Replicated Procedure Call

Eric C. Cooper

Computer Systems Research Group
Computer Science Division — EECS
University of California
Berkeley, CA 94720

Abstract

A new mechanism for constructing highly available distributed programs is described. It combines remote procedure call with replication of program modules for fault tolerance.

The set of replicas of a module is called a troupe. In a program constructed from troupes, what appears to the programmer as a single inter-module procedure call results in a replicated procedure call. A distributed program constructed in this way will continue to function as long as at least one member of each troupe survives.

The semantics of replicated procedure calls and troupes are defined and algorithms are presented that support these semantics.

1. Introduction

This paper describes *replicated procedure call*, a new mechanism for constructing highly available distributed programs. Replicated procedure call combines remote procedure call with replication of program modules for fault tolerance.

Remote procedure call allows program modules to be located on different machines. Replicated procedure call generalizes this by allowing modules to be replicated any number of times. The set of replicas of a module is called a *troupe*. When a client troupe makes a replicated procedure call to a server troupe, each member of the server troupe performs the requested procedure exactly once, and each member of the client troupe receives the results, as shown in Figure 1. A distributed program constructed from troupes will continue to function as long as at least one member of each troupe survives. Replicated procedure call has the following properties:

Transparency
: Replicated procedure calls appear to the programmer like normal procedure calls. The details of module replication are invisible.

Flexibility
: The degree of replication of individual modules can be changed to achieve varying levels of reliability without modifying or recompiling the program. Reconfiguration can occur while a program is executing.

Generality
: Many distributed algorithms, particularly those involving replicated servers and broadcast communication, can be formulated in terms of replicated procedure calls.

This work was sponsored by a National Science Foundation Graduate Fellowship and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the U.S. Government.

This paper will appear in the *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, August 1984.

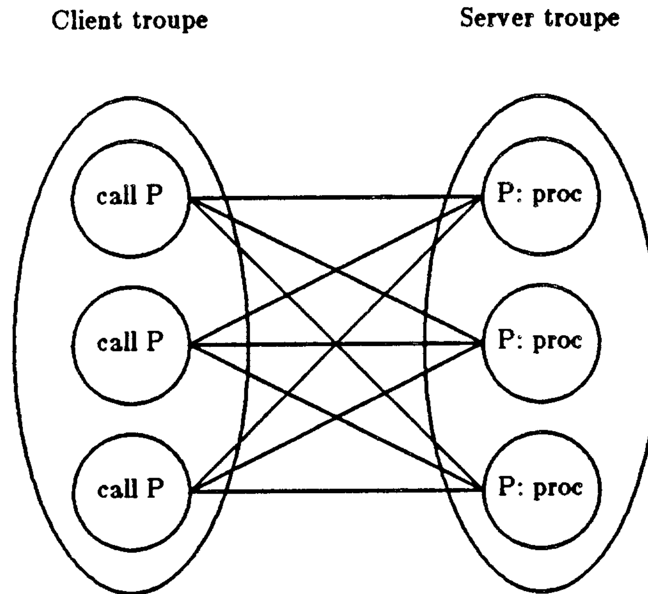

Figure 1: Replicated procedure call

Efficiency
The replicated procedure call algorithms are well suited to broadcast local-area networks.

This paper describes the semantics of replicated procedure call and the algorithms required to implement these semantics.

2. Background and Related Work

Remote procedure call [3, 27] enables programmers to write distributed programs in the same style as conventional programs for centralized computers. Details of communication are hidden, and the syntax of a remote call is identical to that of the local case. The protocol implemented in the course of this research began as an attempt to transfer the Xerox RPC ideas [39] to an environment based on Berkeley UNIX [17] and DARPA Internet protocols [30,31].

The idea of replication as a means of masking the failures of individual components dates back to von Neumann [37]. Triple-modular and N-modular redundancy have long been familiar to designers of fault-tolerant computer systems [1,25]. Early applications of redundancy to software fault tolerance include the SIFT system [38] and the PRIME system [12]. Replication is also the basis of methods proposed by Lamport [19] and Schneider [35] for constructing distributed systems that meet given reliability requirements. These forms of modular redundancy, in which each component performs the same function, are to be distinguished from primary/standby schemes, in which only a single component functions normally and the remaining replicas are on stand-by in case the primary fails [2,5].

