right.$$

A processor failure can cause the Combining Condition to be violated by increasing $F(\tau)$, thereby decreasing $P(\tau) - F(\tau)$.

When Byzantine failures are possible, if a faulty processor can be identified, then removing it from the ensemble decreases *Enuf* without further decreasing $P(\tau) - F(\tau)$; this can prevent the Combining Condition from being violated. When only fail-stop failures are possible, increasing the number of non-faulty processors—by adding one that has been repaired—is the only way to prevent the Combining Condition from being violated because increasing $P(\tau)$ is the only way to keep $P(\tau) - F(\tau) > 0$. Therefore, provided the following conditions hold, it may be possible to maintain the Combining Condition forever and thus tolerate an unbounded total number of faults over the life of the system.

    F1:   If Byzantine failures are possible, then state machine copies being executed by faulty processors are identified and removed from the ensemble before the Combining

Condition is violated.

F2:   State machine copies running on repaired processors are added to the ensemble so that the Combining Condition is not violated.

F1 and F2 constrain the rates at which failures and repairs occur.

Removing faulty processors from an ensemble of state machines can also improve system performance. This is because the number of messages that must be sent to achieve Agreement is usually proportional to the number of state machine copies that must agree on the contents of a request. In addition, some protocols to implement Agreement execute in time proportional to the number of processors that are faulty. Removing faulty processors clearly reduces both the message complexity and time complexity of such protocols.

Adding or removing a client from the system is simply a matter of changing the state machine so that henceforth it responds to or ignores requests from that client. Adding an output device is also straightforward—the state machine starts sending output to that device. Removing an output device from a system is achieved by *disabling* the device. This is done by putting the device in a state that prevents it from affecting the environment. For example, a CRT terminal can be disabled by turning off the brightness so that the screen can no longer be read; a hydraulic actuator controlling the flap on an airplane wing can be disabled by opening a cutoff valve so that the actuator exerts no presure on that control surface. However, as shown by these examples, it is not always possible to disable a faulty output device: turning off the brightness might have no effect on the screen and the cutoff valve might not work. Thus, there are systems in which no more than a total of t actuator faults can be tolerated because faulty actuators cannot be disabled.

The *configuration* of a system structured in terms of a state machine and clients can be described using three sets: the clients C, the state machine copies S, and the output devices O. S is used by the implementation of the Agreement abstraction and therefore must be known to clients and state machine copies. It can also be used by an output device to determine which **send** operations made by state machine copies should be ignored. C and O are used by state machine copies to determine from which clients requests should be processed and to which devices output should be sent. Therefore, C and O must be available to state machine copies.

Two problems must be solved to support changing the system configuration. First, the values of C, S, and O must be available when required. Second, whenever a client, state machine copy, or output device is added to the configuration, the state of that *element* must be updated to reflect the current state of the system. These problems are considered in the following two subsections.

9.1. Managing the Configuration

The configuration of a system can be managed using the state machine in that system. Sets C, S, and O are stored in state variables and changed by commands. Each configuration is *valid* for a collection of requests—those requests r such that $uid(r)$ is in the range defined by two successive configuration-change requests. Thus, whenever a client, state machine copy, or output device performs an action connected with processing r, it uses the configuration that is valid for r. This means that a configuration-change request must schedule the new configuration for some point far enough in the future so that clients, state machine copies, and output devices can find out about the new configuration before it actually comes into effect.

There are various ways to make configuration information available to the clients and output devices of a system. (The information is already available to the state machine.) One is for clients and output devices to query the state machine periodically for information about relevant pending configuration changes. Obviously, communication costs for this scheme are reduced if clients and output devices share processors with state machine copies. Another way to make configuration information available is for the state machine to include information about configuration changes in messages it sends to clients and output devices in the course of normal processing. Doing this requires regular and periodic communication between the state machine and clients and between the state machine and output devices.

Requests to change the configuration of the system are made by a failure/recovery detection mechanism. It is convenient to think of this mechanism as a collection of clients, one for each element of C, S, or O. Each of these *configurators* is responsible for detecting the failure or repair of the single object it manages and, when such an event is detected, for making a request to alter the configuration. A configurator is likely to be part of an existing client or state machine copy and might be implemented in a variety of ways.

