Replicated RPC Using Amoeba Closed Group Communication

Mark D. Wood*
Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

Since RPC has become the method of choice for client-server communication in a distributed operating system, providing a fault-tolerant RPC mechanism is crucial to ensuring system reliability. This paper presents a replicated RPC library for the Amoeba distributed operating system. The library's RPC protocol is presented in detail, including preliminary performance figures. The protocol is distinguished by its use of closed process groups in conjunction with the coordinator-cohort method of computation. The method presented here is applicable to any system supporting closed process groups and totally ordered multicast.

Keywords: *fault-tolerant RPC, Amoeba, active replication, multicast, broadcast, process groups.*

1 Introduction

Distributed systems are often structured as a set of clients making requests of a set of servers, with the remote procedure call (RPC) being the primary communication method. In the basic RPC mechanism, a single client process makes a procedure call requesting a remote server to compute some result; the client blocks until the server returns the result. The great attraction to RPC lies in its similarity to the normal, local procedure call. RPC mechanisms have been around for about a decade, with [5] providing a detailed description of an important early implementation. A good discussion of RPC may be found in [15].

Ideally, RPC provides a location-transparent way for programming a distributed system, but in practice, location transparency is difficult to achieve. Distributed systems are prone to partial failures—failures in which some components operate incorrectly while other components remain operable. Since the RPC mechanism is so important for programming distributed systems, a considerable amount of research has been directed towards developing reliable, fault-tolerant RPC protocols. By using a reliable RPC, key system services may be accessed despite partial communication or processor failures. Such fault-tolerant execution is provided by replicating the server and client programs.

This paper presents a reliable RPC mechanism for the Amoeba distributed operating system [12]. This RPC package, which is provided as a C library, is used to construct fault-tolerant services. To write a fault-tolerant server program, the programmer declares to the library the otherwise non fault-tolerant server routines. The programmer may additionally supply state-transfer routines to be used by the library when integrating new replicas. Multiple replicas of the server program are then run to implement a reliable service. Client programs make fault-tolerant remote procedure calls merely by using the RPC call provided in the library. Programs being replicated using this method must be deterministic, since an optimized version of the state-machine approach [14] is used to provide fault-tolerance.

This work is distinguished in part by its use of closed process groups [10] and by its use of hardware-based ordered multicast for intragroup communication. The *coordinator-cohort* method of computation (see, *e.g.*, [4]) is used by the replicas of a service to carry out the RPC. In the coordinator-cohort method, a designated coordinator makes the request and then forwards the reply to the other replicas, the cohorts. This method is an optimization of the pure state-machine approach, providing the same degree of resiliency to fail-stop failures [13] with less computational and communication overhead in the absence of failures.

*The work presented in this paper was done while the author was a visitor at the Vrije Universiteit, Amsterdam, The Netherlands.

0-8186-3770-6/93 $3.00

Initial results in using this RPC package to construct fault-tolerant services, including a simple lock server, have shown it to greatly simplify the construction of fault-tolerant applications. Although this package was developed for Amoeba, the technique behind the protocol is applicable to any system supporting closed groups and ordered multicast communication. As such systems continue to gain in popularity, this RPC protocol will provide a generally applicable model for constructing fault-tolerant services.

2 Replicated RPC for Amoeba

The Amoeba distributed operating system is a microkernel-based operating system for heterogeneous multicomputer systems. Most standard operating system functions, such as file, time-of-day and process management, are provided by servers, with clients interacting with the servers via RPC. Amoeba's provision of functionality through servers accentuates the importance of fault-tolerant server communication.

All processes communicate in Amoeba via the use of ports, which are merely random numbers generated over a large domain. Each server listens to a port; clients communicate with a given server by sending a message to the server's port and waiting for a reply.

In addition to synchronous point-to-point communication, Amoeba also supplies an orthogonal mechanism of asynchronous group communication [9]. Members of a group may broadcast a message to the entire membership of the group. Amoeba groups are *closed*: only members of a group may send messages to the group. Group broadcasts are delivered using a broadcast protocol which guarantees that all members receive messages in the same total order, that the order messages are delivered in respects causality, and that delivery is atomic even in the presence of failures. Ordering is achieved by designating one member of the group to be the *sequencer*; the sequencer determines the order in which messages are delivered.

