MESSAGES AS ACTIVE AGENTS*

David W. Wall

Computer Science Department
The Pennsylvania State University
University Park, PA 16802

Abstract

Network algorithms are usually stated from the viewpoint of the network nodes, but they can often be stated more clearly from the viewpoint of an active message, a process that intentionally moves from node to node. This paper gives some examples of this notion, and then discusses a means of implementing it. This implementation applied in both directions also demonstrates the logical equivalence of the two viewpoints.

1. Introduction

The difference between a clear algorithm and an obscure one is often no more than a matter of finding the right viewpoint from which to describe it. For instance, an algorithm that makes active and confusing use of a growing and shrinking stack may well have a concise, elegant formulation in terms of recursion.

Algorithms for use in a distributed network are often confusing. A process at one node decides that something must be done, so it sends a message to another node to do something about it. No one node has a complete picture of what is happening, and nodes throw messages back and forth in a manner reminiscent of circus jugglers. The program at each node is likely to be written as a catalog of responses to the various message stimuli that can arrive, obscuring the actual algorithm.

My claim is that there is a simple reason for this confusion: In many cases, it is misleading to think of the processor node as the active agent in the computation. Instead, the active agent is the message, moving around the network performing different acts at each node. An algorithm stated from the viewpoint of the messages will often be simpler and clearer than if stated from the viewpoint of the processor nodes.

* This material is based upon work supported by the National Science Foundation under Grant MCS-8102278.

2. Background

Researchers have considered many ways of describing parallel and distributed algorithms. Early work in parallel algorithms emphasized communication by means of shared variables, using critical regions or monitors to arbitrate the sharing [4, 5, 11].

Growing interest in distributed systems has led to the exploration of message-based schemes. These include synchronous unbuffered approaches like Hoare's communicating sequential processes [10], Brinch Hansen's distributed processes [3], and Ada's notion of a rendezvous [12]. The idea is that a process trying to send a message or receive a message is suspended until there is a matching process trying to communicate with it in the other direction.

We can contrast this with the asynchronous buffered approach exemplified by PLITS [8], Gypsy [9], actors [2], mailboxes in SUE/360 [18], and UNIX pipes [17]. In this case a process may wait to receive a message, but it can send a message without waiting; the message is delivered to the destination process and buffered until the destination decides to receive it, and the sending process can meanwhile go on with whatever it was doing.

Recent papers by Lauer and Needham [13] and Andrews [1] are notable attempts to unify various of the approaches. The concept of active messages can be taken as yet another approach to the problem of writing distributed algorithms, but it also represents another attempt to unify aspects of the different existing approaches. If we view the world through the eyes of a message, it is no longer of critical importance whether that message was passed synchronously or asynchronously. Moreover, since active messages interact with each other by changing the states of the nodes, much of the work on communication via shared variables may prove more useful in distributed networks than might otherwise have been expected.

3. A Simple Example

A very simple distributed algorithm is the ARPANET algorithm for routing a message from a source to a destination [14, 15]. Each node in the network maintains a table called NEXT, which tells which of its neighbors is the first node on the shortest path to each possible destination. If a node is trying to forward a message to some destination D, it looks in its table and forwards the message to node NEXT[D], which looks in its own

table and forwards it another step, and so on.

Since a message can come in at any time, this algorithm would be stated as follows.

```
do forever
   wait for a message;
   D := destination;
   if D = me then
      process message
   else
      N := NEXT[D];
      send message to N
   fi
od
```

Each node loops forever waiting for messages to arrive. Each time a message arrives, the node looks to see if the message is addressed to this node ("me"), and reads it if so; otherwise the node looks in its forwarding table to decide which neighbor it should pass the message on to.

This is straightforward. But this statement of the algorithm has a strange property: It is an endless loop! As a result, it hides the fact that there is something terminating, because after all any message does eventually reach its destination. If we state the algorithm from the point of view of the message, we get a somewhat different program.

```
D : = destination;
while D ≠ here do
   N := NEXT[D];
   go to N
od;
process message
```

The go to in this program is not a transfer of control, but instead represents a movement of this active message from one node to another. This formulation is slightly simpler, in the sense that it is shorter and not as deeply nested. But more important, it makes it clearer that the goal of the algorithm is to get the message to the destination! A message repeatedly looks in a table for the right next node, and goes there, where it finds a new table, until it finally manages to make D=here.

