Chapter 7:

Replication Management using the State Machine Approach

Fred B. Schneider*

Department of Computer Science
Cornell University
Ithaca, New York 14853 U.S.A.

This chapter reprints my paper "Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial" which orginally appeared in *ACM Computing Surveys 22* (Dec. 1990). The paper has been reformatted, but otherwise remains unchanged.

Most distributed systems employ replicated services in one form or another. By replicating a service, we can support fault-tolerance as well as improving overall throughput by placing server replicas at sites where the service is needed. Protocols for replication management can be divided into two general classes. The first, called "the state machine approach" or "active replication", has no centralized control. This class is the subject of this chapter. The second class of protocols is called the "primary-backup approach", and it is discussed in Chapter 8 (???FS et al-primary-backup).

The state machine approach ties together a number of the fundamental problems that have been discussed in previous chapters. Chapter 2 (??FS-models) should be consulted to put in perspective the distributed systems models of this chapter. Chapter 4 (???Ozalp+Keith) discusses the logical clocks used here to order client requests. And, Chapter 5 (???Toueg+Hadzlacos) discusses semantics for various communications primitives that support the state machine approach.

*This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-8701103, and DARPA/NSF Grant No. CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

1. Introduction

Distributed software is often structured in terms of *clients* and *services*. Each service comprises one or more *servers* and exports *operations* which clients invoke by making *requests*. Although using a single, centralized, server is the simplest way to implement a service, the resulting service can only be as fault-tolerant as the processor executing that server. If this level of fault tolerance is unacceptable, then multiple servers that fail independently must be employed. Usually, replicas of a single server are executed on separate processors of a distributed system, and protocols are employed to coordinate client interactions with these replicas. The physical and electrical isolation of processors in a distributed system ensures that server failures are independent, as required.

The *state machine approach* is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas.[1] The approach also provides a framework for understanding and designing replication management protocols. Many protocols that involve replication of data or software—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.

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; some of these are mentioned in §9. Section 2 describes how a system can be viewed in terms of a state machine, clients, and output devices. Coping with failures is the subject of §3 through §6. An important class of optimizations—based on the use of time—is discussed in §7. Section 8 describes dynamic reconfiguration. The history of the approach and related work is discussed in §9.

2. State Machines

Services, servers, and most programming language structures for supporting modularity define *state machines*. A *state machine* consists of *state variables*, which encode 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 execute 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.

---

[1]The term "state machine" is a poor one but, nevertheless, is the one used in the literature.

In this tutorial, we will describe a state machine simply by listing its state variables and commands. As an example, state machine *memory* of Figure 2.1 implements a time-varying mapping from locations to values. A *read* command permits a client to determine the value currently associated with a location, and a *write* command associates a new value with a location.

For generality, our descriptions of state machines deliberately do not specify how command invocation is implemented. Commands might be implemented

- using a collection of procedures that share data and are invoked by a **call**, as in a monitor,
- using a single process that awaits messages containing requests and performs the actions they specify, as in a server, or
- using a collection of interrupt handlers, in which case a request is made by causing an interrupt, as in an operating system kernel. (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. In it, $x \circ y$ denotes the result of appending y to the end of list x, $head(x)$ denotes the first element of list x, and $tail(x)$ denotes the list obtained by deleting the first element of list x. This state machine would probably be implemented as part of the supervisor-call handler of an operating system kernel.

Requests are processed by a state machine one at a time, in an order that is consistent with potential causality. Therefore, clients of a state machine can make the following assumptions about

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

```
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 ○ 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 order in which requests are processed:

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

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'.

Note that due to communications network delays, O1 and O2 do not imply that a state machine will process requests in the order made or in the order received.

To keep our presentation independent of the interprocess communication mechanism used to transmit requests to state machines, we will program client requests as tuples of the form

$$\langle \mathit{state\_machine.command}, \mathit{arguments} \rangle$$

and postulate that any results from processing a request are returned using messages. For example, a client might 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.

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 a 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.

Not all collections of commands necessarily satisfy this characterization. Consider the following program to solve 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, communicates the value read to state machine *pc*, and delays approximately D seconds:

```
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*.

In practice, having to structure a system in terms of state machines and clients does not constitute a real 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

Before turning to the implementation of fault-tolerant state machines, we must introduce some terminology concerning failures. A component is considered *faulty* once its behavior is no longer consistent with its specification. In this paper, we consider two representative classes of faulty behavior:

> **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 *t fault-tolerant* if it satisfies its specification provided that no more than t 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 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 t fault-tolerant makes explicit the assumptions required for correct operation; MTBF and other statistical measures do not. Moreover, t 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. MTBF and other statistical reliability measures of a t fault-tolerant system can be derived from statistical reliability

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

measures for the components used in constructing that system—in particular, the probability that there will be t or more failures during the operating interval of interest. Thus, t is typically chosen based on statistical measures of component reliability.

