Circus: A Replicated Procedure Call Facility

Eric C. Cooper

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

Abstract

The Circus replicated procedure call facility for Berkeley UNIX* is described. The main components of the system are a paired message protocol, a runtime library implementing replicated procedure call semantics, a binding service for replicated modules, and a stub compiler for the C programming language. The implementation of each of these components is discussed.

1. Introduction

This paper describes a replicated procedure call facility, called Circus, that has been implemented for Berkeley UNIX. Replicated procedure call [7] is a new mechanism for constructing highly available distributed programs. It combines remote procedure call with replication of program modules for fault tolerance.

The main components of the Circus system are a paired message protocol, a runtime library implementing replicated procedure call semantics, a binding service for replicated modules, and a stub compiler for the C programming language [17].

Figure 1 shows the structure of the Circus system. The protocol layers used in Circus are shown in figure 2.

2. Remote Procedure Call

The goal of remote procedure call [24] is to allow distributed programs to be written in the same style as conventional programs for centralized computers. Details of communication are hidden, and the syntax of a remote call is similar or identical to the local case.

A complete remote procedure call facility must address the following issues.

(1) Reliable communication of CALL and RETURN messages of variable length.

(2) Procedure invocation semantics.

(3) Binding (importing and exporting remote modules).

(4) Representation of parameter and result types.

---

*UNIX is a trademark of Bell Laboratories.

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 4th Symposium on Reliability in Distributed Software and Database Systems*, October 1984.

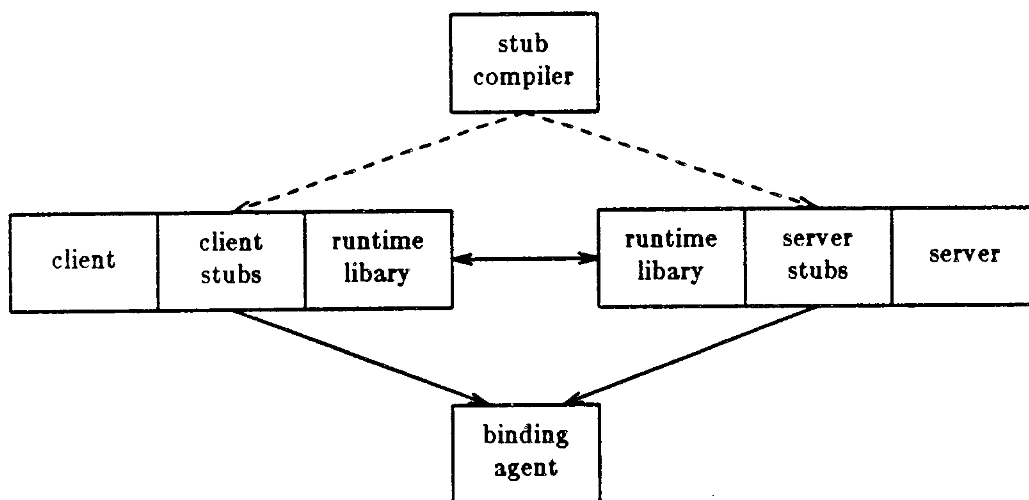

Figure 1: Structure of the Circus system

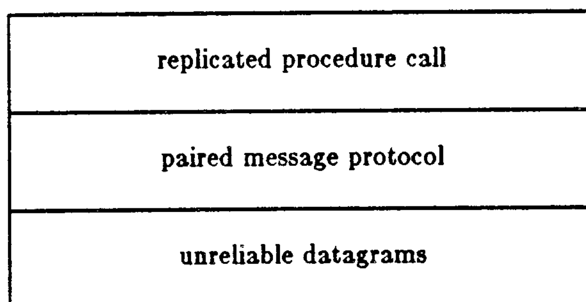

Figure 2: Circus protocol layers

3. Troupes and Replicated Procedure Call

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 all the results. Figure 3 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.