I have intentionally been a little sloppy about the variables in this example. In an active message program, some of the variables are local to the nodes, like "here", which identifies the node at which the message currently is active, and the forwarding table NEXT. Others are local to the active message, like "D"; these correspond to the content of the message. We can distinguish between these two kinds of variables in their declarations; in the examples that follow, variables local to the message will be declared as such, and all others are assumed to be local to the nodes.

4. Broadcast and Reverse Broadcast in a Tree

Another distributed algorithm that benefits from a change of viewpoint is the forwarding of broadcast messages on a tree. A node can initiate a broadcast by sending a copy to each of its neighbors in the tree, and a node that receives such a broadcast will forward it by sending a copy to each neighbor except the one on which it arrived. From the node's point of view it looks something like this.

```
do forever
  wait for a broadcast message;
  if it was initiated locally then
     LIST := all neighbors;
     send msg to LIST
  else
     N := neighbor that sent it;
     LIST := all neighbors except N;
     send message to LIST;
     process message
  fi
od
```

Once again we have an infinite loop simply because a message can be initiated by any user at any time. Moreover, this version obscures the fact that we carry out a broadcast by having an initial node send a copy to all of its neighbors and subsequent nodes send it to all but one. The sequentiality of those two cases does not appear here, since from the node's viewpoint it can freely intermix the forwarding of foreign broadcasts and the initiating of its own broadcasts.

If we state this in terms of active messages, this becomes clearer.

```
{P is local to the message}
P := here;
LIST := all neighbors;
go to LIST;
pass message to local user;
while number-of-neighbors > 1 do
  LIST := all neighbors except P;
  P := here;
  go to LIST;
  pass message to local user
od
```

Note the implicit fanning out of the message produced by going to a set of nodes. A copy goes to each node in the destination set, and each copy continues executing immediately after the go to.

We can run a broadcast in reverse to collect values from the nodes and select the smallest one. Each leaf sends its value to its neighbor; each internal node waits until it has received values from all but one of its neighbors, and then sends the smallest value it has seen, possibly its own, to the remaining neighbor. Eventually these incoming signals cross at some edge in the middle of the tree, and the endpoints of that edge can decide what to do with the answer. Because a node does not know whether it is an endpoint of the edge at which the messages will happen to cross, the node-oriented program must force each node to wait for a crossing message to arrive, as we see in the following.

```
VAL := value;
LIST := neighbors;
while size(LIST) > 1 do
   wait for message;
   MSGVAL := value in message;
   LIST := LIST - {sender};
   if MSGVAL < VAL then
      VAL := MSGVAL
   fi
od;
send VAL to only member of LIST;
wait for message;
MSGVAL := value in message;
if MSGVAL < VAL then
   VAL := MSGVAL
fi;
```

```
   Now VAL is minimum; negotiate
      with sender of last message
      for next action.
```

This has several ugly properties. First of all, most nodes never finish; they get hung up at the last wait, which might force us to follow this reverse broadcast by a forward broadcast whose only purpose is to assure the waiting nodes that all is well and they can stop waiting. Moreover, the code for comparing old and new VALs appears twice, which seems odd when we see what the active message version looks like.

```
{P, MSGVAL are local to message}
VAL := value;
LIST := neighbors;
while size(LIST) = 1 do
   MSGVAL := VAL;
   P := here;
   go to only member of LIST;
   LIST := LIST - {P};
   if MSGVAL < VAL then
      VAL := MSGVAL
   fi
od;

if size(LIST) = 0 then
   VAL is minimum; negotiate with
   sender of last message for action
fi
```

Now we see that any message tries to minimize VAL whenever it finds itself in a new location. Moreover, we don't have cases when somebody gets hung up early; messages that come into a node earlier than others simply update the VAL in that node if appropriate and then quietly terminate. The last message that arrives sees that it is last and takes its minimum VAL on to the next node inward. When two messages cross in the middle, those two are the only ones active, and they don't need to worry about anybody else.

I am a little disappointed that we can't make messages "fan in" implicitly as we made them "fan out" by going to a list in the example of forward broadcasting. To do this we would need something like a fork/join construct for fanning out and back in. This does not seem as elegant as simply going to a set of destinations; moreover it is hard to imagine what we do with all of those VALs at the join. Even without implicit fanning in, the algorithm as stated from the viewpoint of active messages seems more attractive and easier to understand than the active node program.

