2001 Society for Design and Process Science
Printed in the United States of America

MESSENGERS: DISTRIBUTED PROGRAMMING USING MOBILE AGENTS

Munehiro Fukuda
Lubomir F. Bic
Michael B. Dillencourt
Fehmina Merchant

Information and Computer Science
University of California, Irvine, CA 92697

Messengers are agents, each capable of navigating through the underlying network and performing various tasks at each node. Their use facilitates a programming paradigm shift allowing applications to be written not as collections of communicating processes but from the point of view of each Messenger as it navigates through the system. Using several different applications, we demonstrate the MESSENGERS *programming style and its implications for distributed programming. The advantages of programming in* MESSENGERS *include the ability to compute in unknown network topologies, the ability to modify or extend the applications' functional capabilities at runtime, and the ability to dynamically exploit computational resources. Furthermore,* MESSENGERS *programs result in a smaller semantic gap between the abstract algorithms and their implementations, which makes program construction a more intuitive process.*

1. Introduction

Distributed applications are collections of multiple concurrent activities executing on a loosely coupled network of computer nodes. The majority of distributed applications are developed using explicit concurrency, that is, using language where the programmer is required to divide the problem into separate activities and specify their dependencies. This is accomplished using various constructs, such as *fork, cobegin, parfor,* or concurrent process declarations.

Implicit concurrency relies on the compiler to extract the potentially parallel activities and to provide both the mapping and the coordination primitives. Performing automatic parallelization of programs written in conventional imperative languages has been the subject of several decades of research and important discoveries have been made, especially in the area of scientific computation. More recently, non-conventional languages have been the focus of attention. Functional programming languages have been found useful in writing programs for dataflow and other parallel architectures, since they only express

necessary dependencies among functions, rather than prescribing the order in which these are to be evaluated. This avoids the various false dependencies present in imperative programs, thus simplifying the compiler's task.

In this paper we present and explore a new programming paradigm for distributed systems, which differs from those mentioned above in its basic philosophy. The main concept is mobile agents, which are self-contained autonomous programs that have their own identity and behavior. Mobile agents can navigate through the underlying computational network, perform various tasks at the nodes they visit, and coordinate their activities with other agents participating in the distributed computation. We have developed a mobile agent system, called MESSENGERS, which we will use to illustrate concepts underlying general-purpose distributed programming using mobile agents. The mobile agents in the MESSENGERS system are referred to as *Messengers* (Bic, 1995; Bic, et al., 1996; Fukuda, et al., 1999; Fukuda, et al., 1998; Fukuda, et al. 1997).

The use of Messengers in developing applications implies a *style of programming* that is quite different from programming with send/receive primitives. Message passing forces the programmer to think in terms of concurrent activities that must be created, mapped onto physical node, and coordinated in time. Hence the application is viewed as a collection of stationary processes, operating concurrently and communicating with one another using message-passing; the programmer has a *global* view of all the concurrent activities. MESSENGERS programming, on the other hand, views the application as a collection of autonomous mobile entities, each having a "mind of its own," that is, capable of navigating through the underlying computational network, performing various tasks at the nodes they visit, and coordinating their activities with each other in both time and space. Hence the programmer's point of view is that of a *navigator*, sitting in the "driver's seat" of a Messenger, and guiding it on its way through the computation.

Following a brief overview of the MESSENGERS system in Section 2 we present examples from several different application domains that illustrate the new programming style enabled by Messengers (Section 3). We will demonstrate that this new paradigm offers elegant solutions to problems that would be difficult to achieve using more traditional approaches.

2. Messengers

2.1. Principles of Operation

MESSENGERS (Fukuda, et al., 1999) is a system that supports the development and use of distributed applications structured as collections of mobile agents, called Messengers. To allow Messengers to navigate autonomously through the network and carry out their tasks, the MESSENGERS system is implemented as a collection of daemons instantiated on all physical nodes participating in the distributed computation. The daemon's task is to continuously receive Messengers arriving from other daemons, schedule them, supervise their execution, and send them on to their next destinations in accordance with the navigational commands they invoke. The scheduling of Messengers on a daemon is non-preemptive: once a particular Messenger starts executing, it continues to run until it either terminates or explicitly gives up the CPU (e.g., by executing a navigational statement or issuing a *wait* command). CPU Scheduling is done using a FIFO queue, called the *ready queue*. The communication channels are also FIFO.

The MESSENGERS system involves three levels of networks as illustrated in Figure 1. The lowest level is the *physical network* (a LAN or WAN), which constitutes the underlying computational resource (Sun Workstations or PC's running UNIX/LINUX in our present implementation). Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process running a MESSENGERS server. The daemon network topology is described by the user at system's initialization. This involves selecting a subset of the physical nodes to run the daemon process on. The *logical network* is an

