Distributed Computing using Autonomous Objects[1]

Lubomir F. Bic
Munehiro Fukuda
Michael B. Dillencourt

Department of Information and Computer Science
University of California, Irvine, CA 92717

Abstract

Autonomous Objects are a new paradigm for distributed systems, based on the concept of intelligent messages that carry their own behavior as they navigate autonomously through the underlying computational network.

Keywords: distributed computing, automonous messages, simulation, process migration, open-ended applications, individual-based models, coordination

Contact author: Lubomir Bic
`bic@ics.uci.edu`
Phone: (714)-824-5248

Revised February 26, 1996

[1] **This is a draft; the final copy-edited version will appear in IEEE COMPUTER in 1996**

1 Introduction

Most distributed systems today are structured as collections of statically-compiled communicating processes. These are mapped onto the different nodes of the underlying computational network and they communicate with one another via messages. What is characteristic of these message-based paradigms is that the system's control or "intelligence" is embodied in the communicating node programs, while messages carry only simple pieces of information in the form of passive data. We will refer to these as *Communicating Objects* paradigms.

In this paper we concentrate on several novel paradigms for distributed computing, which depart from the above conventional view by elevating messages to higher-level entities that embody some degree of autonomy or "intelligence". A message can be viewed as an object itself, which has its own identity and which can decide at runtime where it wishes to propagate next and what tasks it is to perform there. In the extreme, an approach completely complementary to the Communicating Objects paradigm can be considered, where only messages are the active components performing all computations, while the nodes are generic *interpreters*, enabling the messages to navigate through the network and carry out the computations specified by each message. In this case, all "intelligence" of the application is embodied in and carried by messages as they propagate through the network, much like a human agent or a robot would move in space, visiting or examining different locales, in the process of performing its tasks. We shall refer to this paradigm as *Autonomous Objects*.

The first objective of this paper is to survey and classify several seemingly disparate systems by showing that they all embody the general philosophy of autonomous objects. In Section 3 we will then present a new system based on the Autonomous Objects paradigm, called MESSENGERS, which embodies and extends many of the features of the systems surveyed in Section 2. We demonstrate the paradigm's strengths and weaknesses vis-a-vis the better known Communicating Objects paradigm by presenting solutions to problems from different application domains in Section 4.

2 Autonomous Objects

In this section we survey several recent paradigms where messages have been elevated from being simple carriers of passive data to a higher form, such that some behavioral information may be carried by each message and interpreted or executed by the receiving sites. This gives messages a certain degree of autonomy in navigating through the underlying network and allows them to be viewed as first-class objects.

2.1 RPCs, Method Invocations, and Remote Evaluation

Remote Procedure Calls (RPCs) [BN84] are one of the most common mechanisms for high-level communication in distributed systems, especially in client-server type applications. The basic idea is for the client to send a message containing the name of a procedure to be invoked on a remote node, together with the necessary parameters. The server then executes the procedure and returns the results in a reply message.

In terms of autonomy, RPC messages carry no navigational information other than their final destination. However, RPC's differ from ordinary low-level messages exchanged via send/receive primitives in that they are not passively received. Rather, they trigger actions by determining which function is to be invoked at the remote site. Thus they embody some degree of coordination capability.

Another recent approach to higher-level distributed programming is to use an object-based paradigm. An application based on this paradigm is viewed as a collection of objects residing at different sites and communicating with one another via messages, which invoke specified functions (called methods in object-based terminology) in a manner similar to a RPC. Hence, in terms of their autonomy and coordination capability, RPCs and invocations of methods are at the same level.

The basic RPC concept has been extended by a mechanism called Remote Evaluation [SG90]. The essential idea is for the caller to supply the procedure body to be evaluated on the remote computer. This code is carried by the request message along with the necessary parameters and the results are returned to the caller in a way analogous to RPCs. The main advantage of Remote Evaluation is that it allows the server's capabilities to be arbitrarily extended by providing new functionalities with each request.

2.2 Echo Algorithms

One of the first approaches to distributed computing based on the philosophy of propagating self-contained intelligent messages through a system of simple interpretive nodes were Echo Algorithms, developed to solve a variety of graph-based problems, such as finding shortest paths or biconnected components of a given graph [Cha82]. The basic idea of Echo Algorithms is to start a wave of messages spreading from one or more initial nodes into neighboring nodes until all network nodes have been visited. The forward propagating messages are termed explorers, since they replicate themselves according to the given topology into all possible directions. When a message reaches a node that has already been visited, it stops its forward movement and starts retracing its path to its origin. The returning messages result in a second wave, termed the echo, which is the reversal of the explorer wave. At each node, messages can collect information about the graph they are traversing. This information is then composed into a global solution to the given problem during the echo phase and reported to the original starting point(s) of the wave.