Our approach is similar to one taken by Gunningberg, who proposed a fault-tolerant message protocol based on triple-modular redundancy [15]. We have extended this idea to a system based on remote procedure calls rather than messages, and with more general replication and voting schemes than triple-modular redundancy and majority voting.

A methodology known as N-version programming uses multiple implementations of the same module specification to mask software faults [6]. This technique can be used in conjunction with replicated procedure call by constructing troupes from independently implemented modules, thereby increasing software as well as hardware fault tolerance.

3. Processor Failures

We assume that the only type of processor failure is a crash, in which the processor simply halts, losing its volatile state information. Schneider [35] has termed machines with this property *fail-stop* processors.

If processors were not fail-stop, troupe members would have to reach *Byzantine agreement* [20] about the contents of incoming messages, because a malfunctioning processor might send different messages to different troupe members. Byzantine agreement could be added to the algorithms presented here, but would result in a significant loss of performance. There is no evidence that failures other than crashes occur often enough to warrant the increased expense.

4. Troupes

A *troupe* is a set of replicas of the same module. A troupe behaves like a single logical module, but remains available as long as at least one of its members continues to function. Increasing the number of processors spanned by a troupe increases its resilience to crashes. In addition, the technique of N-version programming mentioned above can be used to increase tolerance of software failures. The cost of increasing the reliability of a distributed program by replication can be traded off against the cost of providing crash recovery facilities based on stable storage, such as checkpointing and message logging [2,5,21,32].

5. Module Semantics

In order to describe the semantics of troupes, we must first describe the semantics of the individual modules of which troupes are composed. This section presents a model of the execution of a procedure in a module. Using this model, we can state precisely our assumptions about the degree to which modules behave deterministically.

We adopt the usual definition of a module as consisting of private state information together with a set of procedures that manipulate this state. At any moment, the state information summarizes the net effect of the history of calls to the module. The module state is used explicitly in our model; in a sense, the effects of procedure call execution are described by looking inside the module. It is also possible to formulate an equivalent model, in terms of the call history rather than the module state, which views the behavior of the module from the outside. This duality between the two models of the behavior of a module is reflected in the two standard methods of implementing a module that survives crashes: checkpointing the module's state and logging the module's interactions [14,21].

For simplicity, we assume that module M consists of a single state variable S and a single procedure P. We use the expression M.P(x) to denote the act of calling procedure P with parameter x. An *execution* of M.P(x) in state S consists of the result of the procedure, the new state of the module, and the trace of remote procedures called during the execution. Because of nondeterminism, there may be a set of possible executions of M.P(x) in state S.

We say that M is *completely deterministic* if there is exactly one possible execution of M.P(x) in state S.

Complete determinism is required in roll-forward crash recovery schemes such as replay of messages [5,32] or re-execution of intention lists [14,21]. Previous methods for fault tolerance based on modular redundancy [1,25,15] also assume complete determinism. Although this requirement may be acceptable in certain applications, such as transaction processing, it is too restrictive to be practical in the context of more general distributed programs.

The requirement of complete determinism can be relaxed by incorporating an application-specific definition of what constitutes an acceptable degree of nondeterminism. We associate with module M a *state equivalence relation*, and with each procedure in M a *parameter equivalence relation* and a *result equivalence relation*. The module-specific equivalence relations induce an equivalence relation on procedure executions in the module: two executions are equivalent if

(1) the results are result-equivalent,

(2) the new states are state-equivalent, and

(3) the traces are identical up to parameter-equivalence of the remote procedures called.

We define module M to be *deterministic up to equivalence* if

(1) all executions of M.P(x) in state S are equivalent, and

(2) whenever x_1 is parameter-equivalent to x_2 and S_1 is state-equivalent to S_2, all executions of $M.P(x_1)$ in state S_1 are equivalent to all executions of $M.P(x_2)$ in state S_2.

6. Troupe Semantics

When a program is constructed from troupes, what appears to the programmer as a single inter-module procedure call actually results in a replicated procedure call. One way to achieve this kind of transparency is to use a *stub compiler* [27]. The details of a stub compiler for replicated procedure call are described elsewhere [7]. This section concentrates on the semantics of replicated procedure calls between troupes.

