Replicated Distributed Programs

Eric C. Cooper

Computer Science Division—EECS
University of California
Berkeley, California 94720

Abstract

A *troupe* is a set of replicas of a module, executing on machines that have independent failure modes. Troupes are the building blocks of replicated distributed programs and the key to achieving high availability. Individual members of a troupe do not communicate among themselves, and are unaware of one another's existence; this property is what distinguishes troupes from other software architectures for fault tolerance.

Replicated procedure call is introduced to handle the many-to-many pattern of communication between troupes. The semantics of replicated procedure call can be summarized as exactly-once execution at all replicas.

An implementation of troupes and replicated procedure call is described, and its performance is measured. The problem of concurrency control for troupes is examined, and a commit protocol for replicated atomic transactions is presented. Binding and reconfiguration mechanisms for replicated distributed programs are described.

1 Introduction

This paper addresses the problem of constructing highly available distributed programs. (The adjectives *highly available*, *fault-tolerant*, and *nonstop* will be used synonymously to describe a system that continues to operate despite failures of some of its components.) The goal is to construct programs that automatically tolerate crashes of the underlying hardware. The problems posed by incorrect software or by hardware failures other than crashes are only addressed briefly.

The key to tolerating component failures is replication; this approach was proposed by von Neumann thirty years ago [29]. The idea is to replicate each component to such a degree that the probability of all replicas failing becomes acceptably small. The advent of inexpensive distributed computing systems (consisting of computers connected together by a network) makes replication an attractive and practical means of tolerating hardware crashes.

The ability to vary replication on a per-module basis is desirable because it allows software systems to adapt gracefully to changing characteristics of the underlying hardware. Even if perfectly reliable hardware were possible, there would still be periods during which hardware would be unavailable: scheduled down-time for preventive maintenance or reconfiguration, for example. The mechanisms described in this paper permit distributed programs to be reconfigured, while they are executing, so that their services remain available during such periods.

Incorporating replication on a per-module basis is more flexible than previous approaches, such as providing fault tolerance in hardware or writing it into the application software. The first method is too expensive because it uses reliable hardware everywhere, not just for critical modules. The second approach burdens the programmer with the complexity of a non-transparent mechanism.

The fundamental mechanisms presented in this paper are:

- *troupes*, or replicated modules, and
- *replicated procedure call*, a generalization of remote procedure call for many-to-many communication between troupes.

The following important property is what distinguishes troupes and replicated procedure call from previous software architectures for fault tolerance: individual members of a troupe do not communicate among themselves, and are unaware of one another's existence. This property is also what gives these mechanisms their flexibility and power: since each troupe member behaves as if it had no replicas, the degree of replication of a troupe can be varied dynamically, with no recompilation or relinking.

Author's present address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1985 ACM-0-89791-174-1-12/85-0063 $00.75

Previous papers presented the author's initial ideas about replicated procedure calls [10] and a description of the Circus system [11]. This paper presents a portion of the author's Ph.D. dissertation [12].

2 Background and Related Work

The idea of achieving fault tolerance by using replication to mask the failures of individual components dates back to von Neumann [29]. The two architectures for fault-tolerant software are *primary-standby* systems and *modular redundancy*. In a primary-standby scheme, only a single component functions normally; the remaining replicas are on standby in case the primary fails. With modular redundancy, each component performs the same function; there is some form of voting on the outputs to mask failures.

A classic primary-standby architecture is the method of *process pairs* in Tandem's Guardian operating system [1]. The processes in a process pair execute on different processors. One process is designated as the primary, the other as the standby. Before each request is processed, the primary sends information about its internal state to the standby, in the form of a checkpoint. The checkpoint enables the standby to complete the request if the primary fails.

The Auragen architecture combines a primary-standby scheme with automatic logging of messages [6]. If a primary crashes, the log is used to replay the appropriate messages to a standby.

The Isis project at Cornell uses a primary-standby architecture for replicated objects [3]. In each interaction with a replicated object in Isis, one replica plays the role of *coordinator*, and only it performs the operation. The coordinator then uses a two-phase commit protocol to update the other replicas.

The mechanisms used in primary-standby schemes to allow a standby to take over after the primary crashes are isomorphic to crash recovery mechanisms based on stable storage. Under this isomorphism, a standby corresponds to stable storage while the primary continues to function, but assumes the role of the recovering machine when the primary fails.

Triple-modular and N-modular redundancy have long been familiar to designers of fault-tolerant computer systems [22]. Early applications of modular redundancy to software fault tolerance include the SIFT system [30] and the PRIME system [14].

Replication is also the basis of methods proposed by Lamport [21] and Schneider [27] for constructing distributed systems that meet given reliability requirements.

Gifford's weighted voting scheme uses quorums and version numbers to provide replication transparency for files [15]. Herlihy applied Gifford's quorums to replicated abstract data types [19] by taking advantage of the particular semantics of the data types.

Gunningberg's design of a fault-tolerant message protocol based on triple-modular redundancy [17] is similar to, but less general than, the replicated mechanisms presented in this paper.

A methodology known as N-version programming uses multiple implementations of the same module specification to mask software faults [7]. This technique can be used in conjunction with the replicated modules proposed in the present work by using independently implemented modules instead of exact replicas, thereby increasing software as well as hardware fault tolerance. The problems posed by incorrect software are not otherwise addressed in this research.

The protocols implemented in the course of this research began as an attempt to transfer the Courier remote procedure call protocol [32] and the Xerox PARC RPC ideas [5,23] to an environment based on the UNIX* operating system [20] and DARPA Internet protocols [25,26].

Sun Microsystems has proposed a remote procedure call protocol that includes a facility for *broadcast RPC* [28], and Cheriton and Zwaenepoel have studied *one-to-many communication* in the context of the V system [8]. These types of communication are equivalent to a special case of replicated procedure calls: the one-to-many calls discussed in Section 8.

3 A Model of Replicated Distributed Programs

3.1 Modules

A *module* packages together the procedures and state information needed to implement a particular abstraction, and separates the *interface* to that abstraction from its *implementation*. Modules are used to express the static structure of a program when it is written.

This paper discusses troupes and replicated procedure call in the context of modules, but these concepts apply equally well to instances of abstract data types.

*UNIX is a trademark of Bell Laboratories.

3.2 Threads

A thread of control is an abstraction intended to capture the notion of an active agent in a computation. A program begins execution as a single thread of control; additional threads may be created and destroyed either explicitly by means of **fork**, **join**, and **halt** primitives [9], or implicitly during the execution of a **cobegin** ... **coend** statement [13].

Each thread is associated with a unique identifier, called a *thread ID*, that distinguishes it from all other threads.

A particular thread runs in exactly one module at a given time, but any number of threads may be running in the same module concurrently. Threads move among modules by making calls to, and returning from, procedures in different modules. The control flow of a thread obeys a last-in first-out (or stack) discipline.

