Fault Tolerant Remote Procedure Call

Kiam S. Yap
Pankaj Jalote
Satish Tripathi

Department of Computer Science, and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

ABSTRACT: Remote Procedure Call (RPC) is a desirable primitive for distributed programming. RPC makes building distributed programs easier by providing a high level abstraction and hiding some of the complexities involved. However, if the node on which a remote procedure is executing fails, the caller may be indefinitely suspended. In this paper, we present a scheme that makes an RPC mechanism fault tolerant to hardware failures. Fault tolerance is provided by replicating the procedure at a group of nodes, called a cluster. The copies in a cluster are linearly ordered. A call to a procedure is sent to the first copy in the cluster, and is propagated internally to all other copies. In the event of failures, the first copy in the cluster that has not failed returns the result to the caller. The scheme is transparent to the user and supports nested procedure calls. It has been implemented on a network of Sun Workstations making use of SUN's existing RPC mechanism.

1. INTRODUCTION

Distributed systems are potentially more reliable than centralized ones. Computation tasks when faced with node failures could be completed using services of other nodes. Unfortunately, developing applications for distributed systems is far more difficult than for single processors. A major difficulty in distributed programming is to hide the complications involved in communication, concurrency, transmission errors and node failures [11]. Remote Procedure Call (RPC) has been proposed as a tool that will shield the users from these complications. RPC is a programming language primitive defined "...synchronous language level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel" [15]. The use of shared memory is explicitly excluded.

The procedure call mechanism is well understood and is provided by almost all modern programming languages. RPC extends the procedure call mechanism to the distributed environment by allowing a procedure to reside in another node. When a remote call is invoked, the calling environment (known as the caller or the client) is suspended and the parameters are passed across the network to the remote procedure (known as the callee or the service) where the execution begins. When the callee completes its execution, the result is passed back through the network to the caller, whose execution is then resumed.

With hardware failures, a caller or callee may fail. If a caller fails, its callee is unable to return the result and becomes an orphan procedure. On the other hand, if a callee fails, the caller will be suspended indefinitely, waiting for the result of the call. In this paper, we present a scheme which makes an RPC mechanism fault tolerant to hardware failures. The proposed scheme has been implemented on a network of Sun Workstations using the existing Sun's RPC mechanism [20]. The scheme is transparent to the user.

In the proposed scheme, fault tolerance is provided by having copies of a procedure reside on multiple nodes. Each copy is known as an incarnation. The incarnations are organized in a linear chain. For the ith incarnation, the (i+1)st incarnation forms its backup. A service request is made to the primary incarnation, which is the first copy in the chain that has not failed. The primary callee then propagates the call to its backup, which in turn sends the call to its own backup. In this manner, all the incarnations are invoked. The result of the call is returned to the client by the primary callee. If the primary callee fails, its backup incarnation assumes the role of the primary and replies to the client.

In a nested RPC call, a service invokes a call to another remote procedure. The service acts as a client when making the nested call. Only the primary incarnation will actually make the call; other caller incarnations just wait for the result. The result received by the primary caller is propagated to all other caller incarnations in the cluster. If the primary caller fails to acknowledge the result, another copy of the result will be sent to its backup incarnation.

This work was supported in parts by the NSF grant DCI-8610337, and by the ONR grant N00014-87k-0241. K. S. Yap is now with Information Technology Institute, National Computer Board, 71 Science Park Drive, Singapore 0511

The paper is organized as follows. In the next section, we examine the various schemes that have been proposed to provide reliable distributed computing services. Section 3 introduces the system model used in our protocol. Various terms and functions used by the protocol are defined. In section 4, an overview of the protocol for the fault tolerant RPC is presented. Section 5 discusses the implementation. In section 6, we present some performance measurements.

2. RELATED WORK

The schemes that use replication to support fault tolerance against hardware faults can be classified in two categories, the primary-standby approach and the modular redundancy approach.