When a client troupe makes a replicated call to a server troupe, each client troupe member makes a *one-to-many call* to the server troupe, and each server troupe member handles a *many-to-one call* from the client troupe. In the absence of crashes, a distributed program constructed from troupes should behave exactly as if each troupe were a single module. Therefore, a replicated procedure call should invoke the same procedure and produce the same side effects exactly once at each server troupe member. The assumption of complete determinism or determinism up to equivalence guarantees that the client troupe members all issue the same call, and that the server troupe members all produce the same results.

When a troupe consists of completely deterministic modules, its members respond identically to incoming procedure calls and make the same outgoing procedure calls.

When a troupe consists of modules that are deterministic up to equivalence, an incoming procedure call may cause different (but equivalent) behavior at each troupe member. The restriction on the trace of outgoing procedure calls guarantees that all troupe members will make the same calls to other troupes in the same order while performing the incoming call.

When troupes are used to implement different levels of abstraction, a single procedure call at the highest level may give rise to a chain of replicated procedure calls from troupe to troupe. But if each troupe is deterministic in either of the two senses above, it follows that the program as a whole will exhibit the correct behavior.

7. Algorithm Structure

Figure 2 shows the relationship between the various layers that make up replicated procedure call. A *paired message protocol* [7] is a distillation of the communication requirements of conventional remote procedure call protocols [3,27,39]. It provides

(1) reliably delivered, variable-length, paired messages (e.g. CALL and RETURN), and

(2) *message sequence numbers* that uniquely identify each pair of messages among all the ones exchanged by a given pair of processes.

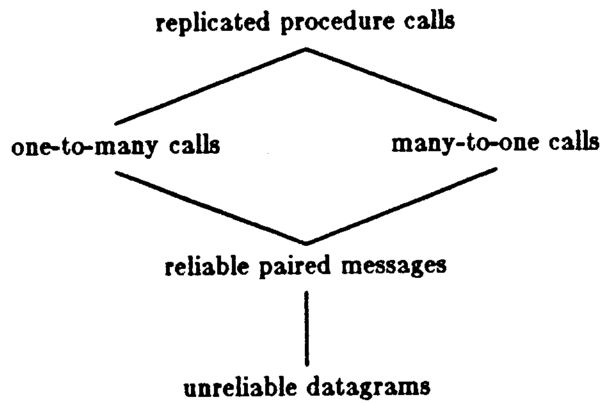

Figure 2: Abstraction structure

The algorithms that follow assume such an underlying paired message layer.

8. One-to-many calls

The client half of the replicated procedure call algorithm is shown in Figure 3. The client sends the same CALL message with the same sequence number to each member of the server

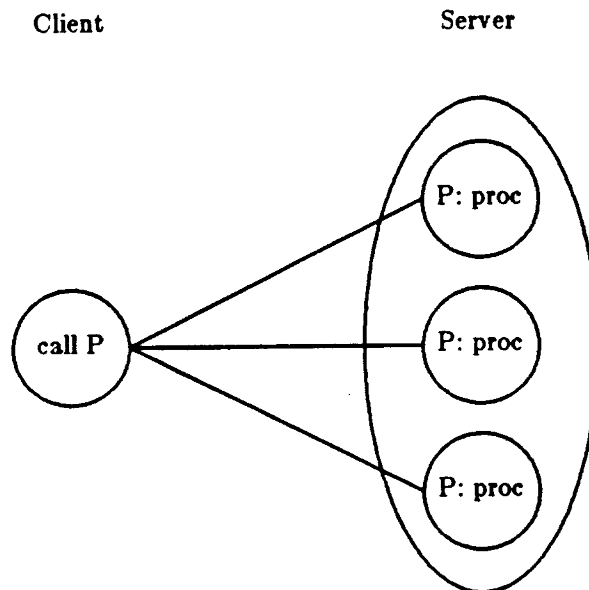

Figure 3: A one-to-many call

troupe. The paired message layer guarantees that the resulting RETURN messages all carry the same sequence number, so it is a simple matter to gather together the set of responses to the call.

In a language with multiple processes, a one-to-many call could be expressed as many concurrent processes, each performing a conventional remote procedure call. Our single-process formulation shows more clearly how the algorithm can be implemented using the multicast operations provided by local-area networks [4, 11].