4. Fault-tolerant State Machines

A t fault-tolerant version of a state machine can be implemented by replicating that state machine and running a replica on each of the processors in a distributed system. Provided each replica 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. Thus, if we assume that each failure can affect at most one processor, hence one state machine replica, then by combining the output of the state machine replicas of this *ensemble*, we can obtain the output for the t fault-tolerant state machine.

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

The key, then, for implementing an t fault-tolerant state machine is to ensure

Replica Coordination. All replicas receive and process the same sequence of requests.

This can be decomposed into two requirements concerning dissemination of requests to replicas in an ensemble.

Agreement. Every non-faulty state machine replica receives every request.

Order. Every non-faulty state machine replica processes the requests it receives in the same relative order.

Notice that Agreement governs the behavior of a client in interacting with state machine replicas and that Order governs the behavior of a state machine replica with respect to requests from various clients. Thus, while Replica Coordination could be partitioned in other ways, the Agreement-Order partitioning is a natural choice because it corresponds to the existing separation of the client from the state machine replicas.

Implementations of Agreement and Order are discussed in §4.1 and §4.2. These implementations make no assumptions about clients or commands. While this generality is useful, knowledge of commands allows Replica Coordination, hence Agreement and Order, to be weakened, and thus allows cheaper protocols to be employed for managing the replicas in an ensemble. Examples of two common weakenings follow.

First, Agreement can be relaxed for read-only requests when fail-stop processors are being assumed. When processors are fail-stop, a request r whose processing does not modify state variables need only be sent to a single non-faulty state machine replica. This is because the response from this replica is—by definition—guaranteed to be correct and, because r changes no state variables, the state of the replica that processes r will remain identical to the states of replicas that do not.

Second, Order can be relaxed for requests that commute. Two requests r and r' *commute* in a state machine *sm* if the sequence of outputs and final state of *sm* that would result from processing r followed by r' is the same as would result from processing r' followed by r. An example of a state machine where Order can be relaxed 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 unnecessary—different replicas 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 $2MAJ \le Cno$, then reordering requests might change the outcome of the election.