When elements are fail-stop, a configurator need only check the failure-detection mechanism of that element. When elements can exhibit Byzantine failures, detecting failures is not always possible. When it is possible, a higher degree of fault tolerance can be achieved by reconfiguration. A non-faulty configurator satisfies two safety properties.

C1: Only a faulty element is removed from the configuration.

C2: Only a non-faulty element is added to the configuration.

However, a configurator that does nothing satisfies C1 and C2. Changing the configuration enhances fault-tolerance only if F1 and F2 also hold. For F1 and F2 to hold, a configurator must also (1) detect faults and cause elements to be removed and (2) detect repairs and cause elements to be added. Thus, the degree to which a configurator enhances fault tolerance is directly related to the degree to which (1) and (2) are achieved. Here, the semantics of the application can be helpful. For example, to infer that a client is faulty, a state machine can compare requests made by different clients or by the same client over a period of time. To determine that a processor executing a state machine copy is faulty, the state machine can monitor messages sent by other state machine copies during execution of an Agreement protocol. And, by monitoring aspects of the environment being controlled by actuators, a state machine copy might be able to determine that an output device is faulty. Some elements, such as processors, have internal failure-detection circuitry that can be read to determine whether that element is faulty or has been repaired and restarted. A configurator for such an element can be implemented by having the state machine periodically poll this circuitry.

In order to analyze the fault-tolerance of a system that uses configurators, failure of a configurator can be considered equivalent to the failure of the element that the configurator manages. This is because with respect to the Combining Condition, removal of a non-faulty element from the system or addition of a faulty one is the same as that element failing. Thus, in an f fault-tolerant system, the sum of the number of faulty configurators that manage non-faulty elements and the number of faulty components with non-faulty configurators must be bounded by f.

9.2. Integrating a Repaired Object

Not only must an element being added to a configuration be non-faulty, it also must have the correct state so that its actions will be consistent with those of rest of the system. Define $e[r_i]$ to be the state that a non-faulty system element e should be in after processing requests r_0 through r_i. An element e joining the configuration immediately after request r_{join} must be in state $e[r_{join}]$ before it can participate in the running system.

An element is *self-stabilizing* [Dijkstra 74] if its current state is completely defined by the previous k inputs it has processed, for some fixed k. Obviously, running such an element long enough to ensure that it has processed k inputs is all that is required to put it in state $e[r_{join}]$. Unfortunately, the design of self-stabilizing state machines is not always possible.

When elements are not self-stabilizing, processors are fail-stop, and logical clocks are implemented, cooperation of a single state machine copy sm_i is sufficient to integrate a new element e into the system. This is because state information obtained from sm_i must be correct. In order to integrate e at request r_{join}, sm_i must have access to enough state information so that $e[r_{join}]$ can be assembled and forwarded to e.

- When e is an output device, $e[r_{join}]$ is likely to be only a small amount of device-specific set-up information—information that changes infrequently and can be stored in state variables of sm_i.

- When e is a client, the information needed for $e[r_{join}]$ is frequently based on recent sensor values read and can therefore be determined by using information provided to sm_i by other clients.

- And, when e is a state machine copy, the information needed for $e[r_{join}]$ is stored in the state variables and pending requests at sm_i.

The protocol for integrating a client or output device e is simple—$e[r_{join}]$ is sent to e before the output produced by processing any request with a unique identifier larger than $uid(r_{join})$. The protocol for integrating a state machine copy sm_{new} is a bit more complex. It is not sufficient for sm_i simply to send the values of all its state variables and copies of any pending requests to sm_{new}. This is because some client request might have been received by sm_i after sending $e[r_{join}]$ but delivered to sm_{new} before its repair. Such a request would neither be reflected in the state information forwarded by sm_i to sm_{new} nor received by sm_{new} directly. Thus, sm_i must, for a time, relay to sm_{new} requests received from clients.[8] Since requests from a given client are received by sm_{new} in the order sent and in ascending order by request identifier, once sm_{new} has received a request directly (i.e. not relayed) from a client c, there is no need for requests from c with larger identifiers to be relayed to sm_{new}. If sm_{new} informs sm_i of the identifier on a request received directly from each client c, then sm_i can know when to stop relaying to sm_{new} requests from c.