2.1 The Amoeba Replicated RPC Package

As said in the introduction, the RPC package is a C language library containing the necessary infrastructure to construct fault-tolerant servers and clients. The replication protocol assumes that each program replica functions as a state machine [14], *i.e.*, each replica is deterministic, executing the same sequence of instructions, and therefore each replica will generate the same sequence of responses when given the same sequence of inputs.

Each program, or set of program replicas, defines a *context*. A context may be thought of as simply a high-level process group; we introduce the term to distinguish between the Amoeba process group and this higher-level group. A context is replicated by running multiple replicas of the same program. It is transparent to one context whether another context is replicated or not.

A program calls the `rpc_context` routine to declare to the RPC stub the context name by which the program will be known; the name is valid across the system. Context names are chosen by the programmer.

A program containing server routines declares each routine separately to the RPC stub by calling the `rpc_define_entry` routine. This routine takes three parameters: a pointer to the server function, the name of the function as a string, and the entry number to be associated with the function. When the stub receives an RPC for a given entry number, it will call the associated function.

Programs containing server routines must declare all such routines using `rpc_define_entry` before calling `rpc_context` to ensure that the replica starts in a consistent state. The `rpc_context` routine first determines if any other replicas of this context are already running. If so, the stub will listen to requests on the same port as the existing replicas and will join the same Amoeba group that the existing replicas already belong to.

When joining an existing group, the new replica needs to receive protocol-specific state from the existing replicas. In addition to this protocol state, the application program may also have state that needs to be transferred. For example, a new replica joining a lock manager service will need to acquire the current lock information. To provide a way for application-level state to be transferred, the `rpc_context` routine takes as parameters a receive-state routine and a transfer-state routine. When a new replica joins a group, the transfer-state routine is called in the active replicas to produce a sequence of messages which are transferred to the new replica by calling the new replica's receive-state routine. The state transfer takes place atomically with the new member joining the group.

If a program joins a context for which no other replica is currently active, then an Amoeba port is generated and registered with the Amoeba directory service, and a new Amoeba process group is created for this context.

To invoke a remote procedure on some server context, the context making the RPC request—the client—calls the `ctxt_rpc` routine. This routine takes

four parameters: the context containing the remote procedure, the procedure's entry number and the request message; the reply message is returned as an "out" parameter. If multiple replicas of a context exist, then all client replicas, because of their deterministic nature, make identical `ctxt_rpc` calls.

The stubs perform the RPC using the coordinator-cohort model, with one member of the context group serving as the coordinator and the others functioning as cohorts. The coordinator for the client context makes a normal Amoeba RPC request to the server port. The client coordinator is chosen to be the sequencer of the Amoeba group for performance reasons. However, the coordinator could be chosen by some other means; the protocol merely requires that all the members be able to deterministically chose the same member as coordinator without requiring any explicit communication. For example, in a group system that ranks members by age, the oldest member could be chosen to be the coordinator.

Each program replica of the server context is listening to the same server port; the Amoeba RPC protocol causes the client coordinator's request to be delivered to an arbitrarily[1] chosen replica. The stub of that replica, rather than directly accepting the message, broadcasts it to the server group. Each server replica therefore receives the request, and generates identical reply messages. However, only the coordinator server, the one that received the original RPC, sends back a reply. If there is only one replica functioning, then the broadcast step is skipped and the receiving server replies directly to the RPC caller.

Upon receipt of the reply, the coordinator for the client context broadcasts the reply message to the client context group. This reply message is used by all client replicas as the reply. If there is only one replica functioning for the client context, then as an optimization the broadcast step is skipped.

Figure 1 illustrates the basic flow of events, with the numbers in parentheses indicating the order of communication. The next section presents the complete protocol, including its handling of failures.

3 The Complete Protocol

Figure 2 presents the client side of the protocol. The `ctxt_rpc` routine first attachs a sequence number to the message (line 2) and then checks to see if multiple replicas are running or not (line 3); the case

[1]The Amoeba RPC protocol does an initial broadcast to resolve the address. The first server to respond to that address is the replica to which the message is delivered.

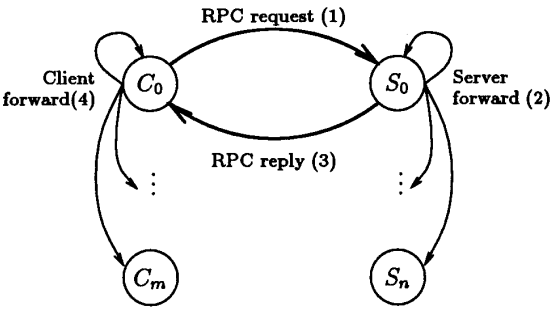

Figure 1: Communication in the basic replicated RPC

where there is only a single replica running is handled specially as an optimization.