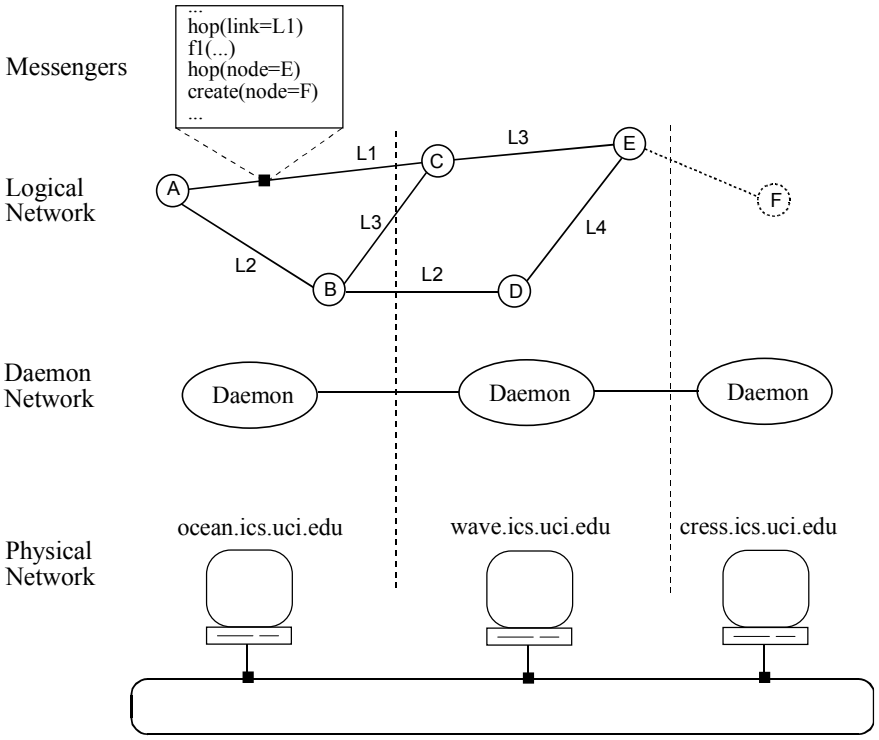

Fig. 1 Network Layers.

application-specific computation network created on top of the daemon network. Multiple logical network nodes may be created on the same daemon network nodes, thus running on the same physical node, and they may be interconnected by logical links into an arbitrary topology. The links in a logical network can be directional, and this directional information may be used by Messengers to navigate in the network.

2.2. Messengers Programming

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C and are compiled into native code. Conceptually, each script is carried in its entirety by the Messenger as it propagates through the network and is replicated each time the Messenger needs to follow more than one logical link. In the actual implementation of the MESSENGERS system, code is shared in the form of libraries, which are loaded on demand when a Messengers arrives for the first time at the daemon node (Gendelman, et al., 2001).

Programming in MESSENGERS facilitates a programming style different from the programming style used with other approaches to distributed computing such as message passing. In particular, the user does not supply any node programs to execute on the various network nodes. Rather, all node programs and their communications are part of the underlying infrastructure of daemons. The application itself consists of only the logical network and the corresponding Messengers navigating this network. Hence the application programmer can put himself into the "driver seat" and write the application from the point of view of the navigating Messengers, rather than from the point of view of a stationary process communicating with other such processes.

Messengers scripts distinguish three types of variables.

1. *Messenger variables* are carried by each Messenger as it propagates through the logical computational network. They are private to each Messenger, and form the local storage for the Messenger. Together with the program counter, they comprise the entire state of a migrating Messenger (Wicke, et al., 1999; Wicke, et al., 1998).
2. *Node variables* are resident in nodes of the logical network and shared by all Messengers currently visiting the same logical node. They provide the primary means for Messengers to communicate with one another. (In addition, rendezvous and signal/wait primitives are also provided.)
3. *Network variables* are predefined at each logical node. These variables allow each Messenger to access the network information local to the current node (e.g., the current node's address and name, and the last traversed link's name.)

Every Messenger script starts with a variables declaration that has the same style as in C. The declaration allows all standard C data types, although certain restrictions are placed on the use of pointer variables. The remainder of a Messenger script consists of statements, each of which is of one of the following three types:

1. **Computational statements:** A Messenger residing in a particular node may read and update this node's variables, read and update its own messenger variables, and read the current node's network variables. Arbitrary expressions are permissible in the assignment and may include all of the common arithmetic and logic operators provided in C. A Messenger may perform all the common control statements supported in C, such as if-then-else, while, do-while and break. A Messenger may also call C functions, which may either be defined in the same script or separately compiled functions that are linked dynamically.
2. **Navigational statements:** A Messenger may create new logical links and nodes, change or delete existing ones, and move arbitrarily through the network by following links or jumping to specific nodes. These operations are specified using the following statements:

   - *hop*: When a Messenger executes this statement, the statement parameters are used to determine a set of destination nodes, and then the Messenger moves to these destinations. (More precisely, if there is exactly one destination, the Messenger moves there; if there are multiple destinations, a separate copy of the Messenger is propagated to each destination.) In its most general form, the command is quite flexible; a complete description of its semantics can be found in (Fukuda, et al., 1998). In particular, the set of destination nodes may be specified by specifying the logical node name, the daemon node name, the logical link(s) to be followed, or the link(s) in the daemon topology to be followed. For example, in Figure 1, the Messenger hops from node *A* to node *C* by specifying the link *L1*, and after executing function *f1()* on node *C*, it then hops from node *C* to node *E* by specifying the destination node (*E*). The set of destination nodes may include the node on which the Messenger is currently running; in this case, it continues running on the current node and sends copies of itself to all the other destination nodes. The parameters support "wild card" matching, which results in a powerful multicast mechanism for each Messenger. There are two forms of the wild-card syntax: * means all nodes in the appropriate network (logical or daemon), while ** means all nodes in the appropriate network that are neighbors of the (logical or daemon)

node on which the Messenger currently resides. Each daemon node has an initial logical node, denoted by 0. Thus the command *hop(node=0; daemon = **)* means hop to the initial node on each neighboring daemon of the daemon on which the current logical node resides.

