Messages versus Messengers in Distributed Programming

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

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

e-mail: {mfukuda, bic, dillenco, fmerchan}@ics.uci.edu

Abstract

Messengers are autonomous objects, each capable of navigating through the underlying network and performing various tasks at each node. Messengers applications are written using navigational *commands rather than the* send/receive *primitives of conventional message-passing approaches. In this paper we contrast the two programming styles. The navigational style generally results in a smaller semantic gap between abstract algorithm descriptions and their actual implementations, which makes programs easier to construct, understand, and maintain. Other advantages of the navigational programming style include the ability to compute in unknown or dynamically changing network topologies.*

1 Introduction

The objective of this paper is to contrast two different styles of programming in distributed systems. The first, which is by far the most prevalent among all distributed programming languages, is based on *message-passing*. The application is viewed as a collection of concurrent tasks (processes, threads), which communicate with one another by exchanging passive data messages. This is accomplished using low-level send/receive primitives or some higher-level abstraction, like remote procedure calls or rendezvous. The second approach is based on the concept of *autonomous messages*, that is, messages that have their own identity and behavior, which permit them to actively navigate through the underlying computational network, perform various tasks at the nodes they visit, and coordinate their activities with other such messages constituting the distributed computation. To distinguish such autonomous messages from simple passive data messages, we refer to them as *Messengers*.

Autonomous messages have been used for a variety of applications, as will be discussed in Section 4. In this paper we concentrate on general-purpose distributed computation, as developed in WAVE [19], BPEM [6], and most recently, our MESSENGERS system [5]. In these systems, autonomous messages serve as active mobile entities that coordinate the computation in both time and space.

There are two fundamental characteristics that distinguish Messengers-based systems from message-passing systems:

- Messengers are capable of creating and using a *logical* network. Nodes may contain arbitrary variables or data structures, while links may be used by a Messenger for navigation, including replicating itself by following multiple links. The logical network thus represents a data structure *external* to and independent of any ongoing activity. Unless explicitly destroyed, it will persist even after all Messengers have ceased to exist. To use a biological metaphor, the logical network is like the *exogenous* skeleton of crustaceans and other low-level organisms, which defines the organism's external form and is preserved beyond the organism's lifespan. In contrast, the structures that bind together conventional message-passing tasks are *internal* to the tasks. Each task will typically contain information about other tasks with which it may communicate. This information may be passed to the task at creation or determined dynamically. In either case, it is similar to an *endogenous* skeleton of higher-level organisms and will disappear when the tasks terminate. This has important implications for the spawning of new activities. Under message-passing, the focus is on *task* creation—a new task is spawned by an existing task and the parent-child relationship is made known to the respective tasks. Under Messengers, the focus is on *node/link* creation. The nodes and links form the places where activities can occur.

- All functionality of the application is embedded in the individual Messengers, i.e., the programs

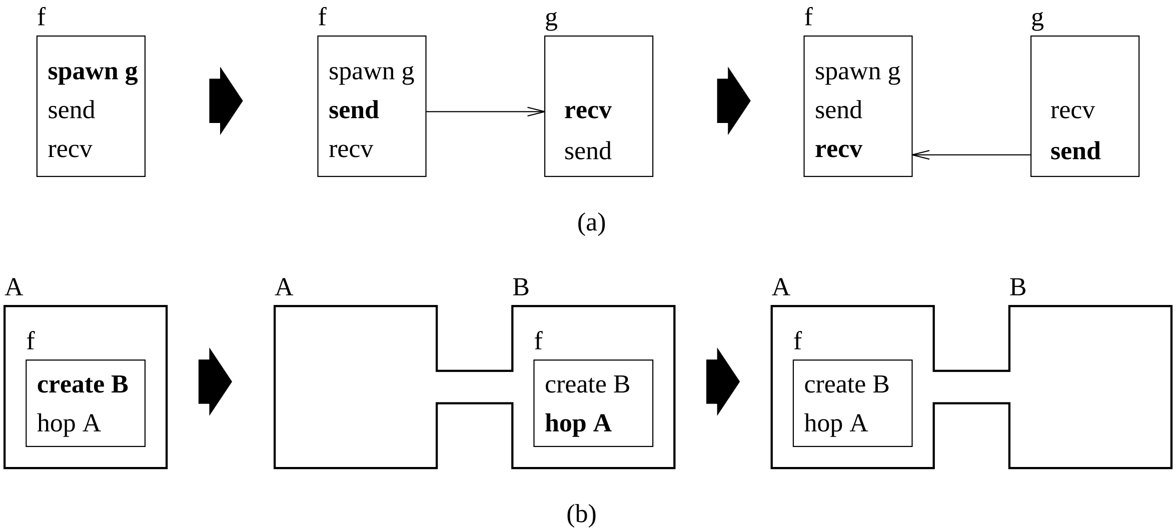

Figure 1: (a) Task creation and message-passing. (b) Logical node creation and navigation