In the replicated case, the coordinator makes an RPC to the server port (line 8), broadcasts the result to the group (lines 11 and 13), and returns the message. The Amoeba RPC routine returns `RPC_FAILURE` if the server fails before completing the RPC; should this happen, the client coordinator retries the RPC with another server.

The messages broadcast by the coordinator are received by all members of the group at the entry point `FORWARDED_REPLY` (equivalently, `FORWARDED_ERROR`, line 27). If a stub receives a message which it had sent, then it was the coordinator for the computation and should simply ignore the forwarded message. The cohorts queue the reply messages they receive using `queued_msg_list` until the client program makes the corresponding `ctxt_rpc` call. When a client program running as a cohort calls `ctxt_rpc` and determines that the queue is empty (line 5), the program blocks (line 16), waiting for either a message to be added to the queue or for the coordinator to fail. A single condition variable is used to signal both of these occurrences, with the code for signaling when the coordinator fails being part of the group management routines (not shown).

The failure of the coordinator client can result in a duplicate request being generated. The new coordinator will not know for certain whether or not the previous coordinator had made an RPC request before dying but had not survived long enough to forward on the reply; in this case, the new coordinator's first RPC request will be a duplicate. This is the only possible duplicate request a server may receive, since each result received from a server is reliably broadcast by the coordinator client to its cohorts before the coordinator may make a new request of that server. Moreover,

```
ctxt_rpc(in server, in entry, in message request,
        out message reply)

  next_seqnum ← next_seqnum + 1                          1
  seqnum(request) ← next_seqnum                          2
  if multiple_replicas then                              3
    status ← dequeue(queued_msg_list, reply)             4
    while status = Q_EMPTY                               5
      if coordinator then                                6
        do                                               7
          status ← rpc(server, entry,                    8
                       request, reply)
        while status = RPC_FAILURE                       9
        if status = RPC_NOTFOUND then                    10
          group_send(clients,                            11
                     FORWARDED_ERROR,
                     status)
        else                                             12
          group_send(clients,                            13
                     FORWARDED_REPLY,
                     status, reply)
        return status                                    14
      else                                               15
        wait(reply_avail_OR_coord_failed)                16
        status ← dequeue(queued_msg_list,                17
                         reply)
    end                                                  18
  else                                                   19
    status ← dequeue(queued_msg_list, reply)             20
    if status = Q_EMPTY then                             21
      do                                                 22
        status ← rpc(server, entry,                      23
                     request, reply)
      while status = RPC_FAILURE                         24
  return status                                          25


FORWARDED_REPLY :                                        26
FORWARDED_ERROR :                                        27
  if not i_am_original_sender(msg) then                  28
    enqueue(queued_msg_list, msg)                        29
    signal(reply_available)                              30
```

Figure 2: Replicated RPC protocol, client side

failures of group members are totally ordered with respect to group communication, so a new coordinator will receive all messages sent by the old coordinator before its failure.

We turn now to the server code, Figure 3. The

```
RPC_REQUEST :                                            1
  if multiple_replicas then                              2
    group_send(servers, FORWARDED_REQ,                   3
               entry, request)
  else                                                   4
    if seqnum(request) = last_seqnum[ctxt] then          5
      if last_avail_seqnum[ctxt] < seqnum(request)       6
      then
        await(reply_generated)                           7
    else                                                 8
      last_seqnum[ctxt] = seqnum(request)                9
      last_reply[ctxt] ←                                 10
        server_rpc_routine[entry](request)
      last_avail_seqnum[ctxt] = seqnum(request)          11
      signal(reply_generated)                            12
      rpc_reply(sender(request), last_reply[ctxt])       13


FORWARDED_REQ :                                          14
  if seqnum(request) = last_seqnum[ctxt] then            15
    if original_recipient(request) then                  16
      if last_avail_seqnum[ctxt] < seqnum(request)       17
      then
        await(reply_generated)                           18
      rpc_reply(sender(request), last_reply[ctxt])       19
  else                                                   20
    last_seqnum[ctxt] = seqnum(request)                  21
    last_reply[ctxt] ←                                   22
      server_rpc_routine[entry](request)
    last_avail_seqnum[ctxt] = seqnum(request)            23
    signal(reply_generated)                              24
    if original_recipient(request) then                  25
      rpc_reply(sender(request), last_reply[ctxt])       26
```