The main characteristic of Echo Algorithms is that asynchronous message passing may be used to explore properties of arbitrary networks without any centralized control, centralized memory, global clock, or any *a priori* knowledge of the network's topology. The capabilities of Echo Algorithms to solve complex problems, however, are limited by the fundamental structure of the paradigm, which is based on first creating a spanning tree within the underlying network using explorer messages and then retracing the paths of explorers using echo messages. Nevertheless, it is very natural for the programmer to think in terms of waves of autonomous messages when solving certain classes of graph-related problems. Hence this paradigm, together with Dijkstra's diffusing computations [DS80] can be viewed as the foundation for later approaches, where messages propagating autonomously through networks are the primary vehicle for solving problems.

2.3 BPEM

Another paradigm based on the principles of intelligent messages propagating asynchronously and autonomously through a network of interpreters is the Binary Predicate Execution Model (BPEM) model developed at the University of California, Irvine [BL87]. BPEM is a computational model designed to facilitate the parallel processing of knowledge, represented in the form of semantic nets. In a traditional implementation, these nets are viewed as passive repositories of knowledge, where programs are the active objects that search for and process the recorded information. BPEM[2] proposed a different philosophy in the processing of knowledge nets. It views the knowledge base as a network of labeled nodes and edges, and each query as a network template, consisting of the same nodes and edges as the underlying knowledge net, but also allowing free variables to be used as nodes or edges. Answering a query then corresponds to the problem of finding a topological match for a given template in the underlying knowledge net such that each free variable is bound to a node label in the knowledge net. The answer, if one exists, is the set of bindings for the free variables.

The above concept of expressing and solving certain classes of problems in the form of non-procedural graph matching has been incorporated naturally into other paradigms, including WAVE (described below) and MESSENGERS. The basic idea is to let a given graph template be carried by autonomous objects and used as a "road map" to navigate through the underlying network in the search for a topological match.

2.4 WAVE

Another representative of the Autonomous Objects paradigm is WAVE [SB94], developed at the University of Surrey, UK, and the University of Karlsruhe, Germany. Like BPEM and Echo Algorithm, WAVE also rejects the notion of precompiled programs running in each node of the underlying network and communicating with other programs via data messages. Instead, each message is a self-contained object, which carries along its complete functionality in the form of a specialized program, that is, instructions specifying what to compute and what to pass on to other neighboring nodes via messages. Unlike BPEM and Echo Algorithms, however, WAVE is a complete language, capable of expressing arbitrary computations and navigation strategies. Each WAVE program is a sequence of computational and navigational commands carried by a message. It is a completely autonomous object, which may travel and replicate itself through the underlying network of logical nodes (interpreters). When the receiving node encounters a computational command, it carries it out locally. When it encounters a navigational command, it propagates the program to one of more nodes in the network. This is repeated until the given program terminates, causing the corresponding WAVE program instance to cease to exist.

WAVE features an elaborate set of commands to control both computation and navigation, both of which are expressed using a highly-specialized low-level interpretive language. As part of the computation, WAVE programs may also spawn new shell processes, which may invoke arbitrary precompiled functions as their subcomputations and return their results back to the WAVE program. Hence WAVE, unlike BPEM or Echo Algorithms, is not limited to only autonomous objects computations but may also include conventional node-resident programs. WAVE is currently operational on SUN-based networks, in both LAN and WAN configurations.

2.5 "Intelligent" Mobile Agents

The term mobile agents has been used by a number of recent projects to describe autonomous, largely "intelligent" programs, capable of physically moving through wide-area communication networks (notably the Internet), and performing a variety of service tasks

[2] BPEM derives its name from the fact that a (directed) graph edge may be described textually as a binary predicate p(x,y), where x and y are two nodes and p is the edge label. Hence it is possible to describe arbitrary graphs in the form of logic clauses, where finding information in the net corresponds to the process of resolution in logic programming.

on behalf of their users.

The infrastructure necessary to support such mobile agents consists of specialized servers, capable of interpreting the agent's behavior, and communicating with each other using various protocols. For example, the infrastructure developed at the University of Frankfurt [LuPD95] is based on a specialized World Wide Web server using the HTTP protocol. Perhaps the best known representative of the mobile agents paradigm is Telescript, a commercial product developed by General Magic, Inc. (Mountain View, CA). While the details of the language have not yet been released, the general operational principles are known and have been reported on in trade journals and technical reports [Whi94].

At the core of Telescript are two languages, High Telescript and Low Telescript. The former is object-oriented and translated into the latter, which is interpreted rather than compiled. Using these languages, it is possible to write programs to describe the behaviors of autonomous objects (referred to as Intelligent Agents in Telescript). Similar to the objects in WAVE, a Telescript object is also autonomous in that it can request—as part of its behavior—to be sent to some other node in the underlying network. The receiving node then interprets the object's program, which may in turn result in the sending of new objects to other nodes.

While Telescript employs the same basic principles of interpretive autonomous objects as the previous three systems, it differs from these in its primary objectives. Echo Algorithms, BPEM, and WAVE all strive for high-performance distributed computing. Each attempts, in some way, to harness the power of the underlying network to speedup some computational task. Telescript, on the other hand, aims at providing a vehicle for "electronic commerce", that is, utilizing the various geographically distributed services provided on the Internet. One of its main goals is to reduce network communication, resulting from client-server interactions. This is achieved by dispatching an autonomous agent to the remote server site where it performs the necessary interactions locally. When the task is completed, it carries the answer back to the original client. Hence, instead of a low-level interactive negotiation between the client and the server, only a single "round trip", traveled by the agent, is necessary.

