Interaction monitors in a distributed system

by RAJIV MALHOTRA
Burroughs Corporation
Detroit, Michigan

INTRODUCTION

By utilizing the parallelism and redundancy inherent in a network, a set of interacting processes could be made more effective. But this should not require the user to know the network details: the network should be transparent enough to allow him to treat it as a single entity rather than as a collection of computers. The concept of relocatable programs within a machine could be generalized to allow processes to be relocatable anywhere within the network. Then, several mappings could exist from a given set of processes to the set of computers in the network, with the final results being independent of the mapping used.

As a step toward such generality, one needs adequate constructs to conceptualize and represent the interactions among processes independently of their geographical locations. It is, therefore, desirable to have some "atoms" of interaction that are equally valid for processes in separate computers and for processes in the same computer. The implementation philosophy for constructs which perform this interaction should be such as to allow easy additions of further constructs.

There are two aspects of a set of asynchronous processes in a network: the dynamic creation and deletion of processes and the interaction among these processes once they have started. In this paper, we deal only with the second of these aspects.

Existing IPC (interprocess communication) schemes merely allow messages to be transferred between processes. Although this facility is necessary, it is not always a convenient programming tool. Accordingly, some higher-level (or function-oriented) protocols have been introduced to provide specific types of interfaces, and thereby isolate the user from the mechanics of communication.

In parallel with this approach, we suggest that each type of interaction among a set of processes should be controlled and coordinated by a special process called a monitor. Some synchronization monitors are discussed as examples in the appendix. Other kinds of monitors may be introduced in a similar fashion. The notion of synchronization is introduced below.

Synchronization is any timing constraint placed on interactions among concurrent processes. In a typical situation, one has a set of interacting processes whose speed ratios are not predictable; yet constraints exist on the relative sequencing of certain operations performed by them. At one extreme, such a set of processes could have almost unrestricted concurrency. At the other extreme, the processes could require synchronization to run "in step" most of the time. In many applications, there are "units of action" (often called transactions) that are allowed to interleave arbitrarily but operations within such units have to be synchronized. This creates a degree of serialism in a set of concurrent processes.

In the following sections, references will be made to some specific synchronization constructs found in the appendix. Their syntax is similar to that suggested in the literature for conventional operation systems although their effects span across machine boundaries. An open problem in the design of interaction constructs is to allow compilers to check for most time-dependent errors.

The remainder of this paper contains a general description of monitors and their implementation in a distributed system.

MONITORS

A set of processes that wish to interact may do so by referring to shared objects. A shared object is an entity that may be known to more than one process. Each process using a given shared object must declare it with the attribute "SHARED".

Just as in the case of ordinary objects local to a given program, shared objects are of various types. The type of a shared object determines a valid set of operations on that subject. Examples of types of shared objects are: file, semaphore, event and conditional region. (Some of these shared objects are discussed in the appendix.) One shared object could be appropriate for a particular kind of interaction but not for another.

A shared object may be either a constant or a variable. For instance, if s is a constant of type semaphore and x is a variable of type semaphore, PROCURE (s) will have the same effect as:

\[ x := s \;
\]

To manage a shared object, there is a special kind of process called a monitor. Each time a shared object is
Figure 1—Processes A and B referencing a shared object e

MONITOR

A

B

e

If e were an event, then each time a process executed an operation to WAIT for e, that process would get suspended and a message would be sent to the monitor of e (by the system software). Monitor e would maintain a list of all such processes waiting for e, and when any process executed a CAUSE operation on e, monitor e would send messages to certain nodes asking the system software that all the processes waiting for e be restarted.

There is a distinction between real and virtual times. Real time is made available by a hardware clock. Virtual time is relative to a given process and corresponds to state changes of that process. Thus, if a particular process remains suspended for a while, then its virtual time does not elapse although the real time continues progressing.

Each operation on a shared object is instantaneous in the virtual time of the process but will necessarily have a delay in real time. For instance, a PROCURE operation could involve a long period of real time but the state of the process remains unchanged.