9. Many-to-one calls

We now consider what occurs at a single server when a client troupe calls a server troupe. The server receives CALL messages from each client troupe member, as shown in Figure 4. The server is supposed to perform the requested procedure exactly once and send the same RETURN message to each member of the client troupe. Two problems must be solved by the many-to-one call algorithm:

(1) When a lone CALL message arrives, how does the server decide if it is a single call or part of a replicated call?

(2) When more than one CALL message arrives, how can the server decide whether they are unrelated calls or part of the same replicated call?

Our solution to the first problem requires a unique ID for each troupe (assigned by the binding agent) and a *client troupe ID* field in each CALL message. When a server receives a CALL message, it maps the client troupe ID into the set of module addresses of the members of the client troupe, by consulting a local cache or contacting the binding agent. In this manner, the server determines from whom to expect CALL messages as part of the many-to-one call.

To avoid interaction with the binding agent in the common special case of an unreplicated client, a distinguished troupe ID is reserved to indicate that the client troupe is a singleton. This

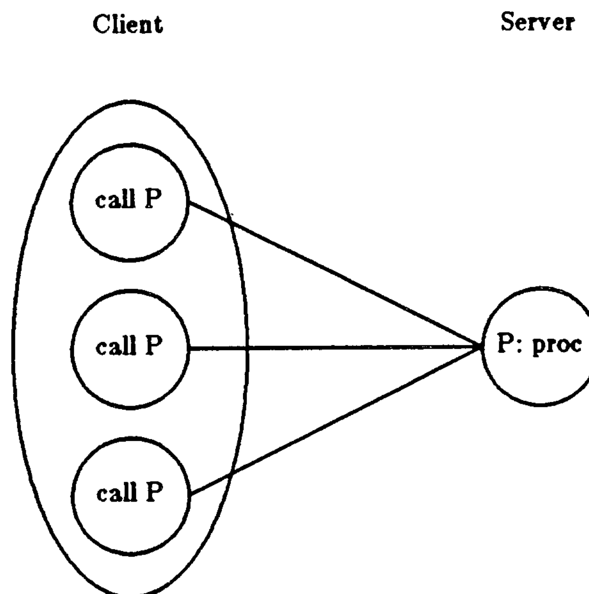

Figure 4: A many-to-one call

insures that the degenerate case of a one-to-one call is equivalent to a conventional remote procedure call not only in function, but also in performance (in terms of the number of messages sent and the number of calls to the binding agent).

We now turn our attention to the second problem: gathering together the set of CALL messages that forms a particular many-to-one call. Gunningberg [15] proposed using message sequence numbers for this purpose, as in the one-to-many case. This requires that the message sequence numbers of troupe members remain completely synchronized. The assumption of deterministic troupe members is sufficient to insure this, but note that this constrains the implementation of the underlying paired message layer to manage message sequence numbers on a per-process basis. This is not the case, for instance, in the Xerox PARC implementation [3], which shares a single sequence number counter among all the processes on a machine by taking advantage of the fact that the sequence numbers of successive remote procedure calls need only be monotonically increasing.

We present a more general solution that does not require the message sequence numbers of troupe members to remain synchronized. This is important when troupe members are permitted a greater degree of nondeterminism, although the complete algorithm for the nondeterministic case is beyond the scope of the present paper.

We first observe that it is a simple matter to extend the familiar notion of a call stack to the case of programs constructed from remote procedure call. The *distributed call stack* consists of the sequence of contexts (possibly spanning machine boundaries) that have been entered but not yet returned from. At the base of the stack is the top-level context that originated the chain of calls. At the top of the stack is the context of the procedure currently being executed.

Next, we extend this idea to programs constructed from troupes by introducing the notion of a *replicated call stack*. An entry for an active procedure in a replicated call stack consists of a set of contexts, one from each member of the troupe implementing that procedure.

Given an active replicated procedure call, we can trace back the replicated call stack to a unique troupe at the base. We call this the *root troupe* of the replicated procedure call. It is the originator of the chain of replicated calls of which the given call is a part.

