Abstractions for Fault Tolerance in Distributed Systems*

Fred B. Schneider

TR 86-745
April 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This work is supported by NSF Grant DCR-8320274 and Office of Naval Research contract N00014-86-K-0092.

To appear, Proceedings 10th World Computer Congress, (IFIP Congress '86). Dublin, Ireland, (September 1986).

To appear, Proceedings 10th World Computer Congress, (IFIP Congress 86).
Dublin, Ireland, (September 1986).

Abstractions for Fault Tolerance in Distributed Systems*

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

Abstractions useful in fault-tolerant and distributed systems are described. The abstractions are [illegible] as properties of protocols, hence they have a different flavor from abstractions prevalent in sequential and concurrent programming. Among the abstractions discussed are agreement, order, failure detection, and stable storage.

1. Introduction

Distributed computing and fault tolerance are closely linked. Fault tolerance can only be achieved by replication of function using components with independent failure modes. In a distributed system, the physical separation and isolation of processors linked by a communications network ensure that processors have independent failure modes. Thus, achieving fault tolerance in a computing system can lead to solving problems traditionally associated with distributed computing systems. Distributed systems, on the other hand, frequently involve replicated function and data for performance reasons. Protocols to manage replication are clearly integral to such systems. Finally, as the size of a distributed system increases, so does the probability that one or more of its components will fail. A large system must continue to function despite some failures, or it will be unusable most of the time. Thus, distributed systems—at least large ones—must be fault tolerant.

One cannot simply employ "separation of concerns" to decompose a problem into its fault-tolerance aspects and its distributed computing aspects. The two concerns cannot be separated because each requires the other in its implementation. Moreover, support for distribution and fault tolerance pervade the lower levels of a system, so retrofitting the necessary support onto an existing application can result in an unacceptable perf[illegible] penalty. When constructing a system, the problems associated with fault tolerance and distribution must be confronted together and from the outset.

Only by using *abstractions*, which capture the important properties of an object of interest and suppress irrelevant details, can one hope to master the complexity associated with supporting fault tolerance and distribution. Of course to be useful, an abstraction must be [illegible]t-able. For example, it is easy to implement a fault-tolerant system by assuming fault-free processors, but implementing a fault-free processor is impossible, rendering it a useless abstraction for building real systems.

A *virtual circuit* [34] is an example of a good abstraction. It is a communications channel that allows processes to exchange messages according to:

VC1: Messages sent by one process to another are delivered uncorrupted.

VC2: Messages sent by one process to another are delivered in the order sent.

The important properties of the virtual circuit abstraction are given by VC1 and VC2. Irrelevant details concern how VC1 and VC2 are achieved—including the message routing protocol in use and the hardware that [illegible] processors—and how the virtual circuit is used—such as whether asynchronous **send** or Ada-style rendezvous is supported. And, the virtual circuit abstraction is implementable—the networking world contains numerous implementations.

This paper describes abstractions that are useful in implementing fault-tolerant and distributed systems. The same abstractions serve both fault tolerance and distribution, supporting our belief that the two concerns are not separable. The abstractions we present have a somewhat different flavor from abstractions prevalent in sequential and concurrent programming, which encapsulate state information and/or provide operations to manipulate that state (e.g. a stack or a monitor). Our abstractions are best thought of as properties of protocols or control.

Section 2 describes abstractions of processors that can fail and section 3 reviews some fundamentals for coping with failures. Section 4 discusses the state machine approach, a general way to construct fault-tolerant distributed computing services. The state machine approach motivates two abstractions: Agreement and Order. Section 5 discusses a second approach for constructing a fault-tolerant computing service and this motivates two more abstractions: Failure Detection and Stable Storage. Implementing abstractions by exploiting hardware is discussed in section 6. Section 7 discusses related work.

