Paradigms for Process Interaction in Distributed Programs

GREGORY R. ANDREWS

Department of Computer Science, The University of Arizona, Tucson, Arizona 85721

Distributed computations are concurrent programs in which processes communicate by message passing. Such programs typically execute on network architectures such as networks of workstations or distributed memory parallel machines (i.e., multicomputers such as hypercubes). Several paradigms—examples or models—for process interaction in distributed computations are described. These include networks of filters, clients, and servers, heartbeat algorithms, probe/echo algorithms, broadcast algorithms, token-passing algorithms, decentralized servers, and bags of tasks. These paradigms are applicable to numerous practical problems. They are illustrated by solving problems, including parallel sorting, file servers, computing the topology of a network, distributed termination detection, replicated databases, and parallel adaptive quadrature. Solutions to all problems are derived in a step-wise fashion from a general specification of the problem to a concrete solution. The derivations illustrate techniques for developing distributed algorithms.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*multiple-instruction-stream, multiple-data-stream processors (MIMD)*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: clients and servers, distributed and parallel algorithms, distributed programming, distributed programming methods, heartbeat algorithms, networks of filters, patterns for interprocess communication, probe/echo algorithms, replicated servers, token-passing algorithms

INTRODUCTION

Concurrent programming is the activity of constructing a program containing multiple processes that cooperate in performing some task. Given a specification of the problem to be solved, decisions have to be made about what and how many processes to use and how they should interact. These decisions are affected by the application and by the underlying hardware on which the program will run. Whatever choice is made, a critical problem is ensuring that communication between processes is properly synchronized.

The history of concurrent programming has followed the same stages as other experimental areas of computer science. The topic arose due to hardware developments and has developed in response to technological changes. Over

CONTENTS

time, the initial ad hoc approaches have evolved into a collection of core principles and general programming techniques.

Operating systems were the first significant examples of concurrent programs and remain among the most interesting. With the advent of independent device controllers in the 1960s, it became natural to organize an operating system as a concurrent program, with processes managing devices and execution of user tasks. Processes in such a single-processor system are implemented by multiprogramming, with processes executed one at a time in an interleaved manner.

Technology has since evolved to produce an amazing variety of multiprocessor systems. In a *shared-memory multiprocessor*, multiple processors share a common memory. In a *multicomputer*, several processors, called *nodes*, are connected by high-speed message-switching hardware. In a *network* system, independent processors are connected by a communication network (e.g., an Ethernet). Several hybrid combinations also exist (e.g., networks of multiprocessor workstations). The operating systems for multiprocessors are concurrent programs in which at least some processes can execute in parallel. The processors themselves range in power from microcomputers to supercomputers.

There are many examples of concurrent programs beside operating systems. They arise whenever the implementation of an application involves real or apparent parallelism. For example, concurrent programs are used to implement

- window systems on personal computers or workstations,
- transaction processing in multiuser database systems,
- file servers in a network, and
- scientific computations that manipulate large arrays of data.

The last kind of concurrent program is often called a *parallel program* since it is typically executed on a multiprocessor or multicomputer.

A *distributed program* is a concurrent (or parallel) program in which processes communicate by message passing. The name results from the fact that such programs typically execute on distributed architectures—such as multicomputers or network computers—in which processors do not share memory. A distributed program can, however, also be executed on a shared-memory multiprocessor or even on a single processor.

There are four basic kinds of processes in a distributed program: filters, clients, servers, and peers. A *filter* is a data transformer; it receives streams of data values from its input channels, performs some computation on those values, and sends streams of results to its output channels. Many of the user-level commands in the UNIX[1] operating system

[1]UNIX is a trademark of AT&T, Bell Laboratories.

are filters (e.g., the text formatting programs *tbl, eqn,* and *troff*).

A *client* is a triggering process; a *server* is a reactive process. Clients make requests that trigger reactions from servers. Thus, a client initiates activity, at times of its choosing; it often then delays until its request has been serviced. On the other hand, a server waits for requests to be made, then reacts to them. A server is usually a nonterminating process and often provides service to more than one client. For example, a file server in a distributed system typically manages a collection of files and services requests from any client who wants to access those files.

A *peer* is one of a collection of identical processes that interact to provide a service or compute a result. For example, two peers might each manage a copy of a data file and interact to keep the two copies consistent. Or several peers might interact to solve a parallel programming problem, with each solving a piece of the problem.

The remainder of this paper examines several process interaction patterns that occur in distributed programs. Each *interaction paradigm* is an example or model of an interprocess-communication pattern and has an associated programming technique that can be used to solve a variety of interesting distributed programming problems. The following process-interaction paradigms are described:

- one-way data flow through networks of filters,
- requests and replies between clients and servers,
- back-and-forth (heartbeat) interaction between neighboring processes,
- probes and echoes in graphs,
- broadcasts between processes in complete graphs,
- token passing along edges in a graph,
- coordination between decentralized server processes, and
- replicated workers sharing a bag of tasks.

These are illustrated by solving a variety of problems, including parallel sorting, disk scheduling, computing the topology of a network, distributed termination detection, replicated files, and parallel adaptive quadrature. Additional examples are mentioned throughout. (See Raynal [1988a, 1988b] for discussions of many of these problems and several additional ones that can be solved using the above paradigms.)

The different problems are solved by starting with a specification of the problem and proceeding in a series of steps to a complete solution. This procedure illustrates several techniques for developing distributed algorithms and showing that the solution is correct. Some problems are solved by starting with a shared-variable solution, then developing a distributed solution. Others are solved by making simplifying assumptions, then extending the initial solution to handle the general case.

The next section presents the programming notation that will be used. Subsequent sections examine the above interaction paradigms.

1. PROGRAMMING NOTATION

In message-passing programs, processes share *channels*. A channel is an abstraction of a physical communication network in that it provides a communication path between processes. Channels are accessed by means of two kinds of primitives: send and receive. To initiate a communication, a process sends a message to a channel; another process later acquires the message by receiving from the channel.

All programming notations based on message-passing provide channels of some form and primitives for sending to and receiving from them. Many different notations have been proposed; see Andrews and Schneider [1983] for a survey of different language mechanisms and Bal et al. [1989] for a survey of specific distributed programming languages. These programming notations vary in the way channels are provided and named,

the way channels are used, and the way communication is synchronized. For example, channels can be global to processes or directly associated with processes, and they can provide one-way or two-way information flow. Also, communication can be asynchronous (nonblocking) or synchronous (blocking).

Five general combinations of these design choices have proved the most popular since each is especially well-suited to solving some programming problems and each can be implemented with reasonable efficiency. With *asynchronous message passing*, channels have conceptually unbounded capacity, and hence the send primitive does not cause a process to block. With *synchronous message passing*, on the other hand, communication and synchronization are tightly coupled. In particular, a process sending a message delays until the other process is ready to receive the message. Thus, the exchange of a message represents a synchronization point between two processes and channels never need to contain stored messages. (In between these two is *buffered message passing* in which a channel has a fixed capacity and hence send delays only when the channel is full.)

Generative communication [Gelernter 1985] is similar to asynchronous message passing. The main difference is that with generative communication, processes share a single communication channel called *tuple space*. With generative communication, associative naming is used to distinguish different kinds of messages stored in tuple space, whereas with asynchronous message passing, different channels are used for different kinds of messages.

Remote procedure call (RPC) and *rendezvous* combine aspects of monitors [Hoare 1974] and synchronous message passing. As with monitors, a module or process exports operations and the operations are invoked by a call statement. As with synchronous message passing, execution of a call is synchronous: The calling process delays until the invocation has been serviced and any results have been returned. An operation is thus a two-way communication channel, from the caller to the process that services the invocation, then back to the caller. An invocation is serviced in one of two ways. One approach is to create a new process. This is called *remote procedure call* since the servicing process is declared as a procedure and might execute on a different processor than the calling process. The second approach is to *rendezvous* with an existing process. A rendezvous is serviced by means of an input (or accept) statement that waits for an invocation, processes it, then returns results.

All five approaches are equivalent in the sense that a program written in one notation can be rewritten in any of the others. Each approach, however, is better for solving some problems than others. Moreover, the different mechanisms have different performance [Atkins and Olsson 1988].

Asynchronous message passing is used in this paper for three reasons. First, it is the most flexible mechanism; in essence it is the lowest common denominator. Second, asynchronous message passing is typically what is provided by system routines in network and multicomputer operating systems. Finally, asynchronous message passing is the most natural approach to programming several of the examples that are considered. Some of the examples, however, would best be programmed using one of the other approaches; this is pointed out in relevant places.

With asynchronous message passing, a channel is a queue of messages that has been sent but not yet received. In this paper, a channel declaration has the form

chan *ch*($f_1 : t_1, \ldots, f_n : t_n$)

Identifier *ch* is the channel's name. The f_i and t_i are names and types, respectively, of the data fields in messages transmitted via the channel. The field names are optional; they will be used when it is helpful to document what each field contains. For example,

chan *input*(**char**)
chan *disk_access*(*cylinder* : **int**, *block* : **int**, *count* : **int**, *buffer* : **ptr** [*] **char**)

declares two channels. The first, *input*, contains single-character messages. The second, *disk_access*, contains messages having four fields, with the field names indicating the role of each field. In many examples, arrays of channels will be used. These are declared by appending subscript ranges to the channel identifier.

A process sends a message to channel *ch* by executing

send $ch(expr_1, \ldots, expr_n)$

The $expr_i$ are expressions whose types must be the same as the types of the corresponding message fields. The effect of executing **send** is to evaluate the expressions, then append a message containing these values to the end of the queue associated with channel *ch*. Because this queue is conceptually unbounded, execution of **send** never causes delay; hence **send** is a *nonblocking* primitive.

A process receives a message from channel *ch* by executing

receive $ch(var_1, \ldots, var_n)$

The var_i are variables whose types must be the same as those of the corresponding fields in the declaration of *ch*. The effect of executing **receive** is to delay the receiver until there is at least one message on the channel's queue. Then the message on the front of the queue is removed and its fields are assigned to the var_i. Thus, in contrast to **send**, **receive** is a *blocking* primitive since it might cause delay. The **receive** primitive blocks so the receiving process does not have to busy-wait polling the channel if it has nothing else to do until a message arrives.

In this paper, message delivery is assumed to be reliable and error free. Thus, every message sent is eventually delivered, and messages are not corrupted. In addition, because each channel is a first-in/first-out queue, two messages sent to a channel by the same process will be received in the order in which they were sent. (See, e.g., Tanenbaum's [1988] book on computer networks for how these attributes can be realized.)

As a simple example, the process below receives a stream of characters from one channel, *input*, assembles the characters into lines, and sends the resulting lines to a second channel, *output*. The carriage-return character, CR, indicates the end of a line; a line is at most MAXLINE characters long. Both CR and MAXLINE are symbolic constants. The example follows:

```
chan input(char)
chan output([1:MAXLINE] char)
Char_to_Line::
  var line[1:MAXLINE] : char
  var i : int := 1
  do true →
    receive input(line[i])
    do line[i] ≠ CR and i < MAXLINE →
    {line[1:i] contains last i input characters}
      i := i + 1; receive input(line[i])
    od
    send output(line); i := 1
  od
```

This process is an example of a filter; namely, it transforms a stream of characters into a stream of lines.

The above example also illustrates other aspects of the programming notation used in this paper. Processes will be declared as shown, with a capitalized name followed by a double colon. Declarations and statements are programmed in a conventional way. (The specific notation used here is that of SR [Andrews et al. 1988].) Comments are introduced by a sharp character (#); they are terminated by the end of the line. Finally, assertions are predicates enclosed in braces; they indicate what is true at the corresponding point in execution and can be thought of as another form of comment.

Channels will be declared global to processes, as above, since they are shared by processes. Any process may send to or receive from any channel. When channels are used in this way, they are sometimes called *mailboxes*. In many of the subsequent programs, however, each channel will have exactly one receiver, although it may have many senders. In this case, a channel is often called an *input port* since it provides a window

(port hole) into the receiving process. If a channel has just one sender and one receiver, it is often called a *link* since it provides a direct path from the sending to the receiving process.

Usually a process will want to delay when it executes **receive** but not always. For example, the process might have other useful work to do if a message is not yet available for receipt. Or a process such as a scheduler may need to examine all queued messages in order to select the best one to service next. To determine whether a channel's queue is currently empty, a process can call the Boolean-valued function

empty(*ch*)

This primitive returns true if channel *ch* contains no messages; otherwise it returns false.

2. FILTERS: A SORTING NETWORK

The key to understanding message-based programs is to understand communication assumptions. Hence, the key to deriving a process that uses message passing is to specify the communication assumptions. Since the output of a filter process is a function of its input, the appropriate specification is one that relates the value of messages sent on output channels to the values of messages received on input channels. The actions a filter takes in response to receiving input must ensure this relation every time the filter sends output.

To illustrate how filters are developed and programmed, consider the problem of sorting a list of n numbers into ascending order. The most direct way to solve the problem is to write a single filter process, *Sort*, that receives the input from one channel, uses one of the standard sorting algorithms, then writes the result to another channel. Let *input* be the input channel, and let *output* be the output channel. Assume the n values to be sorted are sent to *input* by some unspecified process(es). Then the goal of the sorting process is to ensure that the values sent to *output* are ordered and are a permutation of the values received from *input*. Let *sent*[*i*] indicate the *i*th value sent to *output*. Then the goal is specified precisely by the predicate:

SORT: $(\forall i: 1 \leq i < n: sent[i] \leq sent[i+1] \wedge$ values sent to *output* are a permutation of values received from *input*

An outline of *Sort* is as follows:

chan *input*(**int**), *output*(**int**)
Sort:: declarations of local variables
receive all numbers from *input*
sort the numbers
send the sorted numbers to *output*

Since **receive** is a blocking primitive, a practical concern is for *Sort* to determine when it has received all the numbers. One solution is for *Sort* to know the value of n in advance. A more general solution is for n to be the first input value and the numbers themselves to be the next n input values. An even more general solution is to end the input stream with a *sentinel*, which is a special value that indicates all numbers have been received. This solution is the most general since the process producing the input does not itself need to know in advance how many values it will send to *input*.

If processes are "heavyweight" objects, as they are in most operating systems, the above approach would often be the most efficient way to solve the sorting problem. A different approach, however—which is amenable to direct implementation in hardware—is to use a network of small processes that execute in parallel and interact to solve the problem. (A hybrid approach would be to use a network of medium-sized processes.) There are many kinds of sorting networks, just as there are many different internal sorting algorithms [Akl 1985]. Here, a *merge network* is presented.