Figure 3: Replicated RPC protocol, server side

variable `ctxt` in Figure 3 is set to the context of the client who sent the current message. Upon receipt of a request (line 1), the server broadcasts the request to the entire server group (line 3) if the server is replicated; the case where there is only one replica is again handled specially. Upon receipt of the broadcast, each server first determines if the request is a duplicate (line 15); if the sequence number of the current message matches the last request from the associated context, then the request is a duplicate and so the last reply message sent to that context is returned. It is possible that though the message is a duplicate, the server has not yet finished computing the reply. This condition is determined by comparing the sequence number

to `last_avail_seqnum` (line 17), blocking on a per message condition `reply_generated` if necessary (line 18).

If the request is not a duplicate, then each server replica computes the result (line 22). The server remembers the sequence number of the last request from each context, and the associated reply message (lines 21 and 22). Only the process that received the original request, the coordinator, performs the actual RPC reply (line 26). The code for the unreplicated case (lines 5 to 13) performs the same sequence number handling, but avoids the group broadcast.

There is an asymmetry in the protocol that should be pointed out: clients and servers do not handle messages forwarded to the group in the same manner. When a server thread listening for RPC requests receives a message, it rebroadcasts the message to the entire group. Since different server replicas can receive requests from different clients concurrently, it is necessary for servers to wait to process a request until messages are delivered via the ordered broadcast mechanism. The group broadcast establishes a common ordering of requests for all the servers. An ordering problem does not arise when a client context is awaiting for reply, since there is only one possible sender of that reply. Consequently the client coordinator does not need to wait for its broadcast to be delivered; when the message is delivered to the sender, the sender simply ignores it.

We note that the protocol as presented does not permit multiple threads in a single program to make concurrent requests of the same service. This restriction arises because a server remembers only the last reply sent to a given context; to remove this restriction, servers would have to remember last replies on a per thread basis. Although adding this feature to the protocol would be a straightforward exercise, the use of multiple threads in the state machine approach does require some care. In particular, if threads are preemptively scheduled, then extra work is required to ensure that the behavior of each replica is identical.

3.1 Correctness

The correctness of the RPC protocol relies heavily upon the properties of the Amoeba group communication primitives. These primitives support reliable broadcast of a message by a member of a group to the entire membership of the group, such that the following two properties are satisfied:

ATOMICITY A message m is delivered to all surviving replicas, or to none of them.

ORDER If two processes both receive messages m and m', then they both receive them in the same order.

These properties are satisfied even in the presence of message loss and up to $n-1$ fail-stop process failures, where n is the size of the group.

In order for a fault-tolerant RPC protocol to be correct, its behavior should be equivalent to the failure-free execution of a non fault-tolerant RPC protocol. In the absence of replication and failures, a correct RPC protocol guarantees that an action is performed exactly once. In keeping with our use of the state machine approach, we adopt as our definition of correctness that each surviving replica will receive identical results to identical calls, and that each surviving server will perform the requisite action exactly once.

We establish the following properties:

S1 All surviving server replicas execute each RPC procedure exactly once.

S2 All surviving server replicas see the same sequence of RPC actions and generate the same result.

C1 All surviving client replicas receive the same result to the RPC.

The ATOMICITY property of the ordered broadcast ensures that if any server replica executes an action, then each surviving replica executes that action at least once. The protocol's suppression of duplicates guarantees that each action is performed no more than once. The client's retrial in case of failures implies that as long as a server replica survives, the action is executed at least once. These facts together guarantee Property S1.

The replicated servers only perform an action after it has been broadcast to the server group, and the broadcast, by satisfying the ORDER property, ensures that all replicas receive the request in the same order. By Property S1, each surviving replica receives each request only once. Since the replicas are deterministic, they all generate the same reply. These facts together ensure Property S2.

Property C1 follows in a similar fashion. When a client receives the reply for an RPC, it broadcasts it to the client group. By the broadcast's ATOMICITY property, a broadcast is successful, resulting in the message being delivered locally, if and only if the message is delivered to all surviving replicas. So if any client returns a result, then all surviving clients will return the same result.

As long as one client and one server replica are functioning, the protocol will continue to operate, thereby satisfying the following reliability property:

RELIABILITY Given s server replicas and c client replicas, the protocol guarantees exactly-once semantics in the presence of up to $s-1$ server fail-stop failures and $c-1$ fail-stop client failures.