2. The Fine Print

To be able to program a processor, one typically consults a specification that defines its behavior. The specification may be formal, but it is more likely to be a manual written in some natural language. The specification explains the architecture of the processor, the effects of executing each instruction, the way instructions and data are represented, how the console switches affect execution, etc. Invariably, the specification does not explain what behavior is possible or likely by a faulty processor. This omission is problematic when fault tolerance is desired, since it deprives the programmer of information necessary for designing software that can detect and cope with failures.

*This work is supported by NSF Grant DCR-8320274 and Office of Naval Research contract N00014-86-K-0092.

A *processor failure* occurs when the processor no longer satisfies its specification. Behavior in response to a failure can be classified according to the nature of the disruption it causes.

Byzantine Failures. A processor can exhibit arbitrary and malicious behavior, perhaps involving collusion among other faulty processors [16].

Crash Failures. A processor halts in response to a failure [8].

Fail-stop Failures. A processor halts in response to a failure; other processors can detect that the failure has occurred [28].

Note that these categories of failures can be viewed as defining processor abstractions. For example, a Fail-stop Processor is one that is restricted to fail-stop failures.

Byzantine failures are the most disruptive. Crash failures are less disruptive because processors never perform erroneous actions—every message sent and state transformation performed is consistent with the program being executed. However, unless processor execution speeds are known or processor clocks are approximately synchronized, it is not possible to distinguish between a processor that is executing very slowly and one that has halted due to a crash failure. Yet, the ability to make this distinction can be important [9]. A processor that has crashed can take no further action, but a processor that is merely slow can. Other processors can safely perform actions on behalf of a crashed, hence halted, processor, but not on behalf of a slow one, because subsequent actions by the slow processor might not be consistent with actions performed on its behalf by others. Fail-stop failures are the easiest to cope with because processor failures can be detected by other processors—other processors can safely perform actions on behalf of a failed processor. And, in a system where processors have approximately synchronized clocks and message delivery delays are bounded, crash failures can be converted into fail-stop failures by using timeouts.

A system that can tolerate Byzantine failures can tolerate anything. Since an application that makes assumptions about possible behavior of faulty processors runs the risk of failing if these assumptions are not satisfied, it is prudent that life-critical control systems tolerate Byzantine failures. However, while there is anecdotal evidence that Byzantine failues do occur, there is no published data concerning the likelihood of such failures. For most applications, it suffices to tolerate crash failures and, where necessary, convert these into fail-stop failures.

3. Replication

Failures can be partitioned into two classes, depending on whether repair is required following a failure. A component can exhibit a single *hard* failure but multiple *transient* failures between repairs. The occurrence of a hard failure influences the future operation of the device, while the occurrence of a transient failure does not. Byzantine failures are hard because the failure might destroy state information that would render subsequent operation meaningless; crash and fail-stop failures are hard because the device halts and subsequent operation is impossible. Communications lines often exhibit transient failures in response to a noise burst—a message that is in transit might be corrupted, but subsequent transmissions will succeed.

Failures—be they hard or transient—can be detected only by replicating actions in failure-independent ways. One way to do this is by performing the action using components that are physically and electrically isolated; we call this *replication in space*. The validity of the approach follows from an empirically justified belief in the independence of failures at physically and electrically isolated devices. A second approach to replication is for a single device to repeatedly perform the action. We call this *replication in time*. Replication in time is valid only for transient failures.

If the results of performing a set of replicated actions disagree, a failure has occurred. Without making further assumptions, this is the strongest statement that can be made. In particular, if the results agree, we cannot assert that no failure has occurred and the results are correct. This is because if there are enough failures, all of these replicas might be corrupted, yet still agree.

Observe that $t+1$-fold replication permits failure detection but not failure masking when there are as many as t failures. When there is disagreement among $t+1$ independently obtained results, one cannot assume that the majority value is correct. Masking failures requires $2t+1$-fold replication, since then as many as t values can be faulty without causing the majority value to be faulty.