The idea behind a merge network is to merge repeatedly—and in parallel—two sorted lists into a longer sorted list. The network is constructed out of instances of *Merge* filters. Each *Merge* process receives values from two ordered input streams, *in1* and *in2*. It merges these values to produce one ordered output

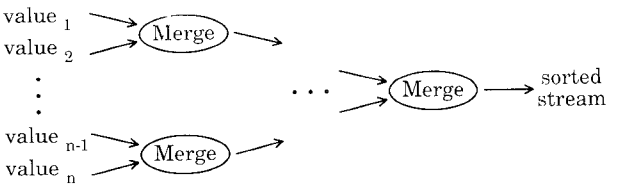

Figure 1. Sorting network of *Merge* processes.

stream, *out*. Assume the ends of the input streams are marked by a sentinel, EOS, as discussed above. Also assume that *Merge* appends EOS to the end of the output stream. If there are a total of n input values (not counting the sentinels), then when *Merge* terminates the following should be true:

MERGE:
in1 and *in2* are empty $\wedge$ *sent*[$n + 1$] = EOS $\wedge$
($\forall i$: $1 \leq i < n$: *sent*[i] $\leq$ *sent*[$i + 1$]) $\wedge$
values sent to *out* are a permutation of values received from *in1* and *in2*

Again, *sent* is the stream of values sent to the output channel. The first line of *MERGE* says that all input has been consumed and EOS has been appended to the end of *out*; the third line says that the output is ordered; the fourth line says that all input data has been output.

One way to implement *Merge* is to receive all input values, merge them, then send the merged list to *out*. This, however, requires storing all input values. Since the input streams are ordered, a much better way to implement *Merge* is repeatedly to compare the next two values received from *in1* and *in2* and to send the smaller to *out*. Let *v1* and *v2* be these values. This suggests the following process outline:

```
chan in1(int), in2(int), out(int)
Merge:: var v1, v2 : int
        receive in1(v1); receive in2(v2)
        do more input to process →
          send smaller of v1 and v2 to out
          receive another input value from
            in1 or in2
        od
        send out(EOS) {MERGE}
```

As a special case, after all values from one input stream have been consumed, further values from the other input stream can simply be appended to *out*. Expanding the loop and handling the special cases yields the final program:

```
chan in1(int), in2(int), out(int)
Merge:: var v1, v2 : int
        receive in1(v1); receive in2(v2)
        do v1 ≠ EOS and v2 ≠ EOS →
           if v1 ≤ v2 →
             send out(v1); receive in1(v1)
           [] v2 ≤ v1 →
             send out(v2); receive in2(v2)
           fi
        [] v1 ≠ EOS and v2 = EOS →
           send out(v1); receive in1(v1)
        [] v1 = EOS and v2 ≠ EOS →
           send out(v2); receive in2(v2)
        od
        send out(EOS) {MERGE}
```

The above loop is a guarded command [Dijkstra 1976] with three branches, one for each of the cases in which there is more input to process. When the loop terminates, both *v1* and *v2* are equal to sentinel value EOS. The *Merge* process sends the sentinel to *out*, then terminates. As required, predicate *MERGE* is true when *Merge* terminates.

To form a sorting network, a collection of *Merge* processes and arrays of input and output channels are connected together. Assuming the number of input values n is a power of 2, the processes and channels are connected so the resulting communication pattern forms a tree as depicted in Figure 1. Information in the sorting network flows from left to right. Each node at the left is given two input values, which it merges to form a

stream of two sorted values. The second-level nodes form streams of four sorted values, and so on. The right-most node produces the final sorted stream. The network contains $n - 1$ processes; the depth of the network is $\log_2 n$.

To realize the sorting network in Figure 1, the input and output channels need to be shared. In particular, the output channel used by one instance of *Merge* needs to be the same as one of the input channels used by the next instance of *Merge* in the graph. This can be programmed in one of two ways. The first approach is to use *static naming*: Declare all channels to be a global array, and have each instance of *Merge* receive from two elements of the array and send to one other element. The tricky part in this case is to have the channels accessed by $Merge_i$ be some function of i. The second approach is to use *dynamic naming*: Declare all channels to be global as above, but parameterize the processes so that each is passed three channels as arguments when it is created. This makes the programming of the *Merge* processes easier since each is textually identical. It, however, requires having a main process that dynamically creates and passes channels to the *Merge* processes.

A key attribute of filters like *Merge* is that they can be interconnected in different ways. All that is required is the output produced by one filter meet the input assumptions of another. An important consequence of this attribute is that as long as the input assumptions and output behaviors are the same, one filter process—or a network of filters—can be replaced by a different filter process or network. For example, the single *Sort* process described earlier can be replaced by a network of *Merge* processes plus a process (or network) that distributes the input values to the merge network. Similarly, *Sort* could be replaced by a network that implements any other sorting strategy.

Networks of filters can be used to solve a variety of other parallel programming problems. For example, Hoare [1978] describes a prime number sieve and a matrix multiplication network.

3. CLIENTS AND SERVERS

Recall that a server is a process that repeatedly handles requests from clients. This section shows how to program servers and their clients. The first example shows how to turn monitors into servers. It shows how to implement resource managers using message passing and also points out the duality between monitors and message passing: Each can be directly simulated by the other.

The second example describes different ways to implement a disk scheduler and a disk server. One solution illustrates a programming technique called *upcalls*; another solution shows how to implement a self-scheduling disk server. The third example shows how to implement a file server. In this case, the solution illustrates a programming technique called *conversational continuity*.

3.1 Centralized Servers: Active Monitors

A centralized server is a resource manager: It has local variables that record the state of a resource and services requests to access that resource. Thus, a centralized server is similar to a monitor [Hoare 1974]. The main differences are that a server is active, whereas a monitor is passive and that clients communicate with a server by sending and receiving messages, whereas clients call monitor procedures. To illustrate these similarities and differences, this section presents both monitor and server implementations of a resource allocator.

A monitor is a synchronization mechanism that is commonly used in concurrent programs that execute on shared-memory machines [Hoare 1974]. It encapsulates permanent variables that record the state of some resource and exports a set of procedures that are called to access the resource. The procedures execute with mutual exclusion; they use condition variables for internal synchronization. The general structure of a

```
monitor Resource_Allocator
  {ALLOC: avail ≥ 0 ∧ (avail > 0) ⇒ empty(free)}
  var avail : int := MAXUNITS, units : set of int,
      free : condition
  code to initialize units to appropriate values
  proc acquire( res id : int )
    if avail = 0 → wait(free)
    [] avail > 0 → avail := avail-1
    fi
    id := remove(units)
    end
  proc release( id : int )
    insert(id, units)
    if empty(free) → avail := avail+1
    [] not empty(free) → signal(free)
    fi
  end
end
```

Figure 2. Resource allocation monitor.

monitor is

```
monitor Mname   # Invariant MI
  var permanent variables
  initialization code
  proc op1(formals1) body1 end
  .
  .
  proc opn(formalsn) bodyn end
end
```

Associated with each monitor is an invariant *MI*: a predicate about the permanent variables that is true when no process is executing in the monitor.

As a concrete example, Figure 2 contains a monitor that manages a multiple unit resource—such as memory pages or file blocks. The two operations are *acquire* and *release*. For simplicity, clients acquire and release units one at a time. The free units of the resource are stored in a set, which is accessed by *insert* and *remove* operations. A process that calls *acquire* delays if there are no available units. When a unit is released, if there is a delayed process it is awakened and takes the unit. Monitor invariant *ALLOC* states the relation between the number of available units and the status of condition variable *free*.

The monitor in Figure 2 is programmed the way it is so that it works correctly independent of whether signaling is preemptive—as in Hoare [1974]—or nonpreemptive—as in the Mesa language [Mitchell et al. 1979] or the UNIX operating system [Thompson 1978]. If signaling is preemptive, the bodies of *acquire* and *release* can be simplified.

To simulate a monitor using message passing, one server process is used. The permanent variables of the monitor become the server's local variables. After initializing the variables, the server executes a permanent loop in which it repeatedly services "calls" of operations. The monitor invariant becomes the loop invariant in the server: It is true before and after each operation is serviced. A call is simulated by having a client process send a message to a request channel, then receive the result from a reply channel. The server thus repeatedly receives from the request channel and sends results to the reply channel. The formal parameters in the different monitor operations become additional variables local to the server. To avoid having one client see the result intended for another, each client needs its own private result channel. If these are declared as a global array, a client thus needs to pass the index of its private element of the result array to the server as part of the request message.

Figure 3 contains a resource allocation server with the same functionality as the resource allocation monitor in Figure 2. The server contains an outer if state-

```
type op_kind = enum(ACQUIRE, RELEASE)
chan request(index : int, op_kind, unitid : int)
chan reply[1:n](int)
Allocator:: var avail : int := MAXUNITS, units : set of int, pending : queue of int
            var index : int, kind : op_kind, unitid : int
            code to initialize units to appropriate values
            {ALLOC: avail ≥ 0 ∧ (avail > 0) ⇒ empty(pending)}
            do true → receive request(index, kind, unitid)
                      if kind = ACQUIRE →
                          if avail > 0 →     # honor request now
                              avail := avail-1; unitid := remove(units)
                              send reply[index](unitid)
                          [] avail = 0 →     # save index of requesting process
                              insert(pending, index)
                          fi
                      [] kind = RELEASE →
                          if empty(pending) →     # release unitid
                              avail := avail+1; insert(units, unitid)
                          [] not empty(pending) →     # give unitid to first requester
                              index := remove(pending); send reply[index](unitid)
                          fi
                      fi
            od
Client[i: 1..n]:: var unitid : int
                  ...
                  # acquire a unit of the resource
                  send request(i, ACQUIRE, 0)      # unitid not needed
                  receive reply[i](unitid)
                  # use resource unitid and then later release it
                  send request(i, RELEASE, unitid)
```

Figure 3. Resource allocator and clients.

ment that serves as a case statement with branches for each kind of operation. Within each branch, a nested if statement implements the body of the corresponding operation, much as in Figure 2. A key difference between the server and monitor, however, is that the server cannot wait when servicing a request; it must continue to receive other operations until a unit is released. Thus, the server needs to save a request when no units are available and defer sending a reply. Later, when a unit is released, the server needs to honor one saved request, if there is one, by allocating the unit to the requester and sending a reply.

Figure 3 also gives an outline of the client interface, which illustrates how calls are simulated using message passing. After sending an ACQUIRE message, a client waits to receive a unit. After sending a RELEASE message, however, the client does not wait for the message to be processed since no reply is needed (assuming nothing can go wrong that the client cares about).

The program in Figure 3 uses static naming since in this paper channels are global to the processes and are referenced directly. Consequently, each process must be coded carefully so that it uses the correct channels. For example, *Client*[*i*] must not use the reply channel of some other *Client*[*j*]. Alternatively, dynamic naming could be used by having each client create a private reply channel, which it then passes to *Allocator* as the first field of *request* in place of the integer index. This would ensure that clients could not access each other's reply channels. It would also permit the number of clients to vary dynamically. (Here, there is a fixed number n of client processes.)

The *Resource_Allocator* monitor and the *Allocator* server point out the duality between monitors and message passing: There is a direct correspondence between the various mechanisms in

Monitor-Based Programs	*Message-Based Programs*
permanent variables	local server variables
procedure identifiers	*request* channel and operation kinds
procedure call	**send** *request*; **receive** *reply*
monitor entry	**receive** *request*
procedure return	**send** *reply*
wait statement	save pending request
signal statement	retrieve and process pending request
procedure bodies	arms of case statement on operation kind

Figure 4. Duality between monitors and centralized servers.

monitors and those in the client and server [Lauer and Needham 1978]. In particular, as shown in Figure 4, the mechanisms in monitors serve the same purpose as do the ones listed opposite them in a message-based program.

Since the bodies of monitor procedures have direct duals in the arms of the server case statement, the relative performance of monitor-based versus message-based programs depends only on the relative efficiency of the implementation of the different mechanisms. On shared-memory machines, procedure calls and actions on condition variables tend to be more efficient than message-passing primitives. For this reason, most operating systems for such machines are based on a monitor-style implementation. On the other hand, most distributed systems are based on message passing since that is both efficient and the appropriate abstraction for such machines.

Although there is a duality between monitors and centralized servers programmed using asynchronous message passing, the duality would be even stronger if the server were programmed using RPC or rendezvous. With RPC, a server module is programmed in almost the same way as a monitor; the only difference is that semaphores or something comparable need to be used to simulate condition variables. With rendezvous, different operations can be used for each kind of server request; an enumeration type and case statement are not required. For example, the *Allocator* in Figure 3 could be programmed in Ada using a **select** statement, with one arm for each of the *acquire* and *release* operations. Another important consequence of using RPC or rendezvous is that the client interface to the server would be identical to the interface to a monitor. In particular, a client would call the *acquire* and *release* operations. The client would not first have to send a message then receive a reply. More importantly, an array of *reply* channels would not be needed and could not be misused.

3.2 Disk Scheduling and Disk Access

Consider now the problem of accessing a moving-head disk. In a distributed system, it is appropriate to use one server process for each disk. Each disk server executes on the machine to which the disk is attached. Clients, which may execute on any machine, request access by sending read and write requests to the server.

The physical address of each data item includes a cylinder number, a track number, and an offset. For a moving-head disk, the largest component of access time is the time it takes to move the read/write head to the appropriate cylinder. Thus, it is important to reduce head-seek time by ordering pending read/write requests. There are several different disk-scheduling algorithms, as described in most operating systems texts [Peterson and Silberschatz 1985]. For example, the Shortest Seek Time (SST) algorithm minimizes head movement by always selecting the pending request whose cylinder is closest to the current head position.

As shown in Figure 5, there are three main ways to structure a solution to the disk-scheduling problem. In all cases, the disk is assumed to be controlled by a

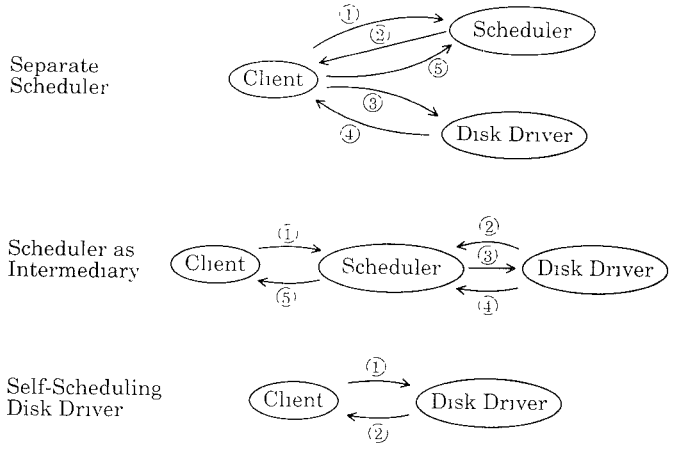

Figure 5. Disk scheduling structures with message passing.

server process that performs all disk access. The principal differences between the three structures are the client interface and the number of messages that must be exchanged per disk access.

One approach is to have the scheduler be a server separate from the disk server. Thus clients first "call" the scheduler to request access, then use the disk, and finally "call" the scheduler to release access. Both the scheduler and disk server are active monitors (there is one such pair for each physical disk). Hence, the servers and their clients are programmed as described in Section 3.1.