The RELIABILITY property follows from the reliability guarantees of the Amoeba group communication and properties S1, S2 and C1.

4 Critique of the Protocol Structure

The task of designing and implementing a replicated RPC protocol for Amoeba was greatly simplified by Amoeba's support for reliable group communication. At the same time, the closed group nature of Amoeba determined to a large extent the structure of our replicated RPC protocol. A logical solution to the problem would have been to use the many-to-many approach of [6], in which each client replica directly broadcasts its request to the entire server group. In applying the coordinator-cohort optimization, it would seem most natural to have the client coordinator directly broadcast a request to the server group.

Since Amoeba supports only closed group communication, having direct many-to-many communication between the client replicas and the server replicas requires all client and server replicas to belong to a common group. If we were to require all processes in the system to belong to a single, common group, then this group would quickly become large and unwieldy. Moreover, since Amoeba does not support a multicast to a group subset, each broadcast would be received by the entire group membership, with recipients having to filter out unwanted messages.

Alternatively, we could have a distinct group for each client-server pair, or even a separate group for each pairing of a client replica with a set of server replicas. These approaches require a potentially large number of groups, which would be expensive to maintain. In order for a client to make a request of some service, the client would first have to join the appropriate group, a costly operation. And these approaches still suffer from the problem of group members receiving messages not intended for them.

Despite the structural disadvantages of having servers and clients belong to a common group, such a method can result in decreased latency when compared to the approach presented here. A protocol permitting clients and servers to directly multicast to each other requires only one round of communication, as opposed to the three rounds required by our protocol.

Although these other approaches may offer increased performance, we consider them to be unacceptable as general solutions because of the degree of cooperation they require between clients and servers. The protocol presented in this paper adheres to the principle of client-server independence, with the only interdependence being the unavoidable reliance a client has upon the availability of a service. The replication of a server (or client) is completely transparent to its clients (or servers). Client-server independence is important because it enables the implementation of the one to be changed without effecting the other, which facilitates program development and management. Moreover, maintaining client-server independence provides a degree of failure isolation. Hardware or software faults causing the failure of one program context do not needlessly cause the failure of another context.

5 Performance

The following preliminary performance figures were obtained using Version 5.0 of Amoeba, running on 20.0 MHz MC68030 processors connected by a 10 Mbps Ethernet.

Table 1 presents the observed RPC times for a null RPC and an RPC in which a 2K byte packet is sent and received. These numbers reflect the performance of the RPC package in the absence of failures. The null RPC performance numbers are also shown graphically in Figure 4. The reported times are those for the client coordinator; the times for the client cohorts are virtually identical. The server function did no computation, it merely returned the received packet.

These performance numbers were obtained by performing five trials of between 1250 and 10000 back-to-back RPC's, with the exact number depending on the size of the data packet and the number of processes involved. The results of the five trials were averaged together, with the expected error in the reported numbers being less than 0.5 milliseconds (much less for most trials).

Table 1: Replicated RPC time (milliseconds)

# of Clients	null RPC Number of Servers				2K Byte Packet Number of Servers			
	1	2	3	4	1	2	3	4
1	3.0	7.4	8.2	8.8	10.4	19.0	19.7	20.3
2	6.6	11.1	11.8	12.4	17.4	25.7	26.7	27.1
3	7.4	12.1	12.7	13.5	18.5	27.6	28.0	28.7
4	8.3	12.7	13.5	14.1	19.5	28.2	28.7	29.6

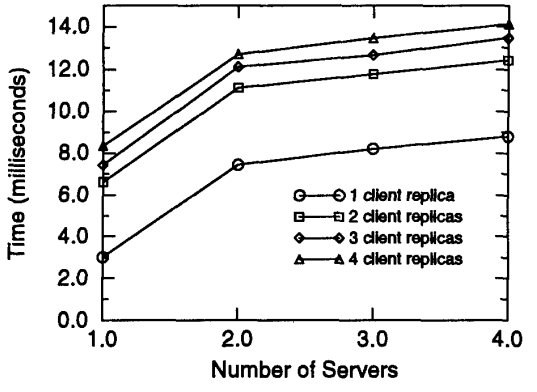

Figure 4: Execution Time for null RPC

The cost of an unreplicated null RPC, 3.0 milliseconds, is twenty percent more than the 2.5 millisecond cost of a "raw" null Amoeba RPC [8]. The raw Amoeba RPC interface is simply a byte-transfer mechanism. The replicated RPC package imposes an extra layer of interface on top of the basic Amoeba RPC, providing a flexible message type at the cost of additional protocol overhead. For sake of comparison, the time for a 2K Amoeba RPC is 6.1 milliseconds.