Theories for constructing state machine ensembles that do not satisfy Replica Coordination are proposed in [Aizikowitz 89] and [Mancini & Pappalardo 88]. Both theories are based on proving that an ensemble of state machines implements the same specification as a single replica does. The approach taken in [Aizikowitz 89] uses temporal logic descriptions of state sequences, while the approach in [Mancini & Pappalardo 88] uses an algebra of actions sequences. A detailed description of this work is beyond the scope of this tutorial.

```
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

4.1. Agreement

The Agreement requirement can be satisfied by using any protocol that allows a designated processor, called the *transmitter*, to disseminate a value to some 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 on which they agree.

Protocols to establish IC1 and IC2 have received considerable attention in the literature and are sometimes called *Byzantine Agreement* protocols, *reliable broadcast protocols*, or simply *agreement* protocols. The hard part in designing such protocols is coping with a transmitter that fails part way through an execution. See [Strong & Dolev 83] for 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.

If requests are distributed to all state machine replicas by using a protocol that satisfies IC1 and IC2, then the Agreement requirement is satisfied. Either the client can serve as the transmitter or the client can send its request to a single state machine replica and let that replica serve as the transmitter. When the client does not itself serve as the transmitter, however, the client must ensure that its request is not lost or corrupted by the transmitter before the request is disseminated to the state machine replicas. One way to monitor for such corruption is by having the client be among the processors that receive the request from the transmitter.

4.2. Order and Stability

The Order requirement can be satisfied by assigning unique identifiers to requests and having state machine replicas process requests according to a total ordering relation on these unique identifiers. This is equivalent to requiring the following, where a request is defined to be *stable* at sm_i once no request from a correct client and bearing a lower unique identifier can be subsequently delivered to state machine replica sm_i:

> **Order Implementation**. A replica next processes the stable request with smallest unique identifier.

Further refinement of Order Implementation requires selecting a method for assigning unique identifiers to requests and devising a stability test for that assignment method. Note that any method for assigning unique identifiers is constrained by O1 and O2 of §2, which imply that if request r_i could have caused request r_j to be made then $uid(r_i) < uid(r_j)$ holds, where $uid(r)$ is the unique identifier assigned to a request r.

In the subsections that follow, we give three refinements of the Order Implementation. Two are based on the the use of clocks; a third uses an ordering defined by the replicas of the ensemble.

Using Logical Clocks

A *logical clock* [Lamport 78a] is a mapping $\hat{T}$ from events to the integers. $\hat{T}(e)$, the "time" assigned to an event e by logical clock $\hat{T}$, is an integer such that for any two distinct events e and e', either $\hat{T}(e) < \hat{T}(e')$ or $\hat{T}(e') < \hat{T}(e)$, and if e might be responsible for causing e' then $\hat{T}(e) < \hat{T}(e')$. It is a simple matter to implement logical clocks in a distributed system. Associated with each process p is a counter $\hat{T}_p$. In addition, a *timestamp* is included in each message sent by p. This timestamp is the value of $\hat{T}_p$ when that message is sent. $\hat{T}_p$ is updated according to:

LC1: $\hat{T}_p$ is incremented after each event at p.

LC2: Upon receipt of a message with timestamp τ, process p resets $\hat{T}_p$:

$$\hat{T}_p := \max(\hat{T}_p, \tau) + 1.$$

The value of $\hat{T}(e)$ 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 $\hat{T}_p$ when e occurs.

Figure 4.2 illustrates the use of this scheme for implementing logical clocks in a system of three processors, p, q and r. Events are depicted by dots and an arrow is drawn between events e and e' if e might be responsible for causing event e'. For example, an arrow between events in different processes starts from the event corresponding to the sending of a message and ends at the event corresponding to the receipt of that message. The value of $\hat{T}_p(e)$ for each event e is written above that event.

If $\hat{T}(e)$ is used as the unique identifier associated with a request whose issuance corresponds to event e, the result is a total ordering on the unique identifiers that satisfies O1 and O2. Thus, a logical

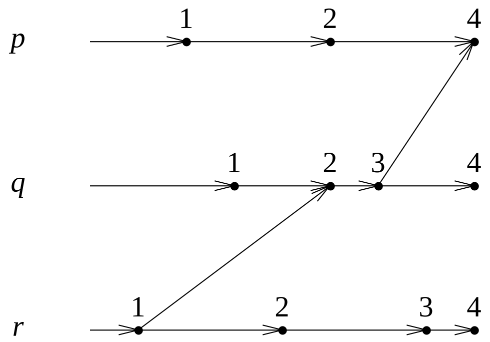

Figure 4.2. Logical Clock Example

clock can be used as the basis of an Order Implementation if we can formulate a way to determine when a request is stable at a state machine replica.

It is pointless to implement a stability test in a system where Byzantine failures are possible and a process or message can be delayed for an arbitrary length of time without being considered faulty. This is because no deterministic protocol can implement agreement under these conditions [Fischer et al 85].[3] Since it is impossible to satisfy the Agreement requirement, there is no point in satisfying the Order requirement. The case where relative speeds of non-faulty processors and messages is bounded is equivalent to assuming that they have synchronized real-time clocks, and will be considered shortly. This leaves the case where fail-stop failures are possible and a process or message can be delayed for an arbitrary length of time without being considered faulty. Thus, we now turn to devising a stability test for that environment.

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.

For fail-stop processors, we can 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.

Under these two assumptions, the following stability test can be used.

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

To see why this stability test works, we show that once a request r is stable at sm_i, no request with smaller unique identifier (timestamp) will be received. First, consider clients that sm_i does not detect as being faulty. Because logical clocks are used to generate unique identifiers, any request made by a client c must have a larger unique identifier than was assigned to any previous request made by c. Therefore, from the FIFO Channels assumption, we conclude that once a request from a non-faulty client c is received by sm_i, no request from c with smaller unique identifier than $uid(r)$ can be