In the second structure, the scheduler is an intermediary between clients and the disk server. In this case, the scheduler is an active monitor with three operations. One is used by clients to request disk access. The second is used by the disk server to acquire the next read/write request. The third operation is used by the server to indicate completion of a read/write action and to return results, which the scheduler then sends back to the appropriate client. (The latter two operations could be combined.) When the scheduler is an intermediary, the disk server's "calls" to acquire the next request and to return results are termed *upcalls* [Clark 1985]. This is because they come up to a higher level server—one closer to clients—from a lower level server—one closer to the physical architecture. This upcall structure is needed since the scheduler cannot know which request to give the disk server until that server indicates it is free.

In the third solution structure, the scheduler and disk server are combined into a single, self-scheduling server process. The disk server now has one operation, to which clients send requests. To do scheduling, the driver must examine all pending requests, which means it must receive all messages queued on the request channel. It does this by executing a loop that terminates when the request channel is empty and there is at least one saved request. The driver then selects the best request according to its internal scheduling policy, accesses the disk, and finally sends a reply to the client who sent the request.

When the scheduler is a separate process, five messages must be exchanged per disk access: two to request scheduling and get a reply, two to request disk access and get a reply, and one to release the disk. The client is involved in all five communications. When the scheduler is an intermediary, four kinds of messages have to be exchanged: The client has to send a request and wait to receive one reply, the disk driver has to ask the

```
chan request(index : int, cylinder : int, other argument types)
    # other arguments indicate read or write, disk block, memory buffer, etc.
chan reply[1:n](result types)
Disk_Driver:: var lower, higher : ordered queue of (int, int, other argument types)
                  # contain index, cylinder, and other arguments of pending request
              var headpos : int := 1, nsaved := 0
              var index : int, cyl : int, args : other argument types
              { SST: lower is an ordered queue from largest to smallest cyl ∧
                     all values of cyl in lower are ≤ headpos ∧
                     higher is an ordered queue from smallest to largest cyl ∧
                     all values of cyl in higher are ≥ headpos ∧
                     (nsaved = 0) ⇒ (both lower and higher are empty) }
              do true →
                  do not empty(request) or nsaved = 0 →
                      # wait for first request or receive another one, then save it
                      receive request(index, cyl, args)
                      if cyl ≤ headpos := insert(lower, (index, cyl, args))
                      [] cyl ≥ headpos := insert(higher, (index, cyl, args))
                      fi
                      nsaved := nsaved+1
                  od
                  # select best saved request; there is at least one in lower or higher
                  if size(lower) = 0 → (index, cyl, args) := remove(higher)
                  [] size(higher) = 0 → (index, cyl, args) := remove(lower)
                  [] size(higher) > 0 and size(lower) > 0 →
                      remove element from front of lower or higher depending
                      on which saved value of cyl is closer to headpos
                  fi
                  headpos := cyl; nsaved := nsaved-1
                  access the disk
                  send reply[index](results)
              od
```

Figure 6. Self-scheduling disk driver.

scheduler for the next request and get the reply. (The driver process can return the results of one disk access request when it asks for the next one.) As can be seen in Figure 5, a self-scheduling disk driver presents the most attractive structure. In particular, only two messages need to be exchanged.

Figure 6 outlines a self-scheduling disk driver that uses the SST scheduling policy. A message sent to the *request* channel indicates which cylinder to access. These values are assumed to be between zero and the maximum cylinder number. The driver keeps track of the current head position in local variable *headpos*. To implement the SST policy, the driver keeps two ordered queues: *lower* and *higher*. When a request arrives, the driver stores it in queue *lower* if the requested cylinder is lower than the current head position; otherwise, it stores it in queue *higher*. (Either queue is used for requests for the current head position.) Requests in queue *lower* are ordered by decreasing cylinder value; those in *higher* are ordered by increasing cylinder value. The invariant for the outer loop of the driver process is as indicated; variable *nsaved* is a count of the total number of saved requests.

In Figure 6, the **empty** primitive is used in the guard of the inner loop to determine whether there are more messages queued on the *request* channel. This is an example of the programming technique called *polling*. In this case, the disk driver process repeatedly polls the *request* channel to determine if there are pending requests. If there are, another one is received so the driver has more requests from which to choose. If there are not (at the time **empty** is evaluated), the driver services the best

pending request. Polling is also useful in other situations and is often used within hardware, such as in the implementation of communication using a shared bus.

The *Disk_Driver* process in Figure 6 could readily be changed to use other disk-scheduling strategies such as the elevator algorithm [Hoare 1974]. All that would change are the kinds of sets the server maintains and the criterion for selecting the best saved request.

When asynchronous message passing is used, as here, the *Disk_Driver* process has to manage queues of pending requests. With a powerful rendezvous mechanism that permits requests to be scheduled—such as that in SR [Andrews et al. 1988] or Concurrent C [Gehani and Roome 1986]—the server could be greatly simplified. For example, in SR the body of the main server loop in Figure 6 could be simplified to

```
in request(cylinder, ...)
  by abs(cylinder - headpos) →
    headpos := cylinder
    access the disk
    results := return values
ni
```

Here, requests are serviced one at a time in an order that minimizes the distance between the requested cylinder and the current head position. As in Section 3.1, another virtue of using rendezvous is that the client simply calls the *request* operation; it does not explicitly have to send a request and receive a reply.

3.3 File Servers: Conversational Continuity

As a final example of client/server interaction, this section presents one way to implement file servers, which are processes that provide access to files on secondary storage such as disk files. To access a disk file, a client first opens the file. If the open is successful—the file exists and the client has permission to access it—the client makes a series of read and write requests. Eventually the client closes the file.

Suppose up to n files may be open at once and access to each open file is provided by a separate file server process. Hence, there are n such processes. To open a file, a client needs to acquire a file server that is free, then interact with it. If all file servers are identical, any free one will do.

File servers could be allocated to clients by a separate allocator process. Since all are identical and communication channels are shared, however, there is a simpler approach. In particular, let *open* be a global channel. To acquire a file server, a client sends a request to *open*. When idle, file servers try to receive from *open*. A specific open request from a client will thus be received by any one of the idle file servers. That server sends a reply to the client, then proceeds to wait for access requests. A client sends these to a different channel, *access*[*i*], where i is the index of the file server that allocated itself to the client. Thus, *access* is an array of n channels. Eventually, the client closes the file, at which time the file server becomes idle so it again waits for an open request.

Outlines for the file servers and their clients are given in Figure 7. The file access requests—READ and WRITE—are sent to the same channel. This is necessary since the file server cannot in general know the order in which these requests will be made and hence cannot use different channels for each. For the same reason, when a client wants to close a file, it sends a CLOSE request to the same access channel.

The interaction between a client and a server in Figure 7 is an example of *conversational continuity*. In particular, a client starts a "conversation" with a file server when that server receives the client's open request. The client then continues to converse with the same server. This is programmed by having the server first receive from *open*, then repeatedly receive from *access*[*i*].

Figure 7 presents one possible way to implement file servers. It assumes *open* is a shared channel from which any file server can receive a message. If each channel can have only one receiver, a separate file allocator process would be needed. That process would receive open

```
type kind = enum(READ, WRITE, CLOSE)
chan open(fname : string[*], clientid : int)
chan access[1:n](kind, other types)     # other types give buffer, number of bytes, etc.
chan open_reply[1:n](int)     # field is server index or error indication
chan access_reply[1:n](result types)     # result types are file data, error flags, etc.
File_Server[i: 1..n]:: var fname : string[*], clientid : int
                       var k : kind, args : other argument types
                       var file_open : bool := false
                       var local buffer, cache, disk address, etc.
                       do true →
                            receive open(fname, clientid)
                            # open data file; if successful then:
                            send open_reply[clientid](i); file_open := true
                            do file_open →
                                 receive access[i](k, args)
                                 if k = READ → process read request
                                 [] k = WRITE → process write request
                                 [] k = CLOSE → close file; file_open := false
                                 fi
                                 send access_reply[clientid](result values)
                            od
                       od
Client[j:1..m]:: ...
                 send open("foo", j)     # open file "foo"
                 receive open_reply[j](serverid)     # get back id of server
                 # use and eventually close file by executing the following
                 send access[serverid](access arguments)
                 receive access_reply[j](results)
                 ...
```

Figure 7. File servers and clients.

requests and allocate a free server to a client; file servers would thus need to tell the allocator when they are free.

The solution in Figure 7 uses a fixed number n of file servers. In a language that supports dynamic process and channel creation, a better approach would be to create file servers and access channels dynamically, as needed. This is better since at any point in time there would only be as many servers as are actually being used, and more importantly, there would not be a fixed upper bound on the number of file servers. At the other extreme, there could simply be one file server per disk. In this case, however, either the file server or client interface will be much more complex than shown in Figure 7. This is because either the file server has to keep track of the information associated with all clients who have files open, or clients have to pass file state information with every request.

Yet another approach, used in the Sun Network File System (NFS) [Sandberg et al. 1985], is to implement file access solely by means of remote procedures. Then, "opening" a file consists of acquiring a descriptor (called a file handle in NFS) and a set of file attributes. These are subsequently passed on each call to a file access procedure. Unlike the *File_Server* processes in Figure 7, the access procedures in NFS are themselves stateless—all information needed to access a file is passed as arguments on each call to a file access procedure. This increases the cost of argument passing but greatly simplifies the handling of both client and server crashes. In particular, if a file server crashes, the client simply resends the request until a response is received. If a client crashes, the server need do nothing since it has no state information. (See Sandberg et al. [1985] for further discussion of these issues.)

4. HEARTBEAT ALGORITHMS

In a hierarchical system, servers at intermediate levels are often also clients of lower level servers. For example, the file

server in Figure 7 might well process read and write requests by communicating with a disk server such as the one in Figure 6. This and the next several sections examine additional kinds of server interaction in which servers at the same level cooperate in providing a service. This type of interaction arises in distributed computations in which no one server has all the information needed to service a client request.

Consider the problem of computing the topology of a network, which consists of processors connected by bidirectional communication channels. Each processor can communicate only with its neighbors and knows only about the links to its neighbors. The problem is for each processor to determine the topology of the entire network, that is, the entire set of links. During the computation, the topology is assumed to be static; that is, links are not added or removed.[2]

Each processor is modeled by a process, and the communication links are modeled by shared channels. The problem is solved by first assuming that all processes have access to a shared memory [Lamport 1982]. Then the solution is refined into a distributed computation by replicating global variables and having neighboring processes interact to exchange their local information. In particular, each process executes a sequence of iterations. On each iteration, a process sends its local knowledge of the topology to all its neighbors, then receives their information and combines it with its own. The computation terminates when all processes have learned the topology of the entire network.

In this paper, this type of process interaction is called a *heartbeat algorithm* since the actions of each node are like the beating of a heart: first expand, sending information out; then contract, gathering new information in. (For this specific problem, this kind of interaction has been called a *wave algorithm* [McCurley and Schneider 1986] since information spreads out in waves from a node to its neighbors, then to the neighbor's neighbors, and so on, until it reaches all processes.) The same type of algorithm can be used to solve many other problems, especially those arising from parallel iterative computations. For example, a grid computation can be programmed by partitioning the grid into blocks; each block is handled by a different process, which communicates with its neighbors to exchange boundary values. Grid computations are used to solve problems such as elliptic partial differential equations by finite differences [Fox et al. 1988].

4.1 Network Topology: Shared-Variable Solution

In the network topology problem, there is a set of n nodes, one per processor. Each node can communicate only with its neighbors, and initially all it knows about the topology is its set of neighbors. The neighbor relationship is assumed to be symmetric: For each pair of nodes p and q, p is a neighbor of q if and only if q is a neighbor of p. The problem is to compute the set *top* of all links, namely, all pairs (p, q) such that p is a neighbor of q.

Each node is represented by a process $Node[p: 1 \ldots n]$. Within each process, the neighbors of a node can be represented by a Boolean (bit) vector $links[1{:}n]$, with element $links[q]$ being true in $Node[p]$ if q is a neighbor of p. These vectors are assumed to be initialized to the appropriate values. The final topology *top* can then be represented by an adjacency matrix, with $top[p, q]$ being true if p and q are neighboring nodes. Specifically, when the computation terminates, the following predicate is to be true:

$$TOPOLOGY: (\forall p, q: 1 \le p \le n, 1 \le q \le n: top[p, q] \Leftrightarrow links_p[q])$$

In *TOPOLOGY*, $links_p$ is the neighbor set of node p.

[2] See Lamport [1982] and Elshoff and Andrews [1988] for discussions of how to handle a dynamic network for this specific problem. See Afek et al. [1987] for a general discussion of dynamic network protocols.

```
var top[1:n, 1:n] : bool := ([n*n] false)    # topology to be stored as set of all links
Node[p: 1..n]:: var links[1:n] : bool
                    # initialized so links[q] is true if q is a neighbor of Node[p]
                top[p, 1:n] := links    # fill in p'th row of top
                { top[p, 1:n] = links[1:n] }
```

Figure 8. Network topology using shared variables.

To solve the network topology problem, assume for now that *top* is global to all *Node* processes and that initially all elements of *top* are false. Then all that each process need do is store the value of its neighbor set in the appropriate row in *top*. In fact, since each process assigns to a different row of *top*, the assignments to *top* can execute in parallel without interference. This yields the shared-variable solution shown in Figure 8. When each process *Node*[*p*] terminates, *top*[*p*, *q*] is true if *q* is a neighbor of *p* and is false otherwise. The final state of the program is the union of the final state of each process; hence upon termination, *TOPOLOGY* is true as required.

4.2 Network Topology: Distributed Solution

One way to turn the solution in Figure 8 into a distributed program is to have a single process *T* compute the topology. This would be done by having each node send its value of *links* to *T*, which would copy the message into the appropriate row of *top*. This approach uses a minimal number of messages but has a drawback: How does a node communicate with *T* if *T* is not one of its neighbors? Nodes that are neighbors of *T* could forward messages from other nodes, but the resulting algorithm would be very asymmetric. A symmetric algorithm in which each node executes the same program is generally preferable since it is usually easier to develop and understand. Moreover, a symmetric algorithm is easier to modify to cope with potential failures of processors or communication links.

To get a symmetric, distributed algorithm to solve the network topology problem, each node needs to be able to compute for itself the entire topology. Initially node *p* knows about the links to its neighbors. If it asks those nodes for their neighbor links—by sending a message to and receiving a reply from each—after one round of messages, *p* will have learned about the topology within two links of it; that is, it will know its links and those of its neighbors. If every node does the same thing, after one full round each node will know about the topology within two links of it. If each node now executes another round in which it again exchanges what it knows with its neighbors, after two rounds each node will know about the topology within three links of it. In general, after *r* rounds the following will be true in each node *p*:

ROUND: $(\forall q: 1 \le q \le n: (dist(p, q) \le r) \Rightarrow$ *top*[*q*, *] filled in)

Here $dist(p, q)$ is the distance from node *p* to node *q*; that is, the length of the shortest path between them. In words, *ROUND* says that after *r* rounds, the links to neighbors of every node *q* within distance *r* of node *p* are stored in *p*'s set of links *top*.