> carried by Messengers as they navigate through space. This is in stark contrast to message-passing, where the communicating tasks contain all the functionality, while messages are only passive carriers of data. Hence *send/receive* operations are replaced with *navigational* operations that move a Messenger between nodes.

Figure 1 illustrates the above philosophical differences graphically. Part (a) shows a task f, which first spawns a child task g, as indicated by the bold font. The two concurrent tasks then exchange messages with one another using send/receive operations. Part (b) shows a logical node A, currently containing a Messenger f. This Messenger creates another logical node, B, connected to A by a logical link. As part of the create operation, the Messenger automatically moves to the new node B. It may continue executing there or, as shown in the figure, it may move back to A using a navigational operation (hop). We note the following fundamental differences between the two programming styles: (1) Under message-passing, two distinct programs, f and g, must be developed; (2) Under message-passing, there are two concurrent activities to deal with at runtime. The execution of these two activities must be carefully coordinated to ensure a proper alignment of the corresponding send/receive pairs. Under Messengers, there is only a single locus of activity at any point in time; (3) Under Messengers, the logical network (nodes A and B, and their connecting link) continues to persist even when the Messenger f terminates or moves to some other logical node. The logical network is available for use by other Messengers that may be created as progenies of f or injected independently by either the user or another Messenger.

Following an overview of our system in Section 2, we illustrate the new programming style imposed by Messengers by presenting examples from two different application domains (Section 3). Our examples 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 [5, 12] is a system that supports the development and use of distributed applications structured as collections of autonomous objects, called Messengers[1]. 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. A daemon's task is to continuously receive Messengers arriving from other daemons, interpret their behaviors, described as programs carried as part of each Messenger, and send them on to their next destinations as dictated by their behaviors.

The MESSENGERS system involves three levels of

[1] The individual autonomous objects are denoted by mixed case (Messengers), while the system as a whole is denoted by small capitals (MESSENGERS).

networks. The lowest level is the *physical network* (a LAN or WAN), which constitutes the underlying computational nodes. Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process running a MESSENGERS language interpreter. The *logical network* is an application-specific computation network created on top of the daemon network. At system startup, a single logical node, named *INIT*, is created on every daemon node. Any Messenger may be injected (from the shell or by another Messenger) into any of the INIT nodes and it may start creating new logical nodes and links on the current or any other daemon.

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C and are compiled into a form of byte code for more efficient transport and parsing [7]. 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. This requires a programming style different from the commonly used approaches to distributed computing. 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. 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.

A Messenger script is a sequence of statements, which can be of one of the following types: (1) Computational statements enable the Messenger to perform arbitrary computations. They include all standard C assignment and control statements, involving arbitrary variables and constants; (2) Navigational statements endow the Messenger with mobility, permitting it to create and destroy logical nodes and/or links, and to move within the logical network; (3) Function invocation statements provide an interface to the system's environment. They permit the dynamic loading and invocation of precompiled C functions to be executed in native mode.

Given the focus of this paper, we will concentrate on only the navigational statements. The remainder of this section will introduce the basic syntax and semantics of the statements *hop*, *create*, and *delete*.

The *hop* Statement. The *hop* statement permits a Messenger to move around the logical network. Its syntax is as follows:[2]

$$hop(ln = n; ll = l; ldir = d)$$

where ln stands for "logical node", ll stands for "logical link", and $ldir$ stands for the link's direction. Together, the triple (n, l, d) is a destination specification in the logical network where n can be an address, a variable, a constant (including the special node INIT), or a wild card ($*$) that matches any name; l can be a variable, a constant, a wild card, or a "virtual link" (corresponding to a direct jump to the designated node); finally, d can be one of the symbols $+$, $-$, or $*$, denoting "forward," backward," or "either," respectively. The default for all three parameters is $*$ and thus may be omitted.

The semantics of the *hop* statement are as follows: From the current node c, replicate the Messenger to all nodes that match n and are connected to c by links matching l and d. The Messenger executing the *hop* in node c then ceases to exist.