The reverse broadcast example has another interesting property. I stated it in a form that required a message to start at each node, but messages that started at internal nodes of the tree did nothing except initialize VAL and LIST. We may be able to accomplish this initialization in some other way, for instance by making it the last step of a broadcast that initiates the reverse broadcast. If so, messages need to be created only at the leaves of the tree, and the algorithm for those messages is exactly the same. We can easily imagine a broadcast performed as described previously, with the addition that a message reaching a leaf does not terminate, but instead goes on to perform a reverse broadcast. Combining the two active message algorithms is a simple matter of concatenation. Combining the two node-oriented algorithms is messier; how do we make the node follow a broadcast by a reverse broadcast and still allow unrelated broadcasts to take place?

5. Construction of a Minimum Spanning Tree

The reader might reasonably argue that the previous examples are somewhat contrived. After all, the ARPANET algorithm and the broadcasting algorithm (and to a smaller extent the reverse broadcast) are in fact message routing algorithms; should we be surprised that we can state them more conveniently from the viewpoint of the messages being routed?

To demonstrate that the notion of active messages is more generally applicable, let us consider an algorithm developed by Dalal [7] and simplified by Wall [19] by which a network can construct its own minimum spanning tree. Although an important reason for constructing such a tree is to use it for routing messages, the algorithm is not itself a routing algorithm, and the construction of the tree is typical of distributed graph algorithms. Each node begins with a Local Image of the network, which tells it the cost of each incident edge, and finishes with a Local Image of the minimum spanning tree, which tells it which incident edges are branches of the MST.

The algorithm is based on Prim's principle [16], which states that if we consider any fragment of a minimum spanning tree, even one consisting only of a single node, then the cheapest edge connecting the fragment to a node outside the fragment is a branch of the MST. We can describe the algorithm in terms of obligations. Each node starts out with an obligation to mark a branch, which it first tries to fulfill by marking the cheapest incident edge. If both endpoints of an edge mark it as a branch, one of them must resume its obligation by examining the larger fragment that was created by the branch in question and trying to mark a new branch connecting this fragment to some other. If this results in another doubly-marked branch, the conflict is resolved the same way. Eventually a node will find that it cannot fulfill its obligation because its fragment contains all the nodes, and at this point the algorithm is finished.

As it performs the algorithm, each node maintains a description of its perception of the fragment to which it belongs. The actual fragment may be bigger than this, since branches may have been created about which this node has not yet received word. This fragment description includes a list of the edges that lead to nodes outside this node's perception of the fragment. The node transmits this fragment description with each message it sends, and the destination node merges it with its own fragment description, giving that node a possibly more complete picture of the actual fragment to which both nodes belong.

When two endpoints of the edge mark it as a branch, one of them must resume its obligation. We assume a subsidiary routine called DECIDE that each endpoint can call to decide if it should resume its obligation. DECIDE can make this decision in any manner whatsoever, as long as it will decide consistently from one node to another.

Dalal's algorithm consists of a main program and a subroutine that we can call FULFILL-OBLIGATION, which is performed by a node trying to add a branch to the tree. FULFILL-OBLIGATION can be stated as follows.

```
routine FULFILL-OBLIGATION:
examine the fragment description
   and let the edge {NEAR, FAR}
   be the smallest edge leaving
   the fragment, where NEAR is
   the endpoint in the fragment;
if there is no such edge then
   TREE-DONE := true
elseif NEAR = me then
   if edge is already marked then
      send "obligation" to FAR
   else
      mark the edge as a branch;
      send "mark this branch" to FAR
   fi
else
   send "obligation" to NEAR
fi
```

Each node starts by initializing its fragment description to contain only itself, and the edges leading away from the fragment to be exactly those edges incident to that node. The main program can then be stated as follows.

```
TREE-DONE := false;
FULFILL-OBLIGATION;
while not TREE-DONE do
   wait for a message and
      merge its fragment state;
   if message is "obligation" then
      FULFILL-OBLIGATION
   else {message is "mark this branch"}
      if edge not already marked then
         mark it as a branch
      else {doubly-marked}
         if DECIDE selects me over
               the message source then
            FULFILL-OBLIGATION
         fi
      fi
   fi
od
```

A correctness proof of this algorithm [19, 20] argues that partial correctness follows from Prim's principle, and that the algorithm terminates because a given obligation cannot be passed around very far without acquiring a fragment description that is bigger than it used to be. This suggests that the algorithm might be easier to understand if described in terms of active messages, as follows.

```
{SUCCESS, TREE-DONE, the two endpoints,
   and a fragment description are local
   to the message.  Each node also has
   a fragment description.}
SUCCESS := false;
TREE-DONE := false;
repeat
   select smallest edge {NEAR,FAR}
   while (there is such an edge)
         and (NEAR ≠ here) do
      go to NEAR and merge fragment;
      select smallest edge {NEAR,FAR}
   od;
   if there is no such edge then
      TREE-DONE := true
   elseif edge is already marked then
      go to FAR and merge fragment
   else
      mark it as a branch;
      go to FAR and merge fragment;
      if edge not already marked then
         mark it as a branch;
         SUCCESS := true
      elseif DECIDE picks FAR then
         SUCCESS := true
      fi
   fi
until SUCCESS or TREE-DONE
```