2.6 Intelligent Email

Another line of research related to autonomous objects contains several related approaches to creating what we shall collectively refer to as Intelligent Email systems. The objective is to elevate electronic mail messages from simple carriers of data to entities of a higher rank, called active messages, intelligent messages, active entities, or envoys by the different projects [Gee91, RD89]. The purpose is to allow messages, having reached their remote destinations, to perform actions of their own, including collecting data, interacting with other processes or users on the remote host, or sending themselves to other destinations. An example of an active message application given in [RD89] is the publicizing of a conference. Initially, an active message would be sent to a list of potential attendees. Once at their destinations, each message would ask its recipient for additional names of potential participants, to whom it would send a copy of itself, thus incrementally disseminating the announcement through the network. Intelligent Email is similar to Telescript in that autonomous agents are released into the communication network to perform intelligent tasks on behalf of their senders. Intelligent Email is more restrictive than Telescript in that it is confined to the electronic mail domain. Unlike Telescript, however, it does not require each participating host computer to run a special interpreter for the programs carried by the active messages.

2.7 Browsers

Hypertext and network browsers represent another type of application that fits into the Autonomous Object paradigm discussed in this paper. These programs are capable of following hypertext links. The user chooses the link to be followed and the local program downloads the corresponding data. This data may, in turn, contain links to other collections of data.

We distinguish two types of browsers, static and dynamic. Static browsers are represented by early versions of Web browsers such as Mosaic or Netscape. These do not have the ability to extend their behavior by invoking new functions that are not already part of their implementation. Consequently, they need to understand all the protocols and conventions used by the various pieces of information posted on the net. Any new features require recompilation.

Dynamic browsers include HotJava and other Java-enabled browsers such as Netscape 2.0. They are capable of dynamically downloading new functions in the form of "applets" (mini-applications) through the net and invoking them as part of the ongoing program execution. This is possible because the underlying languages (e.g., Java) are interpreted and hence portable across platforms.

2.8 A Classification

In this section we classify the various paradigms surveyed in the previous sections along two orthogonal axes. The first axis indicates the level of *navigational autonomy* of messages. By this we mean the degree to which a message can be viewed as an object with its own innate behavior, capable of making decisions about its own destiny, rather than being just a passive data object passed around by communicating nodes. We distinguish three levels along the axis of navigational autonomy: *none, potential*, and *inherent*. The first level (none) contains paradigms where messages contain no navigational information (other then their immediate destination). The third level (inherent) contains paradigms where the fundamental operational principles are based on messages representing completely autonomous objects with their own behaviors. The second level (potential) lies between these two extremes: it includes systems where messages could contain some navigational information but this capability has not been explicitly addressed by the developers.

The second axis captures the level of *dynamic composition* supported by the paradigm. This represents the extent to which autonomous objects can invoke functions compiled in —it native mode and resident at the nodes, thus incorporating them dynamically into their own behavior., and carry new functional behavior with them to the nodes. We distinguish four levels along this axis: *none, process-based, function-based (resident)*, and *function-based (carried)*. The first level consists of paradigms where messages have no ability to invoke node-resident functions or interact with other processes. The second level includes those where autonomous objects can invoke functions already resident (or accessible via the file system) on the receiving node by spawning a new process for their execution. The third level includes paradigms where autonomous objects can invoke and execute node-resident functions not only as separate child processes but by linking them dynamically into the currently running process. The fourth level includes paradigms where the autonomous objects can also carry with them a function specification in some form (e.g., source code, intermediate code, native code) and invoke it at the receiving node. These levels are cumulative, in the sense that the paradigms at a particular level also support the dynamic composition behaviors at all previous levels.

Figure 1 shows how the paradigms surveyed in the previous section can be catego-

NAVIGATIONAL AUTONOMY

DYNAMIC COMPOSITION

	none	potential	inherent
none	Static Browsers	Echo Algorithms	BPEM
process			WAVE
function (resident)	Methods RPC		Intelligent Email Mobile Agents
function (carried)	Dynamic Browsers	Remote Evaluation	MESSENGERS

Figure 1: Classification of autonomous objects paradigms according to navigational autonomy and dynamic composition.

rized based on the two criteria. Along the axis of navigational autonomy, RPC's and method invocations in (distributed) object-based systems fall into the first category: a message contains only the name of the procedure/method to be invoked, together with the necessary parameters. In terms of dynamic composition capabilities, they fall into the third category, since their main purpose is to invoke remote functions efficiently as part of the current server process (typically as a separate thread).