4 Implementing Distributed Modules and Threads

No mention has been made of machine boundaries as part of the semantics of modules and threads. A distributed implementation of these abstractions must provide *location transparency*. A programmer need not know the eventual configuration of a program when it is being written; the fact that a program is distributed is invisible at the programming-in-the-small level.

A module in a distributed program can be implemented by a *server* whose address space contains the module's procedures and data. A distributed thread can be implemented by using remote procedure calls to transfer control from server to server, and viewing such a sequence of remote procedure calls as a single thread of control.

5 Adding Replication

The distributed modules and threads of Section 4 provide location transparency in the absence of failures. As long as the underlying hardware works correctly, the programmer need not be aware of machine boundaries.

Processor and network failures, however, give rise to new classes of *partial failures* of the distributed program as a whole. Partial failures violate transparency, since they can never occur in a single-machine program. These failures must therefore be masked if transparency is to be preserved.

The key to masking failures is replication, but it introduces another transparency requirement: *replication transparency*.

5.1 Troupes

The approach taken in this research is to introduce replication into distributed programs at the module level. A replicated module is called a *troupe*, and the replicas are called *troupe members*.

Troupe members are assumed to execute on fail-stop processors [27]. If the processors were not fail-stop, troupe members would have to reach byzantine agreement 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 in this paper, but would result in a significant loss of performance. There is no evidence that failures other than crashes occur often enough to warrant this increased expense.

A *deterministic troupe* is a set of replicas of a deterministic module. Section 5.2 shows that the assumption that all troupes are deterministic is sufficient to guarantee replication transparency.

In contrast to the work on replicated abstract data types by Herlihy [19], troupes are a simple approach to achieving high availability: no knowledge of the semantics of a module is required, other than the fact that it is deterministic.

Interactions between troupes occur by means of replicated procedure calls in which all troupe members play identical roles. Furthermore, troupe members do not know of one another's existence; there is no communication among the members of a troupe. It follows that each troupe member behaves exactly as if it had no replicas. In this sense, troupes contrast sharply with the replicated objects in Isis [3], although the goal of high availability is the same.

In replicated distributed programs, crash recovery mechanisms are required only for total failures, in which every troupe member crashes. The probability of total failures can be made arbitrarily small by choosing an appropriate degree of replication. Replication can therefore be used as an alternative to crash recovery mechanisms such as stable storage.

5.2 Replication Transparency and Troupe Consistency

A troupe is *consistent* if all its members are in the same state. If a troupe is consistent, then its clients need not know that it is replicated. Troupe consistency is therefore a sufficient condition for replication transparency.

Troupe consistency is a strong requirement, but it cannot be weakened without knowledge of the semantics of the objects being replicated. *In the absence of application-specific knowledge, troupe consistency is both necessary and sufficient for replication transparency.* This is one area in which troupes differ from other replication schemes. Gifford's weighted voting for replicated files, for example, uses quorums and version numbers to mask the fact that not all replicas are up to date [15], and Herlihy has extended Gifford's approach to abstract data types [19]. Troupe consistency is not necessary in these schemes, because they take advantage of the semantics of the objects being replicated.

In a program constructed from troupes, an inter-module procedure call results in a replicated procedure call from a client troupe to a server troupe. One of the distinguishing characteristics of troupes is that their members do not communicate among themselves, and do not even know of one another's existence. Consequently, when a client troupe makes a replicated call to a server troupe, each server troupe member must perform the procedure, just as if the server had no replicas.

The execution of a procedure can be viewed as a tree of procedure invocations. When a deterministic server troupe is called upon to execute a procedure, the invocation trees rooted at each troupe member are identical: the members of the server troupe make the same procedure calls and returns, with the same arguments and results, in the same order. It follows that if there is only a single thread of control in a globally deterministic replicated distributed program, and if all troupes are initially consistent, then all troupes remain consistent.

Additional mechanisms are required if there is more than one thread of control, because concurrent calls to the same server troupe may leave the members of the server troupe in inconsistent states. The problem of maintaining troupe consistency in the presence of concurrently executing threads is addressed in Section 11.

6 Replicated Procedure Calls

The goal of remote procedure call [23] is to allow distributed programs to be written in the same style as conventional programs for centralized computers. When modules are replaced by troupes, the natural generalization of remote procedure call is *replicated procedure call.* The troupe consistency requirement identified in Section 5.2 determines the semantics of replicated procedure call: 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 all the results. These semantics can be summarized as *exactly-once execution at all troupe members.* Figure 1 shows a replicated procedure call from a client troupe to a server troupe. A replicated distributed program constructed in this way will continue to function as long as at least one member of each troupe survives.

To guarantee replication transparency, troupe members are required to behave deterministically: two replicas in the same state must execute the same procedure in the same way. In particular, they must call the same remote procedures in the same order, produce the same side effects, and return the same results.

7 The Circus Paired Message Protocol

A *paired message protocol* is a distillation of the communication requirements of conventional remote procedure call protocols [5,23,32]. It provides

- reliably delivered, variable-length, paired messages (e.g. call and return), and
- *call sequence numbers* that uniquely identify each pair of messages among all those exchanged by a given pair of processes.

The paired message protocol is responsible for segmenting messages that are larger than a single datagram (in

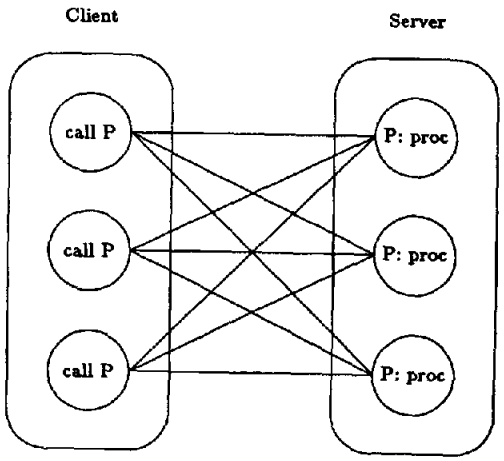

Figure 1: Replicated procedure call

order to permit variable-length messages), and for retransmission and acknowledgment of message segments to ensure reliable delivery. The Circus paired message protocol is based on the RPC protocol of Birrell and Nelson [5]. Circus uses UDP, the DARPA User Datagram Protocol [25]. The Circus protocol is connectionless and geared towards the fast exchange of short messages.

The difference between Birrell and Nelson's RPC protocol and the Circus protocol lies in the treatment of multiple-segment **call** and **return** messages. The Xerox PARC protocol requires an explicit acknowledgment of every segment but the last. This doubles the number of segments sent, but since there is never more than one unacknowledged segment in transit, only one segment's worth of buffer space is required per connection.