In the primary-standby approach one copy of the replicated element is declared as primary and the others as backups. All requests are sent to the primary, which is responsible for providing the service. The backups are passive and their states are periodically updated through checkpoints by the primary so that they are capable of assuming the role of the primary. If the primary fails, a pre-specified backup takes over as the primary. The new primary starts executing from the last checkpoint. Schemes for proper recovery of messages since the last checkpoint are also needed. The primary-standby approach has been used in [1, 5, 6, 21]. The major disadvantage of this approach is that it requires a considerable amount of system support for checkpointing, and often also requires complex message recovery schemes.

In the modular redundancy approach there is no distinction between the replicated elements. Requests are sent to and performed by all the replicated elements. The caller usually waits for all the results to return, and may emply voting to decide the correct result. Schemes based on this approach have been used in [2, 4, 7, 8, 9, 23]. The major disadvantage of such schemes is the high message overhead they often require. For example, in a system for remote operations, like the one described in [7], if there are m requesters and n copies of the requested service, the number of messages needed to complete the request is $O(m*n)$.

The scheme proposed in this paper employs a combination of the modular redundancy approach and the primary-standby approach, in an effort to exploit the advantages of both the approaches. It is a form of modular redundancy in which all the replicated modules execute each call concurrently. However, unlike most of the protocols in this class, the caller is only required to make one call, which is transmitted to others. If there are m callers and n callees then this approach requires $O(n+m)$ messages to service a call. Furthermore, by having all the backups active concurrently, our scheme avoids the expensive operation of checkpointing. The proposed protocol, unlike many schemes [3, 6, 17], does not require any specialized hardware support.

3. SYSTEM MODEL

The words crash and failure are used interchangeably in the paper to represent the failure of a node. The failures discussed here are those in which the failing element simply halts and does not produce spurious side effects. This form of failure is known as sane failures [16]. The fail-stop processor [18] has this simple failure mode operating characteristic.

Each incarnation of the procedure is identical and resides on a different node. The group of replicated incarnations constitute a cluster. There are two types of clusters. The cluster that services a call is the callee cluster. The cluster that makes the call is the caller cluster. The user process that invokes an RPC call is not replicated and is a special instance of a caller cluster with only one incarnation. The interaction of a user process with a cluster is shown in Figure 1. In a nested RPC call, a cluster may function both as a callee and a caller. It is a callee cluster with respect to the cluster that it receives the call from, and a caller cluster with respect to the cluster it sends a call to.

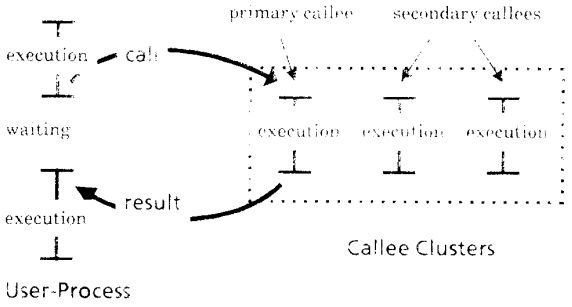

Figure 1. User-process and Cluster

There are two types of incarnations, primary and secondary. When a cluster is first created, an incarnation is designated as primary. The primary incarnation handles the communications between clusters. The primary of the caller cluster (primary caller) sends the RPC call. It also receives the result of the call from the callee cluster. In the callee cluster, the incarnation that is designated to receive RPC calls is the primary callee. It sends the result of a call back to the caller. All other incarnations in a cluster are the secondary incarnations. If the primary fails, a secondary incarnation assumes its function and becomes the primary.

Within a cluster, the incarnations are ordered according to their respective precedence in assuming the role of the primary. An incarnation that has precedence over another in becoming the primary is the superior; the other is its subordinate. A secondary incarnation becomes a primary only when all its superiors have failed.

A secondary incarnation receives a call or result of a call directly from its immediate superior. It will then send the same call or result to its immediate subordinate. The hierarchy amongst the incarnations in a cluster is determined when the service is created. We assume that each cluster is deterministic. A cluster is deterministic if each of its incarnations, on receiving the same call, produces the same result and has the same side effects.