If each node executes an adequate number of rounds, then from *ROUND* it follows that each node will have computed the entire topology. Assume for now that the diameter *D* of the network is known. (This is the distance between the farthest pair of nodes.) Then the network topology problem is solved by the program in Figure 9. As indicated, *ROUND* is the loop invariant in each node. It is made true initially by setting *top* to the node's neighbors and initializing local variable *r* to 0. It is preserved on each iteration since every node exchanges all its local information with its neighbors on each round. When a node

```
chan topology[1:n](top[1:n, 1:n] : bool)     # one channel per node
Node[p: 1..n]:: var links[1:n] : bool
                    # initialized so links[q] is true if q is a neighbor of Node[p]
                var top[1:n, 1:n] : bool := ([n*n] false)     # local view of topology
                var r : int := 0
                var newtop[1:n, 1:n] : bool
                top[p, 1..n] := links     # fill in row for my neighbors
                { top[p, 1:n] = links[1:n] ∧ r = 0 }
                { ROUND: ( ∀ q: 1 ≤ q ≤ n: (dist(p, q) ≤ r) ⇒ top[q, *] filled in }
                do r < D →
                    # send local knowledge of topology to all neighbors
                    fa q := 1 to n st links[q] → send topology[q](top) af
                    # receive their local topologies and or it with top
                    fa q := 1 to n st links[q] →
                        receive topology[p](newtop)
                        top := top or newtop
                    af
                    r := r+1
                od
                { ROUND ∧ r = D } { TOPOLOGY }
```

Figure 9. Heartbeat algorithm for network topology; first refinement

terminates, it has executed D rounds. Since by the definition of D no node is further than D links away, *top* contains the entire topology.

In Figure 9, the for-all statements (**fa** ...) are iterative statements that execute their body once for each different value of the bound variable (q) such that (**st**) the indicated Boolean expression (*links*[q]) is true. Logical "or" is used to union a neighbor's topology, which is received into *newtop*, with the local topology, which is stored in *top*. To simplify channel naming, the communication channels in Figure 9 are declared global to all the processes. Each process *Node*[p], however, only receives from its private channel *topology*[p] and only sends messages to the channels of its neighbors.

There are two problems with the algorithm in Figure 9. First, a node cannot know a priori the value of diameter D. Second, there is excessive message exchange. This is because nodes near the center of the network will know the entire topology as soon as they have executed enough rounds to have received information from nodes at the edges of the network. On subsequent rounds these nodes will not learn anything new, yet they will continue to exchange information with every neighbor. As a concrete example, consider a network that is a chain of five nodes. The node in the center will have learned the topology of the chain after two rounds. After one more round, the nodes between the centers and the ends will know the topology. After the fourth round, the nodes at the ends will finally know the topology.

Loop invariant *ROUND* and the above observations suggest how to overcome both problems. After r rounds, node p will know the topology within distance r of itself. In particular, for every node q within distance r of p, the neighbors of q will be stored in row q of *top*. Since the network is connected, every node has at least one neighbor. Thus, node p has executed enough rounds to know the topology as soon as every row in *top* has some true value. At this point, p can terminate after sharing *top* with its neighbors. This last round is necessary since some neighbors may be one further link away from the edge of the network than p, as in the above example of a chain. To avoid leaving unprocessed messages in message channels, neighboring nodes also need to let each other know when they are done and need to receive one last round of messages before terminating. In particular, after learning the

```
chan topology[1:n](sender : int, done : bool, top[1:n, 1:n] : bool)
Node[p: 1..n]:: var links[1:n] : bool
        # initialized so that links[q] true if q is a neighbor of Node[p]
    var active[1:n] : bool := links     # neighbors who are still active
    var top[1:n, 1:n] : bool := ([n*n] false)     # local view of topology
    var r : int := 0, done : bool := false
    var sender : int, qdone : bool, newtop[1:n, 1:n] : bool
    top[p, 1..n] := links     # fill in row for my neighbors
    { top[p, 1:n] = links[1:n] ∧ r = 0 ∧ ¬done}
    { ROUND ∧ (done ⇒ all rows in top are filled in) }
    do not done →
        # send local knowledge of topology to all neighbors
        fa q := 1 to n st links[q] → send topology[q](p, false, top) af
        # receive their local topologies and or it with top
        fa q := 1 to n st links[q] →
            receive topology[p](sender, qdone, newtop)
            top := top or newtop
            if qdone → active[sender] := false fi
        af
        if all rows of top have some true entry → done := true fi
        r := r+1
    od
    { ROUND ∧ all rows in top are filled in } { TOPOLOGY }
    # send topology to all neighbors who are still active
    fa q := 1 to n st active[q] → send topology[q](p, true, top) af
    # receive one message from each to clear up message queue
    fa q := 1 to n st active[q] → receive topology[p](sender, qdone, newtop) af
```

Figure 10. Heartbeat algorithm for network topology; final version.

topology, node p should exchange a round of messages only with those neighbors who did not finish on the previous round.

The final program for the network topology problem is shown in Figure 10. The comments indicate what is going on at each stage. In the program, r is now an auxiliary variable [Owicki and Gries 1976]. In particular, r is used only to facilitate specification of predicate *ROUND*. The loop invariant in the solution is *ROUND* and a predicate specifying that *done* is true only if all rows of *top* are filled in. Thus, when the loop terminates, every other process has been heard from so *top* contains the complete network topology.

The program in Figure 10 is deadlock free since sends are executed before receives and a node receives only as many messages on each round as it has active neighbors. The loop terminates in each node since the network is connected and information is propagated to each neighbor on every round. The final round of sends and receives ensures that every neighbor sees the final topology and that no unreceived messages are left buffered. If the algorithm were run only once, this would probably not be a problem (depending on what the underlying implementation does about nonempty buffers). An algorithm like this, however, might well be run periodically on a real network since the topology invariably changes over time.

The main loop in any heartbeat algorithm will have the same basic structure shown in Figures 9 and 10: Send messages to all neighbors then receive messages from them. What the messages contain and how they are processed depends on the application. For example, in a grid computation, nodes would exchange boundary values with their neighbors and the computation in each round would be to compute new values for local grid points.

Another difference between instances of heartbeat algorithms is the termination criterion and how it is checked. For the network topology problem, each node

can determine for itself when to terminate. This is because a node acquires more information on each round and the information it already has does not change. In many grid computations, the termination criterion is also based on local information—for example, the values of grid points after one round are within epsilon of their values after the previous round.

Termination cannot always be decided locally, however. For example, consider using a grid computation to label regions of an image, with each node in the grid being responsible for a block of the image. Since a region might "snake" across the image, a node might see no change on one round and get new information several rounds later. Such a computation can terminate only when there is no change anywhere after a round. Thus, the processes need to communicate with a central controller or exchange additional messages with each other.

5. PROBE/ECHO ALGORITHMS

Trees and graphs are used in many computing problems (e.g., game playing, databases, and expert systems). They are especially important in distributed computing since the structure of many distributed computations is a graph in which processes are nodes and communication links are edges.

Depth first search (DFS) is one of the classic sequential programming paradigms for visiting all the nodes in a tree or graph. In a tree, the DFS strategy for each node is to visit the children of that node and then to return to the parent. This is called depth first search since each search path reaches all the way down to a leaf; for example, the path in the tree from the root to the left-most leaf is traversed first. In a general graph —which may have cycles—the same approach is used, except nodes need to be marked as they are visited so edges out of a node are traversed only once.

This section describes the probe/echo paradigm for distributed computations on graphs. A probe is a message sent by one node to its successor; an echo is a subsequent reply. Since processes execute concurrently, probes are sent in parallel to all successors. The probe/echo paradigm is thus the concurrent programming analog of DFS. First, the probe paradigm is illustrated by showing how to broadcast information to all nodes in a network. Then, the full probe/echo paradigm is illustrated by a different algorithm for constructing the topology of a network. Additional examples of the use of the paradigm are given in Chang [1979, 1982], Dijkstra and Scholten [1980], and Francez [1980].

5.1 Broadcast in a Network

Assume as in Section 4 that there is one node per processor and that each node can communicate only with its neighbors. Suppose one initiator node i wants to broadcast a message—that is, to send some information to all other nodes. For example, i might be the site of the network coordinator, which wants to broadcast new status information to all other sites.

If every other node is a neighbor of i, broadcast would be trivial to implement: Node i would simply send a message directly to every other node. In the more realistic situation in which each node has only a few neighbors, however, the nodes need to forward information they receive until all have seen it. In short, i needs to send a probe that reaches all nodes.

If node i has a local copy *top* of the entire network topology—computed for example as shown in Figure 10—then an efficient way for i to broadcast a message is first to construct a spanning tree of the network, with itself as the root of the tree. (A spanning tree of a graph is a tree whose nodes are all those in the graph and whose edges are a subset of those in the graph [Aho et al. 1974].) For example, node i might construct a spanning tree T that contains the shortest paths from i to every other node. Given T, node i can then broadcast a message *msg* by sending *msg* together with T to all

```
chan probe[1:n](span_tree[1:n, 1:n] : bool, message_type)
Node[p: 1..n]:: var span_tree[1:n, 1:n] : bool, msg : message_type
                receive probe[p](span_tree, msg)
                fa q := 1 to n st q is a child of p in span_tree →
                    send probe[q](span_tree, msg)
                af
Initiator:: var i : int := index of node that is to initiate broadcast
            var top[1:n, 1:n] : bool    # initialized with topology of the network
            var span[1:n, 1:n] : bool, msg : message_type
            compute spanning tree of top rooted at i and store it in span
            msg := message to be broadcast
            send probe[i](span, msg)
```

Figure 11. Broadcast using a spanning tree.

its children in T. Upon receiving the message, every node examines T to determine its children in the spanning tree, then forwards both T and *msg* to all of them. The spanning tree is sent along with *msg* since nodes other than i would not otherwise know what spanning tree to use.

The full algorithm is given in Figure 11. Since T is a spanning tree, eventually the message will reach every node; moreover, each node will receive it exactly once, from its parent in T. A separate process on node i initiates the broadcast. This makes the broadcast part of the algorithm on each node symmetric.

The broadcast algorithm in Figure 11 assumes that the initiator node knows the entire topology, which it uses to compute a spanning tree that guides the broadcast. Suppose instead that each node knows only its neighbors. In this case, a message *msg* is broadcast to all nodes as follows. First, node i sends *msg* to all its neighbors. Upon receiving *msg*, a node sends it along to all its other neighbors. If the links defined by the neighbor sets happen to form a tree, the effect of this approach is the same as before. In general, however, the network will contain cycles. Thus, some node might receive *msg* from two or more neighbors. In fact, two neighbors might send the message to each other at about the same time.

It might appear that it would be sufficient to ignore multiple copies of *msg* that a node might receive. This, however, leads to the following problem. After receiving *msg* for the first time and sending it along, a node cannot know how many times to wait to receive *msg* from a different neighbor. If the node does not wait at all, extra messages could be left buffered on some of the *probe* channels. If a node waits some fixed number of times, it might wait forever unless at least that many messages are sent; even so, there might be more.

The solution to the problem of unprocessed messages is again to have a fully symmetric algorithm. In particular, when a node receives *msg* for the first time, it sends *msg* to all its neighbors, including the one from whom it received *msg*. Then the node waits to receive redundant copies of *msg* from all its other neighbors; these it ignores. The algorithm is given in Figure 12.

The broadcast algorithm using a spanning tree (Figure 11) causes $n - 1$ messages to be sent, one for each parent/child edge in the spanning tree. The algorithm using neighbor sets (Figure 12) causes two messages to be sent over every link in the network, one in each direction. The exact number depends on the topology of the network, but in general it will be much larger than $n - 1$. For example, for a tree rooted at the *Initiator* process, $2 \cdot (n - 1)$ messages will be sent; for a complete graph, $2 \cdot n \cdot (n - 1)$ will be sent. The neighbor-set algorithm does not, however, require that the initiator node know the topology and compute and

```
chan probe[1:n](message_type)
Node[p: 1..n]:: var links[1:n] : bool := neighbors of node p
                var num : int := number of neighbors of p, msg : message_type
                receive probe[p](msg)
                # send msg to all neighbors
                fa q := 1 to n st links[q] → send probe[q](msg) af
                # receive num-1 redundant copies of msg
                fa q := 1 to num-1 → receive probe[p](msg) af
Initiator:: var i : int := index of node that is to initiate broadcast
            var msg : message_type := message to be broadcast
            send probe[i](msg)
```

Figure 12. Broadcast using neighbor sets

disseminate a spanning tree. Instead, a spanning tree is constructed dynamically; it consists of the links along which the first copies of *msg* are sent. Also, the messages are shorter in the neighbor-set algorithm since the spanning tree (n^2 bits) need not be sent in each message.

Both broadcast algorithms assume the topology of the network does not change. In particular, neither works correctly if there is a processor or communication link failure while the algorithm is executing. If a node fails, obviously it cannot receive the message being broadcast; if a link fails, it might or might not be possible to reach the nodes connected by the link. Several people have investigated the problem of reliable or fault-tolerant broadcast, which is concerned with ensuring that every functioning and reachable processor receives the message being broadcast and that all agree upon the same value. For example, Schneider et al. [1984] present an algorithm for fault-tolerant broadcast in a tree, assuming that a failed processor stops executing and that failures are detectable (i.e., that failures are fail stop [Schlichting and Schneider 1983]). On the other hand, Lamport et al. [1982] show how to cope with failures that can result in arbitrary behavior (i.e., so-called Byzantine failures).

5.2 Network Topology Revisited

Section 4 presented an algorithm for computing the topology of a network by starting with a shared-memory algorithm, then generating multiple copies of the shared data. In this section, the same problem is solved in a different manner. In particular, one node first gathers the local topology data of every other node, then disseminates the full topology back to the other nodes. The topology is gathered in two phases. First, each node sends a probe to its neighbors, much as in Figure 12. Later, each node sends an echo containing local topology information back to the node from which it received the first probe. Eventually, the initiating node has gathered all the echoes. It can then broadcast the complete topology using either the algorithm in Figure 11 or the one in Figure 12.