The Circus protocol allows multiple segments to be sent before one is acknowledged, which reduces the number of segments sent to the minimum, but requires an unbounded amount of buffering. An alternate implementation of the Circus protocol could easily bound the amount of buffer space required for a connection by dropping all segments outside a fixed allocation window, and simply requiring the sender to retransmit them. These retransmissions could be reduced by informing the sender of the size of the allocation window; this is precisely what is done in the flow-control mechanisms of reliable stream protocols such as TCP [26], but since single-segment messages are expected to occur most often in remote procedure calls, these optimizations are probably not worthwhile.

The paired message abstraction can be provided on top of reliable stream protocols like TCP [26], but implementations of these protocols are typically tuned for bulk data transfers. The Berkeley 4.2BSD implementation of TCP, for example, does not even begin to transfer data until the connection has been established by a three-way handshake, although this restriction is not inherent in the protocol specification. Since **call** and **return** messages are usually short, a specially designed, datagram-based paired message protocol like Circus can complete a message exchange using the same number of packets that a stream protocol requires merely to establish a connection. Nelson makes this same point, with performance measurements to support his claim, in his dissertation [23].

The Circus protocol is currently implemented in user code under Berkeley 4.2BSD [20]. Asynchronous events, specifically the arrival of datagrams and the expiration of timers, must be handled in parallel with the activity of the client or server. For instance, a probe may arrive while a server is performing a procedure. If multiple processes sharing the same address space were available under Berkeley 4.2BSD, a separate process could be devoted to listening for incoming segments and handling timers. Since this is not possible, these events are modeled as software interrupts using the signal mechanism, the interrupt-driven I/O facility, and the interval timer [20]. Protection of critical regions is achieved by using system calls that mask and enable interrupts.

A project is under way at Berkeley to produce an implementation of a remote procedure call protocol for the Berkeley UNIX kernel [31]. The initial specification was an unreplicated version of the Circus protocol, but the desire to limit the required amount of kernel buffer space led to a protocol similar to Birrell and Nelson's.

The unifying communication abstraction provided by the Berkeley 4.2BSD kernel is the *socket* [20], an endpoint for process-to-process communication. Each socket has a protocol type that is used to dispatch generic operations like `read` and `write` to the appropriate protocol implementation. The interface to the kernel RPC protocol is by means of a new protocol type (RPC) with two subtypes: client and server. The implementation enforces write-read alternation for client sockets and read-write alternation for server sockets.

8 Implementing Replicated Procedure Calls

Replicated procedure calls are implemented on top of the paired message layer. There are two subalgorithms involved in a *many-to-many* call from a client troupe to a server troupe: each client troupe member performs a *one-to-many* call to the entire server troupe, and each server troupe member handles a *many-to-one* call from the entire client troupe.

The algorithms for these two cases are described in the following sections. In Circus, these algorithms are implemented as part of the run-time system that is linked with each user's program. The run-time system is called by stub procedures that are produced automatically from a module interface; the replicated procedure call algorithms themselves are thus hidden from the programmer. When the algorithms below refer to various client and server actions, the reader should bear in mind that those actions are performed by the protocol routines in the corresponding run-time systems, rather than by the portions of the program written by the user.

8.1 One-To-Many Calls

The client half of the replicated procedure call algorithm performs a *one-to-many call* as shown in Figure 2. The purpose of the one-to-many call algorithm is to guarantee that the procedure is executed at each server troupe member.

The same **call** message is sent to each server troupe member, with the same call number at the paired message

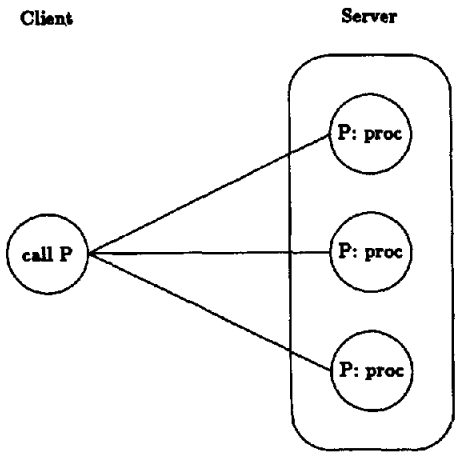

Figure 2: A one-to-many call

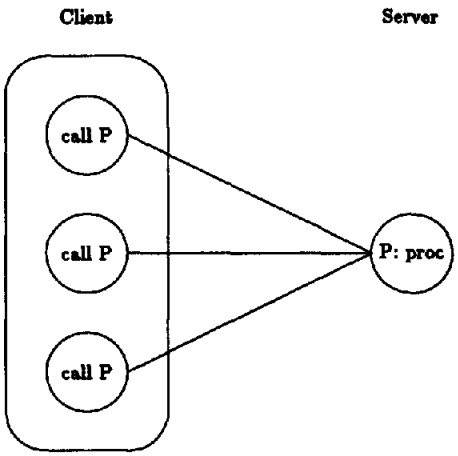

Figure 3: A many-to-one call

level. The client then awaits the arrival of the **return** messages from the members of the server troupe.

In the Circus replicated procedure call implementation, the client will normally wait for all the **return** messages from the server troupe before proceeding. The client receives notification if any server troupe member crashes, so it can proceed with the **return** messages from those that are still available. The return from a replicated procedure call is thus a *synchronization point*, after which each client troupe member knows that all server troupe members have performed the procedure, and each server troupe member knows that all client troupe members have received the result. Alternatives to this strategy are discussed in Section 8.4 below.

8.2 Many-To-One Calls

Now consider what occurs at a single server when a client troupe makes a replicated call to it. The server will receive **call** messages from each client troupe member, as shown in Figure 3; this is called a *many-to-one* call. The semantics of replicated procedure call require the server to execute the procedure only once and return the results to all the client troupe members. The many-to-one call algorithm must therefore solve the following two problems:

1. The server must be able to distinguish unrelated **call** messages from ones that are part of the same replicated call.

2. When one **call** message of a replicated call arrives, the server must be able to determine how many other **call** messages to expect as part of the same replicated call.

A complete description of the algorithm may be found in the author's dissertation [12].

In Circus, the server waits for **call** messages from all available client troupe members before proceeding. Alternatives to this strategy are discussed in Section 8.4 below.

8.3 Many-To-Many Calls

In general, a replicated procedure call is a *many-to-many* call from a client troupe to a server troupe, as shown in Figure 1. A many-to-many call involves the following steps:

1. Each client troupe member sends a **call** message to each server troupe member.

2. Each server troupe member receives a **call** message from each client troupe member.

3. Each server troupe member performs the requested procedure.

4. Each server troupe member sends a **return** message to each client troupe member.

5. Each client troupe member receives a **return** message from each server troupe member.

The key to the many-to-many case is the observation that steps 1 and 5 are the same steps that an unreplicated client performs when making a one-to-many call to a server troupe, and steps 2, 3, and 4 are the same steps that an unreplicated server performs when handling a many-to-one call from a client troupe. The general case therefore factors into the two special cases already described; no additional algorithms are required for the general case. Each client troupe member executes the one-to-many algorithm (as if it were an unreplicated client calling the server troupe), and each server troupe member executes the many-to-one algorithm (as if it were an unreplicated server handling an incoming call from the client troupe).