Examples (showing the complete syntax and the equivalent default forms):

- $hop(ln = a; ll = x; ldir = *)$
  $\equiv hop(ln = a; ll = x)$
  from the current node c replicate the Messenger to all nodes a connected to c by link x (regardless of link direction)

- $hop(ln = *; ll = *; ldir = -)$
  $\equiv hop(ldir = -)$
  from the current node c replicate the Messenger to all nodes connected to c by a backward-oriented link

- $hop(ln = *; ll = *; ldir = *)$
  $\equiv hop()$
  from the current node c replicate the Messenger to all nodes connected to c, i.e., all neighboring nodes

The *create* Statement. The *create* statement permits a Messenger to create new logical nodes and/or links. Its syntax is as follows:

$$\begin{aligned} create(&ln = n_1, ..., n_k; ll = l_1, ..., l_k; ldir = d_1, ..., d_k;\\ &dn = N_1, ..., N_k; dl = L_1, ..., L_k; ddir = D_1, ..., D_k;\\ &[ALL]) \end{aligned}$$

where each triple (n_i, l_i, d_i) specifies a *new* logical node, n_i, connected to the current node by a (possibly directed) link, l_i. The node n_i is created on the

[2] The syntax shown here is slightly simplified. In its full generality, a single *hop* statement supports multiple hop specifications, similar to the *create* statement discussed below.

daemon node specified by the triple (N_i, L_i, D_i), which is a destination specification in the daemon network.

The semantics of *create* are then as follows: For each destination specification (N_i, L_i, D_i) determine the set of daemon nodes that match the given daemon name (N_i), daemon link (L_i), and the daemon link direction (D_i). If the optional parameter ALL is omitted, choose one of the possible daemons and create the logical node n_i on it. (The choice is made by a set of rules that are beyond the scope of this paper [13].) The newly created logical node n_i can be named (using a variable or constant) or unnamed ($\sim$), and can be connected to the current node c by the link l_i, which could be named (using a variable or constant) or unnamed ($\sim$). When the optional parameter ALL is included, the new node n_i is created on *all* the daemons matching the destination specification and a copy of the Messenger continues executing in each of the newly created nodes n_i.

The defaults for the logical network parameters n_i, l_i, d_i are "$\sim$" and those for the daemon network parameters N_i, L_i, D_i are "$*$".

Examples. Assume that a MESSENGERS is executing in a logical node c mapped onto a daemon node C.

- $create(ln =\sim; ll =\sim; ldir =\sim;$
  $dn = *; dl = *; ddir = *; ALL)$
  $\equiv create(ALL)$
  create an unnamed node connected by an unnamed and undirected link to c on all neighboring daemons, i.e., those connected to C by any daemon link.

- $create(ln = a, b; ll = x, y; ldir =\sim, \sim;$
  $dn = *, *; dl = *, *; ddir = *, *)$
  $\equiv create(ln = a, b; ll = x, y)$
  create a connected to c by x on a neighboring daemon, and create b connected to c by y on another neighboring daemon

The *delete* Statement. The *delete* statement has the same syntax as *hop* and performs the same navigational operations. However, in addition to moving among nodes, it also deletes all logical links it traverses. If a node becomes a singleton, it is also deleted.

3 Applications

3.1 Smart Workers vs. Smart Managers

Consider the program in Figure 2, which shows a message-passing implementation of the common manager-worker (or master-slave) paradigm. This

```
 (1) manager(){
 (2)  for (i = 0; i < ntask; i++)
 (3)    child[i] = spawn(worker);
 (4)  for (i = 0; i < ntask; i++)
 (5)    send(worker[i], next_task() )
 (6)  while(next_task() != NULL){
 (7)    res = recv(any_worker)
 (8)    i = who_sent(res)
 (9)    send(worker[i], next_task() )
(10)    deposit(res) }
(11)  for (i = 0; i < ntask; i++){
(12)    res = recv(any_worker)
(13)    i = who_sent(res)
(14)    kill(worker[i] )
(15)    deposit(res) }
(16) }

(17) worker(){
(18)   while(TRUE){
(19)     task = recv(manager)
(20)     res = compute(task)
(21)     send(manager, res) }
(22) }
```

Figure 2: Manager-Worker using message-passing

program is an adaptation of a PVM program [15], where unnecessary details, such as data packing/unpacking, buffering, etc, have been abstracted away for clarity. The program is divided into two distinct functions. The behavior of the worker is very simple: it repeatedly receives a task (line 19), computes the result (line 20), and sends it back to the manager (line 21). The manager is more complicated. It first spawn a number of workers (lines 2-3) and sends each one a task, obtained using the function *next_task()*, to work on (lines 4-5). It then enters a while loop (line 6) during which it repeatedly receives a result from any worker that finished its work, determines the worker's ID, sends it the next task to work on, and deposits the received result for subsequent processing and output (line 7-10). When there is no more work to do, the manager enters the final loop (line 11) during which it collects the remaining results and kills the workers (lines 12-15).