Assume for now that the topology of the network is acyclic; since it is an undirected graph, this means the structure is a tree. Let node i be the node that initiates a topology computation. Then the topology is gathered as follows. First, i sends a probe to all its neighbors. When these nodes receive a probe, they send it to all their other neighbors, and so on. Thus, probes propagate through the tree. Eventually they will reach leaf nodes. Since these nodes have no other neighbors, they begin the echo phase. In particular, each leaf sends an echo containing its neighbor set to its parent in the tree. Upon receiving echoes from each of its children, a node combines them and its own neighbor set and echoes this information to its parent. Eventually the root node will receive echoes from all its children. The union of these will contain the entire topology since the initial

```
const source = i     # index of node that initiates the algorithm
chan probe[1:n](sender : int)
chan echo[1:n](links[1:n, 1:n] : bool)     # contents are part of the topology
chan finalecho(links[1:n, 1:n] : bool)     # final echo to Initiator
Node[p: 1..n]:: var links[1:n] : bool := neighbors of node p
                var localtop[1:n, 1:n] : bool := ([n*n] false)
                localtop[p, 1:n] := links     # put neighbor set in localtop
                var newtop[1:n, 1:n] : bool
                var parent : int     # will be node from whom probe is received
                receive probe[p](parent)
                # probe on to to all other neighbors, who are p's children
                fa q := 1 to n st links[q] and q ≠ parent → send probe[q](p) af
                # receive echoes for all children and union them into localtop
                fa q := 1 to n st links[q] and q ≠ parent →
                     receive echo[p](newtop); localtop := localtop or newtop
                af
                if p = source → send finalecho(localtop)     # this node is the root
                [] p ≠ source → send echo[parent](localtop)
                fi
Initiator:: var top[1:n, 1:n] : bool     # network topology as set of links
            send probe[source](source)
            receive finalecho(top)
```

Figure 13. Probe/echo algorithm for topology of a tree.

probe will reach every node and every echo contains the neighbor set of the echoing node together with those of its descendants in the tree.

The full probe/echo algorithm for gathering the network topology in a tree is shown in Figure 13. The probe phase is essentially the broadcast algorithm from Figure 12, except that no message is broadcast; probe messages merely indicate the identity of the sender. The echo phase returns local topology information back up the tree. In this case, the algorithms for the nodes are not fully symmetric since the instance of *Node*[*p*] executing on node *i* needs to know to send its echo to the *Initiator*. After *Initiator* receives the final topology into *top*, it can broadcast the topology back to the other nodes using the algorithm in either Figure 11 or 12.

To compute the topology of a network that contains cycles, the above algorithm is generalized as follows. After receiving a probe, a node sends it on to all its other neighbors, then waits for an echo from each. Because of cycles and because nodes execute concurrently, however, two neighbors might send each other probes at about the same time. Probes other than the first one can be echoed immediately. In particular, if a node receives a subsequent probe while waiting for echoes, it immediately sends an echo containing a null topology (this is sufficient since the local neighbor set of the node will be contained in the echo sent in response to the first probe). Eventually, a node will receive an echo in response to every probe it sends. At this point, it echoes the union of its neighbor set and the echoes it received.

The general probe/echo algorithm for computing the network topology is shown in Figure 14. Because a node might receive subsequent probes while waiting for echoes, the two types of messages are merged into one channel. (If they came in on separate channels, a node would have to use **empty** and polling to know when to receive from a channel.)

The correctness of the algorithm in Figure 14 results from the following facts. Since the network is connected, every node eventually receives a probe. Deadlock is avoided since every probe is echoed—the first one just before a *Node* process terminates, others while node is waiting to receive echoes in response to all its probes (this avoids leaving messages buffered on the *probe_echo* channels). The last echo sent by a node

```
const source = i     # index of node that initiates the algorithm
type kind = enum(PROBE, ECHO)
chan probe_echo[1:n](kind, sender : int, links[1:n, 1:n] : bool)
chan finalecho(links[1:n, 1:n] : bool)     # final echo to Initiator
Node[p: 1..n]:: var links[1:n] : bool
                    # initialized so that links[q] is true if q is a neighbor of Node[p]
                var localtop[1:n, 1:n] : bool := ([n*n] false)
                localtop[p, 1:n] := links     # put neighbor set in localtop
                var newtop[1:n, 1:n] : bool
                var first : int     # node from whom first probe is received
                var k : kind, sender : int
                var need_echo : int := number of neighbors - 1
                receive probe_echo[p](k, first, newtop)     # first message will be a probe
                # send probe on to to all other neighbors
                fa q := 1 to n st links[q] and q ≠ parent → send probe_echo[q](k, p, ∅) af
                do need_echo > 0 →
                    # receive echoes or probes from neighbors
                    receive probe_echo[p](k, sender, newtop)
                    if k = PROBE → send probe_echo[sender](ECHO, p, ∅)
                    [] k = ECHO → localtop := localtop or newtop; need_echo := need_echo-1
                    fi
                od
                if p = source → send finalecho(localtop)
                [] p ≠ source → send probe_echo[first](ECHO, p, localtop)
                fi
Initiator:: var top[1:n, 1:n] : bool     # network topology as set of links
            send probe[source](PROBE, source, ∅)     # no topology sent with probes
            receive finalecho(top)
```

Figure 14. Probe/echo algorithm for topology of a network.

contains its local neighbor set. Hence, the union of the neighbor sets eventually reaches *Node*[*i*], which sends the topology to the *Initiator*. As with the algorithm in Figure 12, the links along which first probes are sent form a (dynamically computed) spanning tree; the network topology is echoed back up this spanning tree, with the echo from a node containing the topology of the subtree rooted at that node.

This algorithm for computing the topology of a network requires fewer messages than the heartbeat algorithm in Figure 10. Two messages are sent along each link that is an edge in the spanning tree of first probes—one for the probe and another for the echo. Other links carry four messages—one probe and one echo in each direction. To disseminate the topology from the *Initiator* back to all nodes using the broadcast algorithm in Figure 11 would require another n messages. In any event, the number of messages is proportional to the number of links. For computations that disseminate or gather information on graphs, probe/echo algorithms are thus more efficient than heartbeat algorithms. In contrast, heartbeat algorithms are appropriate and necessary for many parallel iterative algorithms in which nodes need to exchange information until they converge on an answer.

6. BROADCAST ALGORITHMS

In most local area networks, processors share a common communications channel such as an Ethernet or token ring. In this case, each processor is directly connected to every other one. In fact, such communications networks often support a special network primitive called *broadcast*, which transmits a message from one processor to all others. Whether supported by the communications hardware or not, broadcast is a useful programming technique.

Let $P[1{:}n]$ be an array of processes and let $ch[1{:}n]$ be an array of channels, one

per process. Then a process $P[i]$ broadcasts a message m by executing

broadcast $ch(m)$

Execution of **broadcast** places one copy of m on each channel $ch[1{:}n]$, including that of $P[i]$. The effect is thus the same as executing n send statements in parallel, with each sending m to a different channel. Process i receives a message from its private channel $ch[i]$ by executing **receive** as usual. The **broadcast** primitive is not assumed to be indivisible. In particular, messages broadcast by two processes A and B could be seen by two other processes C and D in different orders. (See Birman and Joseph [1987] for a discussion of how to implement totally ordered broadcast, which simplifies many algorithms.)

Broadcast can be used to disseminate or gather information; for example, it is often used to exchange processor state information in local area networks. Broadcast can also be used to solve many distributed synchronization problems [Schneider 1982]. This section illustrates the power of broadcast by developing a distributed implementation of semaphores. The basis for distributed semaphores—and many other decentralized synchronization protocols—is a total ordering of communication events. Thus, the next section describes how to implement logical clocks, then shows how to use such clocks to order events [Lamport 1978].

6.1 Logical Clocks and Event Ordering

Processes in a distributed program execute local actions and communication actions. Communication actions are sending and receiving messages. These affect the execution of other processes since they communicate information and are the basic synchronization mechanism. Communication actions are thus the significant *events* in a distributed program. Hence, below, the term event refers to execution of send and receive statements.

If two processes A and B are executing local actions, there is no way to know the relative order in which the actions are executed. If A sends a message to B, however, the send action in A must happen before the corresponding receive action in B. If B subsequently sends a message to process C, the send action in B must happen before the receive action in C. Moreover, since the receive action in B happens before the send action in B, there is a total ordering between the four communication actions: The send by A happens before the receive by B, which happens before the send by B, which happens before the receive by C. "Happens before" is thus a transitive relation between causally related events.

There is a total ordering between events that causally affect each other, as described above. There is, however, only a partial ordering between the entire collection of events in a distributed program. This is because unrelated sequences of events—for example, communications between different sets of processes—might occur before, after, or concurrently with each other.

If there were a single, central clock, communication actions could be totally ordered by giving each event a unique timestamp. In particular, when a process sends a message it could read the clock and append the clock value to the message. When a process receives a message, it could read the clock and record the time at which the receive event occurred. Assuming the granularity of the clock is such that it "ticks" between any send and the corresponding receive, an action that happens before another will thus have an earlier timestamp. Moreover, if processes have unique identities, communication actions could be totally ordered by, for example, using the smallest process identity to break ties if unrelated actions in two processes happen to have the same timestamp.

Unfortunately, it is quite restrictive to assume the existence of a single, central clock. In a local area network, for example, each processor has its own clock. If

these were perfectly synchronized, the local clocks could be used for timestamps. Physical clocks are never perfectly synchronized, however. Clock synchronization algorithms exist for keeping two clocks fairly close to each other, but perfect synchronization is impossible [Marzullo and Owicki 1983]. Thus, physical clocks have to be simulated.

A *logical clock* is a simple integer counter that is incremented when events occur. Let each process have a logical clock and assume that every message contains a timestamp. The logical clocks are then incremented according to the following logical clock update rules:

Let *lc* be the logical clock in process *A*.

(1) When *A* sends or broadcasts a message, it sets the timestamp in the message to the current value of *lc*, then increments *lc* by 1.

(2) When *A* receives a message with timestamp *ts* from any process *B*, it sets *lc* to the maximum of *lc* and *ts* + 1, then increments *lc* by 1.

Since *lc* is increased after every event, every message sent by *A* will have a different timestamp and these values will increase in the order in which the messages were sent. Since a receive event sets *lc* to be larger than the timestamp in the received message, the timestamp in any message subsequently sent by *A* will have a larger timestamp.

Using logical clocks, a clock value can be associated with each event as follows. For a send event, the clock value is the timestamp in the message; that is, the local value of *lc* at the start of the send. For a receive event, the clock value is the value of *lc* after it is set to be at least as big as *ts* + 1 but before it is incremented. The above rules for updating logical clocks ensure that if event *a* happens before event *b*, the clock value associated with *a* will be smaller than that associated with *b*. This induces a partial ordering on the set of causally related events in a program. If each process has a unique identity, then a total ordering between all events results from using the smaller process identity as a tie breaker in case two events happen to have the same timestamp.

6.2 Distributed Semaphores

Semaphores are normally implemented using shared variables and are normally used for synchronizing access to other shared variables. They can be implemented in a message-based program using a server process (active monitor) using the techniques shown in Section 3.1. They can also be implemented in a distributed way as shown below.

A semaphore *s* is an abstract data type accessed by means of two operations: **P** and **V**. These operations are synchronized so that at all times they maintain the following *semaphore invariant*: The number of completed **P** operations is at most the number of completed **V** operations plus the semaphore's initial value.

In a shared-variable program, *s* is usually represented by a nonnegative integer. Execution of **V**(*s*) increments *s* as an atomic action; execution of **P**(*s*) delays until *s* is positive then decrements it, again as an atomic action. A different technique, however, is needed in a distributed program for representing the value of a semaphore and maintaining the semaphore invariant. In particular, what is required are a way to count **P** and **V** operations and a way to delay **P** operations. Moreover, the processes that "share" a semaphore need to cooperate so they maintain the semaphore invariant even though the program state is distributed.

These requirements can be met by having processes broadcast messages when they want to execute **P** and **V** operations and by having them examine the messages they receive to determine when to proceed. In particular, each process has a local message queue *mq* and a logical clock *lc*. To simulate execution of a **P** or **V** operation, a process broadcasts a message to all the user processes, including itself. The message contains the sender's identity, a tag (P or V), and a timestamp. The timestamp in every copy

of the message is the current value of *lc*, which is updated according to the logical clock update rules.

When a process receives a P or V message, it stores the message in its message queue *mq*. This queue is kept sorted in increasing order of the timestamps in the messages; sender identities are used to break ties. Assume for the moment that every process receives broadcast messages in the same order and in increasing order of timestamps. Then every process would know exactly the order in which P and V messages were sent. Thus, each could count the number of corresponding **P** and **V** operations and maintain the semaphore invariant.

Unfortunately, it is unrealistic to assume that broadcast is an atomic operation. Two messages broadcast by two different processes might be received by others in different orders. Moreover, a message with a smaller timestamp might be received after a message with a larger timestamp. Different messages broadcast by one process, however, will be received by the other processes in the order they were broadcast by the first process; these messages will also have increasing timestamps. This is because execution of **broadcast** is the same as concurrent execution of **send**—which is assumed to provide ordered, reliable delivery—and because a process increases its logical clock after every communication event.

The fact that two messages from a process are ordered and have increasing timestamps provides a way to make synchronization decisions. Suppose a process's message queue *mq* contains a message *m* with timestamp *ts*. Then once the process has received a message with a larger timestamp from every other process, it is assured that it will never see a message with a smaller timestamp. At this point, message *m* is said to be *fully acknowledged*. Moreover, once *m* is fully acknowledged, then every other message in front of it in *mq* will also be fully acknowledged since they all have smaller timestamps. Thus, the part of *mq* containing fully acknowledged messages is a *stable prefix*: No new messages will ever be inserted into it.

Whenever a process receives a P or V message, it broadcasts an acknowledgment (ACK) message. Acknowledgments are broadcast rather than merely sent to the sender of the P or V message so that every process sees the acknowledgment. The ACK messages have timestamps as usual, but they are not stored in the message queues nor are they themselves acknowledged. They are used simply to determine when a message in *mq* has become fully acknowledged.

To complete the implementation of distributed semaphores, each process simulates the execution of P and V messages stored in the stable prefix of *mq* in the order in which the messages are stored in *mq*. In particular, each process keeps local counters *nP* and *nV* of the number of fully acknowledged **P** and **V** operations it has processed. (Actually, only one counter is needed and would avoid potential overflow; two are used here to simplify specification of the loop invariant.) When a V message becomes fully acknowledged, the process increments *nV*. When a P message becomes fully acknowledged and $nV > nP$, the process increments *nP*. After incrementing the appropriate counter, the process can delete the V or P message from its message queue. In short, each process maintains the following predicate, which is its loop invariant:

DSEM: nV = number of fully acknowledged V messages $\wedge$
nP = number of fully acknowledged P messages such that $nV \geq nP \wedge$
mq is totally ordered by the timestamps in V and P messages

The different processes might be at different stages in handling P and V messages—since messages might become fully acknowledged in different orders—but every process will handle fully acknowledged messages in the same order.