A replicated RPC with two client replicas and two server replicas costs 3.7 times the cost of an unreplicated RPC. The increase in cost is due to the two ordered group broadcasts which are used to make the RPC fault-tolerant. However, higher degrees of resiliency are achieved with minimal additional expense. The low overhead of adding additional replicas stems from the usage of a hardware-based multicast. Although using additional replicas does provide a higher degree of fault-tolerance with minimal performance degradation, it obviously requires additional computational resources.

The delivery time for an Amoeba group broadcast is not the same for all members of the group. The group communication protocol designates one member of the group to be the sequencer, and group broadcasts originating at the sequencer are delivered faster than messages sent by other group members. The replicated RPC protocol exploits this fact by chosing as the client coordinator the sequencer for the client group.

Amoeba delivers a client coordinator's RPC request to a replicated service to an arbitrary member of the server group, not necessarily the sequencer. For the purpose of obtaining these benchmarks, artificial steps were taken to ensure that requests were delivered to the sequencer, thereby giving the best case performance. Having a request delivered to some server replica other than the sequencer adds approximately 0.8 milliseconds to the execution time for null RPC's, more for those involving more data.

5.1 Recovery Time

A significant advantage of the active replication method over primary-backup schemes [1] is that recovery from a failure is immediate; no recovery procedure needs to be invoked and there is (at least in theory) no adverse impact on performance. This advantage is lost when using a coordinator-cohort scheme. If a coordinator fails, then a new coordinator must be chosen and the last RPC may have to be repeated. The Amoeba group facilities also require the group to be reformed; a group is reformed by using a two-round protocol described in [9, 8] which employs a number of messages linear to the group size.

For a set of server replicas, since each replica has been receiving and acting upon the sequence of requests received up to the point of the failure, each replica has the correct state to carry on. The failure of the server coordinator in the middle of an RPC does require the client to detect the failure and repeat the request. The time required to detect the failure is dependent upon Amoeba system timeout values.

If the client coordinator fails, then the client replicas must likewise reform the client group, and the new client coordinator will have to repeat the last request if its reply value had not been received. The coordinator-cohort method clearly does pay a price for failure recovery as opposed to the pure active replication scheme. However, in the usual case where failures are rare, this price is more than compensated for by the greatly diminished computation required during normal operation.

6 Related Work

The seminal work of [6] describes a replicated RPC protocol for the Circus system. This work uses active replication (the state-machine approach); in particular, each client replica makes an identical request to the set of server replicas, using one-to-many communication. This method of full replication does not employ a hardware multicast; the number of messages required is proportional to the number of server replicas multiplied by the number of client replicas. Moreover, the Circus protocol does not guarantee that the

different server replicas receive requests in the same total order; a higher level protocol must be used to ensure that all replicas serialize requests in the same order. Nevertheless, the simple strategy behind the Circus reliable RPC has served as a starting point for a number of subsequent protocols. For example, the approach described in [11], is basically the same strategy, but extended to provide for flow control and timeouts.

The work of [17] resembles the work described here in that only a single, primary client replica makes the request of a single server replica, which forwards the request to the other server replicas. However, this scheme does not use group-based broadcast communication; instead the different replicas are sequentially chained together, with each replica passing the request (or reply) to a subordinate replica.

A closely related work is the process replication protocol of Manetho [7]. This protocol is like ours in that a designated coordinator for a set of replicas, called a *troupe*, sends messages on behalf of all the troupe members. However, unlike the method outlined in this paper, the coordinator in Manetho multicasts the request to the entire server troupe; the coordinator for the server troupe, called the *leader*, determines the order in which messages are actually accepted by the troupe. Manetho's process replication protocol is not a self-contained library for reliable RPC: the protocol is asynchronous, lacking a reply message, and the protocol guarantees neither reliable nor FIFO inter-troupe communication. Moreover, Manetho requires a complex recovery protocol to be executed after the failure of a leader; the surviving troupe members must elect a new leader who then communicates with the other troupes in order to establish a consistent state. Our protocol does not require a special recovery procedure to be executed, other than the recovery required by the group system, and that recovery only requires intragroup communication.

