FAULT TOLERANCE USING GROUP COMMUNICATION

M. Frans Kaashoek *Andrew S. Tanenbaum*

Vrije Universiteit, Amsterdam, The Netherlands
(email: kaashoek@cs.vu.nl)

ABSTRACT

We propose group communication as an efficient mechanism to support fault tolerance. Our approach is based on an efficient reliable broadcast protocol that requires on average only two messages per broadcast. To illustrate our approach we will describe how the task bag model can be made fault-tolerant using group communication.

1. INTRODUCTION

A common technique used to achieve fault tolerance is to replicate data. To keep replicated data consistent, group communication can be used, in which an arbitrary one of the n members can send a message to the whole group. Although hardware supporting broadcast communication exists, most distributed operating systems do not provide group communication. An important reason for doing so is that reliable group communication is thought to be slow.

In this paper we propose group communication as an efficient mechanism for implementing fault-tolerant and parallel applications on a distributed system. Our approach differs from earlier systems in several aspects. Our protocol for reliable group communication is very efficient: a reliable multicast requires in average only two messages. The semantics of the group primitives are simple and easy to understand, but are still powerful. Our primitive for sending messages, for example, guarantees global ordering of all broadcast messages. Finally, we have a prototype implementation, which has been used for running both fault-tolerant parallel applications.

2. GROUP COMMUNICATION

We are concerned with distributed systems that consist of multiprogrammed nodes that can communicate with each other over a LAN. The protocol to be described runs inside the kernel. Any member of a group can, at any instant, decide to send a broadcast message to its group. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communications, lost packets, finite buffers, and node failures. We assume, however, that byzantine failures (a process sends spurious or contradictory messages) do not appear. If a LAN supports broadcast or multicast, the kernel will use this facility, but it is not essential for the correct functioning of the protocol.

The group communication primitives are given in Fig 1.

3. PROTOCOL FOR COMMUNICATION FAILURES

The basic protocol works as follows. When a group member calls *SendToGroup( )* to send a message, M, it hands the message to the kernel and blocks. The kernel encapsulates M in an ordinary point-to-point message and sends it to a special member called the *sequencer*. The sequencer contains the same code as all other members. The only difference is that a flag tells it to process messages differently.

When the sequencer receives the point-to-point message containing M, it allocates the next sequence number, s and broadcasts a packet containing M and s. Thus all broadcasts are issued from the same node, by the sequencer. Assuming that no packets are lost, it is easy to see that if two members simultaneously want to broadcast, one of them will reach the sequencer first and its message will be broadcast to all the other

CreateGroup(port, resilience-degree) → member-id	Create a group. A process specifies the group and how many member failures must be tolerated without loss of any message.
JoinGroup(port) → member-id	Make a process member.
LeaveGroup(port) → void	Leave a group. The last member leaving causes the group to vanish.
SendToGroup(hdr, buf, bufsize) → mess-number	Atomically send a message to the group. All messages are globally ordered.
ReceiveFromGroup(hdr, buf, bufsize, more) → mess-number	Block until a message arrives. *More* tells if the system has buffered any other messages.
ResetGroup(port, number-of-members) → boolean	Recover from processor failure. If the new group has more than *number-of-members* members, it succeeds.

Fig. 1. Primitives to manage a group and to communicate within a group. A group is addressed by a *port*.

nodes first. Only when that broadcast has been completed will the other broadcast be started. The sequencer provides a global ordering in time. In this way, we can easily guarantee the indivisibility of broadcasting.

When the kernel that has sent M, itself receives the message from the network, it knows that its broadcast has been successful. It unblocks the member that called *SendToGroup( )* and returns s to the member.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast packet eventually arrives, the kernel will immediately notice a gap in the sequence numbers. If it was expecting s next, and it got $s + 1$, it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for copies of the missing message (or messages, if several have been missed). To be able to reply to such requests the sequencer stores broadcast messages in a *history buffer*. The sequencer sends point-to-point the missing messages to the process requesting them. (The other members also keep a history buffer, to be able to recover from sequencer failures, as we will see in the next section.)

As a practical matter, a kernel has only a finite amount of space in its history buffer, so it cannot store broadcast messages forever. However, if it could somehow discover that all machines have received broadcasts up to and including k, it could then purge the first k broadcast messages from the history buffer.

The protocol has several ways of letting a kernel discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message. This information is also included in the message from the sequencer to the other kernels. In this way, a kernel can maintain a table, indexed by member number, showing that node i has received all broadcast messages 0 up to T_i, and perhaps more. At any instant, a kernel can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, a kernel knows that everyone has received broadcasts 0 through 6, so they can be deleted from the history buffer.

If a node does not need to do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval, Δt, send the sequencer a special packet acknowledging all received broadcasts. The sequencer can also request this information when it runs out of space.

In short, to do a broadcast, an application process sends the data to the sequencer, which gives it a sequence number and broadcasts it. There are no separate acknowledgement packets, but all messages to the sequencer carry piggybacked acknowledgements. When a node receives an out-of-sequence broadcast, it buffers the broadcast temporarily, and asks the sequencer for the missing broadcasts. Since broadcasts are expected to be common—many per second—the only effect that a missed broadcast has is causing some application process to get behind by a few tens of milliseconds once in a while, hardly a serious problem. So, on average, a reliable broadcast costs only 2 messages, one point-to-point and one broadcast.

4. PROTOCOL FOR PROCESSOR FAILURES

Processor failures are detected by a kernel, when it tries to reach another kernel to deliver a message. If after a certain number of trials, a kernel has not responded, the member running on that kernel is assumed to be dead. In reality a kernel may be busy doing something else and respond a week later. This does not affect, however, the correct functioning of the group primitives.