The full algorithm for distributed semaphores is given in Figure 15. The *User* processes initiate **V** and **P** opera-

```
type kind = enum(V, P, ACK)
chan sem[1:n](sender: int, kind, timestamp: int)
chan go[1:n](timestamp : int)
User[i: 1..n]:: var lc : int := 0      # logical clock
                var ts : int           # timestamp in go messages
                # execute a V operation
                broadcast sem(i, V, lc); lc := lc+1
                ...
                # execute a P operation
                broadcast sem(i, P, lc); lc := lc+1
                receive go[i](ts); lc := max(lc, ts+1); lc := lc+1
                ...
Helper[i: 1..n]:: var mq : queue of (int, kind, int)   # ordered by timestamps
                  var lc : int := 0                    # logical clock
                  var nV : int := 0, nP : int := 0     # semaphore counters
                  var sender : int, k : kind, ts : int # values in messages
                  do true → { loop invariant DSEM }
                       receive sem[i](sender, k, ts); lc := max(lc, ts+1); lc := lc+1
                       if k = P or k = V →
                            insert (sender, k, ts) at appropriate place in mq
                            broadcast sem(i, ACK, lc); lc := lc+1
                       [] k = ACK →
                            record that another ACK has been seen
                            fa fully acknowledged V messages →
                                 remove the message from mq; nV := nV+1
                            af
                            fa fully acknowledged P messages st nV > nP →
                                 remove the message from mq; nP := nP+1
                                 if sender = i → send go[i](lc); lc := lc+1 fi
                            af
                       fi
                  od
```

Figure 15. Distributed semaphores algorithm.

tions by broadcasting messages on the *sem* channels. The *Helper* processes implement the **V** and **P** operations. There is one *Helper* for each *User*. Each receives messages from the appropriate *sem* channel, manages its local message queue, broadcasts ACK messages, and tells its *User* process when to proceed after a **P** operation. As shown, each process also maintains a logical clock, which it uses to place timestamps on messages.

Distributed semaphores can be used to synchronize processes in a distributed program in essentially the same way they are used in shared-variable programs. For example, they can be used to solve mutual exclusion problems such as locking files or database records [Schneider 1980]. The same basic approach—broadcast messages and ordered queues—can also be used to solve additional problems. For example, Schneider [1982] presents a broadcast algorithm to implement guarded input/output commands of CSP [Hoare 1978] (although more efficient solutions exist for that problem [Bernstein 1980; Silberschatz 1979]). Also, Section 8 mentions how broadcast can be used to coordinate the actions of replicated file servers. Broadcast, however, does not scale well to interactions between large numbers of processes since every one has to handle every message.

When broadcast algorithms are used to make synchronization decisions, every process must participate in every decision. In particular, a process cannot determine when a message is fully acknowledged until it hears from every other process. Thus, a basic algorithm such as the one in Figure 15 needs to be modified if it is to cope with failures. Schneider [1982] shows how this can be accomplished. That paper also describes

how to reintegrate a repaired processor and process into an ongoing algorithm.

7. TOKEN-PASSING ALGORITHMS

This section illustrates yet another communication pattern: token passing between processes. A token is a special kind of message that can be used either to convey permission to take an action or to gather state information. Token passing is illustrated by presenting solutions to two additional synchronization problems. The next section develops a simple, distributed solution to the critical section problem. The two succeeding sections develop algorithms for detecting when a distributed computation has terminated. Token passing is also the basis for several other algorithms; for example, it is used for fair conflict resolution in Chandy and Misra [1984] and for determining global states in Chandy and Lamport [1985]. Section 8.1 describes how tokens can be used to synchronize access to replicated files.

7.1 Distributed Mutual Exclusion

The *critical section problem* is a classic synchronization problem concerned with ensuring that at most one process at a time executes code that accesses a shared resource. Although the problem arises primarily in shared-variable programs, it also arises in distributed programs. Moreover, a solution to the critical section problem is often a component of a solution to a larger problem such as ensuring consistency in a distributed file or database system (see Section 8.1).

One way to solve the critical section problem is to use an active monitor that grants permission to access the critical section. For many problems, such as implementing locks on nonreplicated files, this is the simplest and most efficient approach. At the other extreme, the critical section problem can be solved using distributed semaphores, implemented as shown in Section 6.2. That approach yields a decentralized solution in which no one process has a special role, but it requires exchanging a large number of messages for each semaphore operation since broadcasts have to be acknowledged. (More efficient broadcast-based approaches are described in Lamport [1978], Ricart and Agrawala [1981], Maekawa [1985], and Suzuki and Kasami [1985]; these approaches are also described in books by Raynal [1986] and Maekawa et al. [1987].

Here a token ring is used to solve the problem in a third way [LeLann 1977]. The solution is decentralized, like one using distributed semaphores, but it requires the exchange of far fewer messages. Moreover, the basic approach can be generalized to solve other synchronization problems not easily solved in other ways.

Let $P[1:n]$ be a collection of processes that contain critical and noncritical sections of code. As mentioned, the critical sections access a shared resource; hence at most one process at a time is permitted to execute its critical section. The noncritical sections access noncritical resources and hence can execute concurrently. The task is to develop entry and exit protocols that the processes execute before and after their critical sections. These protocols must ensure that critical sections execute with mutual exclusion. They should also avoid deadlock and unnecessary delay and should ensure eventual entry (fairness).

Entry to the critical section will be controlled by means of a circulating token. In particular, let *Helper*$[1:n]$ be a collection of additional processes, one per $P[1:n]$. The helpers form a ring and share one token, possession of which signifies permission for the corresponding process to execute its critical section. The token circulates between the helpers, being passed from *Helper*[1] to *Helper*[2] to *Helper*[3], and so on, to *Helper*$[n]$, which passes it back to *Helper*[1]. When *Helper*$[i]$ receives the token, it checks to see whether its client $P[i]$ wants to enter its critical section. If not, *Helper*$[i]$ passes the token on. Otherwise, *Helper*$[i]$ tells $P[i]$ that it may enter its critical section, then waits until $P[i]$ exits; at this point

```
chan token[1:n]( )
chan enter[1:n]( ), go[1:n]( ), exit[1:n]( )
Helper[i: 1..n]:: do true →   {DMUTEX}
                      receive token[i]( )              # acquire the token
                      if not(empty(enter[i])) →        # P[i] wants to enter
                           receive enter[i]( ); send go[i]( )
                           receive exit[i]( )
                      fi
                      send token[i mod n + 1]( )       # pass the token on
                  od
P[i: 1..n]:: do true →
                  send enter[i]( )     # entry protocol
                  receive go[i]( )
                  critical section
                  send exit[i]( )      # exit protocol
                  non-critical section
             od
```

Figure 16. Mutual exclusion with a token ring.

Helper[*i*] passes the token on. Thus, the processes cooperate to ensure that the following predicate is always true:

DMUTEX: (∀ *i*: 1 ≤ *i* ≤ *n*:
P[*i*] is in its critical section
⇒ *Helper*[*i*] has the token) ∧
there is exactly one token

The full solution is shown in Figure 16. The token ring is represented by an array of *token* channels, one per *Helper*. For this problem, the token itself carries no data so it is represented by a "null" message. The other channels are used for communication between each client *P*[*i*] and its server *Helper*[*i*]. The client/server pairs communicate as in Figure 2, with **empty** being used to determine whether *P*[*i*] wishes to enter its critical section.

The solution in Figure 16 is fair—assuming processes eventually exit critical sections. This is because the token continuously circulates, and when *Helper* has it, *P*[*i*] is permitted to enter if it wants to do so. As programmed, the token moves continuously between the helpers. This is, in fact, what happens in a physical token-ring network. In a software token ring, however, it is probably best to add some delay in each helper so that the token moves more slowly around the ring.

Again, this algorithm does not work if failures occur. In particular, every *Helper* process must continuously pass the token, and the token must not be lost. Since control is distributed, however, the algorithm can once again be modified to cope with failures. In particular, LeLann [1977] describes how to bypass some node on the ring if it should fail and how to regenerate the token. LeLann's method requires knowledge of maximum communication delays and of process identities. More recently, Misra [1983] has developed an algorithm that overcomes these requirements by using two tokens that circulate around the ring in opposite directions.

7.2 Termination Detection in a Ring

It is trivial to detect when a sequential program has terminated. It is equally simple to detect when a concurrent program on a single processor has terminated: Every process is blocked or terminated and no I/O operations are pending. It is not at all simple, however, to detect when a distributed program has terminated. This is because the global state is not visible to any one processor. Moreover, there may be messages in transit between processors.

The problem of detecting when a distributed computation has terminated can be solved in several ways. For example, Dijkstra and Scholten [1980], Francez [1980], and Misra and Chandy [1982]

present probe/echo algorithms for different kinds of computations; also, Rana [1983] and Morgan [1985] show how to use logical clocks and timestamps. This section develops a token-passing algorithm for detecting termination, assuming all communication between the processes goes around a ring [Dijkstra et al. 1983]. The next section generalizes the algorithm for a complete communication graph [Misra 1983]. In both cases, token passing is used to signify state changes. (See Raynal [1988b] for a description and comparison of several of these algorithms; see also Chandrasekaran and Venkatesan [1990] for a message-optimal algorithm that combines the probe/echo and token-passing paradigms.)

Let $P[1{:}n]$ be the process in some distributed computation, and let $ch[1{:}n]$ be an array of communication channels, one per process. Assume that the computation is such that the interprocess communication in the computation forms a ring. In particular, process $P[i]$ receives messages only from channel $ch[i]$ and $P[i]$ sends messages only to channel $ch[i \textbf{ mod } n + 1]$. As usual, it is assumed that messages from a process are received by its neighbor in the ring in the order in which they were sent.

At any point in time, each process $P[i]$ is active or idle. Initially, every process is active. It is idle if it has terminated or is delayed at a **receive** statement. (If a process is temporarily delayed while waiting for an I/O operation to terminate, it is considered to be active since it has not terminated and will eventually be awakened.) After receiving a message, an idle process becomes active. Thus, a distributed computation has terminated if two conditions hold:

DTERM: every process is idle and
no messages are in transit

A message is in transit if it has been sent but not yet delivered to the destination channel. The second condition is necessary since when the message is delivered, it could awaken a delayed process.

The task is to superimpose a termination detection algorithm on an arbitrary distributed computation, subject only to the above assumption that the processes in the computation communicate in a ring. Clearly termination is a property of the global state—that is, the union of the states of individual processes plus the states of the message channels. Thus, the processes have to communicate with each other to determine if the computation has terminated.

To detect termination, let there be one token, which is a special message that is not part of the computation proper. The process that holds the token passes the token on when it becomes idle. (If a process has terminated its computation, it is idle with respect to the distributed computation but continues to participate in the termination-detection algorithm. In particular, the process passes the token on and ignores any regular messages it receives.)

The token is passed using the same ring of communication channels the computation uses. For example, $P[1]$ passes it to $P[2]$ by sending a message to channel $ch[2]$. When a process receives the token, it knows the sender was idle at the time it sent the token. Moreover, when a process receives the token it has to be idle since it is delayed receiving from its channel and will not become active again until it receives a regular message that is part of the computation proper. Thus, upon receiving the token, a process passes it on to its neighbor by sending it to the neighbor's channel.

The question now is how to detect that the entire computation has terminated. When the token has made a complete circuit of the communication ring, it means every process was idle at some point. But how can the holder of the token determine if all other processes are still idle and there are no messages in transit?

Suppose one process, $P[1]$ say, initially holds the token and hence initiates the termination-detection algorithm when it becomes idle. Suppose the token gets back to $P[1]$ and $P[1]$ has been *continuously idle* since it first passed the token to $P[2]$. Then $P[1]$ can conclude the

computation has terminated. This is because the token goes around the same ring regular messages do and messages are delivered in the order in which they are sent. Thus, when the token gets back to $P[1]$ there cannot be any regular messages either queued or in transit. In essence, the token has "flushed" the channels clean, pushing all regular messages ahead of it.

The algorithm and its correctness can be made more precise as follows. First, associate a color with every process: blue (cold) for idle and red (hot) for active. Initially, all processes are active, so all are colored red. When a process receives the token, it is idle, so it colors itself blue and passes the token on. If the process later receives a regular message, it colors itself red. Thus, a process that is blue has become idle, passed the token on, and remained idle since passing the token.

Second, associate a value, *token*, with the token to indicate how many channels are empty if $P[1]$ is still idle. When $P[1]$ becomes idle, it colors itself blue, sets *token* to 0, then sends *token* to $P[2]$. When $P[2]$ receives the token, it is idle. Hence, $P[2]$ colors itself blue, increments *token* (to 1), and sends the token to $P[3]$. Each process $P[i]$ in turn colors itself blue and increments *token* before passing it on.

These token-passing rules are listed in Figure 17. As indicated, the rules ensure the invariance of predicate *RING*. This follows from the fact that if $P[1]$ is blue, it has not sent any regular messages since sending the token, and hence there are no regular messages in any channel up to where the token resides. Moreover, all these processes have remained idle since passing the token on. Thus, if $P[1]$ is still blue when the token gets back to it, all processes are blue and all channels are empty. Hence, $P[1]$ can announce that the computation has terminated.

7.3 Termination Detection in a Graph

The previous section made a simplifying assumption: that all communication goes around a ring. In general, the communication structure of a distributed computation will form an arbitrary directed graph. The nodes of the graph are the processes in the computation; the edges represent communication paths. There is an edge from one process to another if the first process sends to a channel from which the second receives.

Suppose the communication graph is *complete*: There is one edge from every process to every other. In particular, there are n processes $P[1:n]$ and n channels $ch[1:n]$. Each process $P[i]$ receives from its private input channel $ch[i]$; every other process can send messages to $ch[i]$. Under these assumptions, the previous termination detection algorithm can be extended as described below. The resulting algorithm is adequate to detect termination in any network in which there is a direct communication path from each processor to every other. It can readily be extended to arbitrary communication graphs and multiple channels [Misra 1983].

Detecting termination in a complete graph is more difficult than in a ring since messages can arrive over any edge. For example, consider the complete graph of three processes $P[1:3]$ shown in Figure 18. Suppose the processes pass a token only from $P[1]$ to $P[2]$ to $P[3]$ and back to $P[1]$. Suppose $P[1]$ holds the token; when it becomes idle, it passes the token to $P[2]$. When $P[2]$ becomes idle, it passes the token to $P[3]$. But before $P[3]$ receives the token and becomes idle, it could send a regular message to $P[2]$. Thus, when the token gets back to $P[1]$, it cannot conclude that the computation has terminated even if it has remained continuously idle.18

The key to the ring algorithm in Figure 17 is that all communication uses the same channels and hence the token flushes out regular messages. In particular, the token traverses every edge of the ring. To extend that algorithm to a complete graph, it is sufficient to ensure that the token traverses every edge of the graph, which means that it visits every process multiple times. If *every* process

```
{ RING: P[1] blue ⇒ ( P[1] ... P[1+token] blue ∧ ch[2] ... ch[1 + token mod n] empty ) }
actions of P[1] when it first becomes idle:
    color[1] := blue; token := 0; send ch[2](token)
actions of P[i: 1..n] upon receiving a regular message:
    color[i] := red
actions of P[i: 2..n] upon receiving the token:
    color[i] := blue; token := token+1; send ch[i mod n + 1](token)
actions of P[1] upon receiving the token:
    if color[1] = blue → announce termination and halt fi
    color[1] := blue; token := 0; send ch[2](token)
```

Figure 17. Termination detection in a ring

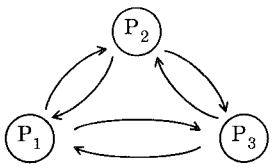

Figure 18. Complete communication graph.

has remained continuously idle since it first saw the token, the computation has terminated.