4. SCHEME FOR FAULT TOLERANT RPC

In this section, we present an outline of our scheme. The details and proof of correctness may be found in [23]. We discuss how the scheme functions both when there are no failures, and under failure conditions.

There are 4 types of messages in the proposed scheme. A *call* message invokes an RPC call. A *result* message contains the return values of a RPC call. A *done* message is sent to inform a secondary callee that the call is completed. These 3 types of messages require their recipients to acknowledge (with an *ack* message). If no acknowledgement is received within a defined time period, the sender will check the status of its receiver. If it has failed, the same message will be sent to the receiver's immediate subordinate. This simple discipline helps to ensure that every active incarnation in the cluster will receive the same message in the event of failures.

4.1. Execution without Failures

In our scheme, the secondary incarnations play an active role. All the incarnations in the callee cluster execute a call, and every incarnation in a caller cluster receives a copy of the result. However, if the primary callee does not fail during a call, the caller makes a single call and only one copy of the result is sent to the caller cluster. All communications between the clusters are carried out by their respective primary incarnations. A secondary incarnation will only interact with external processes if all its superior incarnations have failed and it has assumed the role of the primary.

The incarnations in a cluster are ordered sequentially. Figure 2 shows the sequence of messages that are transmitted in an RPC call from a 2–caller cluster to a 3–callee cluster. On receiving a call, the primary callee sends the same call to its immediate subordinate. The incarnation, in turn, sends the call to its immediate subordinate. On receiving an acknowledgement to the call it has sent, a callee incarnation performs the requested service. A secondary callee on completing a call waits for a go ahead message from its immediate superior. The primary callee will send its result to the primary caller and wait for an acknowledgement. On receiving the acknowledgement, the primary sends a done message to inform its immediate subordinate that the call is completed. This message is propagated to all other secondary callees.