- *create*: This statement takes up to three optional parameters, *node*, *link*, and *daemon*, which are specified using name syntax. In its basic form, it creates a new logical node, connected to the current node by a new link, and hops to the newly created node. The new node is created on the daemon specified by the *daemon* parameter if one is specified; otherwise, the daemon is chosen by the system. For example, in Figure 1, when the Messenger reaches node *E*, it creates a new logical node *F*, and leaves the choice of the daemon to the system. The *create* statement can also be used to create a new link to an existing logical node; the syntax allows us to specify whether the new link is directional and which way it is directed. As with *hop*, all parameters are optional and there is great flexibility in specifying the set of new links and new nodes.
- *delete*: This statement causes the Messenger to move to another logical node, as with *hop*, but it also delete the links traversed by each copy of the Messenger. In addition, if this action erases all links from the departing node and there are no other Messengers currently residing in it, the node may also be deleted.

3. **Task control/coordination statements:** These statements allow Messengers to initiate the execution of other Messengers, to terminate their execution, or to coordinate and synchronize with one another.

   - *inject:* This statement initiates (injects) the execution of one or more instances of another Messenger. The parameters specify the Messenger script to be injected and the initial parameters to be passed to the newly created Messenger. In addition the starting destination(s) may be specified.
   - *exit, kill:* These statements cause a Messenger to be terminated. The exit terminates the Messenger issuing the command, while the kill command terminates another Messenger, specified as a parameter.
   - *signal, wait:* These statements allow Messengers to synchronize with one another using named events. Execution of a wait command with a named event as an argument causes the issuing process to block until the named event is signaled by another Messenger. A wait command without an argument causes the issuing Messenger to give up the CPU and be placed at the end of the ready queue.

3. Applications

3.1 Network Control

The ability to operate in networks without *a priori* knowledge of their topology is of great interest to many applications, especially those operating in large heterogeneous networks. The control of such networks is an important application area in itself, requiring the general ability to deal with a dynamically changing topology as physical nodes are disconnected and reconnected, to query their status and to disseminate information to them. To illustrate the ability of MESSENGERS in this context, we consider the problem of sending (diffusing) some piece of information (a message) to all currently operational nodes

in the network, assuming there is no general broadcast facility. Two standard solutions to this problem using conventional send and receive primitives, taken from (Andrews, 1991), are shown in Figures 2 and 3. The first solution uses a spanning tree, superimposed onto the physical topology, to emulate a broadcasting facility. The second solution uses neighbor set communication to send the message from one node to all its active neighbors. We will refer to these two algorithms as the ST and NS algorithms, respectively.

```
(1)     initiator:
(2)       construct spanning tree ST
(3)       for (each child c of root)
(4)         send(ST, msg, my_ID) to c

(5)     each node:
(6)       receive(ST, msg, sender)
(7)       store msg
(8)       for (each child c of my_ID in ST)
(9)         send(ST, msg, my_ID) to c
```

Fig. 2 The ST (spanning tree) message-passing diffusion algorithm.

```
(1)     initiator:
(2)       n = number of neighbors
(3)       for (each neighbor x of my_ID)
(4)       send(msg, my_ID) to x
(5)       receive n messages

(6)  each node:
(7)    receive(msg, sender)
(8)       n = number of neighbors
(9)       store msg
(10)      for (each neighbor x of my_ID)
(11)        send(msg, my_ID) to c
(12)      receive n-1 messages
```

Fig. 3 The NS (neighbor-set) message-passing diffusion algorithm.

In the algorithm of Figure 2, the initiator of the algorithm (i.e., the process wishing to distribute a message to all nodes) first constructs a spanning tree, ST, over the network, with itself as the root node (line 2). It then sends ST together with the message (msg) to be distributed and its own ID to all its children according to ST (lines 3-4). Each receiving node (line 6) forwards the message to all its children, again using ST (lines 8–9). This is repeated until all leaf nodes have been reached.

In the algorithm of Figure 3, each node sends a message to all its neighbors. To avoid leaving extra messages buffered on the network and to correctly detect termination of the distributed computation, each node then waits to receive redundant messages sent by its neighbors. Each node knows exactly how many messages it will receive: the initiator will receive a redundant message from each of its

neighbors, while all other nodes will receive a redundant message from all but one of its neighbors (i.e., from all but the node that sent it the message that activated it).

```
(1)repeat  {
(2)  hop(node = 0; daemon = *);
(3)  if (visited == FALSE) {
(4)     store(msg);
(5)     visited = TRUE;
(6)  }
(7)  else exit; (8)
(8)}
```

Fig.4 Diffusion using Messengers.

Figure 4 shows a diffusion program using MESSENGERS. The program works by constructing a spanning tree "on the fly," and using this spanning tree to diffuse the message. Note that the program is the behavioral description of a Messenger. Hence it is written from the point of view of the Messenger, whose task it is to repeatedly spread into all neighboring nodes that have not yet been visited. The spreading is accomplished using the *hop*() statement (line 2), which replicates the Messenger along all daemon links of the current node. Each receiving node continues by executing the next command (line 3), which determines if the node has already been visited before. If not, the node variable *visited* is set to TRUE to indicate that it now has been visited (line 5). The Messenger then continues executing its repeat loop (line 1), which again replicates it to all neighboring nodes. If the condition on line 3 fails, indicating that the node has already been visited, the Messenger dies (line 7).

All three programs use essentially the same algorithm, a spanning-tree-based diffusion. The MESSENGERS program is similar to the NS program in that only a knowledge of the neighbors (rather than knowledge of the entire network topology) is required. ST and NS are written from the point of view of stationary processes communicating via messages, while the MESSENGERS program is written from the viewpoint of the Messenger navigating through the network. Although the programming styles are quite different, all three programs are easy to understand. There are, however, two fundamental differences between the two conventional message-passing programs and the MESSENGERS program:

(1) The ST program needs to know the network topology *a priori* in order to compute the spanning tree; the MESSENGERS program constructs the spanning tree on the fly. The NS program, unlike the MESSENGERS program, requires that for every message sent, a corresponding message is received, thus significantly increasing message traffic.