After a member failure is detected, all further group primitives fail (return an error status). Any surviving member may call *ResetGroup( )* to recover from a member failure. *ResetGroup( )* tries to make a new group from the old group. It does this in two phases. In the first phase it invites all other members to cooperate in creating a new group. Every surviving member responds with the highest sequence number it has seen so far. After this phase, the initiator knows who is still alive and it elects the living member that has responded with highest sequence number as the new sequencer. In the second phase it sends this information to all members in the new group. When a member receives this message, it sends back an acknowledgement and enters normal operation again. The initiator goes back to normal operation after it has received all acknowledgements.

The protocol described so far recovers from member failures, but does not guarantee that all surviving members receive all messages that have been sent. If, for example, a member and the sequencer die directly after a call to *SendToGroup( )*, it can happen that all other members miss the message. During recovery this message cannot be retrieved. To guarantee that this cannot happen *SendToGroup( )* returns only after the kernel knows that at least *resilience-degree* kernels have received the message. A kernel can discover this information using the same scheme as for the history buffer. Thus, an increase in fault-tolerance is paid for by a decrease in performance. The trade-off chosen is up to the user.

5. ACHIEVING FAULT TOLERANCE

The protocol described in the previous sections has been used for implementing parallel applications [Kaashoek et al. 1989; Bal et al. 1990]. In this section we will describe how our primitives can be used to obtain fault tolerant parallel applications. Other applications like databases can also be implemented using our primitives, but are not the topic of this paper.

Many parallel applications can be structured using the task bag model. In this model processes repeatedly get a task from a task bag and perform the task. They may generate new tasks while working on a task. An application is finished when all processes are idle and the task bag is empty.

The task bag can be made fault-tolerant by replicating the task bag at each member. When a member wants to add a task to the bag, it uses *SendToGroup( )* to broadcast the task to all members. When a member receives a new task, it adds the new task to its copy of the task bag. A member broadcasts a *gettask*-message to get a task from the task bag. When a member receives a *gettask*-message, it assigns to the task the member identity of the sender, but does not remove the task from the bag. When the same member asks again for a new task, its previous task can be removed from the task bag. As all broadcasts are indivisible and reliable, the task bags at each member are always identical.

When after a processor failure *ResetGroup( )* returns successfully, each surviving member goes through its copy of the task bag and reclaims all the tasks that have been given away to a failed member and that have not been finished yet. When a new get-work message is received, the reclaimed task can be allocated again.

The implementation is efficient: it requires only one reliable broadcast per operation on the task bag. The delay per reliable broadcast is short, as a group with *resilience-degree* 0 is enough to achieve fault-tolerance. Lost messages from a failed processor will be regenerated, when a reclaimed task is executed again. So, an operation on a task bag has the same performance as one RPC. Furthermore, the implementation can be easily extended to incorporate different task distribution algorithms. For example, a task stack or task queue can be implemented without requiring additional messages.

6. DISCUSSION

Much research in group communication has been done. The two closest approaches to our work are Isis and a family of protocols designed by Chang and Maxemchuk [Birman and Joseph 1987; Chang and Maxemchuk 1984]. We will discuss each in turn.

In the Isis toolkit, an application can choose among different primitives to broadcast a message. Each primitive has different semantics and performance. Our *SendToGroup( )* has similar semantics to ABCAST, but our performance is much better. ABCAST uses a two-phase protocol that requires $2 \times n$ messages. To achieve better performance the designers have chosen to weaken the ordering semantics. The resulting primitive, called CBCAST, requires the same number of messages as *SendToGroup( )*, but only provides a weak ordering, whereas our provides full global ordering. In addition, to perform a CBCAST, each machine in the system has to be involved, even nodes that do not belong to the group. Furthermore, applications that need not to be fault-tolerant have to pay a performance penalty for using the fault-tolerant broadcast primitives. The Isis toolkit, however, also guarantees ordering of broadcasts in the presence of overlapping groups.

Chang and Maxemchuk (CM) describe a family of protocols. The protocols differ mainly in the degree of fault-tolerance that they provide. Like our protocol, the CM protocol depends also on a central node, the token site, for ordering messages. However, on each acknowledgement another node takes over the role of token site. Depending on the system utilization, the transfer of the token site on each acknowledgement can cost one extra control message. Thus their protocol requires 2 to 3 messages per broadcast, whereas ours requires only 2 in the normal case. In addition, in the CM protocol all messages are broadcast, whereas our protocol uses point-to-point messages whenever possible, reducing interrupts and context switches at each node. This is important, because the efficiency of the protocol is not determined by the transmission time, but mainly by the processing time at the nodes. In their scheme, each broadcast causes at least $2(n-1)$ interrupts, ours only n. Finally, CM do not describe how they do group management and the semantics of their primitives.

In summary, we have proposed group communication as a mechanism for implementing both fault-tolerant and parallel applications. This is feasible, because the protocol for reliable indivisible broadcast is cheap in terms of number of messages and the delay seen by the sender. In addition, the semantics of our primitives are simple and easy to understand, but are still powerful. This has been illustrated by a description of a fault-tolerant implementation of the task bag model.

REFERENCES

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Experience with Distributed Programming in Orca," *IEEE CS Int. Conf. on Computer Languages*, pp. 79-89, New Orleans, Louisiana, Mar. 1990.

Birman, K. P. and Joseph, T. A., "Reliable Communication in the Presence of Failures," *ACM Trans. on Comp. Syst.*, Vol. 5, No. 1, pp. 47-76, Feb. 1987.

Chang, J. and Maxemchuk, N. F., "Reliable Broadcast Protocols," *ACM Trans. on Comp. Syst.*, Vol. 2, No. 3, pp. 251-273, Aug. 1984.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., "An Efficient Reliable Broadcast Protocol," *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989.