We no longer need FULFILL-OBLIGATION to be a subroutine, since as soon as an active message is sure that it has fulfilled its obligation it simply terminates. The sequence that marks a branch has a certain elegance to it; to mark a branch, a message simply records it at one endpoint, goes to the other endpoint, and records it there. Whether or not it has fulfilled its obligation rests on whether or not some other message has already marked this branch at the other endpoint, a fact that is also made somewhat clearer in this version.

Most striking of all, perhaps, is the fact that the active message algorithm exhibits the nested loop structure of the process. A message proceeds by looking for an edge to mark as a branch, but it can only do this if the edge it finds is incident to the node at which the message currently resides. Otherwise, it must travel to the node in question and verify that this node knows of no better edge. This leads to the inner loop at the beginning. When the message finally reaches a node at which its obligation can be fulfilled, it marks the branch as we discussed a moment ago. If it finds that the branch is doubly-marked, it may have to resume its obligation and start over, which leads to the outer loop.

Of course, all of this is true of the node-oriented version as well, but it appears only in the proof and not in the program.

6. Implementation

The initial reaction to this method of describing a distributed algorithm is often dismay at the thought of passing programs around in the messages. This is one possible implementation, of course, but a better solution exists.

To execute an active node program, we must get copies of it into the nodes. We can do the same thing with an active message program. Each node keeps a set of active message programs; each message identifies the program it is executing and its position in that program, and includes values for its local variables. Each node cycles in an endless loop waiting for such messages. When such a message arrives, the node branches to the appropriate point in the appropriate active message program and executes it until it either terminates or reaches a go to. In the latter case, the node sends a message with the new state of the active message to the set of nodes in the go to; in either case, it then resumes waiting.

In some sense, therefore, all we are doing is giving a more elegant means of describing the responses that a node can have to the various message stimuli that can arrive. It is tempting to compare an active message to a coroutine; but a node resumes this coroutine at a location that depends on where some other node suspended it.

An interesting point to note about all this is that it really doesn't matter whether we do the

message passing synchronously without buffers or asynchronously with buffers. The loop that repeatedly waits for a message and responds to it is now invisible, and we can write it either way depending on the distributed environment in which the algorithm is being performed. If we use synchronous message passing, a node must be able to wait for a message without specifying the node from which it is expected; moreover, the correctness of an algorithm may still depend on which underlying model we are using. Nevertheless the notion of structuring the computation from the viewpoint of the messages does not seem to require the assumption that messages are passed asynchronously.

7. Equivalence of Active Messages and Active Nodes

The previous section sketched a means of implementing active messages without passing programs around, essentially by simulating active messages by means of active nodes. We can use the same idea to simulate active nodes by means of active messages.

To simulate an active message program in an active node notation, we kept the program at the nodes and let the message contain only the location counter. Conversely, we can keep an active node program in the messages (at least logically) and let the node contain only the location counter of the wait statement it is currently stuck at. When a message is received, execution is resumed and continues until another wait is encountered. Because a node can send a message and then go on doing something else before it goes back to waiting, this equivalence does depend on our active message notation including some kind of forking facility beyond merely going to a set of destinations, but this should not distress us too much. The point is that it doesn't matter whether we suppose that the node was spurred to action by receiving the message, or that the message arrived at the node and then looked around to see what needed to be done. The two are entirely equivalent, and our choice between them should be based upon the relative clarity and elegance of the resulting programs.

8. Conclusions and Future Work

We have shown a simple way of describing distributed algorithms from the point of view of the messages rather than the nodes, and have given a simple means of implementing this idea practically. We can use this implementation in both directions to demonstrate the logical equivalence of the two methods of description, which leaves us free to choose between them according to considerations of readability and clarity.

It is entirely possible that the two approaches of active nodes and active messages represent special cases of a general approach in which we can consider both nodes and messages to be equal partners. For instance, while the ARPANET routing algorithm was clearer from the message's viewpoint, the associated algorithm for maintaining the NEXT routing tables, which until recently was done by periodically exchanging these tables with each immediate neighbor [14], would probably be difficult to state as an active message program. Development of a technique in which both nodes and messages are considered to be active agents presents an interesting challenge.