Network browsers support a form of "navigation," but this is different from the notion of navigational autonomy introduced in this paper. Browsers do not physically migrate between nodes. Rather, they import information from selected nodes. As with RPC's or object method invocations, there is no navigational autonomy: a request goes out over the net to a particular location and it brings back some new information. It is actually the user doing the navigating, rather than the browser and hence they are placed in the first column. In terms of dynamic composition, static browsers do not provide any support and thus are placed in the top row. In contrast, dynamic browsers are inherently extensible and thus appear at the fourth level. Their ability to download new functions and execute them is similar to remote evaluation. The main conceptual difference is that with remote evaluation the function is "exported" to another node as it is carried by a message, while dynamic browsers "import" new functionality by requesting it through the network. Another important difference is that, unlike remote evaluation, the interpretive nature of the underlying languages (e.g., Java) makes dynamic browsers inherently open-ended, as new functions can be linked to an ongoing computation at run time. External references are passed to the interpreter as symbolic names, which are resolved the first time the function is actually invoked. This is similar to MESSENGERS.

Echo Algorithms have the potential for navigational autonomy in message passing, even though this aspect has not explicitly been addressed. Applications could be structured as collections of interpreters while all navigational and computational information could be carried on messages. These would then function as fully autonomous objects exploring the underlying network in an asynchronous manner. With Remote Evaluation, each message could potentially carry an arbitrary program to be evaluated on a remote node. This program could in turn spawn other messages, carrying the same or a modified program to other nodes for remote evaluation. In this sense, messages under this paradigm can be considered autonomous objects capable of carrying both computational and navigational information. In terms of coordination capabilities, the two paradigms are very different. Remote Evaluation is intended to perform efficiently the invocation of pre-compiled functions carried through a network. Echo Algorithms, in contrast, have no dynamic composition capabilities. They have been designed as a computational paradigm for network algorithms where the only node programs executed are those that pass explorer and echo messages among nodes.

BPEM, WAVE, Telescript and Intelligent Email, are highly unconventional in their basic philosophy. In all four cases, the emphasis is on completely autonomous objects navigating and propagating through the underlying network. An object's behavior is encoded in the form of a complete program, which determines what is to be done at each receiving node and where the objects should be propagated next. Hence, along the navigational autonomy axis, all three belong to the last category. Their dynamic composition capabilities, however, are very different. BPEM, like Echo Algorithms, is a purely computational paradigm, designed to find given patterns in the underlying network. The emphasis in WAVE is also primarily on computing, accomplished by waves of autonomous objects navigating through the network, and communicating with one another by reading and writing node-resident variables. WAVE programs also possess a fair amount of coordination capability, but only at the process level. They can invoke UNIX shell processes, but they cannot dynamically link to node-resident functions. Telescript and Intelligent Email aim at providing autonomous agents capable of navigating the Internet. The main emphasis of both paradigms is on coordination. Each agent's task is to locate services, possibly by spawning subagents to facilitate the search, to negotiate with each service locally, and to coordinate the results for its original user. Although computational capabilities are secondary, employed only as part of the overall coordination process, both paradigms provide mechanisms for invoking node-resident functions in addition to their communication capabilities. The MESSENGERS paradigm, which will be described in Section 3, falls into the fourth category of coordination paradigms, which permit efficient execution of functions in all three forms: as separate processes, as embedded functions loaded via the file system, or as embedded functions carried along by the autonomous objects.

We wish to emphasize that the concept of *navigational autonomy* as used in this paper should not be confused with process or object *migration*, which is orthogonal to the former. While navigational autonomy refers to the ability of *messages* to navigate, migration refers to the ability of the *sending/receiving programs* to change location. The purpose is to reduce communication overhead during the remote operation. However, unlike Remote Evaluation, the object being moved has no navigational autonomy—it is *being* moved (in the passive sense) by a command executed by some other object rather than as the result of any of its *own* actions.

3 MESSENGERS

MESSENGERS is a general Autonomous Objects paradigm developed at the University of California, Irvine. It is implemented in C under UNIX on a network of Sun workstations and is currently being ported to other platforms. MESSENGERS is intended for the composition and coordination of concurrent activities in a distributed environment.

MESSENGERS' autonomous objects are called *Messengers*[3]. Their behaviors, including both computation and navigation, are expressed using C, as described below. The source programs are compiled into a machine-independent intermediate code, which is interpreted. For increased efficiency, Messengers' programs support the ability to directly call functions in native mode. This code can be made available at a particular node in one of three ways: it can be part of the node-resident code library, it can be carried to the node in native (precompiled) form, or it can be carried to the node as source code and compiled at the node. These options support the goal of dynamic composition, as defined in Section 2. They also provide a variety of tradeoffs between efficiency and run-time flexibility. In particular, they permit Messengers to dynamically change their run-time behavior even as they migrate between physical nodes of different architectures in a heterogeneous network.

3.1 The Language of MESSENGERS

During the course of executing an application, Messengers dynamically construct an application-specific logical network over a network of physical nodes (Sun Workstations in our present implementation). Multiple logical nodes can be mapped onto the same physical node, in which case the corresponding interpreter is multiplexed among these logical nodes. Execution of Messengers' programs within each logical node is serialized.