The complete integration protocol is summarized in the following.

> **Integration with Fail-stop Processors and Logical Clocks.** A state machine copy sm_i can integrate an element e at request r_{join} into a running system as follows.

---

[8]Duplicate copies of some requests might be received by sm_{new}.

If e is a client or output device, sm_i sends the relevant portions of its state variables to e and does so before sending any output produced by requests with unique identifiers larger than the one on r_{join}.

If e is a state machine copy sm_{new}, then sm_i

(1)   sends the values of its state variables and copies of any pending requests to sm_{new},

(2)   sends to sm_{new} every subsequent request r received from each client c such that $uid(r) < uid(r_c)$, where r_c is the first request sm_{new} received directly from c after being restarted.

The existence of synchronized real-time clocks permits this protocol to be simplified because sm_i can determine when to stop relaying messages based on the passage of time. Suppose, as in section 4, there exists a constant Δ such that a request r with unique identifier $uid(r)$ will be received by every (correct) state machine copy in the configuration no later than time $uid(r) + \Delta$ according to the local clock at the receiving processor. Let sm_{new} join the configuration at time τ_{join}. By definition, sm_{new} is guaranteed to receive every request that was made after time τ_{join} on the requesting client's clock. Since unique identifiers are obtained from the real-time clock of the client making the request, sm_{new} is guaranteed to receive every request r such that $uid(r) \geq \tau_{join}$. The first such a request r must be received by sm_i by time $\tau_{join} + \Delta$ according to its clock. Therefore, every request received by sm_i after $\tau_{join} + \Delta$ must also be received directly by sm_{new}. Clearly, sm_i need not relay such requests, and we have the following protocol.

> **Integration with Fail-stop Processors and Real-time Clocks.** A state machine copy sm_i can integrate an element e at request r_{join} into a running system as follows.
>
> If e is a client or output device, then sm_i sends the relevant portions of its state variables to e and does so before sending any output produced by requests with unique identifiers larger than the one on r_{join}.
>
> If e is a state machine copy sm_{new} then sm_i
>
> (1)   sends the values of its state variables and copies of any pending requests to sm_{new},
>
> (2)   sends to sm_{new} every request received during the next interval of size Δ.

When processors can exhibit Byzantine failures, a single state machine copy sm_i is not sufficient for integrating a new element into the system. This is because state information furnished by sm_i is not necessarily correct—sm_i might be executing on a faulty processor. To tolerate f failures in a system with $2f+1$ state machine copies, $f+1$ identical copies of the state information and $f+1$ identical copies of relayed messages must be obtained. Otherwise, the protocol is as described above for real-time clocks.

Stability Revisited

The stability tests of section 4 do not work when requests made by a client can be received from two sources—the client and via a relay. During the interval that messages are being relayed, sm_{new}, the state machine copy being integrated, might receive a request r directly from c but later receive r', another request from c, with $uid(r) > uid(r')$, because r' was relayed by sm_i. The solution to this problem is for sm_{new} to consider requests received directly from c stable only after no relayed requests from

c can arrive. Thus, the stability test must be changed:

> **Stability Test During Restart**. A request r received directly from a client c by a restarting state machine copy sm_{new} is stable only after the last request from c relayed by another processor has been received by sm_{new}.

An obvious way to implement this is for a message to be sent to sm_{new} when no further requests from c will be relayed.

10. Related Work

The state machine approach was first described in [Lamport 78a] for environments in which failures could not occur. It was generalized to handle fail-stop failures in [Schneider 82], a class of failures between fail-stop and Byzantine failures in [Lamport 78b], and full Byzantine failures in [Lamport 84]. The various abstractions proposed for these models are unified in [Schneider 85]. A critique of the approach for use in database systems appears in [Garcia-Molina et al 84]. Experiments evaluating the performance of various of the stability tests in a network of SUN Workstations are reported in [Pittelli & Garcia-Molina 87].