Finally, we can describe our solution to the problem of determining which CALL messages are part of the same replicated call. We add a *root ID* to each CALL message. The root ID contains two components: the troupe ID of the root troupe of the call, and the sequence number of the root troupe's original replicated call (the one that initiated the current chain of calls). The root ID plays the role of a transaction identifier. Whenever a server makes a replicated call on behalf of a client, it *propagates* the root ID of the call it is currently performing.

The sequence number component of the root ID is well-defined only if all the CALL messages sent by the members of the root troupe as part of the original call bear the same sequence number. This is true if the root troupe is a singleton, which will be the most common case in practice. Otherwise, the members of the root troupe must take explicit action to synchronize their sequence numbers. Note that this constraint applies only to the root troupe.

Root IDs have the following essential property: two or more CALL messages arriving at a server have the same root ID if and only if they are part of the same replicated call. The way in which root IDs are propagated guarantees that messages that are part of the same replicated call have the same root ID. To see the converse, observe that the transfer of control during a procedure call (including a replicated one) implies that unrelated calls to the same server must originate in different threads of control. So unrelated replicated calls will have different root troupes and therefore different root IDs.

10. Collators

In both the one-to-many and many-to-one algorithms, a client or server receives a set of messages from the members of a troupe when a conventional remote procedure call client or

server would only receive a single message. If the troupe members are completely deterministic, all the messages in the set should be identical. If the troupe members are only deterministic up to equivalence, the messages in the set should be equivalent according to the application-specific equivalence relation. We introduce *collators* as a general way of allowing applications to perform these checks.

A collator is a function that reduces a set of messages into a single result. A typical collator would check that all the messages in the set were equivalent, and raise an exception otherwise. Application-dependent collators add considerable flexibility to the replicated procedure call mechanism, but this flexibility is achieved by sacrificing transparency, since the programmer must now be aware of the fact that modules are replicated.

The framework of replicated procedure calls and collators is general enough to encompass majority voting [25,36,37], weighted voting [13,29], and a variety of other distributed algorithms, particularly those based on broadcasting information to multiple recipients [22,28].

11. Concurrent Replicated Calls

The question of the semantics of concurrent replicated calls from different client troupes to the same server troupe is beyond the scope of this paper. This is a different problem than that posed by a many-to-one call; the latter is solved by the algorithms already described. For a server module to operate correctly in the presence of concurrent calls from different clients, even without replication, it must appear to execute those calls in some serial order. Serializability may be achieved by any of a number of concurrency control algorithms [18,24].

When the server is a troupe, not only must concurrent calls from different client troupes be serialized in some way by each server troupe member, but they must be serialized in the *same* way by each server troupe member. Le Lann describes this synchronization requirement as follows:

> In this context [fully replicated computing], the purpose of a synchronization mechanism is to guarantee that the ordering of actions processed by consumers is identical for all consumers [24].

Proper coordination between the replicated procedure call mechanism and a concurrency control mechanism such as nested atomic actions [26,34] is required for correct semantics. The design of such a unified mechanism, which also allows a wider spectrum of determinism constraints on troupe members, will appear in the author's dissertation [8].

12. Binding for Troupes

This section describes a binding agent for troupes, with the following characteristics:

(1) Programs import and export troupes by name.

(2) Replication is invisible to importers and exporters.

(3) A single replicated procedure call suffices to import or export a troupe.

The binding agent and its clients make use of the following types of objects:

Module names
: A module name is what a program uses to import or export a module, and is typically determined by the programming environment. Module names are represented as character strings.

Module addresses
: A module address uniquely identifies an instance of a module in the internet. It is a refinement of the internet process address provided by the underlying paired message protocol [7], since processes may import and export any number of modules.

Troupes
: A troupe is represented by the set of module addresses of its members.

Troupe IDs

A troupe ID corresponds to a unique troupe in the internet. Since troupes may be long-lived, a permanently unique ID is recommended.

The interface to the binding agent is specified by the following procedures:

find troupe by name: module name → troupe

find troupe by ID: troupe ID → troupe

join troupe: module name × module address → troupe ID

A client imports a module by calling **find troupe by name**. The set of module addresses associated with that name is returned.

A server exports a module by calling **join troupe**. If there is already a troupe associated with the specified name, the address of the exported module is added to it; otherwise, a new troupe is created. The troupe ID is returned.