(2) Each of the message-passing programs requires a node program specific to the application to be loaded and started at each node. The MESSENGERS program only uses a generic server at each node, which does not change for different applications.

These two properties have several important implications:

- The message-passing programs are not tolerant of node failures or disconnections. In the ST algorithm, if a node becomes disabled between the construction of the spanning tree and its use,

the algorithm will not be able to reach the subtree that has the disabled node as its root, even if the network remains connected. Moreover, the algorithm cannot take advantage of newly re-connected nodes as these are not reflected in the spanning tree. In the NS algorithm, node failures can result in expected messages not being received, which can prevent the algorithm from terminating. If nodes are reconnected during the running of the NS algorithm, it can result in premature termination of the algorithm at a node, and leaving extra buffered messages on the network. The MESSENGERS program, on the other hand, constructs a new (implicit) spanning tree without using acknowledgements each time it is run. Hence the Messengers always explore the current topology. If a node is disabled (before or after it has been visited), it does not prevent other nodes from being reached, provided the network remains connected. Similarly, a new node can simply start the daemon server and thus automatically become reachable by Messengers.

- The NS algorithm requires twice as many data communication operations as the MESSENGERS algorithm, since each link is traversed twice (by a message) in the NS algorithm and once (by a Messenger) in the MESSENGERS algorithm. The ST algorithm requires fewer communication operations, but it is not responsive to communication delay fluctuations since the spanning tree is static and reflects the communication pattern at the time of its creation. In contrast, each execution of the MESSENGERS program implicitly constructs the "shortest-communication-path" spanning tree, based on the current communication pattern. This is because the first Messenger arriving at any given node is the one to continue the exploration.
- The message-passing programs are useful only when multiple diffusions along the network are to be performed. This is because initiating the message-passing computation requires sending a startup command to each node by some mechanism; if only one message is to be diffused, this mechanism can be used to send the message itself to each node rather than the startup command. Additional startup overhead is incurred by the ST-algorithm, since the initiating node must determine the network topology and compute the spanning tree. The Messengers program, in contrast, only needs the server, which runs continuously in every node.
- To change the dissemination algorithm or to run a different network application, the message-passing program requires the corresponding node programs to be started at each node. With Messengers, all application-specific information is carried by the Messengers while all nodes run the same server. Hence arbitrary new applications may be created dynamically without having to know anything about the current network topology.

3.2. Dynamic Data Structures

In this section we consider a class of problems that require the dynamic construction of a data structure, such as a search tree, that is independent of the underlying physical network topology. To illustrate the differences between the message-passing and MESSENGERS styles of programming, we consider a divide-and-conquer algorithms to solving a generic problem, which we simply denote as *compute(work)*. The algorithm operates as follows: If *work* is larger than some threshold value, *max*, *work* is divided into two subsets. This is repeated until the resulting subsets are below the threshold, at which time the *compute* function is applied to each. The partial results are then merged in reverse order of the decomposition. To keep track of the subcomputations, a binary tree is constructed, where each leaf node represents a local computation and each non-leaf node represents the division of *work* and the subsequent merging of the partial results.

```
(1)     f(work){
(2)     if (work > max) {
(3)       l_work = take_left_half(work);
(4)       r_work = take_right_half(work);
(5)      l_child = fork f;
(6)      r_child = fork f;
(7)      send(l_work) to l_child;
(8)       send(r_work) to r_child;
(9)      receive l_res from l_child;
(10)   receive r_res from r_child;
(11)   res = merge(l_res, r_res);
(12)    }
(13)   else
(14)     res = compute(work);
(15)   send(res) to parent;
(16)   }
```

Fig. 5 Divide-and-conquer using message passing.

The easiest way to implement the above algorithm as a sequential program is to use recursion to construct and subsequently collapse the binary tree. In a distributed environment, recursion must be emulated by explicitly creating new processes and specifying the exchange of information among them. Figure 5 shows a possible distributed implementation using message-passing. Lines 2-8 implement the subdivision of *work* into two parts, each of which is sent to a newly created child process, *l_child* and *r_child*. The sending program then awaits the arrival of the partial results (lines 9 and 10), which it combines into a single result (line 11) and passes to its parent (line 15). If the received *work* is less then *max*, the result is computed locally (line 14) and passed to the parent (line 15).

```
(1)     m(work){
(2)      while (work > max) {
(3)       create(link = +"right", +"left");
(4)       if ($link == "left")
(5)          work = take_left_half(work);
(6)       else
(7)          work = take_right_half(work);
(8)      }
(9)      res = compute(in);
(10)     while (node != root) {
(11)      hop(link = -);
(12)      if (stored_res == NULL) {
(13)        stored_res = res;
(14)       exit;
(15)      }
(16)      else
(17)        res = merge(stored_res, res);
(18)     }
(19)    }
```

Fig. 6 Divide-and-conquer using Messengers.

The corresponding MESSENGERS program is shown in Figure 6. Initially, a Messenger called *m* is created and passed *work* as its parameter. The first *while*-loop (lines 2–8) constructs the binary tree. At each iteration, it creates two new links labeled "left" and "right", respectively and replicates itself along these links (line 3). At the destination node, which is created at the same time as the new link, it determines whether it is the left or the right child by testing the network variable $*link*, which contains the name of the last link traversed (line 4). It then takes the appropriate half of *work* for itself (lines 4–7). This is repeated until *work* is small enough to be handled locally in the current node (line 9), which produces the lowest level result *res*. The Messenger then starts backtracking toward the root (lines 10–18). The minus sign in the *hop* statement indicates that it follows the link ("left" or "right") in the opposite direction of its creation (indicated by the plus sign in the *create* statement). In each node the Messenger checks if it is the first to arrive there by examining the variable *stored_res* (line 12). If this is empty, the Messenger deposits the partial result it carries in that variable (line 13) and terminates itself (line 14). The second Messenger arriving at the same node from the other child node will detect that its sibling has already been there and thus continues its journey toward the root after having combined its own partial result (*res*) with that of its sibling (*stored_res*) deposited at the node earlier (line 17). Eventually, a single Messenger will arrive at the root carrying the complete result.