Requirements for fault-tolerance are usually specified in terms of MTBF (mean-time-between-failures), probability of failure over a given interval, and other statistical measures [33]. While it is clear that such characterizations are important to users of a system, there are advantages to describing the fault tolerance of a system in terms of the maximum number of failures that can be tolerated over some interval of interest. We shall say that a system is t-fault tolerant if that system will continue to operate correctly provided t or fewer failures occur.[1] Asserting that a system is t-fault tolerant is a guarantee. This guarantee is independent of 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. Fault tolerance of an actual system will depend on the reliability of the components used in constructing the system—in particular, the probability that there will be t or more failures during the operating interval of interest. In practice, t is chosen based on statistical measures of component reliability. Once t has been chosen, it is a simple matter to derive the usual statistical measures of reliability by computing the probabilities of various configurations of 0 through t failures and their consequences [2].

4. State Machine Approach

One way to employ replication in space for a function is to place a copy of a program that implements that function on each of the processors in a distributed system. Provided the program is deterministic and each copy of the program receives the same requests in the same order, it will do the same thing, hence produce the same results. The results of these copies can then be compared. If $t+1$ or more of the results are the same, this result can be used as the output of a t-fault-tolerant implementation of the function. This technique, known as the *state machine approach*, was first described in [12]. It was generalized to

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

handle fail-stop failures in [27], a class of failures between crash and Byzantine failures in [13], and full Byzantine failures in [14].

The name "state machine" comes from viewing a program as an automaton that repeatedly reads input (requests), performs a computation, and generates results. Not only is this view of programs simple, it is general. For example, process control applications are usually formulated in terms of one or more loops:

```
do true → read from sensors
          compute new state and output values
          write to actuators
od
```

As another example, a memory cell can be viewed as a hardware implementation of a state machine. Two types of requests are serviced: *read* and *write*. In response to a *read*, the current value of the cell is fetched and returned as the output of the state machine; in response to a *write*, the current value of the cell is changed and an acknowledgment message returned.

The state machine approach is based on 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.

Notice that among the (irrelevant) details suppressed by these abstractions are the behavior of faulty processors and the properties of the communications facility. In particular, the Agreement abstraction masks the effects of processor failures on the dissemination of values, yet stipulates nothing about the behavior of faulty processors; the Order abstraction synchronizes message receipt, yet stipulates nothing about message delivery order or system synchronicity. Implementing these abstractions will, of course, require attention to these details.

Abstractions are of value only if they can be implemented, so we now consider implementing Agreement and Order in a distributed system where processors can exhibit Byzantine failures. We assume communications lines that are only subject to transient failures due to noise bursts and that the network has sufficient connectivity so that despite hard failures every processor can communicate with every other.[2] We also assume that speed differences between (non-faulty) processors and message delivery delays have a known bound—in practice, a reasonable assumption—and that clocks on non-faulty processors are approximately synchronized—not a reasonable assumption, but one that can be discharged using any of the Byzantine clock synchronization protocols that have been devised [15][19][20].

Before considering implementations for the Agreement and Order abstractions, we describe an interprocessor communications service that, provided t or fewer failures occur, satisfies properties VC1 and VC2 of a virtual circuit (see section 1), as well as the following authentication property.

Auth: A process can *sign* a message it receives and then forward it to any other process. Any process that receives such a signed message can determine whether the message has been corrupted since it was signed.

[2] There is no way to coordinate processors if they cannot communicate.

Property VC1 can be satisfied in our system by resending each message until $t+1$ identical copies have been received. (To tolerate at most t failures, a maximum of $2t+1$ copies of a message will have to be sent.) This is a form of replication in time and is appropriate because of the assumptions made above about the communications network. The traditional use of check-sums to trigger retransmission of corrupted messages is actually an optimization of this approach that assumes a failure causing corruption of a message always causes inconsistency between a message and its check-sum. The check-sum replaces having agreement on $t+1$ copies of the message; the request for retransmission causes additional copies of the message to be sent only when necessary and only until a copy is received that is (apparently) uncorrupted.

Property VC2 can be achieved by the sender including sequence numbers in messages and having the receiver buffer messages and deliver them in strict order by sequence number.