The corresponding Messengers version of this program is shown in Figure 3. This shows the script of a single Messenger, which is injected into the INIT node of some daemon. Its first course of action is to create logical nodes connected to the current node on every neighboring daemon (line 2). This also causes a replica of the Messenger to be created, each of which

```
 (1) manager_worker(){
 (2)  create(ALL)
 (3)  hop(ll = LAST)
 (4)  while((task = next_task()) != NULL){
 (5)    hop(ll = LAST)
 (6)    res = compute(task)
 (7)    hop(ll = LAST)
 (8)    deposit(res)
 (9)  }
(10) }
```

Figure 3: Manager-Worker using Messengers

now executes in the new node while the original Messenger disappears; that is, it has cloned itself into a number of independent "workers". Each of the Messengers hops back to the original node by following the most recently traversed logical link `LAST` (line 3). It then attempts to get a new task to work on (line 4). If successful, it hops back to its logical node—again by following the most recently traversed link (line 5)—where it computes the result (line 6), carries it back to the central node (line 7), and deposits it there (line 8). It continues hopping back and forth, each time solving a new task, until there is no more work to do. At that point it ceases to exist.

Even though both programs implement the same basic paradigm, the MESSENGERS program offers two important advantages:

- The Messengers code is conceptually much simpler as it only deals with one program. The message-passing version consists of two separate programs resulting in two types of concurrent activities—a manager who embodies most of the "intelligence" and a "mindless" worker. That is, the manager is in charge of the entire operation, including spawning the workers, supplying them with work, and killing them when all work is completed, while each worker only processes the tasks supplied to it. The Messengers version makes the workers "intelligent," thus eliminating the need for a manager. Each worker is capable of shuttling autonomously between the central node containing the tasks and its own node where it carries out its work. This is possible for two reasons. First, Messengers are mobile and hence can get their own work. Second, the logical network created initially (line 2) provides the "exogenous skeleton" for the application. The logical nodes can be thought of as work areas connected to the central node by corridors. Each worker can literally travel to the central place to take a new task and then carry it to its own work place to solve it. This distributes all the complexity of the central manager over the individual workers. The main reason why this is not possible with a non-navigational language is the inability to maintain a data structure on any node without some activity (process) associated with it. Thus, even if we endowed the worker processes with the necessary intelligence to autonomously request new tasks and deposit the results (e.g. using RPCs), an active manager process would still be necessary to "guard" the central repository of tasks and partial results.

- The Messengers program is considerably shorter. This despite the fact that the message-passing version is only written in pseudo code—a lot of detail would have to be added to make this program run under PVM or any other system. The Messengers code, on the other hand, is very close to executable (the main difference is in the syntax of the function calls). The reduction in code length is not the result of using a high-level language; Messengers programming is comparable to message-passing in that both deal with parallelism explicitly. Specifically, our *create* is comparable to a *spawn* operation in PVM while a *hop* is similar to a *send/receive* operation. The reduction is due to our reformulation of the manager-worker paradigm that distributed the intelligence of the central manager over the workers, which is not possible using message-passing.

3.2 Information Diffusion/Collection

The control of large heterogeneous networks requires the ability to deal with a dynamically changing topology as physical nodes are dis- and re-connected, to query their status and to disseminate information to them. To illustrate the ability of MESSENGERS in this context, we consider the relatively simple 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. A standard solution to this problem using conventional send and receive primitives, taken from [2], is shown in Figures 4(a). It uses neighbor set communication to send the message from one node to all its active neighbors.