The two programs, while similar in their length and complexity, are very different in their programming style. The first is written from the point of view of stationary processes cooperating with each other in the construction of the search tree. The second program is written from the point of view of the Messenger, which constructs the tree by replicating itself at each node and then retraces its paths back toward the root. These two distinct point of view have a number of important implications:

- The abstract algorithm distinguishes three separate phases: (1) construct the binary tree by subdividing the work at each level; (2) compute the results at the leaf nodes; (3) merge all partial results by retracing the tree toward the root. The MESSENGERS program follows this intuitive description of the algorithm much more closely than the message-passing program. It clearly preserves these three phases in its structure: the construction and retracing of the tree corresponds to the two while-loops that surround the local computations at the leaf nodes. In contrast, the message-passing program does not preserve this structure. It recursively forks the function *f* to construct the tree but it must also handle the local computations and the merging of the partial results during each cycle. This is because there is no easy way to describe the backtracking from the leaves toward the root as a separate loop. Instead, each newly created process must wait for the partial results to be returned from its two children before it can terminate.
  Another way to look at this is to consider the *locus* of computation. With the MESSENGERS program, it is possible to step through the program sequentially. For any statement, there may be multiple instantiations of that activity but we never have to suspend an activity to consider another. In particular, when the execution on any particular branch reaches the "compute" function, there are no activities earlier in the program suspended and waiting for the current computation to return to them. Hence there is always a single locus of the computation. In contrast, the message-passing program can only be understood by considering the interactions among multiple concurrent activities. In particular, after forking the two child processes, each parent remains active waiting for their replies. Hence, when tracing a child's executions, we also need to keep track of each of the suspended ancestors higher in the tree. It is this distributed locus of the computation that makes the writing and understanding of distributed programs so difficult.
- The message-passing program forces the user to deal with two separate concepts at the same time: the creation (and termination) of concurrent activities, and communication, i.e., the ex-

change of information among these activities. The former is accomplished using the *fork* statement while the letter uses *send* and *receive* primitives. The algorithmic description, however, does not make any such distinction. It only prescribes the construction and subsequent retracing of a binary tree. The MESSENGERS program follows this abstract description very closely by using only the navigational commands *create* and *hop*. The *create* operation is used during the tree construction and accomplishes both the creation of new nodes and the passing of the necessary parameters (including the locus of the computation) to these nodes. The *hop* operation then achieves the backtracking, including the merging of all the partial results. The smaller semantic gap between the abstract algorithm and the MESSENGERS implementation makes the programs easier to understand.

- To control the mapping of the message-passing program onto the physical network, the *fork* statement must use some function at runtime to determine the location of the child process. This is typically part of the logic built into the program and hence any change in the underlying topology requires modification and recompilation of the code. With the MESSENGERS implementation, the code can be left generic, while the actual mapping is controlled by the system as part of the *create* command.

3.3. Spatially-Oriented Applications

In this section we consider a class of applications structured as a collection of entities interacting with one another within a two- or three-dimensional virtual space. Typical examples of such applications are interactive battle simulations (DIS, 1994), particle-level simulations in physics (Hockney and Eastwood, 1988), traffic modeling (Resnik, 1994), and various individual-based simulation models in biology or ecology (Huston, et al., 1988; Huth and Wissel, 1992; Villa, 1992). Related to the latter is also the study of collective behavior in AI, which investigates the mechanisms that result in complex and highly coordinated behaviors of groups of individuals, such as a school of fish or an ant colony, while each individual has only a very limited local knowledge of the "problem" and a set of simple rules to follow (Resnik, 1994). Finally, advanced graphics and animation applications, especially those involving large numbers of individuals, such as the animation of a flock of birds, have also taken advantage of spatially-oriented individual-based modeling (Reynolds, 1987).

To illustrate the programming style differences in this class of applications, we consider the simulation of a school of fish. It has been observed that schools of fish are capable of performing complex maneuvers without any particular individual taking on the role of a leader. The complex behavior of the entire school is the results of local interactions among neighboring fish.

In (Huth and Wissel, 1992), a specific model has been formulated, where each fish periodically adjusts its position and velocity by coordinating its movement with its immediate neighbors within a certain radius of visibility. It chooses a small number of neighbors within its circle of visibility and then uses these to recompute its own vector of motion (speed and orientation).

The simulated environment, which in the simplest case is a two-dimensional ocean, is partitioned into cells and distributed over the network of workstations, each of which handles the fish located in its cell. One important requirement is that the individual fish move not only within the simulated space but actually migrate among processors in the network. This occurs when a fish crosses the boundary from one cell into another.

The resulting close correlation between the simulated space and the underlying computer network significantly reduces inter-processor communication. The reason is that only neighboring fish in the logical space need to communicate with each other. If logical neighbors are always in close proximity in the physical space, only a very small number of processors, if any, need to exchange any information.