Property Auth can be approximated by using check-sums. This approximation works because failures typically result in random, rather than truly malicious behavior. Carefully designed redundancy is usually sufficient to cope with random malicious behavior. Digital signatures [25], which are somewhat more costly than check-sums, can be used if intelligent malicious behavior is anticipated.

4.1. Agreement Abstraction

The Agreement abstraction can be realized by devising a 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.

The hard part in implementing such a protocol is being able to cope with a transmitter exhibiting Byzantine failures. Such a transmitter might send different values to different processors in an attempt to violate IC1. Clearly, processors must exchange the values they receive from the transmitter among themselves before agreeing on any value.

To ensure IC1, it is sufficient that when the protocol terminates, the set of values received by each non-faulty processor is the same, because then each processor can compute some function on the contents of this set and obtain a value on which all will agree. However, the details of ensuring this and that IC2 also holds are subtle.

One problem is that faulty processors might selectively fail to relay values. Then, a processor might be delayed forever awaiting a value. To handle this difficulty, the assumptions made above about clock speeds and message delivery delays are exploited. Approximately synchronized clocks and bounded message delivery delays allow a processor to determine when it can expect no further messages from non-faulty processors. Thus, no (faulty) processor can cause another to be delayed indefinitely awaiting a message that will never arrive.

A second problem is that a faulty processor might relay different values to different processors. However, use of signed messages prevents a (faulty) processor p from forging a message. And, if p receives a message m from q, then an attempt by p to change the contents of m before relaying it will not succeed: q will have signed the message and so p's tampering with its contents will invalidate q's signature, which would be detectable by any recipient of the message due to Auth. Thus, using signatures ensures that every value received by a processor is either incorrectly signed and detected as such, or one of the values originally sent by the transmitter.

The remaining problem is to ensure that not only are the values received by each non-faulty processor a subset of the values sent by the transmitter, but that all non-faulty processors agree on the contents of this set. This is solved by having processors sign and relay the messages they originally receive from the transmitter. If at most t processors can be faulty, then $t+1$ rounds of message relay are necessary and sufficient [7]. To see that $t+1$ rounds are sufficient, note that if the transmitter is faulty then t rounds ensure that every message is seen by at least one non-faulty processor, and a non-faulty processor will follow the protocol and therefore forward the message to all other processors; if the transmitter is non faulty, then its value is the only one that will ever be accepted by a non-faulty processor due to the requirements about signatures.

Putting all this together, we get the following protocol for the Agreement abstraction, assuming there are no more than t faulty processors in the system. A process making a request of the state machine is the transmitter; the copies of the state machine are the other processors.

Agreement Protocol:

> The transmitter signs and sends a copy of its value to every processor.
>
> Every other processor p performs $t+1$ rounds, as follows.[3] Whenever p receives a message in round i with i signatures, indicating that the message has been relayed through i processes without being modified, it adds the value in that message to V_p, the set of values it has received, appends its signature to the message, and relays the signed message to all processors that have not already signed it.
>
> At the end of the $t+1^{st}$ round, to select the agreed upon value, each processor computes the same given function on V_p.

At first, the cost of this protocol—$t+1$ rounds of message exchange—might appear prohibitive. In fact, there exist protocols to establish IC1 and IC2 where the number of rounds required is proportional to the number of failures that actually occur during execution of the protocol [35]. When there are few failures, these protocols are comparable in cost to 2- and 3-phase commit protocols [10][17][29]. However, they are considerably more fault tolerant—classic commit protocols are unable to tolerate Byzantine failures. There is an extensive literature concerning implementing the Agreement abstraction, which is variously called the Byzantine Generals Problem and the Consensus Problem. See [8] and [32].

[3]Since the clocks on non-faulty processors are approximately synchronized and message delivery time is bounded, each processor can independently determine when each round starts and finishes.

4.2. Order Abstraction

Our next task is to support the Order abstraction. For this, we employ timestamps. Every request disseminated to the ensemble of state machine copies is given a timestamp using the clock on the processor making the request. Assuming that these clocks have sufficient resolution to ensure that two requests from the same user of the state machine are assigned different timestamps, the timestamps will define a fixed order on requests. (Two requests with the same timestamp from different users are ordered according to the user names, which are assumed to be unique.)