A related observation is that in this paper I assumed that only one active message existed at any given node at any given time, because the traditional active node approach usually assumed that only one message was received at a time, others being buffered or delayed until the node was ready to receive them. If we view our messages as active processes, and if a network node is capable of multiprocessing or multiprogramming, then there is no reason we can't have several active messages at the same node. This raises some timing problems with regard to accessing the variables local to the node, but the general problem of synchronizing use of shared variables is old and well-explored [4, 5, 11].

Even if it turns out that these two generalizations are of little use, we need to be able to design an environment in which active node programs and active message programs can be intermixed. Organization of the control program at the processor nodes to admit this kind of structure may be a practical problem of some interest.

Postscript

I have an interesting additional argument for the appropriateness of active messages. After writing this paper, I gave it to Cherry's fiendish STYLE program [6], which among other things asked me to reword places where I used the passive voice. A surprising proportion of these occurred in the passages about node-oriented algorithms: As I was writing, phrases like "the message is passed" or "the obligation is transferred" felt like the most comfortable phrases to use, even though the node was theoretically the hero of my tale.

The conclusion I draw is only half in jest. Just as we shouldn't torture our writing style by inappropriately resorting to the passive voice, we shouldn't torture our programming style by restricting our attention to the passive elements of our program.

Acknowledgements

I would like to thank Susan Owicki, Brent Hailpern, John Gilbert, Robert Roos, and V. Ashok for discussions about network algorithms that led directly or indirectly to the idea of active messages.

References

[1] Gregory R. Andrews. Synchronizing resources. Transactions on Programming Languages and Systems 3, 4 (October 1981), pages 405-430.

[2] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. Proceedings of the Fourth Symposium on the Principles of Programming Languages (January 1977), pages 267-280.

[3] Per Brinch Hansen. Distributed processes: A concurrent programming concept. Communications of the ACM 21, 11 (November 1978), pages 934-941.

[4] Per Brinch Hansen. Operating System Principles. Prentice-Hall, 1973.

[5] Per Brinch Hansen. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering SE-1, 2 (June 1975), pages 199-207.

[6] Lorinda Cherry. Computer aids for writers. SIGPLAN Notices 16, 6 (June 1981), pages 61-67.

[7] Yogen Kantilal Dalal. Broadcast Protocols in Packet Switched Computer Networks. PhD thesis, Stanford University, April 1977. (Computer Systems Lab Technical Report 128.)

[8] Jerome A. Feldman. High level programming for distributed computing. Communications of the ACM 22, 6 (June 1979), pages 353-368.

[9] Donald I. Good, Richard M. Cohen, and James Keeton-Williams. Principles of proving concurrent programs in Gypsy. Proceedings of the Sixth Symposium on the Principles of Programming Languages (January 1979), pages 42-52.

[10] C.A.R. Hoare. Communicating sequential processes. Communications of the ACM 21, 8 (August 1978), pages 666-677.

[11] C.A.R. Hoare. Monitors: An operating system structuring concept. Communications of the ACM 17, 10 (October 1974), pages 549-557.

[12] J.D. Ichbiah, et al. Preliminary Ada reference manual. SIGPLAN Notices 14, 6 (June 1979), part A.

[13] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. Proceedings of the Second International Symposium on Operating Systems, IRIA, 1978. Also appeared in Operating Systems Reivew 13, 2 (April 1979), pages 3-19.

[14] John M. McQuillan. Adaptive Routing Algorithms for Distributed Networks. PhD thesis, Harvard University, May 1974 (BBN Report 2831).

[15] John M. McQuillan, Ira Richer, and Eric C. Rosen. An overview of the new routing algorithm for the ARPANET. Sixth Data Communications Symposium, November 1979, pages 63-68.

[16] R.C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal 36, 6 (November 1957), pages 1389-1401.

[17] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. Communications of the ACM 17, 7 (July 1974), pages 365-375.

[18] K.C. Sevcik, J.W. Atwood, M.S. Grushcow, R.C. Holt, J.J. Horning, and D. Tsichritzis. Project SUE as a learning experience. Proceedings of the Fall Joint Computer Conference 1972, Volume 41, pages 331-338. AFIPS Press.

[19] David Wayne Wall. Mechanisms for Broadcast and Selective Broadcast. PhD thesis, Stanford University, June 1980 (Computer Systems Lab Technical Report 190).

[20] David W. Wall and Susan Owicki. The correctness of two network algorithms. In preparation.