The state machine approach has been used in the design of significant fault-tolerant process control applications [Wensley et al 78]. It has also been used to implement distributed synchronization—including read/write locks and distributed semaphores [Schneider 80], input/output guards for CSP and conditional Ada SELECT statements [Schneider 82]—and, more recently, in the design of a fail-stop processor approximations in terms of processors that can exhibit arbitrary behavior in response to a failure [Schlichting & Schneider 83] [Schneider 84]. The state machine approach is rediscovered with depressing frequency, though rarely in its full generality. For example, the (late) Auragen 4000 series system described in [Borg et al 83] and the Publishing crash recovery mechanism [Powell & Presotto 83], both use variations of the approach. A stable storage implementation described in [Bernstein 85] exploits properties of a synchronous broadcast network to avoid explicit protocols for Agreement and Order and employs Transmitting a Default Vote (as described in section 7). The notion of Δ common storage, suggested in [Cristian et al 85], is a state machine implementation of memory that uses the Real-time Clock Stability Test. The method of implementing highly available distributed services in [Liskov & Ladin 86] uses the state machine approach, with clever optimizations of the stability test and Agreement abstraction that are possible due to the semantics of the application and the use of fail-stop processors.

The ISIS project [Birman & Joseph 87] has recently been investigating fast protocols to support *fault-tolerant process groups*—in the terminology of this paper, state machines in a system of fail-stop processors. Their *ABCAST* protocol is a packaging of our Agreement and Order abstractions based on the Logical Clock Stability Test Tolerating Fail-stop Failures; *CBCAST* allows more flexibility in message ordering and permits designers to specify when requests commute.

Another project at Cornell, the Realtime-Reliability testbed, is investigating semantics-dependent optimizations to state machines. The goal of that project is to systematically develop efficient, fault-tolerant, process control software for a hard real-time environment. Starting with a system structured as state machines and clients, various optimizations are performed to combine state machines, thereby obtaining an fast, yet provably fault-tolerant distributed program.

Acknowledgments

References

[Babaoglu 86]   Babaoglu, O. On the reliability of consensus-based fault-tolerant distributed systems. *ACM TOCS 5*, 4 (Nov. 1987), 394-416.

[Bernstein 85] Bernstein, A.J. A loosely coupled system for reliably storing data. *IEEE Trans. on Software Engineering SE-11*, 5 (May 1985), 446-454.

[Birman 85]   Birman, K.P. Replication and fault tolerance in the ISIS system. *Proc. Tenth ACM Symposium on Operating Systems Principles.* (Orcas Island, Washington, Dec. 1985), ACM, 79-86.

[Birman & Joseph 87]   Birman, K.P. and T. Joseph. Reliable communication in the presence of failures. *ACM TOCS 5*, 1 (Feb. 1987), 47-76.

[Borg et al 83]   Borg, A., J. Baumbach, and S. Glazer. A message system supporting fault tolerance. *Proc. of Ninth ACM Symposium on Operating Systems Principles*, (Bretton Woods, New Hampshire, October 1983), ACM, 90-99.

[Cooper 84]   Cooper, E.C. Replicated procedure call. *Proc. of the Third ACM Symposium on Principles of Distributed Computing*, (Vancouver, Canada, August 1984), ACM, 220-232.

[Cristian et al 85]   Cristian, F., H. Aghili, H.R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine agreement. *Proc. Fifteenth International Conference on Fault-tolerant Computing*, (Ann Arbor, Mich., June 1985), IEEE Computer Society.

[Dijkstra 74]   Dijkstra, E.W. Self Stabilization in Spite of Distributed Control. *CACM 17*, 11 (Nov. 1974), 643-644.

[Fischer et al 85]   Fischer, M., N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM 32*, 2 (April 1985), 374-382.

[Garcia-Molina et al 84]   Garcia-Molina, H., F. Pittelli, and S. Davidson. Application of Byzantine agreement in database systems. TR 316, Department of Computer Science, Princeton University, June 1984.