It is not sufficient, however, for state machine copies simply to process requests that have been received in ascending order by timestamp. We must ensure that a copy of the state machine processes a request only if no request with a smaller timestamp can be subsequently received. We shall say that a request is *stable at p* once no request with lower timestamp can be delivered to the state machine copy running on processor p. Clearly, a request should be processed only after it becomes stable.

Testing stability of a request can be accomplished by exploiting the bounds on delivery delays and processor clocks. If requests are disseminated using an agreement protocol like the one above, then there will exist a constant Δ such that a message timestamped T by transmitter p will be received by $T+\Delta$ at every other processor according to each processors local clock. (See [6] for a detailed derivation for Δ in a variety of environments.) Once the clock on a processor p reads time τ, p cannot subsequently receive a request with timestamp less than $\tau - \Delta$. Therefore, a stability test for use with processors that can exhibit arbitrary behavior when they fail is:

Order Protocol:

> Stable requests are processed in ascending order by timestamp. A request is stable at p if the timestamp on the request is T and the clock at p has a value greater than $T+\Delta$.

5. Checkpoints and Restarts

A second general approach for achieving fault tolerance is periodically to save state information as *checkpoints*, and *restart* the computation from its last checkpoint when a failure occurs.[4] This approach requires that two assumptions hold.

(1) Failures are detected before the results of invalid state transformations are saved in a checkpoint.

(2) Checkpoints are unaffected by failures and remain available.

These assumptions permit a computation to be restarted after a failure from one of its prior states that was saved as a checkpoint.

As before, we discharge these assumptions by postulating abstractions. We assume storage is partitioned into *volatile storage* and *stable storage* [17] such that:

> **Failure Detection.** Failures are detected before any visible erroneous state transformation is performed.
>
> **Stable Storage.** Information stored in stable storage remains available, despite failures.

[4]A variety of language constructs have been defined to support this style of programming fault-tolerant applications. See [24] and [26].

Note that implementing the Failure Detection abstraction is equivalent to implementing the Fail-stop Processor abstraction of section 2.

Replication in space can be used to implement the Failure Detection abstraction in a system of processors that can exhibit Byzantine failures and have bounded speed differences and message delivery delays. In order to tolerate t failures, every computation is performed on $t+1$ processors and the results at each step are compared. Whenever a disagreement occurs, a failure has been detected. The details of ensuring that all copies receive the same input in the same order are the same as described above for the state machine approach—we employ our Agreement and Order abstractions. However, for Failure Detection, only $t+1$-fold replication is required, while for a replicated state machine, $2t+1$-fold replication was required. A state machine that is replicated $2t+1$-fold can mask failures, though; our Failure Detection implementation cannot.

The Stable Storage abstraction can also be implemented in such a system by using replication in space. Here, however, data must be replicated $2t+1$-fold since data must remain accessible even if as many as t failures occur. Again, the state machine approach is used. The state machine responds to *read* and *write* requests in the obvious way. The difference between implementing Stable Storage using a state machine and replicating the entire computation $2t+1$-fold is one of cost. If accesses to Stable Storage are infrequent, the processing power allocated to state machine instances running the Stable Storage abstraction can be modest, in contrast to what might be required to run the entire computation replicated $2t+1$ times.

If processors exhibit only crash failures or fail-stop failures, then the implementation of the Failure Detection abstraction remains unchanged, but the Stable Storage abstraction can be implemented with $t+1$-fold replication.

6. Exploiting Hardware

The implementations described above are quite conservative. All assumptions necessary for correctness are explicit, and we generally made no assumptions about behavior of faulty processors. The implementations are also expensive—perhaps too expensive for all but the most critical applications. Other implementations do exist, although we avoided them because invariably they involve stronger assumptions.