Observe also that there is never any communication between members of the same troupe in the five steps listed above; communication occurs only between members of different troupes. This means that nowhere in a troupe member is there any information about other members of its own troupe, or whether it is replicated at all. Neither the protocol routines in the run-time system nor the stub procedures produced by the stub compiler use such information.

Finally, notice that messages are sent only in steps 1 and 4, and in both these steps, the message is sent to an entire troupe. Thus, **call** messages are sent to the entire server troupe, and **return** messages are sent to the entire client troupe. These steps obviously correspond to multicast operations.

A multicast implementation would make a dramatic difference in the efficiency of the replicated procedure call protocol. Suppose that there are m client troupe members and n server troupe members. Point-to-point communication requires a total of mn messages to be sent. In contrast, a multicast implementation requires only $m + n$ messages to be sent. The Berkeley 4.2BSD networking primitives used by Circus do not currently allow access to the multicast capabilities of the Ethernet.

8.4 Waiting for Messages to Arrive

A client making a one-to-many call requires a single result, but it receives a **return** message from each server troupe member. Similarly, a server handling a many-to-one call must perform the requested procedure once, but it receives a **call** message from each client troupe member.

Since troupes are assumed to be deterministic, all the messages in these sets will be identical. When should computation proceed: as soon as the first message arrives, or only after the entire set has arrived?

Waiting for all messages to arrive and checking whether they are identical is analogous to providing *error detection* as well as transparent *error correction*. Any inconsistency among the messages is detected, but the execution time of the replicated program as a whole is determined by the slowest member of each troupe. This *unanimous* approach is used by default in the Circus system.

If one is willing to forfeit such error detection, then a *first-come* approach can be used, in which computation proceeds as soon as the first message in each set arrives. In this case, the execution time of the program as a whole is determined by the fastest member of each troupe.

The first-come approach requires only a simple change to the one-to-many call protocol. The client can use the call sequence number provided by the paired message protocol to discard **return** messages from slow server troupe members.

The many-to-one call protocol becomes more complicated; in this respect, the first-come approach destroys the symmetry between the client and server halves of the protocol. The server must be allowed to start performing a procedure as soon as the first **call** message from a client troupe member arrives. When a **call** message for the same procedure arrives from another member of that client troupe, the server cannot execute the procedure again, because that would violate the exactly-once execution property. The server must therefore retain the **return** message until the corresponding **call** messages from all other members of the client troupe have arrived. Whenever such a **call** message arrives, the **return** message is retransmitted. Execution of the procedure thus appears instantaneous to the slow client troupe members, since the **return** message is ready and waiting.

Note that once a client troupe member has received the results of its call, it is free to go ahead and make more calls. Therefore, as the slower members of the client troupe fall further and further behind the faster ones, the server must buffer more and more **return** messages. When a **call** message arrives from one of the slower client troupe members, the server must be able to find its earlier response from among the buffered **return** messages, in order to retransmit it. The call sequence number associated with each message by the paired message protocol suffices for this purpose, because of the assumption that troupes are deterministic.

A better first-come scheme can be implemented by buffering messages at the client rather than the server. In this case, the server broadcasts **return** messages to the entire client troupe in response to the first **call** message. A client troupe member may receive a **return** message for a **call** message that has not yet been sent; this **return**

message must be retained until the client troupe member is ready to send the corresponding call message.

This approach is preferable to buffering messages at the server, for the following reasons:

1. the burden of buffering **return** messages and pairing them with the corresponding late **call** messages is placed on the client, rather than on a shared and potentially heavily-loaded server;

2. the server can use broadcast rather than point-to-point communication; and

3. no communication is required by a slow client once it is ready to send a **call** message, since the corresponding **return** message has already arrived.

Majority voting schemes require similar buffering of **return** messages. Simulations and queueing models have been used to analyze the buffering requirements in this context as a function of the variation in execution rate [33].

Error detection is desirable in practice, since programmers may not be sure that their programs are deterministic. To provide error detection and still allow computation to proceed early, a *watchdog* scheme can be used. This technique requires that the computation be structured as one or more transactions. Computation proceeds with the first message, but another thread of control (the watchdog) waits for the remaining messages and compares them with the first. If an inconsistency is detected by the watchdog, the main computation is aborted. Note that this scheme also requires buffering (in the form of transaction workspaces) to compensate for the skew in execution rates of different troupe members.

Many other schemes are possible in addition to the approaches described here. Discovering and evaluating such variations is an important area for future research.

8.5 Crashes and Partitions

Whenever a troupe member is waiting for one or more messages in the one-to-many and many-to-one call algorithms, the underlying message protocol uses probing and timeouts to detect crashes. This mechanism relies on network connectivity, and therefore cannot distinguish between crashes and network partitions.

Network partitions raise the possibility of different troupe members continuing to execute, each believing that the others have crashed. To prevent troupe members in different partitions from diverging, one can require that each troupe member receive a majority of the expected set of messages before computation is allowed to proceed there.

8.6 Collators

One way to relax the determinism requirement (at the cost of transparency) is to allow programmers to specify their own procedures for reducing a set of messages to a single message. Such procedures are called *collators*.

A collator is a function that maps a set of messages into a single result. To improve performance, it is desirable for computation to proceed as soon as enough messages have arrived for the collator to make a decision. (This is equivalent to using *lazy evaluation* when applying the collator.)

Three collators are supported at the replicated procedure call protocol level (viewing the contents of **call** and **return** messages as uninterpreted bits): *unanimous*, which requires all the messages to be identical and raises an exception otherwise; *majority*, which performs majority voting on the messages; and *first-come*, which accepts the first message that arrives. The framework of replicated calls and collators is sufficiently general to express weighted voting [15] and other replicated or broadcast-based algorithms [24].

9 Performance Analysis

Experiments were conducted to measure the cost of replicated procedure calls as a function of the degree of replication. The cost of a simple exchange of datagrams was also measured in order to establish a lower bound.

The experiments were run on lightly loaded computer center machines during an inter-semester break at Berkeley. The distributed system consisted of six identically configured VAX*-11/750 systems, connected by a single 10 megabit per second Ethernet cable.

Any implementation of a paired message protocol on top of an unreliable datagram layer must perform at least the following steps during the course of a message exchange:

1. Send a datagram.

2. Receive a datagram, specifying a timeout to detect lost datagrams.

The time required to perform these operations therefore represents a lower bound for any implementation of a remote procedure call protocol using unreliable datagrams.

