Broadcasts: A Paradigm for Distributed Programs*

Fred B. Schneider

TR 80-440

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

October 1980

*This research was supported in part by the National Science Foundation under Grant MCS 76 22360

in Proc. Workshop on Fundamental Issues in Distributed Computing, Fallbrook, California, December 1980

Broadcasts: A Paradigm for Distributed Programs

Fred B. Schneider
Department of Computer Science
Cornell University
Ithaca, New York 14853

1. Introduction

It has been argued that by distributing a computation over a number of processors it is possible to construct systems that are immune to various types of hardware failures, have high throughput and exhibit incremental growth capabilities. Unfortunately, constructing systems that realize these goals is by no means simple. Often, a particular task can be decomposed into disjoint communicating processes in many different ways; some decompositions will satisfy the above criteria, while others will not.

Systems with the following characteristics will, in general, exhibit the above properties. First, processes should be loosely coupled in terms of communications bandwidth and synchronization. Low interprocess communications bandwidth allows use of a network for interprocess communication. Furthermore, use of asynchronous communications protocols seems appropriate because synchronous communications protocols may decrease the overall throughput of a system due to reduced potential for parallelism. Secondly, the system should employ "decentralized" control as much as possible. There should be no entities whose failure or performance is critical to correct operation of the system. Otherwise, the fault tolerance and incremental growth capabilities of the system are threatened, because such entities constitute critical resources. Their saturation or failure would have catastrophic, system-wide implications.

Writing programs within this framework is difficult -- perhaps an order of magnitude more difficult than writing concurrent programs. Due to delays in message delivery, any state information a process receives through messages necessarily reflects a past state of the sending process. Thus, use of asynchronous protocols means that no single process can have complete knowledge of the entire state of the system. In

addition, prohibiting any sort of centralized control precludes use of a single arbiter or process to maintain consistent, albeit approximate, views of system states. Finally, processes must be able to cope with failures in other processes. Notice that all of these difficulties stem from the absence of shared memory (which could provide a consistent view of the system state to all processes) and the presence of more than one processor (so that failure of a processor does not mean failure of all processors).

A distributed program consists of a collection of processes that communicate solely by use of an asynchronous, buffered communications network. Our investigations have been concerned with distributed programs that implement distributed control regimes, since that seems to be the most promising way to support fault tolerance and incremental growth capabilities. In particular, by using broadcast communication we have developed (provably correct) distributed programs that realize the above goals. Also, we have studied some of the problems associated with implementing broadcast communications protocols in computer networks. A family of broadcast protocols has been devised that allows for rapid dissemination and guaranteed delivery of a message to a collection of processors despite processor and link failures. In addition, we have developed solutions to some of the protection problems unique to broadcast communication.

2. Using Broadcasts

Synchronization of processes is among the most difficult problems that must be faced when writing a distributed program. For purposes of synchronization, execution of a process can be viewed as a sequence of phases. The extent of these phases is dependent on the particular application being considered. For example, in the readers/writers problem, three phases are of interest: **reading**, **writing** and **computing**. A phase transition occurs when a process ceases executing in one phase and attempts to begin execution in another phase. A synchronization mechanism is employed to constrain the phase transitions of a collection of processes in accordance with some specification.

A technique for implementing synchronization mechanisms in distributed programs, assuming processes can initiate broadcasts and that messages between a pair of processes are always received in the order sent, is developed in [S79]. There, in order to make a phase transition, a process broadcasts a timestamped phase transition message and waits for a predicate (corresponding to that phase transition) on its local state to become true. The local state information at a process is updated whenever a message is received by that process. Timestamps are generated using logical clocks as described in [La78]. Unlike the synchronization scheme developed in [La78], our approach can be employed in environments in which processors may fail, assuming that processor failures are detected and that a failed processor does not broadcast messages. Provisions exist in the protocol to cope with processor restarts, as well.

Our technique has been used to solve synchronization problems directly, to implement new synchronization mechanisms that are well suited for use in distributed systems, like eventcounts and sequencers [Re77], and to construct distributed versions of existing mechanisms. For example, we have developed an implementation of a distributed semaphore -- a semaphore-like object that does not require shared memory -- and the conditional synchronous message passing primitives of Communicating Sequential Processes [Ho78] and ADA. In addition, using our approach, in [S80] we generalize "locking" solutions for the consistency problem in centralized database systems to distributed database systems.

Detecting distributed termination is another illustration of a programming problem for which simple solutions can be obtained by using broadcasts. The problem is to devise a protocol so that a process can determine that every process is waiting for input from other processes, thus signifying that a particular computation is completed and the next one can be started by all processes. Several solutions for the problem have recently appeared in the literature [DS78] [F80] [L80]. Although all these solution employ decentralized control, none can tolerate process failure. In [LS80] we develop a fault-tolerant distributed termination scheme using broadcast communication. Moreover, it is possible to derive each of the other solutions from our protocol by assuming

processor failures do not occur and applying various optimizations. Thus, although our protocol was first developed in terms of broadcast communication, programs that do not actually use broadcasts can be derived from it.