[Gray 78]   Gray, J. Notes on Data Base Operating Systems. *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, Vol. 60, Springer-Verlag, New York, 1978, 393-481.

[Hammer and Shipman 80]   Hammer, M. and D. Shipman. Reliability mechanisms for SDD-1: A system for distributed databases. *ACM TODS 5*, 4 (December 1980), 431-466.

[Lamport 78a]   Lamport, L. Time, clocks 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 84]   Lamport, L. Using time instead of timeout for fault-tolerance in distributed systems. *ACM TOPLAS 6*, 2 (April 1984), 254-280.

[Lamport et al 82]   Lamport, L., R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS 4*, 3 (July 1982), 382-401.

[Liskov 85]   Liskov, B. The Argus language and system. *Distributed Systems—Methods and Tools for Specification, Lecture Notes in Computer Science*, Vol. 190, Springer-Verlag, New York, N.Y. 1985, 343-430.

[Liskov & Ladin 86]   Liskov, B. and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. *Proc. of the Fifth ACM Symposium on Principles of Distributed Computing*, (Calgry, Alberta, Canada, August 1986), ACM, 29-39.

[Pittelli & Garcia-Molina 87]   Pittelli, F.M. and H. Garcia-Molina. Efficient scheduling in a TMR database system. *Proc. Seventeenth International Symposium on Fault-tolerant Computing*, (Pittsburgh, Pa, July 1987), IEEE.

[Powell & Presotto 83]   Powell, M. and D. Presotto. PUBLISHING: A reliable broadcast communication mechanism. *Proc. of Ninth ACM Symposium on Operating Systems Principles*, (Bretton Woods, New Hampshire, October 1983), ACM, 100-109.

[Schlichting & Schneider 83]   Schlichting, R.D. and F.B. Schneider. Fail-Stop processors: An approach to designing fault-tolerant computing systems. *ACM TOCS 1*, 3 (August 1983), 222-238.

[Schneider 80]   Schneider, F.B. Ensuring Consistency on a Distributed Database System by Use of Distributed Semaphores. *Proc. International Symposium on Distributed Data Bases* (Paris, France, March 1980), INRIA, 183-189.

[Schneider 82]   Schneider, F.B. Synchronization in distributed programs. *ACM TOPLAS 4*, 2 (April 1982), 179-195.

[Schneider 84]   Schneider, F.B. Byzantine generals in action: Implementing fail-stop processors. *ACM TOCS 2*, 2 (May 1984), 145-154.

[Schneider 85]   Schneider, F.B. Paradigms for distributed programs. *Distributed Systems—Methods and Tools for Specification, Lecture Notes in Computer Science*, Vol. 190, Springer-Verlag, New York, N.Y. 1985, 343-430.

[Schneider 86]   Schneider, F.B. A paradigm for reliable clock synchronization. *Proc. Advanced Seminar on Real-Time Local Area Networks* (Bandol, France, April 1986), INRIA, 85-104.

[Schneider et al 84]   Schneider, F.B., D. Gries, and R.D. Schlichting. Fault-Tolerant Broadcasts. *Science of Computer Programming 4* (1984), 1-15.

[Siewiorek & Swarz 82]   Siewiorek, D.P. and R.S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, Bedford, Mass, 1982.

[Skeen 82]   Skeen, D. Crash Recovery in a Distributed Database System. Ph.D. Thesis, University of California at Berkeley, May 1982.

[Spector 85]   Spector, A.Z. Distributed transactions for reliable systems. *Proc. Tenth ACM Symposium on Operating Systems Principles,* (Orcas Island, Washington, Dec. 1985), ACM, 127-146.

[Strong & Dolev 83]   Strong, H.R. and D. Dolev. Byzantine agreement. *Intellectual Leverage for the Information Society,* Digest of Papers, (Compcon 83, IEEE Computer Society, March 1983), IEEE Computer Society, 77-82.

[Wensley et al 78]   Wensley, J., *et al.* SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proc. IEEE 66*, 10 (Oct. 1978), 1240-1255.