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, produce the same side effects, and return the same results. Note that this

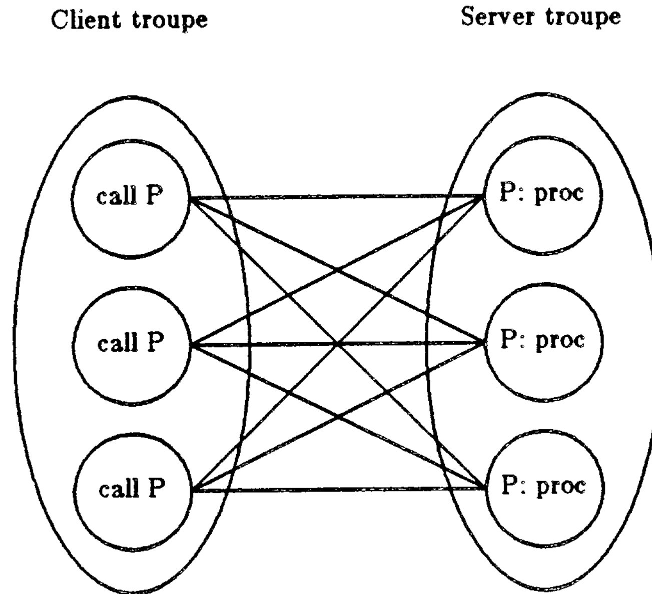

Figure 3: Replicated procedure call

determinism requirement is also implicit in roll-forward crash recovery schemes such as replay of message logs [5,14] or re-execution of intention lists [18]. An advantage of the troupe mechanism is that "same" can be replaced by an application-specific equivalence relation.

Just as replicated procedure call is a conceptual extension of remote procedure call, the Circus implementation described here evolved as an extension of a remote procedure call implementation for Berkeley UNIX. When the degree of module replication is one, Circus functions as a conventional remote procedure call system.

A description of replicated procedure call and the algorithms required to implement its semantics has appeared elsewhere [7]. This paper concentrates on the implementation of these algorithms in a practical system.

3.1. Related Work

This research began as an attempt to transfer the Xerox RPC ideas [4,23,24,33] to an environment based on Berkeley UNIX [16] and DARPA Internet protocols [27,28]. The idea of achieving fault tolerance by means of replication dates back to von Neumann [32]. The present work may be viewed as an extension of the techniques of triple-modular and N-modular redundancy [1,21]. This approach is also similar to the one taken by Gunningberg, who proposed a fault-tolerant message protocol based on triple-modular redundancy [15]. We have extended Gunningberg's idea to a system based on remote procedure calls and fully general replication and voting schemes.

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 to increase software as well as hardware fault tolerance.

We use a form of replication in which each component performs the same function, in contrast to schemes such as those of Tandem [2,30] or Auragen [5] in which only a single

component functions normally and the remaining replicas are on stand-by in case the primary fails.

4. The Paired Message Protocol

The paired message protocol provides reliably delivered, variable-length, paired CALL/RETURN messages. It uses UDP, the DARPA User Datagram Protocol [27].

The paired message protocol is responsible for segmenting messages that are larger than a single datagram, in order to permit variable-length messages, and for retransmission and acknowledgment of message segments to ensure reliable delivery. It is connectionless and geared towards the fast exchange of short messages. Our protocol is based very closely on the RPC protocol of Birrell and Nelson [4]. The only real difference lies in the treatment of messages requiring multiple datagrams; our protocol provides better recovery from lost datagrams in this case.

The paired message protocol does not specify how remote modules or procedures are identified, how clients and servers are bound together, how parameters and results are represented, or how exceptional conditions are handled. The contents of the messages are uninterpreted. It is therefore possible for several remote (or replicated) procedure call systems, with different type representation and module binding requirements, to use this same protocol as a basis for communication. For example, in addition to the Circus system, a simple remote procedure call facility was implemented for Franz Lisp [12] that uses the same paired message protocol, but represents procedures and values symbolically in messages.