A server that receives a CALL message as part of a many-to-one call uses the **find troupe by ID** procedure to map the client troupe ID into the module addresses of the client troupe.

Since binding is such a pivotal component of the replicated procedure call system, it is essential that the binding agent be highly available. Naturally, this is accomplished by replication, in the form of a troupe of binding agents and replicated calls to the binding procedures.

Before a program can call a binding procedure, it must first import the binding agent (to learn the module addresses of the binding agent troupe). But the binding agent cannot be used to import itself. Eliminating this circularity requires what Lampson [22] calls a *sandwich:* a special case of one abstraction (importing the binding agent), supporting a second abstraction (replicated procedure call), which in turn supports the general case of the first abstraction (importing an arbitrary module). The simplest solution is to assign the module addresses of the binding agents in advance. A better approach is to assign a *broadcast address* [11] to the troupe of binding agents; this allows the actual number of binding agents to vary.

13. Crash Detection and Recovery

A troupe is resilient to the crashes of all but one of the processors on which its members reside. We assume that crashes are detected by the paired message protocol (presumably using a timeout mechanism). At some point it becomes desirable to replace troupe members that have crashed, because a diminished troupe is more vulnerable to future crashes. In this section we describe a general mechanism for adding a new troupe member to an existing troupe. The question of how long to wait before replacing defunct troupe members is beyond the scope of this paper.

Adding a new troupe member to an existing troupe requires that the new member be brought into a state consistent with that of the other members. We propose two solutions to this problem, depending on whether all the processors in the distributed system have the same instruction set architecture and operating system.

In the homogeneous case, a variant of process migration [33] can be used. Process migration enables an existing process to be moved to another machine without otherwise changing its state. Such a mechanism can easily be modified to copy rather than move the process.

If process copying is impossible because of heterogeneous machines, a mechanism similar to checkpointing to stable storage can be used. In this variation, the state information of an existing troupe member is sent in a standard external form to the newly created troupe member. The transmission method for abstract data types proposed by Herlihy [16] can be used for this

purpose.

14. Directions for Future Research

The algorithms described in this paper have been incorporated into *Circus*, a replicated procedure call system in operation at Berkeley [7]. We are just beginning to gain experience with the system.

There are several directions we intend to follow in the course of future research [8]. We are investigating the relationship between the replicated procedure call algorithms described in this paper and various algorithms for general-purpose concurrency control via nested atomic actions [26,34]. At the same time, we are pursuing ways of relaxing the strong determinism requirements on troupe members.

We are designing a *configuration language* and a *configuration manager* to cope with the programming-in-the-large aspects of constructing programs from troupes. A configuration is a correspondence between the logical module structure and the physical troupe structure of a program. The configuration language allows the programmer to specify the set of acceptable configurations of a program. The configuration manager uses this specification to perform the necessary creation and binding of troupe members when the program is started or dynamically reconfigured. We intend to build on existing work in this area [9,10,23,27].

We are investigating how to express arbitrary application-specific collators and integrate them into programming languages. In the process, we hope to identify existing distributed algorithms that can be formulated in terms of troupes, replicated procedure calls, and collators.

Acknowledgments

I would like to thank Robert Fabry, Domenico Ferrari, Earl Cohen, and Doug Terry for their many helpful comments and suggestions.

References

[1] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.

[2] Joel F. Bartlett. A NonStop kernel. *Proceedings of the 8th Symposium on Operating Systems Principles, Operating Systems Review* **15**, 5 (December 1981), pp. 22-29.

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems* **2**, 1 (February 1984), pp. 39-59.

[4] David Reeves Boggs. *Internet Broadcasting.* Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Xerox PARC report number CSL-83-3, October 1983.

[5] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. *Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* **17**, 5 (October 1983), pp. 90-99.

[6] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. *Digest of Papers, FTCS-8: 8th Annual International Conference on Fault-Tolerant Computing,* June 1978, pp. 3-9.

[7] Eric C. Cooper. Circus: A replicated procedure call facility. *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems,* October 1984.

[8] Eric C. Cooper. *Mechanisms for Constructing Reliable Distributed Programs.* Ph.D. dissertation, Computer Science Division, University of California, Berkeley, in preparation.

[9] Daniel H. Craft. Resource management in a decentralized system. *Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* **17**, 5 (October 1983), pp. 11-19.