Exploiting hardware is a cost-effective way to support our abstractions in practice. For example, Agreement and Order might be implemented by using a triple-redundant bus, Failure Detection (or. Fail-stop Processors) by hardware monitoring of key points in the processor [11], and Stable Storage by using mirrored (dual) disks. Such hardware implementations are based on engineering data about how components fail. While it is usually impossible to guarantee that these implementations will continue to work in the presence of (any) t failures, it is possible to assert that they will work provided certain other assumptions are valid about the number and nature of failures during some interval of interest. Thus, t-fault tolerance is really a special case of what we might call *A-fault tolerance*, which stipulates that an implementation satisfies its specification provided some set of assumptions A remains valid. Engineering data about component failures allows a set of assumptions A to be chosen.

Computer designers frequently design an architecture and then investigate realizations of that architecture with different price/performance ratios. Our abstractions for fault tolerance give the architect of a fault-tolerant application the opportunity to consider implementations with different price/fault-tolerance ratios. When designing in terms of our abstractions—at least, in theory—one need not decide between hardware and software implementations until the entire design is complete. Moreover, the architect can contemplate A-fault tolerant implementations for different assumptions A, thereby permitting additional tradeoffs between price and fault-tolerance. Finally, the abstractions permit a separation of concerns and a portability of design not otherwise possible.

7. Discussion

By no means is there widespread agreement among researchers or practitioners on what are the right abstractions for distributed systems, fault-tolerant systems, or distributed fault-tolerant systems. This paper describes abstractions that we have found useful. Although the particular abstractions were motivated by specific approaches to achieving fault-tolerance, they are quite general. For example, the Agreement and Order abstractions underly most distributed synchronization mechanisms, even when fault tolerance is not of concern [27]. Failure Detection has application beyond the checkpoint/restart approach described in section 5; the literature is replete with algorithms that require such a service.

The most surprising thing about our abstractions is their form. Abstractions common in sequential and concurrent programming relate state information and operations to manipulate that state. Such abstractions can usually be presented to the programmer as language constructs (e.g. the monitor construct or coroutine control-regime) or as a module instance (e.g. a stack or tree). Our abstractions are best presented to the programmer as guarantees—assertions that can be used to simplify a program by eliminating possible executions. The ISIS project [4] is currently investigating this approach to fault-tolerant distributed systems by providing kernel-level support for Agreement, Ordering, and Failure Detection.

Other approaches to designing distributed (fault-tolerant) systems that are being discussed extend the semantics of instructions and/or data to help the programmer cope with distribution and fault-tolerance. Argus [18], CLOUDS [1], an early version of ISIS [3], LOCUS [22], and TABS [31] all provide transactions as an abstraction for structuring fault-tolerant applications. A *transaction* is like an instruction, except that its execution is atomic. Intermediate states of a transaction are never visible, even if a failure occurs during its execution. Note that a transaction that aborts due to a failure can be re-executed, since the state will be unchanged from what existed when the first execution of the transaction was attempted. Thus, a sequence of transactions specifies a computation not unlike the checkpoint/restart approach described above. In addition to sequential composition, transactions can be composed hierarchically, resulting in nested transactions [21]. By making the grain of computation small—a consequence of nesting transactions—we can decrease the interval that must elapse between failures in order to make progress and therefore increase the probability that progress can be made. Notice that implicit in the notion of a transaction is some form of Stable Storage.

The use of remote procedure calls [23] provides the basis for another approach to designing fault-tolerant distributed systems. A remote procedure call is like an ordinary procedure call, except the procedure body can be executed by a different processor from the one executing the caller. This hides distribution from the programmer. When done properly, it can also hide certain types of failures. Implementing remote procedures so that the procedure body is performed exactly once—not at most once, or at least once—is a hard problem [23][30]. Replicated remote procedures [5] provide a way to support replication in space with a remote procedure call interface. Here, executing a single call causes a copy of the procedure body to be executed on a number of processors; the results of these executions are returned to the caller and compared.