[3] 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.

received by sm_i. This means that once requests with larger unique identifiers than $uid(r)$ have been received from every non-faulty client, it is not possible to receive a request with a smaller unique identifier than $uid(r)$ from these clients. Next, for a client c that sm_i detects as faulty, the Failure Detection Assumption implies that no request from c will be received by sm_i. Thus, once a request r is stable at sm_i, no request with smaller timestamp can be received from a client—faulty or non-faulty.

Synchronized Real-Time Clocks

A second way to produce unique request identifiers satisfying O1 and O2 is by using approximately synchronized real-time clocks.[4] Define $T_p(e)$ to be the value of the real-time clock at processor p when event e occurs. We can 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 §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. Thus, if processor clocks have a resolution of R seconds, then each client can make at most one request every R seconds. O2 follows provided the degree of clock synchronization is better than the minimum message delivery time. In particular, 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.

When unique request identifiers are obtained from synchronized real-time clocks, a stability test can be implemented by exploiting these clocks and the bounds on message delivery delays. Define Δ to be constant 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. Such a Δ must exist if requests are disseminated using a protocol that employs a fixed number of rounds, like the ones cited above for establishing IC1 and IC2.[5] By definition, once the clock on a processor p reaches time τ, p cannot subsequently receive a request r such that $uid(r) < \tau - \Delta$. Therefore, we have the following stability test.

---

[4]A number of protocols to achieve clock synchronization while tolerating Byzantine failures have been proposed [Lamport & Milliar-Smith 84] [Halpern et al 84]. See [Schneider 86] for a survey. The protocols all 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. In practice, these requirements do not constitute a restriction. 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.

[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.

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

This second stability test is never passed if a (faulty) processor refuses to make requests. However, by combining the first and second test, so that a request is considered stable when it satisfies either test, a stability test results that lags clients by Δ only when faulty processors or network delays force it.

Using Replica-Generated Identifiers

In the previous two refinements of the Order Implementation, clients determine the order in which requests are processed—the unique identifier $uid(r)$ for a request r is assigned by the client making that request. In the following refinement of the Order Implementation, the state machine replicas determine this order. Unique identifiers are computed in two phases. In the first phase, which can be part of the agreement protocol used to satisfy the Agreement requirement, state machine replicas propose candidate unique identifiers for a request. Then, in the second phase, one of these candidates is selected and it becomes the unique identifier for that request.

The advantage of this approach to computing unique identifiers is that communications between all processors in the system is not necessary. When logical clocks or synchronized real-time clocks are used in computing unique request identifiers, all processors hosting clients or state machine replicas must communicate. In the case of logical clocks, this communication is needed in order for requests to become stable; in the case of synchronized real-time clocks, this communication is needed in order to keep the clocks synchronized.[6] In the replica-generated identifier approach of this subsection, the only communication required is among processors running the client and state machine

[6]This illustrates an advantage of having a client forward its request to a single state machine replica that then serves as the transmitter for disseminating the request. In effect, that state machine replica becomes the client of the state machine, and so communication need only involve those processors running state machine replicas.

replicas.

By constraining the possible candidates proposed in phase 1 for a request's unique identifier, it is possible to obtain a simple stability test. To describe this stability test, some terminology is first required. We say that a state machine replica sm_i has *seen* a request r once sm_i has received r and proposed a candidate unique identifier for r; and we say that sm_i has *accepted* r once that replica knows the ultimate choice of unique identifier for r. Define $cuid(sm_i, r)$ to be the candidate unique identifier proposed by replica sm_i for request r. Two constraints that lead to a simple stability test are:

UID1: $cuid(sm_i, r) \le uid(r)$

UID2: If a request r' is seen by replica sm_i after r has been accepted by sm_i then $uid(r) < cuid(sm_i, r')$.

If these constraints hold throughout execution, then the following test can be used to determine whether a request is stable at a state machine replica.

> **Replica-Generated Identifiers Stability Test**. A request r that has been accepted by sm_i is stable provided there is no request r' that has (i) been seen by sm_i, (ii) not been accepted by sm_i, and (iii) for which $cuid(sm_i, r') \le uid(r)$ holds.

To prove that this stability test works, we must show that once an accepted request r is deemed stable at sm_i, no request with smaller unique identifier will be subsequently accepted at sm_i. Let r be a request that, according to the Replica-Generated Identifiers Stability Test, is stable at replica sm_i. Due to UID2, for any request r' that has not been seen by sm_i, $uid(r) < cuid(sm_i, r')$ holds. Thus, by transitivity using UID1, $uid(r) < uid(r')$ holds, and we conclude that r' cannot have a smaller unique identifier than r. Now consider the case where request r' has been seen but not accepted by sm_i and—because the stability test for r is satisfied—$uid(r) < cuid(sm_i, r')$ holds. Due to UID1, we conclude that $uid(r) < uid(r')$ holds and, therefore, r' does not have a smaller unique identifier than r. Thus, we have shown that once a request r satisfies the Replica-Generated Identifiers Stability Test at sm_i, any request r' that is accepted by sm_i will satisfy $uid(r) < uid(r')$, as desired.