A reliable byte-stream protocol, such as TCP, is generally considered to be inferior to datagrams for the purposes of a remote procedure call implementation. A TCP-based client and server are included for the purpose of comparison. Unlike the UDP client, the TCP client does not need any timeouts, because TCP provides reliable delivery.

*VAX is a trademark of Digital Equipment Corporation.

degree of replication	real time (msecs/rpc)	total cpu time (msecs/rpc)	user cpu time (msecs/rpc)	kernel cpu time (msecs/rpc)
(UDP)	26.5	13.3	0.8	12.4
(TCP)	23.2	8.3	0.5	7.8
1	48.0	24.1	5.9	18.2
2	58.0	45.2	10.0	35.2
3	69.4	66.8	13.0	53.8
4	90.2	87.2	16.8	70.4
5	109.5	107.2	21.0	86.1

Table 1: Performance of UDP, TCP, and Circus

The first set of experiments measured the time per procedure call in Circus as a function of the degree of replication. For comparison, the time for an exchange of UDP datagrams and the time for an exchange of messages over a TCP byte-stream were also measured. The time of day and the total user-mode and kernel-mode CPU time used by the client process were recorded before and after each replicated procedure call. The entries in Table 1 were calculated by averaging the differences between the before and after values for each component of the execution time.

Note that the TCP echo test is faster than the UDP echo test. Several factors help explain this somewhat surprising result. First, the cost of TCP connection establishment is effectively ignored, since it is amortized over the `read` and `write` loop. Second, the UDP-based test makes two `alarm` calls, and therefore two `setitimer` system calls, which take approximately 1.2 milliseconds each (see Table 2); the corresponding TCP timers are managed by the kernel. Finally, the `read` and `write` interface to TCP byte-streams is more streamlined than the `sendmsg` and `recvmsg` interface to UDP datagrams, which uses *scatter-gather* I/O. The scatter-gather interface uses an array of address/length pairs to specify the location in user space of the datagram to be received or sent. The array is first copied from user to kernel space, and then the pieces of the datagram specified by the array are transferred between user and kernel space. This additional copying does not occur when the `read` and `write` system calls are used.

An unreplicated Circus remote procedure call requires almost twice the time of a simple UDP exchange. This is largely due to the extra system calls required to handle various aspects of the Circus protocol. The use of interrupt-driven I/O and timers, for example, requires substantial trafficking with the software interrupt facilities in order to protect critical regions. It is worth noting that these facilities are used by Circus to compensate for the lack of multiple lightweight processes within the same address space under Berkeley 4.2BSD.

Another added expense is the presence of fairly elaborate code to handle *multi-homed* machines (machines with more than one network address). In the research computer network at Berkeley, some machines have as many as four network addresses. The `sendmsg` system call does not allow a source address to be specified when the sender is multi-homed. This means that a multi-homed server is unable to ensure that its reply to a client bears the same network address that the client used in reaching the server. The only way around this problem in the current Berkeley 4.2BSD system is for a multi-homed server to use an array of sockets, one for each of its addresses, and to use the `select` system call to multiplex among them. This situation is a design oversight in Berkeley 4.2BSD, not a fundamental problem.

The incremental expense of a Circus replicated procedure call as the degree of replication increases is more reasonable. Table 1 shows that each additional server troupe member adds between 10 and 20 milliseconds to the real time per call. The fact that this is smaller than the time for a UDP datagram exchange shows that the replicated procedure call protocol achieves some parallelism among the message exchanges with server troupe members, but it is still the case that each component of the time per call increases linearly with the size of the troupe. This linear increase is shown in Figure 4.

In the second set of tests, an execution profiling tool was used to analyze the Circus implementation in finer detail. The profiles showed that six Berkeley 4.2BSD system calls account for more than half of the total CPU time used to perform a replicated procedure call. Table 2 shows the CPU time for each of these primitives. Table 3 shows the percentage of the total CPU time for a replicated call that each of these system calls accounts for, as a function of the degree of replication.

These measurements show that most of the time required for a Circus replicated procedure call is spent in the simulation of multicasting by means of successive `sendmsg` operations, and that `sendmsg` is the most expensive of the Berkeley 4.2BSD primitives used by the Circus implementation.

system call	msecs/call	description
`sendmsg`	8.1	send datagram
`recvmsg`	2.8	receive datagram
`select`	1.8	inquire if datagram has arrived
`setitimer`	1.2	start interval timer for clock interrupt
`gettimeofday`	0.7	get time of day
`sigblock`	0.4	mask software interrupts to begin critical region

Table 2: CPU time for Berkeley 4.2BSD system calls used in Circus

degree of replication	percentage of total CPU time spent in: `sendmsg`	`select`	`recvmsg`	`setitimer`	`gettimeofday`	`sigblock`	total
1	27.2	11.2	9.2	4.4	2.2	1.7	55.9
2	28.8	12.7	10.6	3.5	2.7	1.2	59.5
3	32.5	11.7	11.9	4.2	2.6	1.0	63.9
4	32.9	10.3	10.7	5.4	2.9	0.8	63.0
5	33.0	9.9	11.1	5.0	3.1	0.9	63.0

Table 3: Execution profile for Circus replicated procedure calls

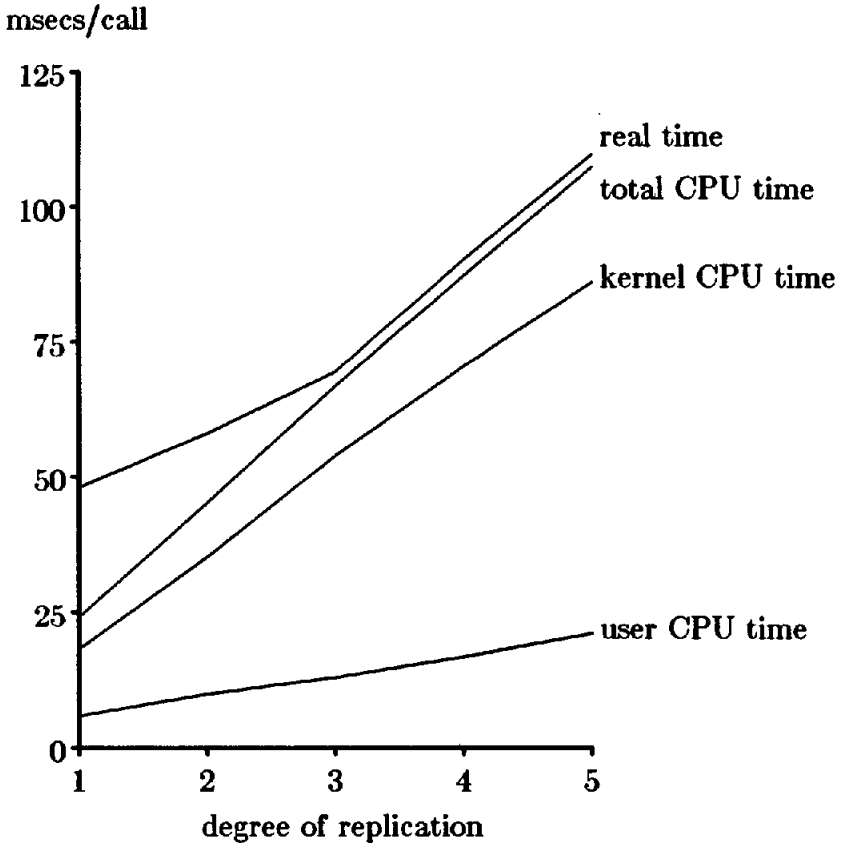

Figure 4: Performance of Circus replicated procedure calls

10 The Synchronization Problem for Troupes

Multiple threads of control give rise to concurrent calls from different client troupes to the same server troupe. This is not the same as a many-to-one call, which is handled by the algorithms described in Section 8. For a 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 can be achieved by any of a number of concurrency control algorithms [2].