The language distinguishes two primary types of variables: *node* and *messenger* variables, which are associated with nodes and Messengers, respectively. The node variables are stationary (i.e. node-resident). The messenger variables carried by the Messenger as it propagates through the network. At any point in time, a Messenger has access to its own (messenger) variables and the node-resident variables of the node currently interpreting the Messenger. There is also a set of predefined variables, referred to as *network* variables, that describe the logical network and its mapping, such as node addresses, node names, link names, and link weights. The network variables associated with nodes behave like ordinary node variables, except that some (e.g., node address) cannot be modified. Link names and weights are associated with links; they can be accessed from either node incident on a link, but they can only be modified when the link is traversed.

Every Messenger program has the following form:

messenger variables;
node variable declaration;
functions;
S_1;
S_2;
...
$\rightarrow$ S_i;
...

Each S_i is either a computational or a navigational statement. The arrow is used to indicate the current statement, i.e., the one to be interpreted next. This corresponds to a program counter in a conventional language but must be made part of the Messenger, rather than the processor state, since the Messenger migrates between different nodes. "Messenger variables" denotes the set of variables local to and carried as part of the Messenger. "Node variable declarations" provide the necessary mapping information between the Messenger code and the node-resident variables. The actual allocation is done by the first Messenger to arrive at a node. Finally, "functions" denotes a (possibly

[3]We use capitalized lower-case when referring to the individual autonomous objects and upper-case for the entire system.

empty) set of functions carried by the Messenger for the purpose of remote evaluation. As noted above, these may be carried either in the form of native (executable) code or C source code.

MESSENGERS statements belong to three possible classes: computation, navigation, and function invocation. The first includes all standard C assignment and control statements, involving arbitrary variables and constants. The other two are expressed as C-functions, specific to MESSENGERS. The following is a simplified list of possible statements, S_i, constituting a Messenger program.

Computational statements:

1. *assignment:* A Messenger may read and update node variables and messenger variables, and read network variables. Arbitrary expressions are permissible in the assignment and include all of the common arithmetic and logic operators provided in C.

2. *control statement:* A Messenger may perform all the common control statements supported in C, such as if-then-else, while, do-while and break.

Navigational statements:

3. *create:* A Messenger may create new nodes or links. Optional parameters allow specification of the logical node name, the physical node, the link name, and the associated weight of the link.

4. *hop:* A Messenger may move to one or more other nodes by specifying its new destination(s) using a variety of optional parameters, including logical and physical node(s), link name(s), and allowable link weights. Wild cards are also allowed. A replica of the Messenger is propagated to all nodes that match the specification.

5. *delete:* A Messenger navigates as with *hop* but it also deletes all traversed links as it leaves the node. In addition, if this action erases all links from the departing node and there are no other Messengers currently residing in it, the node is deleted.

Function Invocation:

6. *exec:* A Messenger may invoke an executable function as a separate concurrent process.

7. *func:* Invoke a specified C function (precompiled in native mode) as a part of the MESSENGERS interpreter, wait for its completion, and return the results back to the invoking Messenger program. The function arguments are specified as part of the statement. The loading of the function is triggered dynamically when it is invoked for the first time.

3.2 The MESSENGERS Library

The MESSENGERS environment provides a library of functions that extend the basic capabilities of the language and its interpreter as described so far. The most important functions implemented thus far are those for creation and destruction of Messengers:

Messengers Creation: The creation of new Messengers is accomplished using the function *inject*. This creates a Messenger from a file containing the Messenger's behavior. It may also supply arbitrary initial parameters to the newly created Messenger. The function has two forms, depending on its intended use: one may be invoked from the

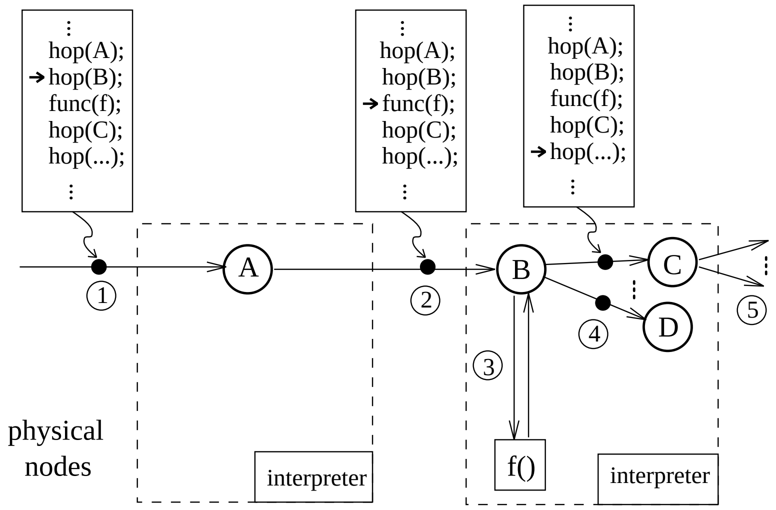

Figure 2: MESSENGERS Architecture

Unix shell, thus allowing the user to create new Messengers on the fly; the other is used inside Messengers programs, thus allowing Messengers to create progenies at run time.