Even though any single operation is instantaneous in virtual time, the overall interaction between a process and its environment could be such that while the interaction is in progress the process is doing some other work. Such constructs allow a process to interleaves several interactions in virtual time. For instance, if a process wishes to procure a semaphore, it could periodically do a TRYTOPROCURE operation and on failure spend its time doing something else. On the other hand, if a process has no other work to do and is waiting for a certain condition, it is more efficient for it to get suspended for the interim period. (This is known as the “non-busy” form of interaction.) The choice of how to interact would be based upon the extent to which a process depends on the interaction. A “tightly coupled” set of processes would prefer this non-busy form of interaction.

In the next section matters related to an implementation of monitors are discussed.

SOFTWARE STRUCTURE

To implement operations on a shared object without explicit message transfers, one needs some intermediate system software, called the interaction system, in each computer. Specifically, if a process executes an operation on a shared object, the local computer's interaction system receives control. The process itself may be suspended. The interaction system searches its tables to find the corresponding monitor for this shared object, and sends a message to it requesting appropriate action or status information. Since the interaction system must be able to si-
multaneously handle many operations, it makes entries in its tables to save the state of each operation in progress. When a reply from a monitor arrives, the interaction system interrogates its tables to determine the processes that are affected by this reply. For instance, if a message arrives from a monitor indicating that a particular event has been caused then all local processes that are waiting on that event will be started. Figure 2 shows the interaction system of a computer interfacing to the local processes on one side and the relevant monitors of the network on the other.

The next matter to be discussed is how the interaction system communicates with the various monitors located in different parts of the network. We assume that the distributed network already has an IPC scheme. At least three such schemes are known.4,6,10,1 Basically, an IPC capability allows any two processes in the network to establish communication under mutual agreement and transfer messages. The various interaction systems and monitors in the network would use this IPC. Figure 3 shows two processes using this facility. From a logical viewpoint, a communication path or channel is established for the message flow. Depending upon the IPC scheme being used, the processes may have to know each other’s names, port numbers, socket numbers, etc. This requires that the interaction system should know which monitors are required by its local processes and what the addressing details of these monitors are. Some IPC schemes would require that a “connection” be established between an interaction system and the monitors. Since the interaction system would have a limited number of logical channels at its disposal, it would allocate these channels as a resource to the individual operations currently active, based on demand and priority.

NAME SPACES

It should be possible for any set of processes in the network to interact by referring to the same shared object. On the other hand, two totally independent processes belonging to different users should not interfere with each other just because each of them declared a shared object with the same name. The basic requirement is that processes should interact if and only if they wish to do so. In this section, the general problem of managing the monitors in a network is discussed.

Note, if an object is declared as being local to a program (although it is shared by the various dependent processes compiled as parts of the same program), then this declaration is invisible outside of this program. The problem of name clashes arises only in the case of objects declared as being shared globally, and hence potentially accessible to separately compiled programs running anywhere in the network. It shall be assumed that the shared objects discussed below are global ones.

A new monitor is created when a user declares a shared object with the attribute “NEW”. This may be done in a job control language or as a system command. One may name any node of the network as the location of this monitor, or else a default location will be provided. It is important to note that the interacting processes are unaware of a monitor’s location; however, its location may be important for efficiency considerations. Any user may maintain his own supply of monitors by creating and deleting them dynamically.

A naming scheme is described below that allows each monitor in the network to be identified uniquely. The name by which a monitor is known throughout the network is called its external name. A process may refer to it using an internal name (relative to that process) which may be equated to the external name at run time. This is analogous to equating file names at run time.

Every user in the network has a unique identification and all processes and monitors created by him are tagged with this identification. For a given user, there is a distinct
set of directories, one per type of shared object. A shared object occurs in exactly one directory. Effectively, the set of all user directories for a given type of shared object constitute a network-wide name space for shared objects of that type. When a process refers to a shared object, it need not specify its type since this can be found by the compiler from the context.