When the server module is a troupe, not only must concurrent calls from different client troupes be serialized by each server troupe member, but they must be serialized in the *same order*. Correct semantics require proper coordination between the replicated procedure call mechanism and a synchronization mechanism, such as transactions.

11 Replicated Transactions

The transaction mechanism for troupes must guarantee serializability and atomicity, so that when a transaction aborts, its tentative updates and the committed updates of its subtransactions can be undone without affecting other concurrently executing transactions.

Conventional transaction mechanisms not only provide these two properties, but they also guarantee the *permanence* of committed updates. Stable storage is used for intention lists and commit records, and the commit algorithm is coupled with a crash recovery algorithm.

This third property (permanence) is not required in programs constructed from troupes, because troupes automatically mask partial failures. Consequently, an implementation of transactions for replicated distributed programs can dispense with the crash recovery facilities based on stable storage and operate entirely in volatile memory.

The correctness condition for conventional transactions is serializability. With troupes, however, independent serialization of transactions at each troupe member is not enough: troupe consistency must also be preserved.

A sufficient condition for preserving troupe consistency is to ensure that all troupe members serialize transactions in the same order. Existing concurrency control algorithms for replicated databases guarantee identical serialization orders at all replicas, but many of these algorithms require communication among replicas [2]. The desire for troupe members to remain unaware of one another's existence rules out the use of such algorithms.

One well-known multiple-copy concurrency control algorithm requires no inter-replica communication: two-phase locking with unanimous update [2]. This algorithm requires each replica to use two-phase locking for local concurrency control. The protocol presented in the next section removes this restriction.

12 A Troupe Commit Protocol

The protocol described in this section is *optimistic*, because it assumes that concurrent transactions are unlikely to conflict, and it is *generic*, because it assumes nothing about the local concurrency control algorithms used by the individual troupe members. The protocol detects any attempt by troupe members to serialize transactions differently, and transforms such an attempt into a deadlock. Deadlock detection is then used to abort and retry one or more of the offending transactions [16].

When a server troupe member is ready to commit or abort a transaction, it calls a `ready_to_commit` procedure. A **true** argument means that the server troupe member is ready to commit the transaction; a **false** argument means that the server troupe member wishes to abort the transaction. If the `ready_to_commit` procedure returns **true**, the server troupe member goes ahead and commits the transaction; otherwise, the transaction is aborted. The server's call is translated into a remote call to the client troupe. The roles of client and server are thus temporarily reversed; this is known as a *call-back protocol*.

The troupe commit protocol has the following essential property: two troupe members succeed in committing two transactions if and only if both troupe members attempt to commit the transactions in the same order.

To see this, let S_1 and S_2 be two members of a server troupe S, and let C and C' be two client troupes. Suppose C performs transaction T at S, and C' performs transaction T' at S.

If both server troupe members commit T and T' in the same order, say T followed by T', then both S_1 and S_2 call `ready_to_commit` first at C and then at C'. Client C tells both server troupe members to go ahead, and C' does the same, so both S_1 and S_2 succeed in committing the transactions.

Now suppose that S_1 tries to commit T first, but S_2 tries to commit T' first. Then S_1 calls `ready_to_commit` at C and S_2 calls `ready_to_commit` at C'. The result is deadlock, because the `ready_to_commit` procedure at each client waits for all members of the server troupe to become ready before responding to any of them. Therefore, neither S_1 nor S_2 succeeds in committing either transaction. The troupe commit protocol therefore ensures that all troupe members commit transactions in the same order.

Note that it is only necessary to transform different serialization orders into deadlocks when the different serialization orders would cause inconsistent states at troupe members. If the transactions being serialized do not conflict with one another, then inconsistency cannot occur, yet

the protocol above may still cause deadlock. To remedy this, the local concurrency control algorithm should commit non-conflicting transactions in parallel. For example, using the notation above, suppose that transactions T and T' do not conflict. Then S_1 and S_2 commit T and T' in parallel, so S_1 and S_2 call both `ready_to_commit` at C and `ready_to_commit` at C' in parallel. The deadlock described above does not occur, because the `ready_to_commit` procedure at each client receives calls from both S_1 and S_2.

13 Binding Agents for Distributed Programs

A binding agent is a mechanism that enables programs to import and export modules by interface name. In the case of distributed programs constructed with remote procedure calls, the interface name must be associated with the address of the server that exports it, and must be looked up by the client that imports it. These functions (registration, lookup, and perhaps deletion) can be provided by a general-purpose name server. For example, Grapevine [4] is used as the binding agent in the Xerox PARC RPC system [5], and Clearinghouse [24] plays the same role for Courier [32].

A natural means of reducing the cost of name server lookups is to have clients cache the results of such lookups. Thus, a client contacts the binding agent only when it imports an interface, and it uses the same information for all subsequent remote calls to that module. This raises the classic *cache invalidation problem*: what happens when a client's binding information becomes stale because the information at the name server has changed?

Suppose a client makes a remote call to a server using its cached information. In the case of programs constructed from conventional remote procedure calls, there are three reasons why the cached information might be stale:

1. There is no longer a server at the specified address.
2. There is a server at that address, but it no longer exports the specified interface.
3. There is a server at that address and it exports that interface, but the actual instance of the module in question is no longer the same as the one originally imported by the client.

If all three of these cases can be detected at or below the remote procedure call protocol level, the run-time system can raise an exception in the client to indicate that rebinding is required. Therefore, the problem of masking stale binding information reduces to the problem of detecting the above three cases.

The first can be detected at the paired message protocol level, since there will be no response to repeated retransmissions. The second can be detected at the remote procedure call protocol level, because the server's run-time system will reject the call. The third case requires some help from the binding agent, in the form of *incarnation numbers* for exported interfaces, as in the system described by Birrell and Nelson [5]. This scheme time-stamps the record created when a server registers itself with the binding agent. The client's run-time system receives the time stamp along with the server address when it imports an interface and includes it in all subsequent calls to that module. It is thus a simple matter for the server's run-time system to detect and reject mismatches.