The creation of Messengers using *inject* should not be confused with the automatic replication of Messengers during navigation. The latter is implicit and is the result of the Messenger's following multiple logical links during a *hop* statement. The function *inject*, on the other hand, causes an explicit creation of a new Messengers inside a node.

Messengers Destruction: A Messenger may terminate itself by executing the *exit* statement or it may be killed by another Messenger using a function *kill*.

3.3 The MESSENGERS Interpreter

The MESSENGERS system is implemented as a daemon running a language interpreter at each physical node. The interpreter exchanges Messengers with other interpreters, and multiplexes logical nodes mapped to its physical node.

For each new Messenger, the interpreter continues processing its statements until it encounters one that is navigational. At that point it passes it on to the appropriate destinations node(s) in the logical network. If this is within the same physical node, the Messenger is simply moved to the appropriate queue, where it awaits its turn as the interpreter is being multiplexed between the different logical nodes. If the destination is in a different physical node, the Messenger is sent there using Unix sockets.

Figure 2 illustrates these basic principles graphically. It shows two physical network nodes, each running the Messenger interpreter as one of its applications. A logical network of 4 nodes (named A through D) is mapped onto this architecture as shown. A Messenger is received by node A (marked as step 1 in the figure). Its current statement is assumed to be navigational (*hop(B))*, which causes it to be forwarded to node B (step 2). The next statement, *func(f)*, causes the invocation of the node-resident function f() (step 3). Upon returning from the function call, the interpreter carries out the next statement, *hop(C)*, which causes a replica of the Messenger to be sent along all outgoing links to the neighbors of the current node B (step 4). Node D discards the Messenger since it does not match the specified node name. Node C matches this label and thus continues interpreting the next statement, denoted by *hop(...)* in the figure (step 5).

4 Capabilities and Applications

The purpose of this section is to illustrate the capabilities of Autonomous Objects paradigms of the type represented by MESSENGERS.

4.1 Open-Ended Distributed Applications

The complex behaviors of many applications cannot be fully specified at the time of development or even after they have started operating. Typical examples are systems that require human intervention while the computation is in progress, e.g., simulation scenarios where humans are integral components of the system being simulated. Conventional simulation techniques require the entire experiment to be set up and compiled prior to starting the execution. This is clearly inadequate for interactive simulation. Autonomous objects, such as Messengers, offer a much more natural paradigm within which such applications can be implemented.

In collaboration with UCI's College of Medicine, we have developed several parallel pharmaco-kinetic models using MESSENGERS to simulate the distribution over time and the metabolism of various toxins by different organs of a living organism, which is of great interest to many branches of medicine, biotechnology, pharmacology. Figure 3 shows one such generic model, consisting of several organs and their interactions. The model is defined by a system of differential equations that govern the rates of absorption and excretion of each organ, the flow rate of blood through each organ, the concentrations of the toxins in the blood stream and in the various organs, their metabolic processes, and other possible interactions. The objective of the simulations is to solve these equations over a given simulated time interval to predict the levels of certain variables, such as the concentrations of toxins in the various organs, as functions of time.

Originally, the models were implemented sequentially, as simple integration routines driven by a fixed time increment. Using such a continuous time-driven simulation, it is very difficult to improve its performance through parallelization. Furthermore, it is impossible to interact with the simulation process, for example, by altering some of the equations or constants, once execution has started.

Our implementation uses MESSENGERS as a control language to coordinate the operation and interaction of compiled node-resident functions, which carry out the actual computations of the model. The basic approach is to map each organ onto a separate node (running the MESSENGERS interpreter). This node contains the necessary sets of differential equations and constants describing the organ's behavior. The toxin-carrying fluids, such as blood, are implemented as waves of consecutive Messengers, which cycle through the organism along the predefined paths, thus mimicking the actual flow through the body over time. As they pass through the organs, they trigger the execution of the appropriate functions to compute the new concentrations and other values for the current simulated time increment.

The circulation of the Messengers through the simulated organs can be driven by time or by other events, such as changes in certain variables. In the first case, the model mimics the original time-driven simulation, where each wave of Messengers that passes from the lung node to the other organs and back represents one time increment in the overall integration process. The lung node becomes the generator of the virtual time increments, which it sends to all other organs with each wave of Messengers. The time

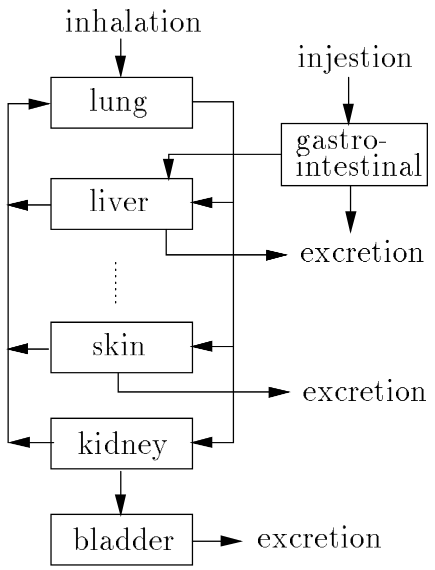

Figure 3: Toxicological Simulation Model

increments can be constant, or can vary with the rate of change in the computed variables to automatically maintain the accuracy of the computations at the same levels.