As before, each process is colored red or blue, with all processes initially red. When a process receives a regular message, it colors itself red. When a process receives the token, it is blocked, waiting to receive the next message on its input channel. (Again, if a process terminates its regular computation, it continues to handle token messages.) Hence the process colors itself blue—if it is not already blue—and passes the token on.

Any complete directed graph contains a cycle that includes every edge. Let c be a cycle in the communication graph and let nc be its length. Each process keeps track of the order in which its outgoing edges occur in c. Upon receiving the token along one edge in c, a process sends it out over the next edge in c. This ensures that the token traverses every edge of the communication graph.

Also as before, the token carries a value that indicates how many times in a row it has been passed on by idle processes. As the above example illustrates, however, in a complete graph a process that was idle might become active again, even if $P[1]$ remains idle. This requires a different set of token-passing rules and a different invariant predicate.

The token starts at any process and initially has value 0. When that process becomes idle for the first time, it colors itself blue then passes the token along the first edge in cycle c. Upon receiving the token, a process takes the actions shown in Figure 19. As shown, if a process is red when it receives the token—and hence was active since last seeing it—the process colors itself blue and sets the value of *token* to 0 before passing it along the next edge in c. This effectively reinitiates the termination-detection process. If, however, the process is blue when it receives the token—and hence has been continuously idle since last seeing the token—the process increments the value of *token* before passing it on.

The token-passing rules ensure the invariance of predicate *GRAPH*. Once the value of *token* gets to nc, the length of cycle c, the computation is known to have terminated. In particular, at that point the last nc channels the token has traversed were empty. Since a process only passes the token when it is idle—and since it only increases *token* if it has remained idle since last seeing the token—all channels are empty and all processes are idle. In fact, the computation had actually terminated by the time the token started its last circuit around the graph. No process could possibly know this, however, until the token has made another complete cycle around the graph

```
{ GRAPH: token has value T ⇒ ( the last T channels token was received from were empty ∧
                               all P[i] that passed it were blue when they passed it ) }
actions of P[i: 1..n] upon receiving a regular message:
    color[i] := red
actions of P[i: 1..n] upon receiving the token:
    if token = nc → announce termination and halt fi
    if color[i] = red → color[i] := blue; token := 0
    [] color[i] = blue → token := token+1
    fi
    set j to the channel corresponding to the next edge to use in cycle c
    send ch[j](token)
```

Figure 19. Termination detection in a complete graph.

to verify that all processes are still idle and that all channels are empty. Thus, the token has to circulate a minimum of two times around the cycle after any activity: once to turn processes blue, the other to verify they have remained blue.

8. REPLICATED SERVERS

The final two process-interaction paradigms involve the use of replicated servers; that is, multiple server processes that each do the same thing. Replication serves one of two purposes. First, it can increase the accessibility of data or services by having more than one process provide the same service. These decentralized servers interact to provide clients with the illusion there is just one, centralized service. Second, replication can sometimes be used to speed up finding a solution to a problem by dividing the problem into independent subproblems that are solved concurrently. This is done by having multiple worker processes share a bag of subproblems. This section illustrates both of these applications by first showing how to implement a replicated file, then developing an adaptive quadrature algorithm for numerical integration.

8.1 Decentralized Servers: Replicated Files

A simple way to increase the likelihood that a critical data file is accessible is to keep a backup copy of the file on another disk, usually one that is attached to a different machine. The user can do this manually by periodically making a backup copy of a file. Or the backup copy could be maintained automatically. In either case, however, users wishing to access the file would have to know whether the primary copy was available and if not, access the backup copy instead. (A related problem is bringing the primary copy back up to date when it becomes reaccessible.)

A third approach is for the file system to provide transparent and automatic replication. In particular, suppose there are n copies of a data file and that each is managed by a separate server process, which handles client requests to read and write the file. Each server provides an identical client interface such as that shown in Figure 7. Thus, a client sends an open request to any server and subsequently continues to converse with that server, sending it read and write requests and eventually sending it a close request. The servers themselves interact to present clients with the illusion there is a single copy of the file. The structure of this interaction pattern is shown in Figure 20.

To present clients with the illusion there is a single file, the file servers need to synchronize with each other. In particular, the *file consistency* problem has to be solved: The results of client read and write requests have to appear to be the same independent of which copy of the file is accessed. Thus, file consistency is an instance of the classic readers/writers problem [Courtois et al. 1971]: Two clients can read the file at the same time, but a client requires exclusive access

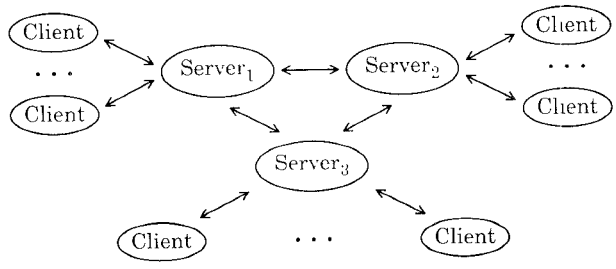

Figure 20. Replicated file server interaction pattern.

when writing the file. There are several ways to implement file consistency as described below. Here it is assumed that entire files are to be kept consistent. The same techniques can be used to ensure consistency at the level of records in files, which is more appropriate in database applications.

One way to solve the file consistency problem is to ensure that at most one client at a time can access any copy of the file. This can be implemented by, for example, the distributed solution to the critical section problem in Figure 16. When a client asks any one of the servers to open the file, that server first interacts with the others to acquire exclusive access. The server then processes the client's read and write requests. For a read request, the server reads the local copy of the file. For a write request, the server updates all copies of the file. When the client closes the file, the server releases exclusive control.

The above approach is of course more restrictive than necessary. Clients cannot be reading and writing the file at the same time, but they can read it concurrently. Assume when a client asks a server to open the file, it indicates whether it will be reading only or will be both reading and writing. To permit concurrent reading, the servers can use a variation on the token-passing algorithm for mutual exclusion (Figure 16). In particular, let there be one token that has an initial value equal to the number *ns* of file servers. Then when a client opens the file for reading, its server waits for the token, decrements it by 1, sends it to the next server (helper process actually), then handles the client's read requests. After the client closes the file, the server increments the value of the token the next time it comes around the ring. On the other hand, when a client opens the file for writing, its server waits for the token to have value *ns*, then holds onto the token while handling the client's read and write requests. After the client closes the file, the server updates all copies, then puts the token back into circulation.

The problem with the above token-passing scheme is that write requests will never get serviced if there is a steady stream of read requests. A somewhat different approach yields a fair solution. Instead of using just one token, use *ns* different tokens. Initially, each server has one token. When a client wants to read a file, its server must acquire one token; when a client wants to write a file, its server must acquire all *ns* tokens. Thus, when a server wants to write the file, it sends a message to all other servers requesting their tokens. Once it has gathered all the tokens, the server handles its client's read and write requests, propagating updates to the other servers as above. When a server wants to read the file, it can do so immediately if it holds a token. If not, it asks the other servers for one of the tokens.

This multiple-token scheme does, however, have two potential problems. First, two servers could at about the same time try to acquire all the tokens. If each is able to acquire some but not all, neither write will ever get executed. Second,

while a server is acquiring all tokens preparatory to writing the file, another server might ask for a token so it can read the file. Both problems can be overcome by using logical clocks to place timestamps on each request for a token. Then a server gives up a token if it receives a request with a timestamp earlier than the time at which it wanted to use the token. For example, if two servers want to write at about the same time, the one that initiated the write request earlier will be able to gather the tokens.[3]

An attractive attribute of using multiple tokens is that they will congregate at active servers. For example, after a server gathers all tokens to perform a write, it can continue to process write requests until some other server requires a token. Thus, if the replicated file is being heavily written by one client, the overhead of token passing can be avoided. Similarly, if the file is mostly read and only rarely updated—which is quite commonly the case—the tokens will generally be distributed among the servers and hence read requests will be able to be handled immediately.

A variation on the multiple-token scheme is *weighted voting* [Gifford 1979; Maekawa 1985; Thomas 1979]. Above, a server requires one token to read but all *ns* to write. This in essence assigns a weight of 1 to reading and a weight of *ns* to writing. Instead, a different set of weights can be used. Let *rw* be the read weight and let *ww* be the write weight. Then, if *ns* is 5, *rw* could be set to 2 and *ww* to 4. This means a server must hold at least two tokens to read the file and at least four to write the file. Any assignment of weights can be used as long as the following two requirements are met:

(1) $ww > ns/2$ (to ensure that writes are exclusive).

(2) $rw + ww > ns$ (to ensure that reads and writes exclude each other).

With weighted voting, not all copies of the file need be updated when a write is processed. It is only necessary to update *ww* copies. Then, however, it is necessary to read *rw* copies. In particular, every write action must set a timestamp on the copies to which it writes, and a read action must use the file with the most recent timestamp. By reading *rw* copies, a reader is assured of seeing at least one of the files changed by the most recent write action.

As mentioned at the start of this section, one of the rationales for replicating files is to increase availability. Yet each of the synchronization schemes above depends on every server being available and reachable. Each scheme can, however, be modified to be fault tolerant. If there is one circulating token and it is lost, it can be regenerated as described in LeLann [1977] or Misra [1983]. If there are multiple tokens and a server crashes, the other servers can interact to determine how many tokens were lost, then can hold an election to determine which one will get the lost tokens. (See Garcia-Molina [1982], Raynal [1988b], and Schneider and Lamport [1985] for descriptions of numerous election algorithms.) Finally, with weighted voting it is only necessary that $max(rw, ww)$ copies of the file are accessible, since only that many are needed to service any read or write.

Independent of the synchronization scheme, after recovery of a server or of a disk holding a copy of the file, the copy needs to be brought up to date before it can be accessed by clients. In essence, a recovered server needs to pretend it is a writer and gain write permission to the other copies of the file; it then reads an up-to-date copy into its local copy of the file and releases write permission. The server can then resume handling client requests.

[3] A variation on having *ns* tokens is to have *ns* locks, one per copy of the file. To read the file, a server acquires the lock for its local copy. To write the file, a server acquires the locks for all copies. If every server acquires the locks in the same order—and if lock requests are handled in first-come, first-served order—then the solution will be fair and deadlock free. This use of locks avoids the need to put timestamps in messages.

8.2 Replicated Workers and a Bag of Tasks: Adaptive Quadrature

Section 8.1 considered an example of data replication. Section 8.2 considers function replication. In particular, this section presents an adaptive, parallel solution to the quadrature problem for numerical integration [Grit and McGraw 1985]. The solution illustrates how to parallelize any divide-and-conquer algorithm, subject only to the requirement that subproblems be independent. (See Carriero et al. [1986], Finkel and Manber [1987], Gentleman [1981], and Thomas and Crowther [1988] for additional examples of problems that can be solved in this way and additional techniques for implementing the solution. The bag-of-tasks approach illustrated here is also used in the implementation of Multilisp [Halstead 1985] and PTRAN [Allen et al. 1988].)

The solution to the adaptive quadrature problem uses a shared channel, which contains a bag of tasks. Initially, there is one task corresponding to the entire problem to be solved. Multiple worker processes take tasks from the bag and process them, often generating new tasks—corresponding to subproblems—that are put into the bag. The computation terminates when all tasks have been processed.

In the quadrature problem, given is a continuous, nonnegative function $f(x)$. Also given are two values l and r, with $l < r$. The problem is to compute the area bounded by $f(x)$, the x axis, and vertical lines through l and r. Thus, the problem is to approximate the integral of $f(x)$ from l to r.

The typical way to approximate the area under a curve is to divide the interval $[l, r]$ into a series of subintervals, then to use a trapezoid to approximate the area of each subinterval. In particular, let $[a, b]$ be a subinterval. Then an approximation to the area under f from a to b is the area of the trapezoid with base $b - a$ and sides of height $f(a)$ and $f(b)$.

The quadrature problem can be solved either statically or dynamically. The static approach uses a fixed number of equal-sized intervals, computes the area of the trapezoid for each interval, then sums the results. This process is then repeated—typically by doubling the number of intervals—until two successive approximations are close enough to be acceptable.

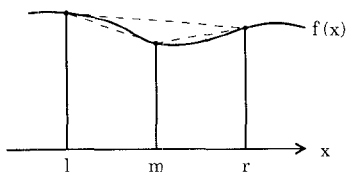

Figure 21. Quadrature problem.

The dynamic approach starts with one interval from l to r and computes the midpoint m between l and r. It then calculates the areas of three trapezoids: (1) the large one bounded by l, r, $f(l)$, and $f(r)$; (2) the small one bounded by l, m, $f(l)$, and $f(m)$; and (3) the small one bounded by m, r, $f(m)$, and $f(r)$. Figure 21 illustrates this. Next, the dynamic approach compares the area of the larger trapezoid with the sum of the areas of the two smaller ones. If these are sufficiently close, it takes the area of the larger trapezoid as an acceptable approximation of the area under f. Otherwise, the process is repeated by solving the two subproblems of computing the area from l to m and from m to r. This process is repeated recursively until the solution to each subproblem is acceptable. The approach then sums the answers to the subproblems to yield the final answer.

The dynamic approach is called *adaptive quadrature* since the solution adapts itself to the shape of the curve. In particular, in places where the curve is flat, a wide trapezoid will closely approximate the area. Hence new subproblems will not be generated. In places where the curve is changing shape, however—and especially in places where the tangent of $f(x)$ is nearly vertical—smaller and smaller subproblems will be generated as needed.

```
chan bag(a, b, fa, fb, area : real)
chan result(a, b, area : real)
Administrator:: var l, r, fl, fr, a, b, area, total : real
                other variables to record finished intervals
                fl := f(l); fr := f(r)
                area := (fl+fr)*(l+r)/2
                send bag(l, r, fl, fr, area)
                do entire area not yet computed →
                     receive result(a, b, area)
                     total := total + area
                     record that have calculated the area from a to b
                od
Worker[1:n]:: var a, b, m, fa, fb, fm : real
              var larea, rarea, tarea, diff : real
              do true → receive bag(a, b, fa, fb, tarea)
                        m := (a+b)/2; fm := f(m)
                        compute larea and rarea using trapezoids
                        diff := tarea – (larea + rarea)
                        if diff small → send result(a, b, tarea)
                        [] diff too large → send bag(a, m, fa, fm, larea)
                                            send bag(m, b, fm, fb, rarea)
                        fi
              od
```

Figure 22. Adaptive quadrature algorithm.