Unlike clock-generated unique identifiers for requests, replica-generated ones do not necessarily satisfy O1 and O2 of §2. Without further restrictions, it is possible for a client to make a request r, send a message to another client causing request r' to be issued, yet have $uid(r') < uid(r)$. However, O1 and O2 will hold provided that once a client starts disseminating a request to the state machine replicas, the client performs no other communication until every state machine replica has accepted that request. To see why this works, consider a request r being made by some client and suppose some request r' was influenced by r. The delay ensures that r is accepted by every state machine replica sm_j before r' is seen. Thus, from UID2 we conclude $uid(r) < cuid(sm_i, r')$ and, by transitivity with UID1, that $uid(r) < uid(r')$, as required.

To complete this Order Implementation, we have only to devise protocols for computing unique identifiers and candidate unique identifiers such that:

- UID1 and UID2 are satisfied. (4.1)
- $r \neq r' \Rightarrow uid(r) \neq uid(r')$ (4.2)
- Every request that is seen eventually becomes accepted. (4.3)

One simple solution for a system of fail-stop processors is the following.

Replica-generated Unique Identifiers. Each state machine replica sm_i maintains two variables:

$SEEN_i$ is the largest $cuid(sm_i, r)$ assigned to any request r so far seen by sm_i, and

$ACCEPT_i$ is the largest $uid(r)$ assigned to any request r so far accepted by sm_i.

Upon receipt of a request r, each replica sm_i computes

$$cuid(sm_i, r) := \max(\lfloor SEEN_i \rfloor, \lfloor ACCEPT_i \rfloor) + 1 + .i \tag{4.4}$$

(Notice, this means that all candidate unique identifiers are themselves unique.) The replica then disseminates (using an agreement protocol) $cuid(sm_i, r)$ to all other replicas and awaits receipt of a candidate unique identifier for r from every non-faulty replica, participating in the agreement protocol for that value as well. Let NF be the set of replicas from which candidate unique identifiers were received. Finally, the replica computes

$$uid(r) := \max_{sm_j \in NF}(cuid(sm_j, r)) \tag{4.5}$$

and accepts r.

We prove that this protocol satisfies (4.1)−(4.3) as follows. UID1 follows from using assignment (4.5) to compute $uid(r)$, and UID2 follows from assignment (4.4) to compute $cuid(sm_i, r)$. To conclude that (4.2) holds, we argue as follows. Because an agreement protocol is used to disseminate candidate unique identifiers, all replicas receive the same values from the same replicas. Thus, all replicas will execute the same assignment statement (4.5) and all will compute the same value for $uid(r)$. To establish that the these $uid(r)$ values are unique for each request, it suffices to observe that maximums of disjoint subsets of a collection of unique values—the candidate unique identifiers—are also unique. Finally, in order to establish (4.3), that every request that is seen is eventually accepted, we must prove that for each replica sm_j, a replica sm_i eventually learns $cuid(sm_j, r)$ or learns that sm_j has failed. This follows trivially from the use of an agreement protocol to distribute the $cuid(sm_j, r)$ and the definition of a fail-stop processor.

An optimization of our Replica-generated Unique Identifiers protocol is the basis for the ABCAST protocol in the ISIS Toolkit [Birman & Joseph 87] developed at Cornell. In this optimization, candidate unique identifiers are returned to the client instead of being disseminated to the other state machine replicas. The client then executes assignment (4.5) to compute $uid(r)$. Finally, an agreement protocol is used by the client in disseminating $uid(r)$ to the state machine replicas. Some unique replica takes over for the client if the client fails.

It is possible to modify our Replica-generated Unique Identifiers protocol for use in systems where processors can exhibit Byzantine failures, have synchronized real-time clocks, and communications channels have bounded message-delivery delays—the same environment as was assumed for using synchronized real-time clocks to generate unique identifiers. The following changes are required. First, each replica sm_i uses timeouts so that sm_i cannot be forever delayed waiting to receive and participate in the agreement protocol for disseminating a candidate unique identifier from a faulty replica sm_j. Second, if sm_i does determine that sm_j has timed-out, sm_i disseminates "sm_j timeout" to all replicas (by using an agreement protocol). Finally, NF is the set of replicas in the ensemble less any sm_j for which "sm_j timeout" has been received from $t+1$ or more replicas. Notice, Byzantine failures that cause faulty replicas to propose candidate unique identifiers not produced by (4.4) do not cause difficulty. This is because candidate unique identifiers that are too small have no effect on the outcome of (4.5) at non faulty replicas and those that are too large will satisfy UID1 and UID2.

5. Tolerating Faulty Output Devices

It is not possible to implement a t fault-tolerant system by using a single voter to combine the outputs of an ensemble of state machine replicas into one output. This is because a single failure—of the voter—can prevent the system from producing the correct output. Solutions to this problem depend on whether the output of the state machine implemented by the ensemble is to be used within the system or outside the system.

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 replica, 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, $2t+1$-fold replication permits up to t faulty output devices to be tolerated. For example, a flap on an airplane wing might be designed so that when the $2t+1$ actuators that control it do not agree, the flap always moves in the direction of the majority (rather than twisting). If output devices exhibit only fail-stop failures, then only $t+1$-fold replication is necessary to tolerate t failures because any output produced by a fail-stop output device can be assumed correct. For example, video display 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 $t+1$ 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 replicas 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 $t+1$ identical responses, each from a different member of the ensemble, and takes that as the response from the t fault-tolerant state machine. When only fail-stop failures are possible, the client can proceed as soon as it has received a response from any member of the ensemble, since any output produced by a replica must be correct.