Figure 4(a) shows the node program that must be running on each node participating in the diffusion. The diffusion is initiated by sending a message (*msg*) to any of the nodes. Upon receiving a message (line

```
(1) diffuse() {
(2)   msg = receive()
(3)   for (i = 0; i < nneighbors; i++)
(4)     send(neighbor[i], msg);
(5)   for (i = 0; i < nneighbors-1; i++)
(6)     msg = receive();
(7)   deposit(msg);
(8) }
```

(a)

```
(1) diffuse(msg) {
(2)   repeat
(3)     create(ln = INIT; ALL)
(4)     if (visited == FALSE) {
(5)       deposit(msg);
(6)       visited = TRUE}
(7)     else exit
(8) }
```

(b)

Figure 4: (a) Diffusion using conventional message-passing. (b) Diffusion using MESSENGERS

2), the node sends that message to all its neighbors (lines 3-4). To avoid leaving extra messages buffered on the network and to correctly detect termination of the distributed computation, each node then waits to receive ($nneighbors - 1$) redundant messages sent by its neighbors. These redundant messages replace the explicit acknowledgments that would otherwise have to be sent before the node programs could safely terminate.

Figure 4(b) shows a diffusion program using MESSENGERS. The program, which can be injected into any node of the network, works by constructing a spanning tree "on the fly," and using this spanning tree to spread the message. The spreading is accomplished using the *create* statement (line 3), which accomplishes navigation using the daemon rather than the logical links. It attempts to recreate the INIT node on each of its neighboring daemons. Since duplicates are not allowed, its effect is to only replicate the Messenger and move each copy into one of the INIT nodes. Each receiving INIT node continues interpreting the next command (line 4), 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 6). The Messenger then continues executing its repeat loop (line 2), which again replicates it to all neighboring daemon nodes. If the condition on line 4 fails, indicating that the node has already been visited, the Messenger dies (line 7).

The MESSENGERS program is similar to the message-passing program in that only a knowledge of the neighbors (rather than knowledge of the entire network topology) is required. Although the programming styles are quite different, both programs are easy to understand. There are, however, several fundamental differences between the message-passing program and the MESSENGERS program:

- The message-passing program is not tolerant of node failures/disconnections. Node failures can result in expected messages not being received, which can prevent the program from terminating. If nodes are reconnected during the running of the program, it can result in premature termination of the program at a node, leaving extra buffered messages on the network. These anomalies occur because each node program must decide when to terminate based on its interaction with its neighbors. The node program makes this decision by counting the number of messages it receives, and node failures or reconnections can disrupt the count.

  The MESSENGERS program, on the other hand, always explores 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 interpreter and thus automatically become reachable by Messengers. The main reason for the increased tolerance of node failures is that a Messenger decides whether to terminate solely on the basis of whether the current node has been visited and does not have to count messages.

- The message-passing algorithm requires more data communication operations than the MESSENGERS algorithm, since each link in the spanning tree is traversed twice (by a message) in the message-passing algorithm and once (by a Messenger) in the MESSENGERS algorithm. Links that are not part of the spanning tree are traversed twice in both cases. The additional data communication overhead incurred by the message-passing algorithm is minor if the network is a dense graph (e.g., a fully connected topology), but it may be significant if the network is a sparse graph (e.g., a planar grid.)

- The message-passing program is useful only when *multiple diffusions* are to be performed, using the same algorithm. This is because initiating the message-passing computation requires sending the node program of Figure 4(a) to each node by some mechanism and then initiating the node program. If only one message is to be diffused, the mechanism used to send the program could be used to send the message itself to each node rather than the program. Changing the dissemination algorithm or running a different network application also requires the corresponding application programs to be sent to each node. With MESSENGERS, all application-specific information is carried by the Messengers while all nodes run the same (application-independent) interpreter, which runs continuously in every node.

4 Related Research

Autonomous messages have been employed in the construction of "intelligent" mobile agents, which are programs capable of physically moving through wide-area communication networks (notably the Internet) and performing a variety of service tasks on behalf of their users [17]. While MESSENGERS and mobile agents share the same basic principle of autonomous navigation, they differ with respect to both intent and structure. Mobile agents typically aim at providing a vehicle for using various distributed services available on wide area networks; MESSENGERS, in contrast, is aimed at general-purpose computing. MESSENGERS provides a logical network, whose size and topology are completely independent from the underlying daemon or physical networks. This is used by the application to organize and manage its data, which is accessed using powerful navigational commands that include implicit replication based on a navigational calculus [13]. In contrast, mobile agents navigate directly in the physical network, and the commands used for this purpose are generally limited to simply transferring the agent to another node.