In the second case, the simulation becomes event- rather than time-driven. This is accomplished by sending Messengers between nodes only when some variable changes by a predefined threshold increment. This not only distributes message traffic more evenly but actually reduces the total traffic volume, since changes are propagated among nodes only when they become significant. At the same time, the model's fidelity increases, since its precision does not depend on an arbitrary time increment (which must be chosen conservatively small) but reflects the actual current changes in the variable. Making such a transition from a continuous to an even-driven simulation in a conventional implementation is very difficult, since the fundamental cyclic structure must be replaced by an event time line.

One of the main advantages of the above MESSENGERS implementations is their inherent parallelism, since each organ is logically a separate computational node. Furthermore, the parallelism is easily scalable to larger numbers of organs—something that cannot be done with the current sequential implementation. Additional parallelism, and, at the same time, a more precise model, can be obtained by subdividing individual organs into subregions using various grids. This then allows the modeling of different levels of toxins in different parts of the same organ as well as their diffusion through the tissues over time. The subdivision offers the possibility to map different grid cells on separate processors, thus further increasing parallelism.

In addition to performance gains through parallelism, the MESSENGERS implementation offers the potential for interactive simulation. In particular, it may be possible for one or even multiple users to observe the simulation as it unfolds and to interact with it by modifying its parameters or its functions. The latter is possible because the computations performed in each node are sequences of function invocations orchestrated by Messengers at runtime. Messengers can also carry new function definitions to all the

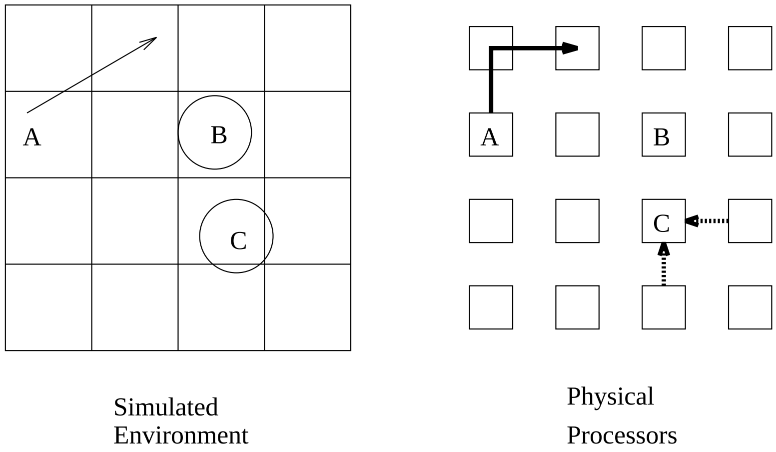

Figure 4: Logical and Physical Object Migration

nodes in the system, effecting dynamic unlinking and relinking of function definitions. Alternatively, a Messenger can circulate through the system, instructing each node to back up to an earlier time in the simulation or run backwards, and even carry with it the instructions for doing this. Hence the user may easily change or steer the computation while it is in progress by modifying or augmenting the Messengers circulating through the model.

4.2 Individual-Based Models

The toxicology application just presented is an example of *intra-object* coordination, in which the MESSENGERS paradigm coordinates lower-level computations to model the behavior of a single complex entity. We now consider another class of applications that illustrates *inter-object-coordination*, in which a collection of Messengers, each representing an individual entity in the application domain, coordinate their respective actions to model complex collective behavior in a two- or three-dimensional space. Typical examples of such applications include interactive battle simulations, particle-level simulations in physics, traffic modeling, artificial life applications, animation, and various individual-based simulation models in biology or ecology. We consider, as a representative example, the simulation of a school of fish. It has been observed that schools of fish are capable of performing complex maneuvers, for example, when avoiding a predator, 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 [HW92], a specific model has been formulated, where each fish periodically adjusts its position and velocity by coordinating its movement with up to four of its neighbors. Each individual fish has a circle of visibility with a certain radius r. It first chooses a small number of neighbors within its circle of visibility and then uses these neighbors to recompute its own vector of motion (speed and orientation). The neighbors are chosen according to certain rules, which give preferences to neighbors that are closest and those that are directly in front of the fish. Based on each of its neighbor's distance, speed, and orientation, the fish then determines a modification to its own speed and/or orientation.

MESSENGERS is a natural paradigm for these types of applications. The simulated environment, which in the simplest case is a two-dimensional ocean, is partitioned and distributed over the network of processing elements. Figure 4 shows a partitioning using a homogeneous grid where each grid cell is mapped onto one physical node.

Each individual fish is implemented as a Messenger, which carries its own behavior. This behavior could include stochastic models for breeding and dying, rules for interaction with fish of the same or other species (e.g., prey or predators), etc. For the above example of simulating schooling behavior, the behavior of each fish consists of swimming indefinitely in the simulated space and repeatedly adjusting its speed and orientation by interacting with its nearby neighbors as dictated by the biological model.