When the client is executed on the same processor as one of the state machine replicas, optimization of client-implemented voting is possible.[7] This is because correctness of the processor implies that both the state machine replica and client will be correct. Therefore, the response produced by the state machine replica running locally can be used as that client's response from the t fault-tolerant state machine. And, if the processor is faulty, then we are entitled to view the client as being faulty, so it does not matter what state machine responses the client receives. Summarizing, we have:

> **Dependent-Failures Output Optimization**. If a client and a state machine replica 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 a t fault-tolerant state machine is not sufficient for implementing a t fault-tolerant system. Faults might result in clients making 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. Therefore, in this section, we discuss various methods for insulating the state machine from faults that affect clients.

---

[7]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 t fault-tolerant system in which clients and state machine replicas have independent failures to one in which they share processors.

6.1. Replicating the Client

One way to avoid having faults affect a client is by replicating the client and running each replica on hardware that fails independently. However, this replication also requires changes to state machines that handle requests from that client. This is because after a client has been replicated N-fold, any state machine it interacts with will receive N requests—one from each client replica—when it formerly received a single request. Moreover, corresponding requests from different client replicas will not necessarily be identical. First, they will differ in their unique identifiers. Second, unless the original client is itself a state machine and the methods of §4 are used to coordinate the replicas, corresponding requests from different replicas can also differ in their content. For example, if a client makes requests based on the value of some time-varying sensor, then its replicas will each read their sensors at a slightly different times and, therefore, make different requests.

We first consider modifications to a state machine *sm* for the case where requests from different client replicas are known to differ only in their unique identifiers. For this case, modifications are needed for coping with receiving N requests instead of a single one. These modifications involve changing each command so that instead of processing every request received, requests are buffered until enough[8] have been received; only then is the corresponding command performed (a single time). In effect, a voter is being added to *sm* to control invocation of its commands. Client replication can be made invisible to the designer of a state machine by including such a voter in the support software that receives requests, tests for stability, and orders stable requests by unique identifier.

Modifying the state machine for the case where requests from different client replicas can also differ in their content typically requires exploiting knowledge of the application. As before, the idea is to transform multiple requests into a single one. For example, in a t fault-tolerant system, if $2t+1$ different requests are received, each containing the value of a sensor, then a single request containing the median of those values might be constructed and processed by the state machine. (Given at most t Byzantine faults, the median of $2t+1$ values is a reasonable one to use because it is bounded from above and below by a non-faulty value.) A general method for transforming multiple requests containing sensor values into a single request is discussed in [Marzullo 89]. That method is based on viewing a sensor value as an interval that includes the actual value being measured; a single interval (sensor) is computed from a set of intervals by using a fault-tolerant intersection algorithm.

6.2. Defensive Programming

Sometimes a client cannot be made fault-tolerant by using replication. In some circumstances, due to the unavailability of sensors or processors, it simply might not be possible to replicate the

[8] If Byzantine failures are possible then a t fault-tolerant client requires $2t+1$-fold replication and a command is performed after $t+1$ requests have been received; if failures are restricted to fail-stop, then $t+1$-fold replication will suffice and a command can be performed after a single request has been received.

client. In other circumstances, the application semantics might not afford a reasonable way to transform multiple requests from client replicas into the single request needed by the state machine. In all of these circumstances, careful design of state machines can limit the effects of requests from faulty clients. For example, *memory* (Figure 2.1) permits any client to write to any location. Therefore, a faulty client can overwrite all locations, destroying information. This problem could be prevented by restricting write requests from each client 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 $user \neq client \rightarrow$ **skip**
[] $waiting = \Phi \wedge user = client \rightarrow user := \Phi$
[] $waiting \neq \Phi \wedge user = client \rightarrow$ **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*, the *acquire* command of *mutex* can automatically schedule *release* on behalf of a client.

We use the notation

schedule $\langle REQUEST \rangle$ **for** $+\tau$

to specify scheduling $\langle REQUEST \rangle$ with a unique identifier at least 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 replicas of the ensemble. This is because each state machine replica will independently **schedule** its own (identical) copy of the request.

We can now modify *acquire* so that a *release* operation is automatically scheduled. In the code that follows, *TIME* is assumed to be a function that evaluates to the current time. Note that *mutex* might now process two *release* commands on behalf of a client that has acquired access to the resource: one command from the client itself and one generated by its *acquire* request. However, the new state variable *time_granted* ensures that superfluous *release* commands are ignored.

$$
\begin{array}{ll}
\mathit{acquire}: & \textbf{command} \\