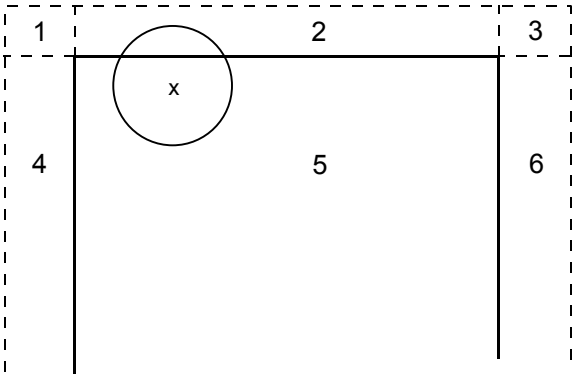

Fig. 7 A cell and its neighbors.

Figure 7 illustrates the representation of the environment as a rectangular grid with each cell having eight neighbors (one on each side, one on each corner). The figure shows a fish in node 5 with its circle of visibility extending into node 2. In order to correctly compute the next position of the fish, node 5 must know about all fish within the circle of visibility; this requires receiving information from node 2. If the fish were closer to the upper left corner of node 5, correctly computing its new position would require receiving information from nodes 1 and 4 as well. The dashed rectangles shown in Figure 7 represent the portions of the neighboring cells whose distance from the boundary of node 5 is less than the radius of visibility. We refer to each of these rectangles as a *boundary strip*. If node 5 knows about all fish in all of its boundary strips, it can then correctly move any fish within its area of responsibility.

We describe a general version of the fish simulation, where the space can be decomposed into any number of dimensions. We compare two different implementations, one using message-passing, the other using MESSENGERS.

```
(1)        Node_prog(){
(2)         for (i = 0; i < sim_run; i++) {
(3)           cur = local_lst;
(4)           while (cur != NULL) {
(5)             upd_pos(cur, local_lst, bndry_strps);
(6)             forall j in directions {
(7)               if (responsible_node(cur->position) == nghbr[j])
(8)                 move *cur from local_lst to out_migr_lst[j];
(9)               }
(10)            cur = cur->next;
(11)          }
        /* migration */
(12)        forall j in directions
(13)          send(out_migr_lst[j]) to nghbr[j];
(14)        forall j in directions {
(15)          rcv(in_migr_lst[j]) from nghbr[j];
(16)          merge(in_migr_lst[j],local_list);
(17)          }
        /* boundary strip exchange */
```

```
(18)        forall j in directions {
(19)          build out_bndry_strp[j] from local_lst;
(20)          send(out_bndry_strp[j]) to nghbr[j]
(21)         }
(22)         bndry_strps = [];
(23)        forall j in directions {
(24)          recv(in_bndry_strp[j]) from nghbr[j];
(25)          merge(in_bndry_strip[j],bndry_strps);
(26)         }
(27)       }
(28)     }
```

Fig. 8 Fish school simulation using message-pasing.

Figure 8 shows the message-passing version of the application. It consists of a node program that is started at each physical node. Each physical node also has a list, *local_lst*, which contains all the fish instances mapped onto this node. The simulation loop is repeated by all nodes for a specified number of simulation steps (line 2). In each simulation step, the program first loops through the fish list (lines 3–4), using the variable *cur* to keep track of the current fish on the list, and updating the position of each fish for the current iteration step (line 5). As mentioned earlier, each fish coordinates its movement with its nearest neighbors, so the *upd_pos* function must find the neighboring fish and examine their motion vector. Since the radius of visibility may extend into a neighboring cell, the function *upd_pos* must have access to the boundary strips of all neighboring nodes. These areas are exchanged periodically among neighboring nodes (lines 18–25); a list called *bndry_strps* of all fish in all neighbors' boundary strips is computed, and this list is used by the *upd_pos* function if necessary.

When the new position of the current fish is determined, it is checked against the node's assigned area of responsibility. If the new position falls in the area of responsibility of one of the neighbor nodes (line 7), the fish is removed from the current list and placed on a special list, *out_migr_lst[j],* containing all the fish to migrate to node *j*. The actual migration takes place at the end of the while-loop, once the position of each fish has been updated. Each node sends the appropriate list of migrating fish to each of its neighbors (lines 12–13) and, in turn, receives the corresponding lists from these neighbors (lines 14–15), which it merges into its local list (line 16). The final task to be performed at each simulation step is the exchange of the boundary strips among neighboring nodes. Each node determines which of its fish lie in boundary strips. For each direction *j*, it builds the boundary list to be sent to the neighbor in that direction by copying those fish from the local list (lines 18–19) and sends these to the neighbor in direction *j* (line 20). It then receives the boundary strips from each of its neighbors and merges them into a single list of fish (line 22–25).

```
(1)        fish(pos) {
(2)          for (i=0; i < sim_run; i++) {
(3)           record_position(i,pos);
(4)           if (responsible_node(pos) == $node) fishcount++;
(5)            else exit;
(6)           forall j in directions
```

```
(7)             wait(event#i#j);
(8)           (nextlinks,pos) = compute_position(i,pos);
(9)            fishcount—;
(10)          hop(link = nextlinks);
(11)         }
(12)       }

(13)       sentinel(dir) {
(14)         for (i=0; i < sim_run; i++){
(15)          signal(event#i#dir);
(16)          forall j in directions
(17)            wait(event#i#j);
(18)          while (fishcount > 0) wait();
(19)          hop(link=dir);
(20)         }
(21)       }

(22)       start(n_fish,n_nodes) {
(23)         for (i=0; i < n_fish; i++) {
(24)           pos = randpos();
(25)          inject(fish(pos),all_nodes(pos));
(26)         }
(27)         for (i=0; i < n_nodes; i++)
(28)          forall j in directions
(29)            inject(sentinel(j),i);
(30)       }
```

Fig. 9 Fish school simulation using Messengers.