With either the static or dynamic approach, every subproblem is independent of the others. Hence either algorithm can readily be parallelized. Following is a parallel algorithm that uses adaptive quadrature since that approach is generally preferred.

One way to solve a parallel divide-and-conquer problem is to use one *administrator* process and several *worker* processes [Carriero et al. 1986; Gentleman 1981]. The administrator generates the first problem and gathers results. Workers solve subproblems, generating further subproblems when that is required. The workers share a single channel, *bag*, which contains the problems to be solved. In this case, a problem is characterized by five values: a left point a, a right point b, the values of $f(a)$ and $f(b)$, and the area of the trapezoid defined by these four points. (The area is included to avoid recomputation.) When a worker computes an acceptable answer to a sub-problem, it sends that answer to the administrator over channel *result*. The program terminates when the administrator detects that the area of the entire interval $[l, r]$ has been computed (or, equivalently, when *bag* is empty and all workers are idle; however, this is difficult to detect).

Figure 22 contains the algorithm for the adaptive quadrature problem. An especially interesting aspect of the algorithm is that there can be any number of worker processes. If there is only one, the algorithm is essentially an iterative, sequential algorithm. If there are more workers, subproblems can be solved in parallel. Thus, the number of workers can be tuned to match the hardware on which the algorithm executes.

The quadrature algorithm assumes all workers can receive from the same channel, *bag*. In many distributed programming languages, a channel can have only one receiver. If this is the case, the shared channel can be simulated by a server process with which the workers communicate. In fact, the administrator itself can serve this role.

9. SUMMARY

This paper has examined the use of several communication structures in distributed computations, including trees,

graphs, rings, and grids. It has also examined several process interaction paradigms that can be used to solve distributed programming problems. Each paradigm illustrates a different use of the **send** and **receive** primitives:

- In networks of filters, data flows in one direction through an acyclic graph.
- With clients and servers, a client sends a request, then waits to receive a reply; a server receives a request and eventually sends back a reply.
- In heartbeat algorithms, processes first send information out to their neighbors, then receive messages from their neighbors.
- In probe/echo algorithms, requests are forwarded along edges of a graph and turned into replies when they reach nodes at the other side of the graph.
- In broadcast algorithms, messages are sent to all other processes.
- In token-passing algorithms, additional messages (tokens) circulate among processes to carry permission or to gather information about state changes.
- Finally, with replicated servers, the server processes either interact with each other to manage replicated data or share a bag of tasks.

The examples used to illustrate the paradigms are both interesting in their own right and representative of the broad spectrum of distributed programming problems.

In addition to illustrating the different interaction paradigms, the examples have illustrated different ways in which to develop solutions to distributed programming problems:

- A filter is developed as an independent process whose output is a function of its input; filters can thus be connected into networks as long as each filter's input/output assumptions are met.
- A server is developed by specifying the operations it provides and specifying a predicate that characterizes the state of the server's local variables when it is idle. Each operation must then be implemented so the predicate is true at communication points.
- The development of the heartbeat algorithm for computing the topology of a network started with a shared variable solution, then refined the solution into one in which each process computed the topology by exchanging information with its neighbors.
- Both the heartbeat and probe/echo algorithms for the network topology problem were solved by first making simplifying assumptions, then extending the initial solutions to handle the general case. The algorithm for termination detection was also developed this way.
- For most of the examples in Sections 4–7, the starting point in solving the problem was specifying a predicate that characterized the answer. Then the actions of each process were designed so they led to a state in which the predicate was true.
- Finally, the adaptive quadrature problem illustrated how to parallelize a divide-and-conquer algorithm.

The examples have been programmed using asynchronous message passing. Hence the programs are appropriate for execution on multiple instruction, multiple data (MIMD) machines. Some of the techniques, however—such as networks of filters, heartbeat, broadcast, and replicated workers—are also applicable to programs that execute on single instruction, multiple data (SIMD) machines such as the Connection Machine [Hillis 1985]. Networks of filters and heartbeat algorithms are also appropriate for systolic machines such as the WARP [Annaratone et al. 1986].

The material in this paper is developed further in Andrews [1991]. That book contains numerous additional examples. It also illustrates how to program these interaction paradigms using the other kinds of message passing: synchronous, remote procedure call, rendezvous, and generative communication. Many of the implementation details differ, but the

basic interaction patterns and program development techniques are the same.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation grants CCR-8701516 and CDA-8822652. Michael Coffin, Irving Elshoff, Udi Manber, Ron Olsson, and Richard Schlichting provided useful comments on an earlier version of this paper. The referees' numerous, insightful comments helped significantly to improve the paper. Finally, Hector Garcia-Molina and Sal March expedited the reviewing process.

REFERENCES

AFEK, Y., AWERBUCH, B , AND GAFNI, E. 1987. Applying static network protocols to dynamic networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science* IEEE Computer Society (Los Angeles, Oct.), pp. 358-370.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

AKL, S. G. 1985. *Parallel Sorting Algorithms.* Academic Press, New York.

ALLEN, F., BURKE, M., CHARLES, P., CYTRON, R., AND FERRANTE, J. 1988. An overview of the PTRAN analysis system. *J. Parallel Distrib. Comput. 5*, 5 (Oct.), 617-640.

ANDREWS, G R. 1991. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, Calif. To be published.

ANDREWS, G. R., AND SCHNEIDER, F. B. 1983. Concepts and notations for concurrent programming. *ACM Comput. Surv. 15*, 1 (Mar), 3-43.

ANDREWS, G. R., OLSSON, R. A., COFFIN, M., ELSHOFF, I., NILSEN, K., PURDIN, T., AND TOWNSEND, G. 1988. An overview of the SR language and implementation. *ACM Trans. Program. Lang. Syst. 10*, 1 (Jan.), 51-86.

ANNARATONE, M., ARNOULD, E , GROSS, T , KUNG, H. T., LAM, M. S., MEZILCIOGLU, O., SAROCKY, K., AND WEBB, J. A. 1986 In *Proceedings of the 13th International Symposium on Computer Architecture.* Computer Science Press, Rockville, Md., pp. 346-356.

ATKINS, M. S., AND OLSSON, R. A. 1988. Performance of multi-tasking and synchronization mechanisms in the programming language SR. *Softw. Pract. Exper. 18*, 9 (Sept.), 879-895.

BAL, H. E , STEINER, J. G., AND TANENBAUM, A. S. 1989 Programming languages for distributed systems. *ACM Comput. Surv. 21*, 3 (Sept.), 261-322.

BERNSTEIN, A. J. 1980 Output guards and nondeterminism in CSP. 1980. *ACM Trans. Program Lang. Syst. 2*, 2 (Apr.), 234-238.

BIRMAN, K. P., AND JOSEPH, T. A. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb.), 47-76.

CARRIERO, N., GELERNTER, D , AND LEICHTER, J. 1986. Distributed data structures in Linda. *Thirteenth ACM Symposium on Principles of Programming Languages.* ALM SIGPLAN and SIGACT (Williamsburg, Virginia, Jan.), pp. 236-242.

CHANDRASEKARAN, S. AND VENKATESAN, S. 1990. A message-optimal algorithm for distributed termination detection. *J. Parallel Distrib. Comput. 8*, 245-292.

CHANDY, K. M., AND LAMPORT, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb.), 63-75.

CHANDY, K. M., AND MISRA, J. 1984. The drinking philosophers problem. *ACM Trans Program. Lang. Syst. 6*, 4 (Oct.), 632-646.

CHANG, E. J. H. 1979. Decentralized algorithms in distributed systems. TR CSRG-103. Ph.D. dissertation, Computer Systems Research Group, University of Toronto, October.

CHANG, E. J. H. 1982. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans Softw. Eng. 8*, 4 (Jul.), 391-401.

CLARK, D. 1985 The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles.* ACM SIGOPS (Orcas Island, Wash., Dec.), pp. 171-180.

COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. 1971. Concurrent control with "readers" and "writers " *Commun. ACM 14*, 10 (Oct.), 667-668.

DIJKSTRA, E. W 1976. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ.

DIJKSTRA, E. W., AND SCHOLTEN, C. S. 1980. Termination detection in diffusing computations. *Inf. Process. Lett. 11*, 1 (Aug.), 1-4.

DIJKSTRA, E. W., FEIJEN, W. H. J., AND VAN GASTEREN, A. J. M. 1983. Derivation of a termination detection algorithm for distributed computation. *Inf. Process. Lett. 16*, 5 (Jun.), 217-219.

ELSHOFF, I. J. P., AND ANDREWS, G. R. 1988. The development of two distributed algorithms for network topology. Tech. Rep. TR 88-13. Dept. of Computer Science, Univ. of Arizona.

FINKEL R , AND MANBER, U. 1987. DIB: A distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst. 9*, 2 (Apr.), 235-256.

FOX, G. C., JOHNSON, M. A., LYZENGA, G. A., OTTO, S. W., SALMON, J. K., AND WALKER, D. W. 1988. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ

FRANCEZ, N. 1980. Distributed termination. *ACM Trans. Program. Lang. Syst. 2*, 1 (Jan.), 42-55.

GARCIA-MOLINA, H. 1982. Elections in a distributed computing system. *IEEE Trans. Comput. C-31*, 1 (Jan.), 48-59.

GEHANI, N. H., AND ROOME, W. D. 1986. Concurrent C. *Softw. Pract. Exper. 16*, 9 (Sept.), 821-844.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan.), 80-112.

GENTLEMAN, W. M. 1981. Message passing between sequential processes: The reply primitive and the administrator concept. *Softw. Pract. Exper. 11*, 435-466.

GIFFORD, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles.* ACM SIGOPS (Pacific Grove, Calif., Dec.), pp. 150-162.

GRIT, D. H., AND MCGRAW, J. R. 1985. Programming divide and conquer on a MIMD machine. *Softw. Pract. Exper. 15*, 1 (Jan.), 41-53.

HALSTEAD, R. H. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct.), 501-538.

HILLIS, W. D. 1985. *The Connection Machine.* MIT Press, Cambridge, Mass.

HOARE, C. A. R. 1974. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct.), 549-557.

HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug.), 666-677.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in distributed systems. *Commun. ACM 21*, 7 (Jul.), 558-565.

LAMPORT, L. 1982. An assertional correctness proof of a distributed algorithm. *Sci. Comput. Program. 2*, 3 (Dec.), 175-206.

LAMPORT, L., SHOSTAK, R. AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 3*, 3 (Jul.), 382-401.

LAUER, H. C., AND NEEDHAM, R. M. 1978. On the duality of operating system structures. In *Proceedings of the 2nd International Symposium on Operating Systems.* (IRIA, Paris, Oct.). Reprinted in *Oper. Syst. Rev. 13*, 2 (Apr. 1979), 3-19.

LELANN, G. 1977. Distributed systems: Towards a formal approach. In *Proceedings of Information Processing 77.* North-Holland Publishing Co., Amsterdam, pp. 155-160.

MAEKAWA, M. 1985. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst. 3*, 2 (May), 145-159.

MAEKAWA, M., OLDEHOEFT, A. E., AND OLDEHOEFT, R. R. 1987. *Operating Systems: Advanced Concepts.* Benjamin/Cummings Publishing Co., Menlo Park, Calif.

MARZULLO, K., AND OWICKI, S. S. 1983. Maintaining the time in a distributed system. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing.* (Montreal, Aug.), pp. 295-305.

MCCURLEY, R., AND SCHNEIDER, F. B. 1986. Derivation of a distributed algorithm for finding paths in directed networks. *Sci. Comput. Program. 6*, 1 (Jan.), 1-9.

MISRA, J. 1983. Detecting termination of distributed computations using markers. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing.* ACM SIGOPS and SIGACT (Montreal, Aug.), pp. 290-294.

MISRA, J., AND CHANDY, K. M. 1982. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst. 4*, 1 (Jan.), 37-43.

MITCHELL, J. G., MAYBURY, W., AND SWEET, R. 1979. Mesa language manual, version 5.0. Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif.

MORGAN, C. 1985. Global and logical time in distributed algorithms. *Inf. Process. Lett. 20*, 4 (May), 189-194.

OWICKI, S. AND GRIES, D. 1976. An axiomatic proof technique for parallel programs. *Acta Inf. 6*, 319-340.

PETERSON, J. L., AND SILBERSCHATZ, A. 1985. *Operating Systems Concepts.* 2nd ed. Addison-Wesley, Reading, Mass.

RANA, S. P. 1983. A distributed solution of the distributed termination problem. *Inf. Process. Lett. 17*, 1 (Jul.), 43-46.

RAYNAL, M. 1986. *Algorithms for Mutual Exclusion.* The MIT Press, Cambridge, Mass.

RAYNAL, M. 1988a. *Networks and Distributed Computation.* The MIT Press, Cambridge, Mass.

RAYNAL, M. 1988b. *Distributed Algorithms and Protocols.* John Wiley & Sons, New York.

RICART, G. AND AGRAWALA, A. K. 1981. An optimal algorithm for mutual exclusion. *Commun. ACM 24*, 1 (Jan.), 9-17.

SANDBERG, R., GOLDBERG, D. KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun network filesystem. In *Proceedings of the Usenix Conference.* (June). pp. 119-130.

SCHLICHTING, R. D., AND SCHNEIDER, F. B. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug.), 222-238.

SCHNEIDER, F. B. 1980. Ensuring consistency in a distributed database system by use of distributed semaphores. In *Proceedings of International Symposium on Distributed Databases.* (Versailles, France, Mar.), pp. 183-189.

SCHNEIDER, F. B. 1982 Synchronization in distributed programs. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr.), 179-195.

SCHNEIDER, F. B., AND LAMPORT, L. 1985. Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course.* Lecture notes in *Comput. Sci. 190.* Springer-Verlag, Berlin.

SCHNEIDER, F. B., GRIES, D., AND SCHLICHTING, R. D. 1984. Fault-tolerant broadcasts. *Sci. Comput. Program. 4,* 1-15.

SILBERSCHATZ, A 1979. Communication and synchronization in distributed systems. *IEEE Trans. Softw. Engr. SE 5,* 6 (Nov.), 542-546.

SUZUKI, I., AND KASAMI, T. 1985. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst. 3,* 4 (Nov.), 344-349.

TANENBAUM, A. S. 1988. *Computer Networks, Second Edition.* Prentice-Hall, Englewood Cliffs, N.J

THOMAS, R. H. 1979. A majority consensus approach to concurrency control in multiple copy databases. *ACM Trans. Database Syst 4,* 2 (Jun.), 180-209.

THOMAS, R. H., AND CROWTHER, W. 1988. The uniform system: An approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the 1988 International Conference on Parallel Processing.* IEEE Computer Society (St. Charles, Illinois, Aug.), pp. 245-254.

THOMPSON, K. 1978. UNIX implementation. *Bell Syst. Tech. J. 57,* 6, part 2 (Jul.-Aug.), 1931-1946

Received October 1989, final revision accepted August 1990.