The ISIS toolkit for distributed programming [3], while not directly supporting replicated RPC, can be very easily used to implement it. ISIS provides both a coordinator-cohort mechanism [4] and a one-to-many synchronous group broadcast; these facilities in conjunction with code to handle duplicates were used in [16] to provide a special-purpose replicated RPC mechanism. The work presented in this paper builds upon the method described in [16] to provide a general purpose reliable RPC for Amoeba. However, the coordinator-cohort scheme used in this paper differs significantly from the coordinator-cohort mechanism provided by ISIS. The ISIS mechanism ensures that if the coordinator fails, a new one is automatically chosen by ISIS. This property requires the members of a group to agree when a coordinator-cohort computation has been completed, which requires an extra multicast. The protocol presented in this paper uses a coordinator-cohort computation at both the client side and the server side. The client multicasts the reply to its cohorts; the same as would be done in ISIS. However, the server coordinator need not multicast a termination message to its cohorts, since the client coordinator will retry the request if the server coordinator fails. Consequently, the protocol presented here requires less messages in the normal, failure-free case, but it may take longer to recover from a failure than ISIS would.

One important difference between the model behind this work and other systems such as ISIS deserves further discussion. ISIS supports *open* group communication, *i.e.*, a process need not be a member of a group in order to send a multicast to the group. The Amoeba distributed operating system provides only closed group communication: only members of a group can send messages to the group. Providing reliable ordered group communication which also permits non-members to send to the group is difficult. ISIS provides this functionality by having the message go to a group coordinator who forwards the message to the entire group. The replicated RPC method presented in this paper also uses a group coordinator and so does not suffer in cost when compared to ISIS; in fact, the performance of our method is much better since Amoeba is faster than ISIS [2, 8].

7 Conclusion

We have presented an implementation of a fault-tolerant RPC library for the Amoeba distributed operating system. The library's RPC protocol is distinguished by its use of closed process groups in conjunction with the coordinator-cohort method of computation. Though designed specifically for Amoeba, the protocol is applicable to any system providing ordered, closed-group communication.

Using the coordinator-cohort optimization of active replication reduces the amount of message traffic and computation in the failure-free case, while still maintaining a high degree of independence between servers and clients. Although a price is paid for using replication, the overall performance is good. Since the method exploits the hardware multicast facilities of Amoeba, using additional replicas provides increased fault-tolerance with negligible degradation of performance.

To summarize, this work is unique in providing a complete package for replicated RPC built upon closed-group communication.

Acknowledgements

This work benefited from discussions with Frans Kaashoek and Leendert van Doorn. Henri Bal, Leendert van Doorn, Frans Kaashoek, Robbert van Renesse, Greg Sharp, Andy Tanenbaum and the anonymous referees provided helpful comments on earlier drafts of this paper.

References

[1] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 627–644, 1976.

[2] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[3] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] Kenneth P. Birman, Thomas A. Joseph, Thomas Raeuchle, and Amr El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.

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

[6] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 63–77. ACM, 1985.

[7] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Replicated distributed processes in Manetho. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 18–27, July 1992.

[8] M. Frans Kaashoek. *Group Communication in Distributed Computer Systems.* PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, December 1992.

[9] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the Eleventh International Conference on Distributed Computer Systems*, pages 222–230. IEEE Computer Society, May 1991.

[10] Luping Liang, Samuel T. Chanson, and Gerald W. Neufeld. Process groups and group communications: Classifications and requirements. *IEEE Computer*, 23(2):56–65, February 1990.

[11] Mark C. Little. *Object Replication in a Distributed System.* PhD thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, England, February 1992.

[12] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba—a distributed operating system for the 1990's. *IEEE Computer*, 23(5):44–53, May 1990.

[13] 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):222–238, August 1983.

[14] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[15] Steve Wilbur and Ben Bacarisse. Building distributed systems with remote procedure call. *Software Engineering Journal*, pages 148–159, September 1987.

[16] Mark D. Wood. *Fault-Tolerant Management of Distributed Applications using the Reactive System Architecture.* PhD thesis, Cornell University, Ithaca, New York, December 1991.

[17] Kiam S. Yap, Pankaj Jalote, and Satish Tripathi. Fault tolerant remote procedure call. In *The Eighth International Conference on Distributed Computing Systems*, pages 48–54. IEEE Computer Society, 1988.