One important implication of the MESSENGERS paradigm 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 of a grid cell. It not only computes the new coordinates but the corresponding Messenger performs a *hop* statement (Section 3.1), which causes it to transfer into the processor responsible for the corresponding grid cell. Figure 4 illustrates the migration of a fish (A) in the logical and physical space.

The resulting close correlation between the simulated space and the underlying computer network significantly reduces inter-processor communication. The reason is that only fish in close proximity in the logical space need to communicate with each other. If they are in close proximity in the physical space as well, only a very small number of processors, if any, need to exchange any information. This problem, which is a manifestation of the general proximity detection problem in simulation, can be seen in Figure 4. If we can always guarantee that all fish in a particular grid cell are executing in the processor corresponding to that grid cell then a processor needs to communicate with only its local neighbors. For example, to examine the visibility range (radius r) of fish C, it only need to communicate with two neighbors. In the case of fish B, no inter-processor communication is required. In general, the need to communicate depends on radius r relative to the grid cell size.

In a conventional message-based implementation of the problem, guaranteeing that the logical environment is always perfectly correlated with the physical network is difficult to achieve due to the high cost of process migration. Hence objects are typically assigned to a given processor at the time of creation and remain stationary at that node for the duration of their execution. Due to their subsequent movement in the logical space, the logical neighbors of any given fish may reside on any processor. Hence each processor must either communicate with all other processors to detect the proximity of other fish, or some complex tracking mechanisms must be implemented.

A second important advantage of the MESSENGERS-based implementation of this application is that it is inherently open-ended. Its functionalities are not bounded by the number of predefined message types to which a node is able to respond. Instead, all functionalities are embodied in messenger programs and carried through the net. New entities with new behaviors can be created and injected into the simulation space on the fly as the simulation is progressing. Similarly, existing behaviors may be changed by modifying the Messengers' programs on the fly.

5 Conclusions

In this paper we have compared several novel paradigms for distributed computing where messages carry some amount of behavioral information and thus can be considered as having a certain degree of navigational autonomy through the network. We have also demonstrated the ability of autonomous objects to serve as a dynamic composition paradigm for applications consisting of node-resident compiled programs distributed throughout the network, and as a vehicle for inter-object coordination in both time and space. The need for specialized coordination paradigms has been argued for eloquently by Gelernter and Carriero in a recent article [GC92]. They observe that most existing programming languages focus primarily on computing, while leaving the aspects of communication (including I/O) and coordination to be handled outside the scope of the computing model, i.e., through ad hoc language extensions or library routines. They observe further that market forces of prepackaged software are already forcing a shift from creating new programs from scratch toward composing complex systems from existing program components. This will require the development of new coordination languages to facilitate the construction of such program ensembles.

The MESSENGERS paradigm, which incorporates the ability of autonomously navigating objects to efficiently invoke native-mode functions on remote nodes and to coordinate their executions, satisfies these requirements much better than other approaches, which typically offer an embedded set of primitives callable by processes to coordinate their activities. MESSENGERS allows the construction of arbitrarily complex control sequences, which are carried on messages through the network and which are capable of invoking node-resident computational programs as well as to coordinate their operation by carrying information among them. We have demonstrated these capabilities in the construction of two applications. The first—biomedical simulation models—are collections of sequential distributed functions threaded together at run time through waves of autonomous objects that cycle through the computational nodes. The second—individual-based models—are collections of entities, each with its own potentially complex individual behavior, coordinating among themselves to achieve realistic models of group behavior. The same basic principles are applicable to problems in other areas. Our current work includes the identification of potential application domains, as well as a further development of the MESSENGERS environment to increase its efficiency and to allow for dynamic utilization of available workstations.

References

[BL87] L. Bic and C. Lee. A data-driven model for a subset of logic programming. *ACM TOPLAS*, 9(4), Oct. 1987.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2, 1984.

[Cha82] E.J.H. Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Trans. Softw. Eng.*, SE-8(4), 1982.

[DS80] E.W. Dijkstra and C.S. Scholten. Termination detection in diffusing computations. *Inf. Process. Letters*, 16(5), August 1980.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, 35(2), Feb. 1992.

[Gee91] T.H. Geesink. The coordination of distributed active messages in a dynamic network topology. *The Computer Journal*, 34(6), 1991.

[HW92] A. Huth and C. Wissel. The simulation of the movement of fish schools. *Journal of Theoretical Biology*, 156:365–385, 1992.

[LuPD95] A. Lingnau and O. Drobnik und P. Doemel. An http-based infrastructure for mobile agents. In *4th International World Wide Web Conference*, Dec 1995.

[RD89] 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.

[SB94] 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.

[SG90] J.W. Stamos and D.K. Gifford. Remote evaluation. *ACM TOPLAS*, 12(4), Oct. 1990.

[Whi94] J.E. White. Telescript technology. Technical report, General Magic, Inc., Mountain View, CA 94040, 1994.