The MESSENGERS implementation is shown in Figure 9. There is no single monolithic program running on each node; rather, each fish is represented by a separate agent. There are a total of three different Messenger scripts: one for the fish, one for a sentinel used to synchronize the fish in a manner to be described, and one to start the computation. On each node, there is an array that contains the current position of all fish relevant to that node (i.e., all fish on that node or on the neighboring boundary strips). This array is a node variable, so it can be shared by all Messengers on the node. A fish that is in the boundary strip of a neighbor makes its presence known to the neighbor by spawning a "shadow" copy of itself that hops to the neighbor, updates the data structure, and then exits.

The first script describes the behavior of the fish, each of which is programmed to "live" for a number of steps corresponding to the simulation run (line 2). At each step, the fish first records its position in the array of fish on the current node (line 3). If the fish is a shadow copy, it exits immediately; otherwise it increments the count of fish being processed at this node (lines 4–5). This determination (whether a fish is a shadow copy) can be performed by testing whether the current position of the fish is in the area of responsibility of the logical node. (The network variable $*node* contains the name of the logical node on which the Messenger is currently running.) The fish then waits until all fish at this node have recorded their position (lines 6–7). Once the positions of all fish have been recorded, the fish computes its new position (line 8). The function *compute_position* returns the new position, and a set of links (*nextlinks*). The set *nextlinks* consists of the links that the fish must follow to get to the node containing its new

position, together with all links to all nodes that will need shadow copies of this fish (i.e., all nodes containing the new position of the fish in their boundary strips). The fish then decrements the count of fish at this node (line 9) and hops along the set of links specified by *nextlinks* (line 10), thus moving itself and creating all shadow copies. Note that there is no distinction made here between the fish and its shadow copies; each copy will determine, in the next iteration, whether it is the fish or a shadow copy via the test at line 4, as described above.

The second script describes the sentinel Messengers, which are used to synchronize the fish. The synchronization uses events: on each node, there is one event (*event#i#j*) for each iteration *i* of the simulation and each direction *j*, including the null direction. At each simulation step, each logical node will have running on it one sentinel for each direction. The key invariant is that on each node, at each iteration, all fish arriving from a particular direction arrive at the node before the sentinel from that direction. We will see shortly that the key invariant always holds. Because the key invariant holds, the sentinel does not get to the front of the ready queue (and hence does not begin executing a particular iteration) until all fish arriving from that direction have updated their position on the logical node. Once this has happened, the sentinel signals an event indicating that all these updates are complete (line 15). The fish Messengers wait until all such events are signaled (lines 6–7), ensuring that all fish have recorded their positions before any fish compute their new positions. Note that when the last event is signaled in a particular iteration *i* on a particular node, all fish Messengers are waiting at line 7, all sentinels are at line 17 (except for the sentinel signaling the event, which is at line 15), and the variable *fishcount* contains the number of fish at that logical node in iteration *i*. Thus the loop at lines 6–7 in the fish Messenger and at lines 16–17 in the sentinel Messenger collectively form a synchronization barrier. Once the sentinel Messenger gets past the synchronization barrier, it waits until all fish have hopped away from this logical node (line 18), and then it hops in its assigned direction (line 19). This waiting by the sentinel Messengers, together with the fact that Messengers hopping along the same link arrive at their destination in FIFO order, ensures that the key invariant holds at the start of the next iteration, for all logical nodes and all directions.

Initially, the simulation is started by injecting all the fish Messengers and sentinel Messengers. The fish Messengers are started in the loop at lines 23–25: each fish is assigned a randomly computed initial position, and it is injected at the node responsible for that position together with all nodes that have that position in one of their boundary strips. The function *all_nodes* computes this set of nodes (these are the shadow fish). As described earlier, it is unnecessary to distinguish a fish from its shadow copies, as this will be done at lines 4–5 of the first iteration of the fish Messenger. The sentinels—one for each logical node/direction—are initiated by the loop at lines 27–29.

We observe the following main distinctions between the two programs:

- The application deals with two separate issues: the behavior of the fish, and the coordination of the fish in the distributed space. These are separated clearly in the MESSENGERS implementation, where a different Messenger program exists for each. The fish Messenger captures the behavior of each fish from its individual perspective: at each iteration step it updates its current position, waits for all fish at the current node to register themselves, and then positions itself and all necessary shadow copies at the correct nodes. The sentinel Messengers act as end-of-stream markers for the fish hopping between a given pair of nodes, providing the information necessary for the fish to synchronize themselves in the form of event signals. Note that the sentinels do not need to know anything about the behavior of the fish or even their count; rather, they only need to know when all fish have hopped away from the node.

  In contrast, the message-passing program is written from a completely different point of view. It combines in a single loop the handling of all the fish instances as well as the management of the

space. It views the fish as a list of passive objects, each of which needs to be advanced during each iteration. At the end of each iteration it handles the boundary strips exchange and the fish migration. Hence, unlike the MESSENGERS implementation, migration is not implemented as part of the fish behavior. Rather, each migrating fish is put on a list during its processing and a global migration is implemented at the end of each iteration. The above differences show clearly that the message-passing program has a much larger semantic gap between the abstract model and its actual instantiation than the MESSENGERS program, thus making it more difficult to design, understand, and maintain.