4.1. Addresses

Messages are exchanged between processes. A *process address* consists of a 32-bit host address together with a 16-bit port number. The host address identifies the machine within the DARPA Internet, and the port number identifies the process within the machine. This is the same address format used by the underlying UDP layer; we rely on the UDP implementation for the assignment of port numbers to processes.

4.2. Messages and Segments

A message is a sequence of bytes, together with a type (CALL or RETURN). Messages are transmitted as one or more *segments* of fixed maximum length. A segment is a UDP datagram of the form shown in figure 4.

A *data segment* consists of a segment header together with some portion of the message data. A *control segment* contains only a segment header; it is used to send acknowledgment information. The *message type* field is a byte containing either 0 for a CALL message or 1 for a RETURN message. The *control bits* field is a byte containing two Boolean values used to control the acknowledgment and retransmission procedures. The least significant bit is the PLEASE ACK flag, and the next least significant bit is the ACK flag. The six most significant bits are unused.

The next two bytes are used to specify the logical position of the segment within the whole message. The *total segments* field is a byte containing the total number of segments in the message, which must be in the range from 1 to 255, inclusive. The *segment number* field is a byte containing a number between 0 and the total number of segments, inclusive. The meaning of this field depends on whether the segment contains data or acknowledgment information.

The *call number* field is a 32-bit unsigned integer, represented most significant byte first. The call number is used to pair CALL messages with the corresponding RETURN messages.

4.3. Sending a Message

This section describes the protocol for sending a message. It is the same for both client and server; the only difference is whether the message type is CALL or RETURN.

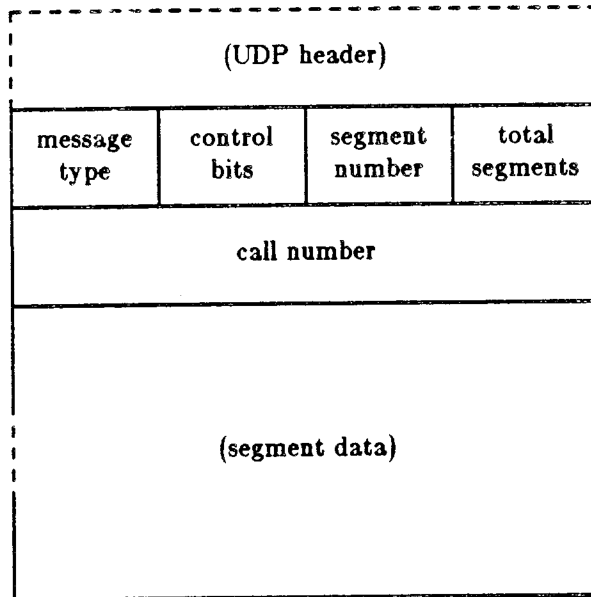

Figure 4: Segment format

A sequence of bytes to be sent as a message is first divided into segments. Each segment is assigned a number, starting at 1, which is placed in the segment number field of its header. The message type, total number of segments, and call number fields of the header are the same for each segment of the message. The sender maintains a queue of the unacknowledged segments of the message, initially containing all the segments.

The sender initially transmits all the segments to the receiver with no control bits set. It then periodically retransmits the first unacknowledged segment on its queue, with the PLEASE ACK bit set. Simultaneously, the sender listens for acknowledgments and removes acknowledged segments from its queue. When all the segments have been acknowledged and the queue is empty, transmission of the message is complete.

An acknowledgment is either explicit or implicit. An explicit acknowledgment is a segment with the ACK bit set and the same message type, call number, and total number of segments as the current message. Acknowledgment segments contain no data; the segment number field is used as an acknowledgment number, indicating that all segments with numbers less than or equal to the acknowledgment number have been received.

An implicit acknowledgment is a data segment sent by the receiver back to the sender. A segment from a RETURN message implicitly acknowledges all the segments of the previous CALL message if it carries the same call number, and a segment from a CALL message implicitly acknowledges all the segments of the previous RETURN message if it carries a later call number. Implicit acknowledgments are possible because processes alternate between sending and receiving.