Entries in a directory contain information such as the physical locations of monitors and the access rights to them. When a new process is created, the local interaction system searches the network name spaces to find the locations of all the monitors required by it. Another approach is to perform the search for a monitor the first time it is referenced. The important point is that a search is not necessary each time a given shared object is referenced.

The owner of a shared object may restrict access to it based on a password. He could also limit the operations that another user may perform on it. In the case of an event, for example, he may allow other users to wait on it but not cause it or delete it. This enables the following to be accomplished:

(1) Any user may set up his own "logical network" of processes that interact using shared objects belonging to him. He may dynamically alter this set of shared objects.
(2) Two separate users will not accidentally interfere due to name clashes between their shared objects, since each user has his own distinct directories.
(3) Under mutual agreement, processes of different users may interact to form larger logical networks. Monitors belonging to one user may be accessed by others subject to constraints imposed by the owner.

To make it convenient to use a shared object's external name, one may implement "short hand" naming conventions. Since the name space of a given type of object is a set of user directories, it forms a tree and any element in it may be referenced by its pathname. A user could be allowed to establish a reference point in the name tree and assume all pathnames to be relative to it. The reference point may be altered dynamically. Basically, this enables one to set up a "working directory". A working directory could also be specified as a set of sub-trees of the name tree and may include shared objects of other users. An appropriate default working directory is the user's own directory. To make the working directory concept more powerful, an installation may maintain a user profile that defines this user's default working directory. A user would be able to change his profile. This subject is further explored in Reference 14.

CONCLUSION

A fundamental premise of this paper is that a network programmer would benefit from the availability of programming constructs that allow him to be unconcerned with the "lower-level" communication in the network. This paper discusses an approach for accomplishing this and uses conventional synchronization constructs as examples. With the use of such constructs, the network appears to be a single, large, multiprocessor system.

A major goal to be accomplished is to develop network operating systems that make resource-sharing more effective. Programming constructs which are effective network-wide would be helpful not only to users but also in the implementation of a network operating system.

Several problems remain that need to be addressed in future work:

(1) Better interaction constructs must be invented that make it easier to prove program correctness.
(2) If some of the interacting processes should fail, those remaining might be adversely affected. Research is needed to prevent individual failures from propagating throughout the network. Perhaps, the various interaction systems should perform periodic "handshakes" with each other to detect node failures, and should continually interrogate the status of processes in their own respective nodes to detect process failures. Upon failure detection, a recovery would have to be coordinated.
(3) Any resource-allocation system is confronted with the problem of deadlocks. Conventional algorithms for detecting and handling deadlocks need to be adapted for distributed systems. Perhaps, a special monitor for the entire network could interrogate all other monitors to detect deadlocks. Either it could run periodically or other monitors could trigger it when suspecting a problem.

Within the environment proposed in this paper, a system designer would have to keep the density of interaction among processes low so that the message traffic in the network does not become unbearable.

It would be desirable to make the interaction constructs extensible. This would allow a user to write his own monitor and define appropriate operations for it.

ACKNOWLEDGMENTS

The author wishes to acknowledge the assistance of Mr. Joe Tuharsky for editing the manuscript and Ms. Sharon Bischoff for typing it.

REFERENCES

5. Crocker, S. D., J. P. Heafner, R. M. Metcalfe and J. B. Postel,

APPENDIX

To illustrate the notion of a shared object, some examples are discussed in this appendix. These are by no means exhaustive. The reader is referred to References 2, 3, 8, 9, 13 for further study.

Semaphores

A semaphore may be used to represent a resource that has to be reserved before it can be used. If s is a semaphore, PROCEDURE(s) suspends the process until s is available and then the process is restarted and "owns" s. The PROCURE construct causes the local interaction system to send a request to the appropriate monitor. When a process wishes to relinquish control over s, it executes a RELEASE(s). The choice of which process wakes up in case several are wanting to procure s is undefined and it is assumed that this does not matter.