- Both the message-passing and the MESSENGERS programs must be mapped onto a distributed physical architecture. In the case of MESSENGERS, this requires creating a logical network and starting up the application on the logical network. Although it is not shown as part of the example, the logical network is created using MESSENGERS. The mapping of the logical network onto the physical network is completely transparent to the application. Once the logical network is created, the start Messenger initiates the simulation by injecting the fish and sentinel Messengers. It distributes the fish and the sentinels based on the size and topology of the logical network. As a result, the application does not have to be changed when the physical network changes, or when the mapping of the logical network to the physical network changes. This has three important consequences: (1) We can choose any subset of the currently available set of machines to map the logical network on. (2) We can change the mapping a run time, which provides a basis for load balancing and also allows us to dynamically change our choice of machines. (3) We can tolerate machine failures, provided we maintain replicated snapshots of the computation from which the failed computation can be restarted on a different machine (Gendelman, et al., 2000; Gendelman, et al., 2001).
  The message-passing implementation requires that each node program be told who its neighbors are. Whenever the physical topology changes, these values need to be recomputed and explicitly supplied to each node program when it is loaded. This requires the use of a different language, e.g., a shell script, to perform the mapping, or it must be done manually. In addition, code must be distributed to each node and initialized. In the MESSENGERS implementation, this is done as part of the code (lines 22–30). In the message-passing implementation, this must be done outside of the code, again either manually or using some other language.

- The message-passing implementation is fixed in terms of its functionality. Any modification or extension would require the program to be modified, recompiled, and redistributed to the physical nodes. The MESSENGERS implementation, on the other hand, is open-ended and thus arbitrarily extensible. Specifically, it is possible to (1) remove existing fish instances at runtime (using the *kill* function); (2) introduce new fish instances (using *inject*); (3) introduce new fish instances with a modified behavior (by modifying the Messenger script before injecting it); and (4) introduce arbitrary new entities of a completely different type. For example, the user could introduce a "predator" by writing its Messenger script and injecting it into the simulation. This would then follow its own behavior, e.g., using a different function to navigate the space, seek out fish, or "kill" these by destroying the corresponding Messengers (using the function *kill*). All of the above can be achieved at runtime, without recompiling or even halting the ongoing simulation. Hence the paradigm facilitates interactive and incremental software development and use.

4. Conclusions

MESSENGERS is a system based on the philosophy of distributed programming using mobile agents. This allows the programmer to adapt a completely different point of view: instead of viewing the application as a collection of concurrent activities interacting with each other via message-passing, the new paradigm puts the programmer into the "driver's seat" of a mobile agent, which it must guide on its journey through the network.

In this paper, we have illustrated how the MESSENGERS programming philosophy yields important properties that would be more difficult to achieve with conventional message-passing. These include the ability to compute in unknown or dynamically changing network topologies, and the ability to modify or extend the applications' functional capabilities at runtime. The MESSENGERS paradigm allows the mapping of the code to the physical architecture and the distribution of the code to be managed using the same language that the application is written in. Furthermore, MESSENGERS programs generally result in a smaller semantic gap between the abstract algorithms and their actual implementations, which makes them easier to construct, understand, and maintain.

For additional information, the reader is referred to our WWW page: http://www.ics.uci.edu/~bic/messengers.

5. References

G.R. Andrews, 1991, Paradigms for process interaction in distributed programs. ACM Computing Surveys, 23(1):49–90.

L.F. Bic., 1995 (August), Distributed computing using autonomous objects. In 5th IEEE CS Workshop on Future Trends of Distributed Computing Systems (FTDCS).

L.F. Bic, M. Fukuda, and M. Dillencourt, 1996, Distributed computing using autonomous objects. IEEE Computer, 29(8).

DIS Steering Committee, 1994, The DIS vision: A map to the future of distributed simulation. Institute for Simulation and Training.

M. Fukuda, L. F. Bic, and M. B. Dillencourt, 1999, Messages versus messengers in distributed programming. Journal of Parallel and Distributed Computing, 57:188–211.

M. Fukuda, L.F. Bic, M. Dillencourt, and F. Merchant, 1998, Distributed coordination with messengers. Science of Computer Programming, 31(2).. Special Issue on Coordination Models, Languages, Applications.

M. Fukuda, L.F. Bic, and M.B. Dillencourt, 1997, Performance of the messengers autonomous-objects-based system. In First Int'l Conf. on Worldwide Computing and Its Applications (WWCA97), Tsukuba, Japan. Springer-Verlag. Lecture Notes in Computer Science 1274.

E. Gendelman, L. F. Bic, and M. B. Dillencourt, 2000, An application-transparent, platform-independent approach to rollback-recovery for mobile-agent systems. In ICDCS 2000: 20th International Conference on Distributed Computing Systems, Tapei, Taiwan.

E. Gendelman, L. F. Bic, and M. B. Dillencourt, 2001, A fast file access for fast agents. In MA 2001: Fifth International Conference on Mobile Agents, Atlanta, Georgia.

R.W. Hockney and J.W. Eastwood, 1988, Computer Simulations using Particles. IOP Publishing Ltd, Bristol, Great Britain, 1988.

M. Huston, D. DeAngelis, and W. Post. New computer models unify ecological theory. BioScience, 38(10):682–691.

A. Huth and C. Wissel, 1992, The simulation of the movement of fish schools. Journal of Theoretical Biology, 156:365–385.

M. Resnick, 1994, Changing the centralized mind. Technology Review, pages 33–40.

C.W. Reynolds, 1987, Flocks, herds, and schools: A distributed behavioral model. Computer Graphics, 21(4):25–34.

F. Villa, 1992, New computer architectures as tools for ecological thought. Trends in Ecology and Evolution (TREE), 7(6):179–183.

C. Wicke, L. F. Bic, and M. B. Dillencourt, 1999, Compiling for fast state capture of mobile agents. In Parallel Computing 99 (ParCo99), Delft, The Netherlands.

C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda, 1998, Automatic state capture of self-migrating computations in messengers. In MA'98: Second International Conference on Mobile Agents, Springer-Verlag, Lecture Notes in Computer Science, Stuttgart, Germany.