4.4. Receiving a Message

The protocol for receiving a message is also the same for both client and server. The receiver maintains a queue of incoming segments for the current message, and an acknowledgment number, initially zero. The acknowledgment number is the highest consecutive segment number

received.

When a segment arrives, it is placed in its proper position in the queue. The segment may have filled a gap in the queue, enabling the acknowledgment number to be advanced. If the PLEASE ACK bit is set in the incoming segment, an explicit acknowledgment segment is sent with the current value of the acknowledgment number in the segment number field.

Reception of the message is complete as soon as all the segments have been received.

4.5. Client Probing

Once a client has sent an entire CALL message and its receipt has been acknowledged, the client may wait arbitrarily long before the remote procedure finishes and sends back the RETURN message. In order to detect crashes during this interval, the client periodically *probes* the server with a PLEASE ACK segment containing no data.

4.6. Correctness and Crash Detection

The send and receive protocols guarantee that messages will be communicated correctly in the presence of lost or duplicated datagrams, (assuming that any segment retransmitted repeatedly will eventually be received).

This assumption does not hold in the event of a crash. In order to detect crashes, an upper bound must be placed on the number of retransmissions with no response before it is assumed that the receiver has crashed. A bound that is too low increases the chance of incorrectly deciding that a receiver has crashed. A bound that is too high introduces a long delay in the detection of true crashes.

4.7. Optimizing Acknowledgments and Retransmissions

Several optimizations are possible to reduce the number of acknowledgments and retransmissions. For instance, when an out-of-order segment arrives during receipt of a multiple-segment message, the receiver knows that one or more segments have been lost. It should therefore immediately send an explicit acknowledgment for the last consecutively received segment, so that the sender will retransmit the first lost segment, rather than an earlier segment.

When a segment that completes a CALL message arrives at a server, acknowledgment of the message can be postponed, even if the PLEASE ACK bit was set, in the hope that the RETURN message will be forthcoming soon enough to serve as an implicit acknowledgment. Subsequent PLEASE ACK segments should be acknowledged promptly.

The retransmission strategy could be changed to retransmit all the remaining unacknowledged segments rather than just the first, depending on the reliability characteristics of the network.

4.8. State Information

The protocol is connectionless in the sense that no initial handshake is needed to establish communication; a client merely sends a CALL message to a server. Clients and servers must maintain state information about active message exchanges (segment queues and acknowledgment numbers). After an exchange has completed, only its call number must be kept, and this may be discarded once sufficient time has passed to guarantee that no delayed segments from the exchange can arrive. This is to prevent the "replay" of delayed CALL messages.

4.9. Maximum Segment Length

The maximum length of a segment is implementation dependent, but must be no larger than the maximum UDP datagram size minus the 8 bytes of segment header. It may be desirable to use a smaller limit in order to prevent fragmentation at the IP level [28]. This requires knowledge

of the maximum transmission unit (MTU) for the physical networks of interest (presumably the local area networks expected to be used most often).

4.10. UNIX Implementation Issues

This section discusses various details of the implementation of the paired message protocol under Berkeley UNIX. The protocol is currently implemented entirely in user code, although a project is under way to transfer it to the UNIX kernel for improved performance.

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 that may take arbitrarily long. If multiple processes sharing the same address space were available under UNIX, a separate process could be devoted to listening for incoming segments and handling timers. Since this is not possible, these events are modelled as software interrupts using the *signal* mechanism, interrupt-driven I/O facility, and interval timer of Berkeley UNIX version 4.2 [16]. Protection of critical regions is achieved by using system calls that mask and enable interrupts.

The protocol package uses timers to handle retransmission, probing, no-response timeouts, and no-activity timeouts. A general timer package was built on top of the single UNIX interval timer for this purpose. It allows a timer to be defined by a timeout interval and a procedure to be invoked upon expiration; any number of timers may be active at the same time.

5. Implementation of Troupes and Replicated Procedure Call

This section describes how replicated procedure calls are implemented on top of the paired message layer.