8. Conclusions

The idea of structuring systems in terms of abstractions is not new. Identifying abstractions for fault-tolerant and distributed systems is. Only experience in using and implementing an abstraction will permit us to evaluate its utility. In the meantime, it behooves the designer of a system to think in terms of abstractions—be they new ones or existing ones—because only then will we stop reinventing the wheel.

Acknowledgments

Discussions with O. Babaoglu, K. Birman, and L. Lamport over the past 5 years have helped me to formulate these ideas. They, along with F. Cristian, D. Gries, E. Hartikainen, R. Schlichting, and H.C. Torng provided helpful comments on an early draft of this paper.

References

[1] Allchin, J.E. and M.S. McKendry. Synchronization and recovery of actions. *Proc. of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983, 31-44.

[2] Babaoglu, O. and R. Drummond. Time-communication tradeoffs for reliable broadcast protocols. Technical Report TR 85-687, Department of Computer Science, Cornell University, Ithaca, New York, June 1985.

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

[4] Birman, K.P. and T. Joseph. Reliable communication in an unreliable environment. Technical Report TR 85-694, Department of Computer Science, Cornell University, Ithaca, New York, July 1985.

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

[6] 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).

[7] Dolev, D. and H.R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, 12, 4 (1983).

[8] Fischer, M.J. The consensus problem in unreliable distributed systems (A brief survey). *Proc. 1983 International Conference on Foundations of Computations Theory, Lecture Notes in Computer Science*, Vol. 158, Springer-Verlag, New York, 1983, 127-140.

[9] Fischer, M.J., N.A. Lynch, and M.S. Paterson. The impossibility of distributed consensus with one faulty process. *Proc. Second ACM SIGACT-SIGMOD Symposium on Principles of D[illegible] Systems*, March 1983, 1-7.

[10] 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.

[11] Katzman, J.A. A fault-tolerant computing system. *Proc. Eleventh Hawaii International Conference on System Sciences*, 1978.

[12] Lamport, L. Time, clocks and the ordering of events in a distributed system. *CACM 21*, 7 (July 1978), 558-565.

[13] Lamport, L. The implementation of reliable distributed multiprocess systems. *Computer Networks 2* (1978), 95-114.

[14] [illegible], L. Using time instead of timeout for fault-tolerance in distributed systems. *ACM TOPLAS 6*, 2 (April 1984), 254-280.

[15] Lamport, L. and P.M. Melliar-Smith. Byzantine clock syn[illegible]tion. *Proc. of the Third ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, (August 1984), 68-74.

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

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

[18] 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.

[19] Lundelius, J. and N. Lynch. A new fault-tolerant algorithm for clock syn[illegible]tion. *Proc. of the Third ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, August 1984, 75-88.

[20] Mahaney, S. and F.B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. *Proc. of the Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Minaki, Ontario, Canada, August 1985, 237-249.

[21] Moss, J.E. Nested transactions and reliable distributed computing. *Proc. Second Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, Penn., July 1982, 33-39.

[22] Mueller, E.T., J.D. Moore and G.J Popek. A nested transaction mechanism for LOCUS. *Proc. Ninth ACM Symposium on Operating Systems Principles*. Bretton Woods, New Hampshire, Oct. 1983, 71-89.

[23] Nelson, B.J. Remote procedure call. Technical Report CMU-CS-81-119, Dept. of Computer Science, Carnegie-Mellon University, (Ph.D. Thesis), May 1981

[24] Randell, B., P.A. Lee, and P.C. Treleaven. Reliability issues in computing system design, *Computing Surveys 10*, 2 (June 1978), 123-165.

[25] Rivest, R., A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM 21*, 2 (Feb. 1978), 120-126.

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

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

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

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

[30] Spector, A.Z. Performing remote operations efficiently on a local computer network. *CACM 25*, 4 (April 1982), 246-260.

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

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

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

[34] Tanenbaum, A. *Computer Networks*. Prentice Hall, New Jersey, 1981.

[35] Toueg, S., K. Perry, and T.K. Srikanth. Fast distributed agreement. *Proc. of the Fourth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Minaki, Ontario, Canada, August 1985, 71-86.