A related problem is *garbage collection*, which is required when some of the binding agent's own registration information becomes obsolete. This can happen if a server crashes or otherwise ceases to export an interface without informing the binding agent. The problem of garbage collection reduces to the cache invalidation problem, since the information maintained by the binding agent is itself just a cached version of the truth. Of the above three ways in which binding information can be out of date, only the first two apply to the binding agent. The third case is detected by the binding agent itself as part of the process of assigning incarnation numbers: when a server re-exports an interface, the binding agent will notice that there is already an entry for that name and address. In the first two cases, however, it is the client that ends up detecting the invalid binding; this fact must somehow reach the binding agent.

One solution is to include a special `rebind` procedure in the interface to the binding agent. Each client, upon detecting an invalid binding, calls `rebind` with the invalid binding as an argument. The binding agent looks up and returns the current binding for the given name, and deletes the old binding if it is still present. (The old binding passed to the `rebind` procedure is only a hint; it need not be deleted immediately, nor should it be blindly accepted as invalid in an insecure environment.)

Another solution is to use a garbage collector: a process which periodically enumerates all the registered modules, probes them with a special null procedure call (an "are you there?" request), and explicitly deletes the bindings for modules that do not respond. The garbage collector need not be part of the binding agent if the binding interface includes enumeration and deletion.

14 Binding Agents for Replicated Programs

Replicated distributed programs import and export

troupes rather than single modules, and therefore require additional support from the binding mechanism. First of all, the binding agent must manipulate sets of module addresses rather than single addresses, and it must manage the troupe IDs required by the replicated procedure call algorithms of Section 8. The binding agent must allow a third party to register an entire troupe. Finally, it must be possible to add or delete individual troupe members, in order to handle troupe reconfiguration.

Since binding is such a pivotal mechanism, it is essential that the binding agent be highly available. An obvious choice is to make the binding agent a troupe and express the interactions with it in terms of replicated procedure calls. The interface to such a binding agent is shown in Figure 5.

The initial registration of a troupe also requires the ability to add a member to an existing troupe. A troupe cannot register itself *en masse* with a single replicated procedure call, because it does not have a troupe ID until it is registered. To avoid this circularity, each troupe member must add itself individually to an initially empty troupe, using `add_troupe_member`. The synchronization requirements of the `add_troupe_member` operation are discussed below.

The cache invalidation problem becomes more complicated when replication is introduced. Let T be the set of members of a troupe, and let C be the cached set of members that a client believes constitutes the troupe. Then C is stale if and only if $C \neq T$. The possibilities for stale information correspond to the possible intersections of these two nonempty sets:

1. $T \cap C = \emptyset$
2. $T \subset C$
3. $T \supset C$
4. $T \cap C \neq \emptyset \wedge T \not\subset C \wedge T \not\supset C$

The semantics of troupes and replicated procedure calls require every member of a server troupe to execute a procedure if any member does. This will be the case if $T = C$, $T \cap C = \emptyset$, or $T \subset C$. The first two possibilities for stale information are therefore harmless; the client will detect that some or all of the members of C are invalid, and perform the necessary rebinding. In the last two cases, however, the client calls some but not all of the troupe members; these calls cannot be allowed to succeed.

The solution is to use troupe IDs as a form of incarnation number. Each call message carries the troupe ID of its destination as well as its source, and each server troupe member rejects any call message whose destination troupe ID is incorrect. If it can be guaranteed that a troupe always changes both its membership and its troupe ID in an atomic operation, then the problem is solved: a server troupe member accepts a call from a client only if it bears the correct server troupe ID, which is the case only if the client knows the correct membership of that server troupe.

The `add_troupe_member` procedure must therefore be an atomic transaction that also changes the troupe ID. This requires informing the existing members of the troupe that their troupe ID has changed, which can be accomplished by running a special `set_troupe_id` procedure at each member. The `set_troupe_id` procedure for each troupe can be generated automatically. If `set_troupe_id` is executed as a subtransaction of `add_troupe_member`, the change in troupe ID and troupe membership will happen atomically and will be correctly serialized with any other calls to the server troupe.

```
binding:
    interface
        troupe_name: type = string
        troupe_member: type = module_address
        troupe: type = set of troupe_member
        troupe_id: type = unique_id

        register_troupe:
            procedure (troupe_name, troupe)
            returns (troupe_id)

        add_troupe_member:
            procedure (troupe_name, troupe_member)
            returns (troupe_id)

        lookup_troupe_by_id:
            procedure (troupe_id)
            returns (troupe)

        lookup_troupe_by_name:
            procedure (troupe_name)
            returns (troupe)

    end interface
```

Figure 5: The interface to the binding agent

15 Reconfiguration and Recovery from Partial Failures

A troupe is resilient to *partial failures*, in which at least one of its members continues to function. Machine crashes are detected (using a timeout) by the paired message protocol, which raises an exception that can be used by higher level software. At some point it becomes desirable to replace troupe members that have crashed, because a dimin-

ished troupe is more vulnerable to future crashes.

Replacing a crashed troupe member or adding a new troupe member to an existing troupe requires the following two steps:

1. the new member must be brought into a state consistent with that of the other members, and

2. the new member must be registered with the binding agent.

This section describes how to perform the first step; the `add_troupe_member` procedure (described in Section 14) is used to accomplish the second step.

The solution is to use a mechanism similar to checkpointing. In this scheme, the state information of an existing troupe member is externalized (converted to a standard external representation), then transmitted to the newly created troupe member, where it is internalized. The transmission method for abstract data types proposed by Herlihy and Liskov is similar [18].

A special `get_state` procedure can be produced automatically for this purpose. The `get_state` procedure copies the module state from the callee to the caller and handles the details of externalization and internalization. This procedure executes as a read-only atomic transaction, so that the state cannot be affected while a new troupe member is being initialized. A new server process wishing to join a troupe initializes its state by making a replicated call to the `get_state` procedure at the existing members of the troupe, and then calls the binding agent's `add_troupe_member` procedure to register itself. Since the states of the existing troupe members are consistent, and since `get_state` is free of side effects at the callee, the replicated call to `get_state` is not strictly necessary; an unreplicated call to any of the existing troupe members would suffice.

Finally, the call to `add_troupe_member` and the call to `get_state` must be bracketed together in a single atomic transaction, to guarantee that the new member joins the troupe and acquires the correct state as an indivisible operation.

16 Summary

The mechanisms described in this paper allow a programmer to add replication transparently and flexibly to existing programs. The resulting replicated distributed programs automatically tolerate partial failures of the underlying fail-stop hardware.

The architecture combines remote procedure calls with replication of program modules for fault tolerance. The replicated modules, called troupes, are the basis for constructing replicated distributed programs.