5.1. Addresses

Replicated procedure calls are made between troupes, which consist of modules. A *module address* is a refinement of a process address, since one process may export several modules. It consists of a process address together with a 16-bit module number that identifies the module among those exported by that process. (The module number is assigned by the export procedure and is an index into a table of exported interfaces.)

A troupe is represented at this level by a sequence of module addresses. This representation is returned by the binding agent when a client imports a server troupe.

5.2. CALL messages

We now describe the contents of a CALL message. Remember that this data is uninterpreted by the paired message layer. A CALL message consists of a header containing a module number and procedure number, and the parameters to the procedure. The module number is the only component of the module address of the destination that is required at this level; the underlying paired message protocol deals with the process address component. The procedure number is assigned by the stub compiler and is the index of the procedure within the module interface. The header also contains other information described below. The parameters are represented in a standard external form by the routines produced by the stub compiler.

5.3. RETURN messages

A RETURN message consists of a 16-bit header, used to distinguish between normal and error results, and the results of the procedure in the standard external representation.

5.4. One-to-many calls

The client half of the replicated procedure call algorithm performs a one-to-many call as shown in figure 5. The same CALL message is sent to each server troupe member, with the same

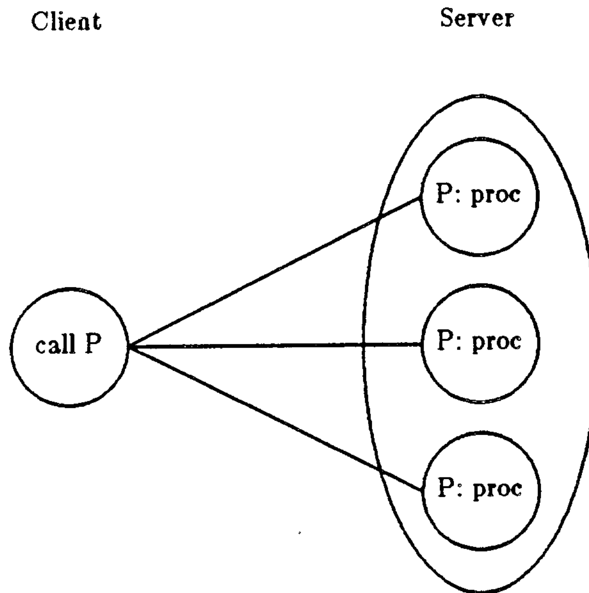

Figure 5: A one-to-many call

call number at the paired message level. The client then awaits the arrival of the RETURN messages from the members of the server troupe.

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.

5.5. Many-to-one calls

Now consider what occurs at a single server when a client troupe calls a server troupe. The server will receive CALL messages from each client troupe member, as shown in figure 6. 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. How can the server tell that all the CALL messages are part of the same replicated call, rather than many unrelated calls? Our solution requires a unique ID for each troupe (assigned by the binding agent) and two more fields in the CALL message header.

First, a *client troupe ID* contains the troupe ID of the client troupe making the call. 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. This is done by consulting a local cache or by contacting the binding agent. The server now knows how many CALL messages to expect as part of the many-to-one call.

Second, a *root ID* uniquely identifies the entire chain of replicated calls of which this one is a part. The root ID consists of the troupe ID of the client that started the chain of calls and the call number of its original CALL message. The root ID is like a transaction ID; it is propagated whenever one server calls another. It can be shown that two or more CALL messages arriving at a server will have the same root ID if and only if they are part of the same replicated call [7].