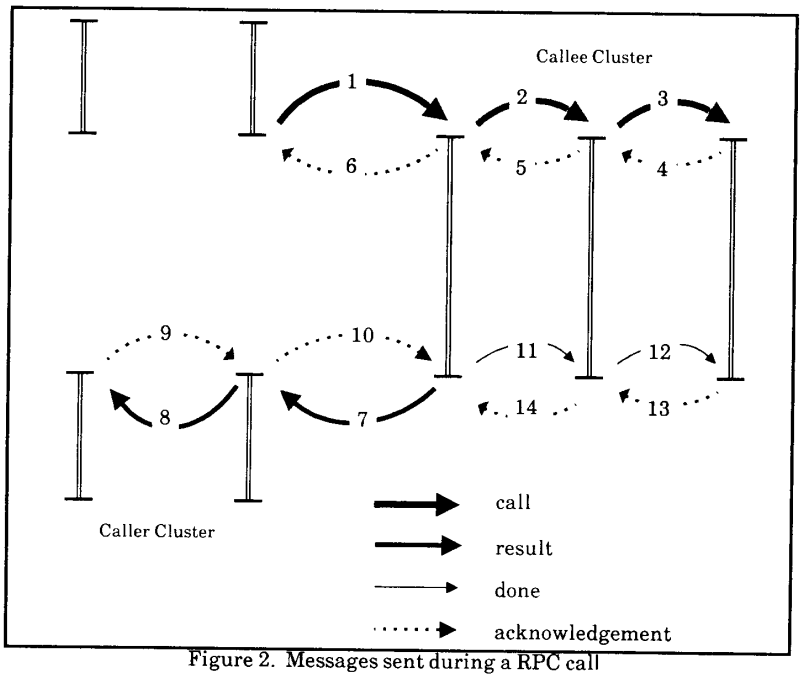

Figure 2. Messages sent during a RPC call

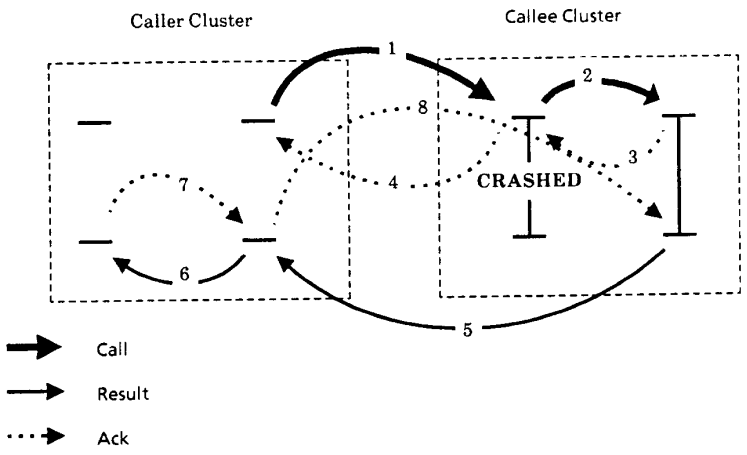

Figure 3. Failure of the Primary Callee

In the caller cluster, the primary, after sending an RPC call, waits for the result. The secondary callers will also wait for the result. However, they are not required to send any call messages. Each secondary caller receives the result from its immediate superior.

4.2. Execution with Failures

After sending an RPC call to the primary callee, the caller waits for an acknowledgement. If, before the acknowledgement is received, the caller detects that the primary callee has failed, the same message is sent to the primary callee's backup. If the backup callee has already received a similar call from the old primary, it acknowledges the message but takes no other actions. If the primary crashes after sending the acknowledgment, the caller takes no action. The immediate subordinate of the crashed primary will take over its role and send the result to the caller. Figure 3 shows the sequence of messages that are sent when the primary of a 2–callee cluster fails before sending the result to the 2–caller cluster.

If the primary callee fails after it has sent its result but before it has sent the done message to its secondary, the caller may receive more than one copy of the same result. The new primary is not aware that the failed primary has already sent the result and will send another copy. The primary caller acknowledges the duplicate result but takes no other actions.

If a secondary callee incarnation fails before it acknowledges a call message, on detecting the failure, its immediate superior will send the same call to the failed incarnation's immediate subordinate. Between sending the call message and the done message, an incarnation need not be aware of its immediate subordinate's status. If a callee incarnation fails to acknowledge the done message, the message will be sent to its immediate subordinate.

If a primary caller fails before it has received and propagated the result of a call, its immediate subordinate will assume its role and send the same call message. The callee cluster may receive duplicates of the same message. The duplicate requests are discarded, though the result of the call is now sent to the new caller. A caller incarnation is not aware of the status of its subordinates before propagating the result of a call. If a caller incarnation fails to acknowledge the receipt of the RPC result, the same message is sent to its immediate subordinate (see figure 4).

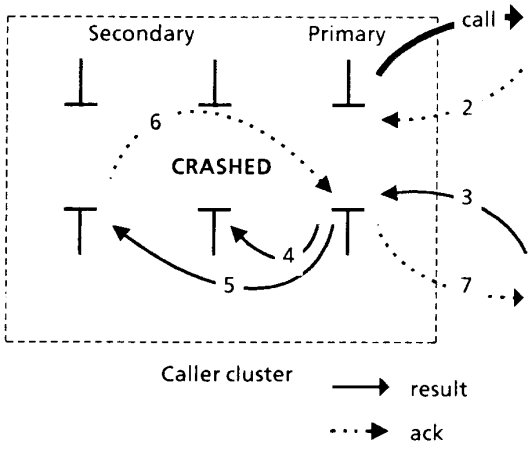

Figure 4. Failure of the Secondary Caller

5. IMPLEMENTATION

The proposed fault tolerant scheme is implemented over the Sun's RPC mechanism. The Sun's RPC, has a fully functional remote procedure call mechanism which allows static data and nested calls. In this section, we briefly describe some aspects of our implementation. The details could be found in [23]. Our implementation is on a network of more than 20 Sun Workstations.