& \textbf{if } \mathit{user} = \Phi \rightarrow \textbf{send } \mathit{OK} \textbf{ to } \mathit{client}; \\
& \qquad\qquad \mathit{user} := \mathit{client}; \\
& \qquad\qquad \mathit{time\_granted} := \mathit{TIME}; \\
& \qquad\qquad \textbf{schedule } \langle \mathit{mutex.timeout}, \mathit{time\_granted} \rangle \textbf{ for } +B \\
& [\,]\ \mathit{user} \neq \Phi \rightarrow \mathit{waiting} := \mathit{waiting} \propto \mathit{client} \\
& \textbf{fi} \\
& \textbf{end } \mathit{acquire} \\
\mathit{timeout}: & \textbf{command}(\mathit{when\_granted} : \textbf{integer}) \\
& \textbf{if } \mathit{when\_granted} \neq \mathit{time\_granted} \rightarrow \textbf{skip} \\
& [\,]\ \mathit{waiting} = \Phi \wedge \mathit{when\_granted} = \mathit{time\_granted} \rightarrow \mathit{user} := \Phi \\
& [\,]\ \mathit{waiting} \neq \Phi \wedge \mathit{when\_granted} = \mathit{time\_granted} \rightarrow \\
& \qquad\qquad\qquad \textbf{send } \mathit{OK} \textbf{ to } \mathit{head}(\mathit{waiting}); \\
& \qquad\qquad\qquad \mathit{user} := \mathit{head}(\mathit{waiting}); \\
& \qquad\qquad\qquad \mathit{time\_granted} := \mathit{TIME}; \\
& \qquad\qquad\qquad \mathit{waiting} := \mathit{tail}(\mathit{waiting}) \\
& \textbf{fi} \\
& \textbf{end } \mathit{timeout}
\end{array}
$$

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 §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 replicas 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 replicas.

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 $Strt+\Gamma$, 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 a vote is cast. For example, the default vote could be "vote for the first client that any client casts a non-default vote for". In that case, the entire election can be conducted as long as one client casts a vote by using actual messages.[9]

8. Reconfiguration

An ensemble of state machine replicas can tolerate more than *t* faults if it is possible to remove state machine replicas running on faulty processors from the ensemble and add replicas 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 replicas 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 $0\le\tau$.

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

A processor failure may 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 keep 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 keep the Combining Condition from being violated because increasing $P(\tau)$ is the only way to ensure that $P(\tau)-F(\tau)>0$ holds. 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.

[9]Observe that if Byzantine failures are possible, then a faulty client can be elected. Such problems are always possible when voters do not have detailed knowledge about the candidates in an election.

F1: If Byzantine failures are possible, then state machine replicas being executed by faulty processors are identified and removed from the ensemble before the Combining Condition is violated by subsequent processor failures.

F2: State machine replicas running on repaired processors are added to the ensemble before the Combining Condition is violated by subsequent processor failures.

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 replicas 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 suggested 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 $\mathcal{C}$, the state machine replicas $\mathcal{S}$, and the output devices $\mathcal{O}$. $\mathcal{S}$ is used by the agreement protocol and therefore must be known to clients and state machine replicas. It can also be used by an output device to determine which **send** operations made by state machine replicas should be ignored. $\mathcal{C}$ and $\mathcal{O}$ are used by state machine replicas to determine from which clients requests should be processed and to which devices output should be sent. Therefore, $\mathcal{C}$ and $\mathcal{O}$ must be available to all state machine replicas.

Two problems must be solved to support changing the system configuration. First, the values of $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{O}$ must be available when required. Second, whenever a client, state machine replica, 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.

8.1. Managing the Configuration

The configuration of a system can be managed using the state machine in that system. Sets $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{O}$ 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 replica, 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 replicas, and output devices all 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 replicas. 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 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 $\mathcal{C}$, $\mathcal{S}$, or $\mathcal{O}$. 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 replica 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 replica is faulty, the state machine can monitor messages sent by other state machine replicas during execution of an agreement protocol. And, by monitoring aspects of the environment being controlled by actuators, a state machine replica 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 a t 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 t.

8.2. 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 replica sm_i is sufficient to integrate a new element e into the system. This is because state information obtained from any state machine replica sm_i must be correct. In order to integrate e at request r_{join}, replica 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 replica, 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 replica sm_{new} is a bit more complex. It is not sufficient for replica 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 be 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.[10] 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 replica 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, 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 replica 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 §4, there exists a constant Δ such that a request r with unique identifier $uid(r)$ will be received by every (correct) state machine replica 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

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

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 replica 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 replica 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 duration Δ.