The client troupe ID and the root ID together allow the server to collect the entire set of CALL messages that form a many-to-one call. The procedure is executed once, and a RETURN

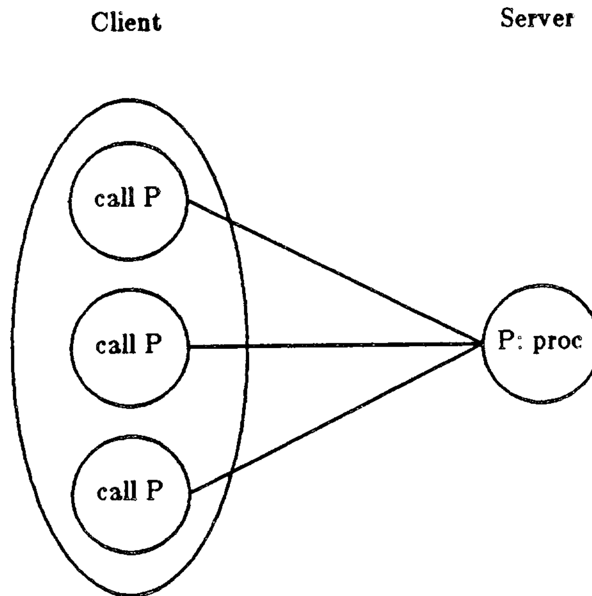

Figure 6: A many-to-one call

message containing the results is sent to each member of the client troupe.

5.5. Collators

A client making a one-to-many call expects a single result, but 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 receives a CALL message from each client troupe member.

Under strong assumptions of determinism, one can require that all the messages in these sets be identical. It is possible to relax this requirement (at the cost of transparency) and allow applications to specify their own procedures for reducing a set of messages to a single message. We call such procedures *collators*.

A collator is basically a function that maps a set of messages into a single result. For performance reasons, 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.) We therefore complicate the definition of a collator slightly. The collator is invoked each time a message in the set arrives, until it returns an indication that it has reached a decision. The collator is applied not to a set of messages, but to a set of status records for the expected messages. Each status record contains one of the following variants:

(1) The contents of the message.

(2) An indication that the message has not arrived but is still expected.

(3) An indication that an error has occurred and the message will never arrive.

Only three collators are currently supported: *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 a variety of voting schemes and broadcast-based algorithms [13, 19, 21, 25, 26, 31, 32].

5.7. Invocation semantics

Nelson argues persuasively that in the presence of concurrency, parallel invocation semantics rather than serial are required in order to match the semantics of the local case [24]. When incoming calls are serialized by arrival time, the possibility of deadlock is introduced. This type of deadlock does not occur when incoming calls are handled by concurrent processes.

Our current implementation suffers from this deficiency because of the lack of multiple processes within the same address space under UNIX. We have provided a partial solution in the form of a simple process mechanism for C that supports several threads of control with synchronization by signalling and awaiting events.

5.8. Use of multicast

The UNIX networking primitives used by Circus do not allow access to the multicast capabilities of the Ethernet [11]. If this were changed, the operation of sending the same message to an entire troupe could be implemented by a multicast operation, and the binding agent, described below, could manipulate Ethernet hardware group addresses.

6. The Binding Agent

This section describes the *Ringmaster*, a binding agent for troupes. The Ringmaster is a specialized name server enabling programs to import and export troupes by name. It plays the same role that Grapevine does in the Xerox PARC RPC system [3,4]. The main differences are that the Ringmaster

(1) manipulates troupes (sets of module addresses),

(2) is a dedicated binding agent, and

(3) is itself a troupe whose procedures are invoked via replicated procedure call.

A client imports a module by calling **find troupe by name**. The procedure returns the set of module addresses associated with that name.

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

The UNIX process ID of the server process is also recorded in the entry for the module, so that the Ringmaster can periodically perform garbage collection of troupe members whose processes have terminated.

A server handling a many-to-one call uses **find troupe by ID** to map a client troupe ID into the set of module addresses of the troupe members.

Access to the binding procedures is by means of stubs produced by the stub compiler from the Ringmaster interface. These stubs are part of the Circus runtime library.

Since the Ringmaster cannot be used to import itself, a special degenerate binding mechanism is used for the Ringmaster module: the Ringmaster troupe is partially specified by means of a well-known port on each machine, and the set of machines running instances of the Ringmaster is determined dynamically.