5.1. Registering a Cluster

During an RPC call, the caller cluster needs information about the callee cluster and vice versa. For example, a primary callee needs to know which secondary caller it should send its result to when the primary caller fails. In our implementation this is done by requiring a cluster to register when it is ready to accept requests. All incarnations in a cluster are registered in the same manner. The codes for the primary and secondary incarnations are indistinguishable; the status of an incarnation is set during compilation.

On registration, the required information is distributed to all the other nodes in the network. This is done by adding the information about a new cluster to a file and then distributing it. The file contains information about all the clusters that are ready to accept requests. Each cluster is represented by its program identification (program–ID), which uniquely identifies a remote service. For each cluster, the nodes on which its incarnations reside are stored in the order of their precedence for becoming the primary.

5.2. Detecting Failures

A mechanism to detect the failure of an incarnation is required by our scheme. The secondary callee needs to ascertain that its immediate superior has failed before it may assume its role. A caller will send a call message to another callee only when it is certain that the primary callee has failed. A process interested in the status of another process at a remote node sends a query to the node. If the reply to the query is not obtained within a timeout period, the process is assumed to have failed.

In our implementation, we utilized the RPC mechanism itself to query the status of a remote process. A dummy RPC call which returns the status of a registered procedure is used. The dummy call goes through the mechanism of making an RPC call; however, no procedure is executed. If a node has failed, the dummy call will timeout. To query the status of a caller incarnation which is in a different cluster, its program–ID is required. The program–ID of a caller is transmitted to a callee as part of the call message" However, to examine the status of an incarnation in the same cluster, no additional information is required.

A secondary callee incarnation needs the program–ID of the caller and all of its superiors. The information is necessary if the secondary is to be capable of assuming the role of any of its superiors and able to handle duplicate messages correctly. This information is stored in a stack. Each incarnation places the information concerning itself on the stack before transmitting the call message to its subordinate. In other words, to the incarnation receiving a stack, the top of the stack contains information about its immediate superior and subsequent lower levels contains information about "higher level" superiors. The information about the caller is at the bottom of the stack.

The height of the stack is an indication of the relative position of the incarnation with respect to becoming the primary of the cluster. When a immediate superior is detected to have failed, the stack is popped and information of the new immediate superior is again on top of the stack. When the height is one, the incarnation is the primary of the cluster.

5.3. Handling Duplicates

As we have seen, duplicate messages may be sent by our scheme in the event of failures. A mechanism that detects and handles duplicates is required. Duplicate handling usually involves generating and assigning appropriate sequence numbers to message [19]. The two best known techniques for generating network–wide unique sequence numbers are the circulating token method [12] and Lamport's synchronized clock approach [10].

In our implementation, a unique sequence number over the entire network system is generated. The sequence numbers are generated by using the unique identity of the registered program and a counter. Each registered RPC cluster is identified by its unique program–ID, which is composed of a program number and a version number. The counter is incremented each time the program makes an RPC call. Different calls made by the same RPC are distinguished by the count value. This implementation avoids the complexity involved in detecting and reinserting lost tokens in LeLann's method, and eliminates the need for synchronizing the local clocks.

Each registered service stores the sequence numbers of all the caller programs from which it has received calls. Only the largest sequence number (the largest count value) of each caller cluster is kept. The information is stored in a linked list called active list. When a message is received, its sequence number is checked against that of the active list. If the program–ID is not found, a new element is added to the list, otherwise, count values are compared. If the count value from active list is larger or equal to that of the incoming message, the message is a duplicate. If the message is not a duplicate, the count value in active list is updated.

This simple algorithm works well for nested calls from one registered cluster to another. In the instances

in which a call is made by a user–process (which has no unique program–ID), the transaction identity is used to form the sequence number. In the Sun's RPC, each remote call is given a transaction identification created by the bitwise exclusive–or on the process identification number, the local time and a randomly generated number. With this scheme it is possible, but very improbable, that another user–process on another node may generate the same identification. This identification may be made unique by including a field which stores the node identification number.

The sequence number of a call from a user–process is always added to the active list. Since the number is unique for every call, active list may grow infinitely large. The problem may be solved by eliminating the sequence number from the list when a call is completed. The list may also be trimmed by invoking a program that searches and eliminates such numbers when the procedure is idle.