Previous fault-tolerant architectures were either too expensive or too inflexible. Simple replication of hardware components, for example, requires all software to be executed redundantly, rather than just critical modules, and permits only a single degree of replication. In contrast, the present approach introduces replication at the program module level, and allows the degree of replication of each module to vary independently and dynamically.

The replication mechanisms introduced in this paper can be used transparently, so that the details of replication are invisible to the programmer. Transparent distributed and replicated mechanisms are an important means of coping with the complexity of fault-tolerant distributed programs. A model of program semantics was used to characterize deterministic programs, a class of programs that can be transparently replicated.

The model is based on program modules and threads of control. In a distributed system, threads must be able to cross machine boundaries to move between modules on different machines. An algorithm to simulate such distributed threads in terms of conventional processes and remote procedure calls was presented.

Transfer of control between troupes requires generalizing remote procedure calls to replicated procedure calls. The semantics of replicated procedure calls can be summarized as exactly-once execution at all replicas.

The Circus replicated procedure call implementation was described. Message transport is provided by a datagram-based paired message layer. The general replicated procedure call protocol, requiring many-to-many communication, is expressed in terms of two sub-protocols, for the one-to-many and many-to-one cases.

In the Circus system, each troupe member waits for all incoming messages before proceeding. Troupe members are thus synchronized at each replicated procedure call and return. Alternative schemes that allow computation to proceed before all messages have arrived were discussed.

Experiments were conducted to measure the performance of the Circus replicated procedure call implementation. The results of the measurements show that six Berkeley 4.2BSD system calls account for more than half of the CPU time of a Circus replicated procedure call. The two most expensive of these system calls use a particularly inefficient interface to copy data between user and kernel address spaces. The other four system calls are used to compensate for the lack of lightweight processes in Berkeley 4.2BSD.

The use of transactions for synchronizing concurrent threads of control within replicated distributed programs was discussed. Serializability, the property guaranteed by concurrency control algorithms for conventional transac-

tions, was shown to be insufficient for the purposes of replicated transactions, because it does not guarantee that transactions commit in the same order at all troupe members. A troupe commit protocol that guarantees a consistent commit order for replicated transactions was presented.

Mechanisms for binding and reconfiguring replicated distributed programs were described. The problem of detecting obsolete binding information was identified; this problem is both more complicated and more critical than the corresponding problem in the unreplicated case. A solution using troupe IDs as incarnation numbers was presented.

17 Directions for Future Research

Replicated procedure calls are useful for more than just fully replicated distributed programs. The troupe commit protocol (presented in Section 12) and other protocols in the author's Ph.D. dissertation [12] are examples of how the use of replicated procedure calls leads to elegant formulations of algorithms traditionally described in terms of asynchronous messages.

An important area for further research is to express more algorithms of this type in terms of replicated procedure calls. For example, the algorithms used in distributed database systems for concurrency control, replicated data, atomic commit and recovery, and deadlock detection would lend themselves to such treatment.

Further research is needed to evaluate the alternative replicated procedure call protocols described in Section 8.4 and to discover new ones. An approach that allowed the choice between such schemes to be made on a per-module basis, as a programming-in-the-large activity, would be attractive.

The troupe commit protocol presented in Section 12 must be implemented and its performance evaluated. Allowing application-specific concurrency control within the context of troupes is another area for further work.

Acknowledgments

I would like to thank Robert Fabry, Domenico Ferrari, Susan Graham, Leo Harrington, Andrew Birrell, Earl Cohen, Daniel Halbert, and Naomi Siegel for their support, encouragement, and advice.

This work was sponsored by a National Science Foundation Graduate Fellowship, the Xerox Corporation, the Digital Equipment Corporation, and by the Defense Advanced Research Projects Agency (DoD), ARPA order number 4031, monitored by the Naval Electronics Systems Command under contract number 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 Advanced Research Projects Agency or of the U.S. Government.

References

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

[2] Philip A. Bernstein and Nathan Goodman.
Concurrency control in distributed database systems.
Computing Surveys 13(2), June 1981, pages 185–221.

[3] Kenneth P. Birman, Thomas A. Joseph, Thomas Räuchle, and Amr El Abbadi.
Implementing fault-tolerant distributed objects.
Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems, October 1984, pages 124–133.

[4] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder.
Grapevine: An exercise in distributed computing.
Communications of the ACM 25(4), April 1982, pages 260–274.

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

[6] 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, pages 90–99.

[7] 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, pages 3–9.

[8] David R. Cheriton and Willy Zwaenepoel.
One-to-Many Interprocess Communication in the V-System.
Report STAN-CS-84-1011, Department of Computer Science, Stanford University, August 1984.

[9] Melvin E. Conway.
A multiprocessor system design.
Proceedings of the AFIPS 1963 Fall Joint Computer Conference, volume 24, pages 139–146.

[10] Eric C. Cooper.
Replicated procedure call.
Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, August 1984, pages 220–232.

[11] Eric C. Cooper.
Circus: A replicated procedure call facility.
Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems, October 1984, pages 11–24.

[12] Eric C. Cooper.
Replicated Distributed Programs.
Ph.D. dissertation, Computer Science Division, University of California, Berkeley, April 1985.
Report UCB/CSD/85/231.

[13] E. W. Dijkstra.
Cooperating sequential processes.
In *Programming Languages*, edited by F. Genuys. Academic Press, 1968, pages 43–112.

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

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

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

[17] 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, pages 182–185.

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

[19] Maurice Peter Herlihy.
Replication Methods for Abstract Data Types.
Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT, May 1984.
Report MIT/LCS/TR-319.

[20] 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,

[21] Leslie Lamport.
The implementation of reliable distributed multiprocess systems.
Computer Networks 2(2), May 1978, pages 95–114.

[22] 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, pages 200–209.

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

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

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

[26] Jon Postel.
Transmission Control Protocol.
RFC 793, Information Sciences Institute, University of Southern California, September 1981.

[27] Richard D. Schlichting and Fred B. Schneider.
Fail-stop processors: An approach to designing fault-tolerant computing systems.
ACM Transactions on Computer Systems 1(3), August 1983, pages 222–238.

[28] Sun Microsystems.
Remote Procedure Call Reference Manual.
Mountain View, California, October 1984.

[29] 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, pages 43–98.

[30] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock.
SIFT: Design and analysis of a fault-tolerant computer for aircraft control.
Proceedings of the IEEE **66**(10), October 1978, pages 1240–1255.

[31] Karen White.
An Implementation of a Remote Procedure Call Protocol in the Berkeley UNIX Kernel.
M.S. report, Computer Science Division, University of California, Berkeley, June 1985.
Report UCB/CSD/85/248.

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

[33] Gary York, Daniel Siewiorek, and Zary Segall.
Asynchronous software voting in NMR computer structures.
Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems, October 1983, pages 28–37.