7. The Stub Compiler

This section describes a stub compiler, called *Rig*, that translates remote module interfaces into client and server stub routines in C [17]. The stub routines take responsibility for sending

parameters and results between client and server troupe members via the replicated procedure call runtime package.

7.1. Specification of Remote Interfaces

The programmer defines module interfaces by means of a specification language derived from Courier [33]. A module consists of a sequence of declarations of types, constants, and procedures. The type algebra is almost identical to that of Courier. Certain of the Courier types require a programming language capable of supporting them: error types (exceptions) that procedures may report in lieu of returning a result, constants of arbitrary constructed types, and procedures that return multiple results. Because of the lack of support for these features in the C language, they are not supported in this implementation.

The predefined types include Booleans, 16-bit and 32-bit signed and unsigned integers, and character strings. The constructed types are enumerations, arrays, records, variable-length sequences, and discriminated unions.

The predefined types and the enumeration, array, and record types have obvious C counterparts. The variable-length sequences and discriminated unions pose some problems when they are mapped into C, because an object of one of these types must contain run-time information (the length of the sequence or which variant is present) that is implicit in the Courier type, but must be made explicit in C. Furthermore, the C programmer must bear the responsibility of keeping this information consistent when these objects are manipulated by functions other than those generated automatically by the stub compiler.

7.2. External Representation of Types

The Courier protocol specifies how objects of each type are represented when transmitted in CALL and RETURN messages; we adopt the same representation. Most of the work of the stub routines consists of translating parameters and results between their external and internal representations. This may involve byte-swapping of integers, realignment of record fields, and storage allocation for objects of variable-length types.

7.3. Transparency of Troupes

The stub compiler also produces binding stubs to import and export each module that it compiles. These routines make replicated procedure calls to the Ringmaster as described above. The representations of troupes that are returned by these binding procedures are used by the client and server stub routines. In this way, once a program has been compiled, no editing or recompilation is required to change the number or location of troupe members.

8. Conclusion

We have described the Circus replicated procedure call facility, a preliminary implementation of our ideas about troupes and replicated procedure call. Currently, the only "production" program using troupes is the Ringmaster binding agent; programmers other than the author have only used Circus in its degenerate capacity as a non-replicated remote procedure call facility.

8.1. Future Research

We are continuing this research in a number of directions [8]. We are investigating the relationship between replicated procedure call and concurrency control mechanisms such as nested atomic actions, in order to clarify the semantics of concurrent replicated calls from unrelated client troupes to the same server troupe.

At the same time, we are exploring the spectrum of possible determinism requirements on troupe members.

The programming-in-the-large issues associated with troupes are also being studied. We are designing a configuration language and a configuration manager for programs constructed from troupes. Our approach will be to extend previous work in this area [9, 10, 20, 22, 24, 29] to handle troupe creation and reconfiguration.

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, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM* 25, 4 (April 1982), pp. 260-274.

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

[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. Replicated procedure call. *Proceedings of the 3rd ACM Symposium on Principles of Distributed Systems,* August 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] John K. Foderaro, Keith L. Sklower, and Kevin Layer. *The Franz Lisp Manual.* Computer Science Division, University of California, Berkeley, June 1983.

[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] 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.

[17] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[18] 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.

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

[20] 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.

[21] 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.

[22] James G. Mitchell, William Maybury, and Richard Sweet. *Mesa Language Manual, Version 5.0.* Xerox PARC report number CSL-79-3, April 1979.

[23] Bruce Nelson and Andrew Birrell. *Lupine User's Guide: An Introduction to Remote Procedure Calls in Cedar.* The Cedar Manual, Computer Science Laboratory, Xerox PARC, July 1982.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] Eric Emerson Schmidt. *Controlling Large Software Development in a Distributed Environment.* Ph.D. dissertation, Computer Science Division, University of California, Berkeley, Xerox PARC report number CSL-82-7, December 1982.

[30] Tandem Computers Inc. *GUARDIAN Operating System Programming Manual, Volumes 1 and 2.* Cupertino, California, 1982.

[31] 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.

[32] 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.

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