6. PERFORMANCE MEASUREMENTS

The performance measurements presented here were conducted on Sun–3/52M Workstations connected on a 10Mbps ethernet. The measurements given are the average taken from a series of experiments. In the experiment, a call with an integer parameter is made to a remote procedure that performs one add instruction and then returns. The input value is added to a static variable which retains its value between calls. The performance parameter that we are interested in is the response time of a call. This is the time elapsed between making a call and receiving the result of the call. The Unix system call, *gettimeofday*, is used to obtain this value. The time of the day is taken before the call is made and after the completion of the call. The difference is the response time.

Table 1 shows the response time of an RPC call under varying degrees of replication. This is the effect of adding redundancy on the response time when no failures occur. There is a linear relationship between the number of secondary callees and the response time. Adding one more secondary callee increases the response time by about 0.291 second. The additional time is used in sending a call message to the secondary and waiting for it to be acknowledged.

Number of Secondary Callees	Response time (seconds)
0	0.291
1	0.583
2	0.874
3	1.164

Table 1: Response time with varying degree of replication

In case of a failure during an RPC call, the response time depends on where the failure occurs. The response time of a call to a 2–callee cluster with different failure scenarios is shown in Table 2. The failure of the RPC service is simulated by terminating the service and unregistering the service. When a client queries the server, it will be informed that the service is not available. In the case of an actual node failure, the query will timeout which may increase the response time.

Failure Scenario	Response time (seconds)
No failure	0.583
Primary fails (after ack, 1 sec wait time)	2.720
Secondary fails (after ack)	0.583

Table 2: Response time of a 2–callee cluster with failure

When a primary callee fails after it has sent an acknowledgement to the call message, it is the task of the secondary callee to assume the function of its primary and return the result of the call. The response time in this class of failures is dependent on the time the secondary will wait for the done message before it checks the status of its immediate superior. The time period may be set by the user. The longer the secondary waits before checking, the longer will be the response time.

It may seem beneficial for a secondary callee to check the status of its immediate superior right after it has finished executing the procedure. However, this may lead to longer response times when there are no failures. The done message may arrive at the secondary callee while it is checking the status of its immediate superior. The callee cluster will therefore complete a call in a longer period of time. Further CPU time is wasted by secondary callee continuously checking its immediate superior's status and by its immediate superior in handling the queries. The optimal waiting time for a secondary callee is dependent on many factors including, the number of incarnations in the caller cluster, the number of superiors it has, and the network delay. The greater the number of caller incarnations, the longer the primary callee has to wait for the acknowledgement to the result it has sent. The secondary callee will not receive the done message before all its superiors have done so.

The failure of a secondary callee after it has acknowledged the call from its primary does not affect the response time. The primary is only aware that the secondary has failed when it attempts to send the done message to it. At that time the caller has already completed the call.

7. CONCLUSION

Remote Procedure Call is a desirable primitive for distributed programming. In this paper, we have described a scheme which makes an RPC mechanism fault tolerant to hardware failures. Fault tolerance is provided by having replicated copies of a service active at distinct nodes. The group of replicated services is known as a cluster. Each replicated service of a cluster is called an incarnation. When a call is invoked, all the incarnations will execute concurrently. However, a client does not have to invoke every incarnation. When there are no failures, a client will only send one call message. The call is then propagated from one callee incarnation to another. In the event of failures, one of the backups which has not failed will return the results to the caller.

The proposed scheme has been implemented over a network of Sun Workstations, utilizing the Sun's RPC mechanism. The Sun's RPC does not support recursive calls, but we believe that the proposed scheme may easily be modified to handle them. We are now looking into ways that may enhance the performance of our implementation. We are also looking into effective ways to reconfigure a cluster when a crashed incarnation recovers.

REFERENCES

[1] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources", In: *Proc. 2nd Int. Conf. on Soft. Engg.*, San Franscisco, CA, Oct 1976, pp. 562–570.