When processors can exhibit Byzantine failures, a single state machine replica sm_i is not sufficient for integrating a new element into the system. This is because state information furnished by sm_i might not be correct—sm_i might be executing on a faulty processor. To tolerate t failures in a system with $2t+1$ state machine replicas, $t+1$ identical copies of the state information and $t+1$ 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 §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 replica 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 replica 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.

9. 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]. These various state machine implementations were first characterized using the Agreement and Order requirements and a stability test in [Schneider 85].

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 in the design of distributed synchronization—including read/write locks and distributed semaphores [Schneider 80], input/output guards for CSP and conditional Ada SELECT statements [Schneider 82]—and in the design of a fail-stop processor approximation using processors that can exhibit arbitrary behavior in response to a failure [Schlichting & Schneider 83] [Schneider 84]. 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 §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 decentralized commit protocol of [Skeen 82] can be viewed as a straightforward application of the state machine approach, while the 2 phase commit protocol described in [Gray 78] can be obtained from decentralized commit simply by making restrictive assumptions about failures and performing optimizations based on these assumptions. The Paxon Synod commit protocol [Lamport 89] also can be understood in terms of the state machine approach. It is similar to, but cheaper to execute, than the standard 3 phase commit protocol. Finally, 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 protocol that are possible due to the semantics of the application and the use of fail-stop processors.

A critique of the state machine approach for transaction management 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 89]. That study also reports on the performance of request batching, which is possible when requests describe database transactions, and the use of null requests in the Logical Clock Stability Test Tolerating Fail-stop Failures of §4.

Primitives to support the Agreement and Order requirements for Replica Coordination have been included in two operating systems toolkits. The ISIS Toolkit [Birman 85] provides ABCAST and CBCAST for allowing an applications programmer to control the delivery order of messages to the members of a process group (i.e. collection of state machine replicas). ABCAST ensures that all state machine replicas process requests in the same order; CBCAST allows more flexibility in message ordering and ensures that causally related requests are delivered in the correct relative order. ISIS has been used to implement a number of prototype applications. One example is the RNFS (replicated NFS) file system, a network file system that is tolerant to fail-stop failures and runs on top of NFS, that was designed using the state machine approach [Marzullo & Schmuck 88].

The Psync primitive [Peterson et al 89], which has been implemented in the x-kernel [Hutchinson & Peterson 88], is similar to the CBCAST of ISIS. Psync, however, makes available to the programmer the graph of the message "potential causality" relation, while CBCAST does not. Psync is intended to be a low-level protocol that can be used to implement protocols like ABCAST and CBCAST; the ISIS primitives are intended for use by applications programmers and, therefore, hide the "potential causality" relation while at the same time including support for group management and failure reporting.

Acknowledgments

Discussions with O. Babaoglu, K. Birman, and L. Lamport over the past 5 years have helped me to formulate these ideas. Useful comments on drafts of this paper were provided by J. Aizikowitz, O. Babaoglu, A. Bernstein, K. Birman, R. Brown, D. Gries, K. Marzullo, and B. Simons. I am also very grateful to Sal March, managing editor of *ACM Computing Surveys*, for his thorough reading of this paper and many helpful comments.

References

[Aizikowitz 89] Aizikowitz, J. *Designing Distributed Services Using Refinement Mappings*. Ph.D. Dissertation, Computer Science Department, Cornell University, Ithaca, New York, August 1989. Also available as technical report TR 89-1040.

[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.

[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. *ACM TODS 11*, 1 (March 1986), 27-47.

[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.

[Halpern et al 84] Halpern, J., B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. *Proc. of the Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Vancouver, Canada, August 1984), 89-102.

[Hutchinson & Peterson 88] Hutchinson, N. and L. Peterson. Design of the x-kernel. *Proc. of SIGCOMM '88—Symposium on Communication Architectures and Protocols* (Stanford, CA, August 1988), 65-75.

[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 89] Lamport, L. The part-time parliament. Technical report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1989.

[Lamport & Melliar-Smith 84] Lamport, L and P.M. Melliar-Smith. Byzantine clock synchronization. *Proc. of the Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Vancouver, Canada, August 1984), 68-74.

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

[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*, (Calgary, Alberta, Canada, August 1986), ACM, 29-39.

[Mancini & Pappalardo 88] Mancini, L. and G. Pappalardo. Towards a theory of replicated processing. *Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science, Vol. 331, Springer-Verlag, New York, 1988, 175-192.

[Marzullo 89] Marzullo, K. Implementing fault-tolerant sensors. Technical Report TR 89-997, Computer Science Department, Cornell University, Ithaca, New York, May 1989.

[Marzullo & Schmuck 88] Marzullo, K. and F. Schmuck. Supplying high availability with a standard network file system. *Proceedings of the Eighth International Conference on Distributed Computing Systems*, (San Jose, CA, June 1988), IEEE Computer Society, 447-455.

[Peterson et al 89] Peterson, L.L, N.C. Bucholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM TOCS 7*, 3 (August 1988), 217-246.

[Pittelli & Garcia-Molina 89] Pittelli, F.M. and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM TOCS 7*, 1 (Feb. 1989) 25-60.

[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.

[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.