There have been previous systems applying autonomous messages to general-purpose distributed computing, seeking to harness the computational capacity of the underlying network to speed up some computational task. Two systems, BPEM and WAVE, were among the first to explore these concepts. BPEM [6] is a computational model designed to facilitate the parallel processing of knowledge, represented in the form of semantic nets. WAVE [19] is a complete environment consisting of a specialized language to express arbitrary autonomous objects behaviors, and a run-time system of interpreters. Both WAVE and BPEM had a major influence on the MESSENGERS system, presented in this paper, which uses some of their basic navigational principles.

A third area of related research is coordination paradigms [1, 10], which provide the synchronization, communication, and creation/destruction of computational activities required to orchestrate the individual computations into a coherent system. Examples of coordination paradigms include Gamma [4], Linda [8], and the IWIM model [3]. MESSENGERS is similar in some respects to PoliS [9], a variant of Linda intended to simplify the design of distributed systems by incorporating explicit partitioning of the underlying state space (called a "tuple space" in the Linda model). One difference is that in PoliS, computational objects (called "agents") communicate by writing messages in other agents' tuple spaces. Thus an agent would move from one tuple space to another by writing a copy of itself in a new tuple space and then finishing its execution in the old tuple space (so navigation is a biproduct of communication). The MESSENGERS point of view, by contrast, is strongly navigation-oriented. Messengers communicate though a rendezvous at a logical node and an exchange of information through node variables. Messengers scripts, which are written from the point of view of a moving entity, are used for all aspects of coordination, including dynamic creation and destruction of spaces (logical nodes), creation and destruction of activities (Messengers), and inter-Messengers communications.

5 Conclusions

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

The MESSENGERS philosophy has been shown to yield a number of important properties that are much 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. Furthermore, as the examples presented here demonstrate, 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, including access to the more detailed technical reports about MESSENGERS referenced in this paper, the interested reader is invited to browse our WWW page: `http://www.ics.uci.edu/~bic/messengers`.

References

[1] J.-M. Andreoli, C. Hankin, and D. Le Métayer (Eds). Coordination Programming: Mechanisms, Models, and Semantics. Imperial College Press, London, UK, 1996.

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

[3] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, pages 34–56, Cesena, Italy, April 1996. Springer Verlag Lecture Notes in Computer Science 1061.

[4] J-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, 36(1):98–111, Jan. 1993.

[5] L. Bic, M. Fukuda, and M. Dillencourt. Distributed Computing using Autonomous Objects. *IEEE Computer*, Aug. 1996.

[6] L. Bic and C. Lee. A data-driven model for a subset of logic programming. *ACM TOPLAS*, 9(4), Oct. 1987.

[7] B. Bidyuk. MESSENGERS-C Compiler Manual. Report MSGR-06, University of California, Irvine, 1996.

[8] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, Apr. 1989.

[9] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251–284, 1994.

[10] P. Ciancarini and C. Hankin (Eds.). Coordination Languages and Models, Proc. First Int'l Conf, COORDINATION'96. Cesena, Italy, Apr., 1996.

[11] M. Fukuda. MESSENGERS System Library Manual. Report MSGR-02, University of California, Irvine, 1996.

[12] M. Fukuda, L. F. Bic, and M. B. Dillencourt. MESSENGERS: A Distributed Computing Environment for Autonomous Objects. Technical report, TR 96-20, Information and Computer Science, University of California, Irvine, 1996.

[13] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant, Distributed Coordination with MESSENGERS. Technical report, TR 97-10, Information and Computer Science, University of California, Irvine, 1997.

[14] T.H. Geesink. The coordination of distributed active messages in a dynamic network topology. *The Computer Journal*, 34(6), 1991.

[15] A. Geist, A. Beguelin, J. Dongarra, W. Jieng, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, 1994.

[16] A. Lingnau, O. Drobnik, and P. Dömel. An HTTP-based infrastructure for mobile agents. In *Fourth International World Wide Web Conference Proceedings*, pages 461–471, Sebastopol, CA, December 1995. O'Reilly and Associates.

[17] MA '97: First International Conference on Mobile Agents. Berlin, Germany, Apr., 1997. Springer-Verlag Lecture Notes in Computer Science, to appear.

[18] P.W. Richardson and T. Danielsen. Intelligent messages or when messages come alive. In K. Boyanov and R. Angelinov, editors, *Network Information Processing Systems*. North-Holland, 1989.

[19] P.S. Sapaty and P.M. Borst. An overview of the WAVE language and system for distributed processing of open networks. Technical report, University of Surrey, UK, 1994.