[2] A. Avizienis, "The N–Version Approach to Fault–Tolerant Software", *IEEE Transaction on Software Engineering*, Vol. SE–11, NO. 12, Dec. 1985, pp. 1491–1501.

[3] J. S. Banino and J.C. Fabre, "Distributed Coupled Actors: A CHORUS Proposal for Reliability", *Proc. 3rd International Conference on Distributed Computing Systems*, Oct. 1982, pp. 128–134.

[4] J. S. Banino et al., "Some Fault–Tolerant Aspects of the CHORUS Distributed System", *Proc. 5th International Conference on Distributed Computing Systems*, May 1985, pp. 530–437.

[5] K. P. Birman, T. A. Joseph, T. Raeuchle and A.E. Abbadi, "Implementing Fault–Tolerant Distributed Objects", *Proc 4th Symp. on Reliability in Distributed Software and Database*, 1984, pp. 124–133.

[6] A. Borg, J. Baumbach and S. Glazer, "A Message System Supporting Fault Tolerance", *9th ACM Symp. on Operating System Principles*, Operating System Review, Vol. 17:5, Oct. 1983, pp. 90–99.

[7] E. C. Cooper, "Replicated Distributed Programs", *Proc. 10th Symp. on Operating Systems Principles*, 1985, pp. 63–78.

[8] P. Gunningberg, "Voting and redundancy management implemented by protocols in distributed systems", *Digest of Papers, FTCS- 13*, June 1983, pp. 182–185.

[9] P. Jalote, "Resilient objects in broadcast networks", *IEEE Transactions on Software Engineering* (to appear).

[10] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communication of the ACM*, Vol. 21, No. 7, July 1978, pp. 558–565.

[11] B. Lampson, "Atomic Transactions" in Davis D.W. et al., editors, *Distributed Systems: Architecture and Implementations: An Advanced Course*, Springer–Verlag, 1985, 3rd printing, pp. 246–264.

[12] G. LeLann, "Distributed Systems: Towards a Formal Approach", *Information Processing 77*, Amsterdam, The Netherlands: North–Holland, 1977, pp. 155–160.

[13] G. LeLann, "Motivations, Objectives and Characterization of Distributed Systems", in Davies et al. editors, *Distributed Systems: Architecture and Implementation: An Advanced Course*, Springer–Verlag, 1985, 3rd printing, pp. 1–8.

[14] K. J. Lin, "Atomic Remote Procedure Call", Ph.D Thesis, University of Maryland, College Park, MD., TR–1552, Aug 1985.

[15] B. Nelson, "Remote Procedure Call", Ph.D. Thesis, Carnegie–Mellon University, Pittsburgh, PA., CMU–CD–81–119.

[16] F. Pittelli, "Database Processing with Triple Modular Redundancy", *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, 1986, pp. 95–103.

[17] M. L. Powell and D. L. Presotto, "PUBLISHING: A Reliable Broadcast Communication Mechanism", *9th ACM Symp. on Operating System Principles*, Operating System Review, Vol. 17:5, Oct. 1983, pp. 100–109.

[18] R. D. Schlichting and F. B. Schneider, "Fail–Stop Processors: An Approach to Designing Fault–Tolerant Computing Systems", *ACM Transactions on Computer Systems*, Aug. 83, pp. 222–238.

[19] S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism", *IEEE Transactions on Computers*, Vol. C–31, No. 7, July 1982, pp. 692–697.

[20] "Remote Procedure Call Reference Manual", Sun Microsystems, Inc., Mountain View, CA, Revision A, May 1985.

[21] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS distributed operating system", In: *Proc. of the 9th symp. on operating systems principles*, Bretton Woods, NH, Oct. 1983, pp. 49–70.

[22] J. Wensley et. al., "SIFT: design and analysis of a fault tolerant computer for aircraft control", *Proc. IEEE*, vol. 60, Oct. 1978, pp. 1240–1245.

[23] K. S. Yap, "Implementing Fault Tolerant Remote Procedure Call", M.S. Thesis, Dept. of Computer Science, University of Maryland, College Park, MD.