[10] Frank DeRemer and Hans Kron. Programming-in-the-large versus programming-in-the-small. *Proceedings of the 1975 International Conference on Reliable Software*, April 1975, pp. 114-121.

[11] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation. *The Ethernet: A Local Area Network*. September 1980.

[12] R. S. Fabry. Dynamic verification of operating system decisions. *Communications of the ACM* **16**, 11 (November 1973), pp. 659-668.

[13] David K. Gifford. Weighted voting for replicated data. *Proceedings of the 7th Symposium on Operating Systems Principles, Operating Systems Review* **13**, 5 (December 1979), pp. 150-162.

[14] J. N. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science* **60**, edited by R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, 1978, pp. 393-481.

[15] Per Gunningberg. Voting and redundancy management implemented by protocols in distributed systems. *Digest of Papers, FTCS-13: 13th International Symposium on Fault-Tolerant Computing*, June 1983, pp. 182-185.

[16] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems* **4**, 4 (October 1982), pp. 527-551.

[17] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher. *4.2BSD System Manual*. Computer Systems Research Group, Computer Science Division, University of California, Berkeley, July 1983.

[18] Walter H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *Computing Surveys* **13**, 2 (June 1981), pp. 149-183.

[19] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks* **2**, 2 (May 1978), pp. 95-114.

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* **4**, 3 (July 1982), pp. 382-401.

[21] Butler W. Lampson and Howard E. Sturgis. *Crash Recovery in a Distributed Data Storage System*. Unpublished paper, Computer Science Laboratory, Xerox PARC, draft of June 1979.

[22] Butler W. Lampson. *Replicated Commit*. Unpublished paper, Computer Science Laboratory, Xerox PARC, January 1981.

[23] Butler W. Lampson and Eric E. Schmidt. Practical use of a polymorphic applicative language. *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, January 1983, pp. 237-255.

[24] Gerard Le Lann. Synchronization. In *Distributed Systems — Architecture and Implementation: An Advanced Course, Lecture Notes in Computer Science* **105**, edited by B. W. Lampson, M. Paul, and H. J. Siegert, Springer-Verlag, 1981, pp. 266-283.

[25] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development* **6**, 2 (April 1962), pp. 200-209.

[26] J. Eliot B. Moss. Nested transactions and reliable distributed computing. *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems*, July 1982, pp. 33-39.

[27] Bruce Jay Nelson. *Remote Procedure Call*. Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University, CMU report number CMU-CS-81-119, Xerox PARC report number CSL-81-9, May 1981.

[28] Derek C. Oppen and Yogen K. Dalal. *The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.* Xerox Office Products Division report number OPD-T8103, October 1981.

[29] W. H. Pierce. Adaptive vote-takers improve the use of redundancy. In *Redundancy Techniques for Computing Systems*, edited by Richard H. Wilcox and William C. Mann, Spartan Books, Washington, D.C., 1962, pp. 229-250.

[30] Jon Postel. *User Datagram Protocol.* Information Sciences Institute, University of Southern California, RFC 768, August 1980.

[31] Jon Postel. *Internet Protocol.* Information Sciences Institute, University of Southern California, RFC 791, September 1981.

[32] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. *Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* 17, 5 (October 1983), pp. 100-109.

[33] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. *Proceedings of the 9th ACM Symposium on Operating Systems Principles, Operating Systems Review* 17, 5 (October 1983), pp. 110-119.

[34] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems* 1, 1 (February 1983), pp. 3-23.

[35] Fred B. Schneider. Fail-stop processors. *Digest of Papers, Spring COMPCON 83: 26th IEEE Computer Society International Conference*, February 1983, pp. 66-70.

[36] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* 4, 2 (June 1979), pp. 181-209.

[37] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, edited by C. E. Shannon and J. McCarthy, Princeton University Press, 1956, pp. 43-98.

[38] John H. Wensley. SIFT — Software implemented fault tolerance. *Proceedings of the AFIPS 1972 Fall Joint Computer Conference, Volume 41, Part I*, December 1972, pp. 243-253.

[39] Xerox Corporation. *Courier: The Remote Procedure Call Protocol.* Xerox System Integration Standard 038112, December 1981.