3. Implementing Broadcasts

"Multi-destination" network organizations -- contention networks (such as Ethernet) and ring networks (such as DCS) -- appear to implement broadcast capability directly in hardware. However, close study reveals that messages may not always be delivered to all processes in such networks [Le79]. In other network organizations, a message must be directed to a single other processor. There, broadcast protocols where each processor sends the message to one other processor require time linear in the number of processors. This usually results in an unacceptable delay for completion of a broadcast. On the other hand, broadcast protocols in which a processor sends the message to more than one other processor require that, should a processor fail, some other processor will assume its duties. In [SS80] such a scheme is developed and proved correct. A broadcast strategy is a formal specification of the manner in which a message is disseminated among processors in order to effect a broadcast. Choice of what broadcast strategy to employ in a given situation depends on what is to be optimized. For example, one strategy might minimize the length of time it takes for all processors (that have not failed) to receive the message, while another minimizes the impact of a processor failure on broadcast completion time. The broadcast protocol developed in [SS80] will work in conjunction with any "reasonable" broadcast strategy. Also, we develop a class of "minimum time for delivery" broadcast strategies that are well suited for use in homogeneous local computer networks. These strategies are parameterized with respect to the speed of the communications network and the speed of the processors.

A second problem that must be addressed when using broadcast communication concerns implementing secure inter-process communication. Given a broadcast facility (which might be implemented in hardware), encryption can be used to allow secure communications between arbitrary

groups of processes. Clearly, if there are N processors, then there are (potentially) 2^N-1 broadcast groups, hence 2^N-1 separate keys are required. However, in [DS80] we have shown how only O(N) secret keys need be stored to generate the required 2^N-1 broadcast keys. In addition, the schemes developed support the existence of master keys for groups. A master key is a key that can be used to decipher any communication among processes that constitute a subset of the group for which it is master. This is useful for performance monitoring applications and hierarchical protection in computer networks.

4. Future Directions

Our ultimate goal is to design a fully decentralized operating system -- one with no centralized resources or control. Use of broadcasts appear to be one technique that can be used in such a venture. Structuring a distributed program around the use of broadcasts can be viewed as a distributed programming paradigm, much as, say, "divide and conquer" is a sequential programming paradigm. We have established that reasonable implementations of broadcasts are feasible in present day computer networks. We have also shown how the broadcast paradigm can be applied to selected distributed programming problems. Clearly, the utility of this approach will only be understood as it is applied to more problems. To this end, we are presently studying decentralized resource allocation schemes (that use broadcasts) and associated deadlock detection strategies.

References

[DS78] Dijkstra, E.W. and C.S. Scholten. Termination detection for diffusing computations. EWD 687a.

[DS80] Denning, D. and F.B. Schneider. The master key problem. Proc. 1980 Symposium on Security and Privacy, April 1980, Oakland, Calif.

[F80] Francez, N. Distributed termination. TOPLAS 2, 1 (Jan. 1980), 42-55.

[Ho80] Hoare, C.A.R. Communicating sequential processes. CACM 21, 8 (Aug 1978), 666-677.

[L80] Levin, G.M. Proof rules for communicating sequential processes. Dept. of Computer Science, Cornell Univ., Ph.D. thesis, 1980.

[La78] Lamport, L. Time, clocks and the ordering of events in a distributed system. CACM 21, 7 (July 1978), 558-565.

[Le79] LeLann, G. An analysis of different approaches to distributed computing. Proc. First International Conference on Distributed Computing Systems, Oct. 1979, Huntsville, Alabama, 222-232.

[LS80] Lermen, C.W. and F.B. Schneider. Distributed termination when processors can fail. In preparation.

[Re77] Reed, D.P. and R.K. Kanodia. Synchronization with eventcounts and sequencers. Proc. 6 Symposium on Operating Systems Principles, Nov. 1977, W. Lafayette, Ind., 91-92.

[S79] Schneider, F.B. Synchronization in distributed programs. TR 79-391, Dept. of Computer Science, Cornell Univ., 1979 (to appear in TOPLAS).

[S80] Schneider, F.B. Ensuring consistency in a distributed database system by use of distributed semaphores. Proc. International Symposium on Distributed Data Bases, March 1980, Paris, France, 183-189.

[SS80] Schneider, F.B. and R.D. Schlichting. Fast Reliable Broadcasts. Technical Report, Dept. of Computer Science, Cornell Univ. Oct. 1980.