The integer function PROCURE(s, t, u, v, ... ) has the "side-effect" of procuring exactly one of the semaphores in the argument list. The value returned denotes the relative position of the semaphore that was procured. A suitable use of this would be in a case statement as follows:

```
CASE PROCURE(s, t, u, v);
    ;
END CASE;
```

In order to specify a time limit on the procurement operation, the integer function PROCURE(<time limit>, s, t, u, v, ... ) may be used. If a timeout occurs, the value returned is one. The integer function TRYTOPROCURE(s, t, u, v, ... ) returns a zero if none of the specified semaphores is immediately available; it is similar to the PROCURE construct with the time limit set to zero.

In a general situation, a certain maximum number of processes may be allowed to own a particular semaphore at any time. This maximum number is called the capacity of the given semaphore and may be defined by its owner. At any time, SUPPLY(s) returns the number of copies of s that are still available; QUEUESIZE(s) returns the number of processes contending for s. These functions could help a programmer in scheduling his "bids" for semaphores.

Events

An event is a type of synchronization object suitable to transmit timing signals to another process or to indicate that a certain condition has been satisfied. At any given time, an event e is either in the set or reset state. When a process does a WAIT(e) and e is already reset, it gets suspended until e gets set. A message is automatically sent to monitor e who is responsible for informing all waiting processes when it's time to wake up. A process may wait on several events, including timeout, using the function WAIT(<time limit>, e, f, ... ) which returns an integer value indicating which of the events occurred. The time limit is optional.

Event e is set (occurs) when a process does a CAUSE(e). Once set, it remains set until a RFSET(e) is executed by one or the processes, and as long as it remains set no process will have to wait on it. The operation CAUSEANDRESET(e) will wake up all processes currently waiting on e and then immediately reset e.

An interrupt I is a procedure that is ATTACHed to an event and will be triggered whenever that event is caused. When the procedure I is finished, control will be returned to the point where the process was interrupted. Several programs may attach one of their procedures to the same event and all of these will be triggered when the event occurs. Since these are "soft" interrupts, there is an unpredictable lag (in both real and virtual times) before the interrupt is actually felt. The monitor of an event keeps track of which interrupts are attached to it. A process may dynamically ATTACH or DETACH an interrupt from an event.

Conditional regions

Although semaphores and events allow a variety of interactions to be programmed, researchers are interested in developing constructs that enforce a better program structure. In sequential programming, a lot of work has been done to expose the trade-offs between "tight", unstructured programs and structured programs with provable properties. Similarly, in concurrent programming, very highly asynchronous processes tend to be unstructured and often have non-deterministic behavior. Interaction constructs have been proposed that would dis-
cipline the nature of asynchronous processing and help to
structure systems with more provable properties.
One of the shared objects proposed for this is called a
region. If r is a region then consider the following code:

\[
\text{WITH } r \text{ DO C END}
\]

Out of all processes that try to execute such a block, only
one is allowed to "own" \( r \) and enter C at a given time.
This way of implementing mutual exclusion allows for bet­
ter compile-time checking than is possible with the use of
semaphores. (A semaphore could be procured without be­
ing released, a condition that cannot be checked by a com­
piler.)

A generalization of this is known as a conditional region.
In this case, one may specify a condition involving \( r \) that
must be satisfied in order to own \( r \). For instance,

\[
\text{WITH } r \text{ WHEN } P(r) \text{ DO C END}
\]

In the above example, C will be executed only if \( P(r) \) is
true and if no other process owns region \( r \). Here, \( r \) may be
of type Boolean, integer, character, etc., as declared. The
compiler checks that \( r \) is not referenced without being
owned.

A more flexible approach might be to associate a condi­
tional region with an event and disallow a process to own
that region until the corresponding event has occurred.
Although conditional regions are of theoretical interest,
they have not been implemented and, hence, there is not
enough experience available on their usefulness.