MESSENGERS: Distributed Computing using Autonomous Objects

Lubomir F. Bic
Department of Information & Computer Science
University of California, Irvine

Technical Report 95-19

May 30, 1995

Abstract

Autonomous Objects is a new computing and coordination paradigm for distributed systems, based on the concept of intelligent messages that carry their own behavior and that propagate autonomously through the underlying computational network. This is accomplished by running an interpreter of the autonomous objects language in each node, which carries out the tasks prescribed by the program contained in a received message. The tasks could be computational, including the invocation of some node-resident compiled programs, or navigational, which cause the message to be propagated to neighboring nodes. Hence interpretation is incremental in that each node interprets a portion of the received program and passes the rest of it on to one or more of its neighboring nodes. This is repeated until the given problem is solved.

We survey and classify several existing systems that fall into this general category of autonomous objects and present a unifying view of the paradigm by describing the principles of a high-level language and its interpreter, suitable to express the behaviors of complex autonomous objects, called Messengers. We discuss the capabilities and applications of this paradigm by presenting solutions to a wide spectrum of distributed computing problems. This includes inherently open-ended applications, such as interactive simulations, where it is not possible to define and precompile the entire experiment prior to starting its execution, and distributed computations where the underlying network topology is unknown or changes dynamically.

Contents

1. Introduction

At the most abstract level, a distributed computing system consists of a collection of nodes communicating with each other via messages. The computations at each node are instruction streams executing asynchronously and operating on local non-shared data. Message passing is used to achieve collaboration among the different asynchronous streams of control toward a common goal.

Within this generic computational framework, different programming paradigms have been developed to describe how data and control are to be distributed and how information can be communicated between the different streams. In most cases, a conventional programming language is extended with primitives to provide for explicit communication and synchronization [BST89]. This includes various forms of send/receive, broadcast, and multicast primitives, as well as some higher-level constructs, which are usually constructed on top of the lower-level primitives.

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. To use an analogy, these paradigms correspond to real-life scenarios where various agencies or individuals communicate with one another via mail, fax, or the telephone. 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. Note that

under this paradigm the interpreters need not be changed to solve different problems, which makes the application inherently open-ended. 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. We will classify these systems in Section 2 along two separate axes:

- the ability of an object to navigate through the network, to possibly replicate itself, and to decide what actions to take at each destination; this captures the degree of *autonomy* of each moving object
- the ability to *coordinate* the activities of a distributed application, which captures the ability of each autonomous object to invoke and control the execution of conventional sequential functions residing on various nodes throughout the network

In Section 3 we will then formulate a general framework for computing with autonomous objects by specifying the structure of a high-level language and its interpreter, suitable to express the behaviors of such objects, called Messengers. In Section 4, we demonstrate the paradigm's strengths and weaknesses vis-a-vis the better known Communicating Objects paradigm by presenting solutions to problems from a variety of application domains.

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, where 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 and Method Invocations

Remote Procedure Calls (RPCs) [BiNe84] 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. All message-passing details, including parameter marshalling and the handling of failures are hidden in the underlying RPC mechanisms, which make the call appear similar to a local call for the client.

In terms of autonomy, RPC messages carry no navigational information other than their final destination. Compared to ordinary low-level messages exchanged via send/receive primitives, however, RPCs embody, by their very design, some degree of coordination capability in that each message determines which function is to be invoked at the remote site. This distinction has been articulated, albeit at a lower level, in [ECGS92], where the concept of Active Messages has been introduced. An Active Message carries in its header a specification of a user-level handler that integrates the message into the current computation. The ability to choose its handler is similar to an RPC, which selects a procedure to be invoked. In fact, Active Messages have been termed ultra-light RPCs [TuMa94]. The main distinction is that an Active Message handler, unlike an RPC server, does not return to its sender. Furthermore, Active Messages are not a programming paradigm but a low-overhead asynchronous communication mechanism that can be used to implement such paradigm more efficiently.

Another recent approach to higher-level distributed programming is to use an object-based paradigm [Ame87, BLL88, DaCh88, GeRo88, YoTo87]. 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 methods invocations are at the same level.

2.2. Remote Evaluation

The basic RPC concept has recently been extended by a mechanisms called Remote Evaluation [StGi90]. The basic 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.

While the authors of Remote Evaluation have not explicitly addressed issues related to autonomy of messages, we include this work in this section because of its potential to provide such autonomy. Since messages carry arbitrary procedures, they may encode any behavior, including code for generating new messages and sending those to other nodes, thus virtually navigating through the network. Furthermore, the objective of providing arbitrarily extensible remote servers is very similar to the goal of open-ended systems, which can be achieved using an Autonomous Objects paradigm, as will be discussed in Section 4.2. Finally, the objective of Remote Evaluation to reduce network traffic by moving some portion of a client's application to the server site for the duration of the interaction coincides with the Autonomous Objects paradigm called Telescript (described in Section 2.6).

2.3. 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 shortests paths or biconnected components of a given graph [CHA82]. The basic idea of Echo Algorithms is to consider the underlying graphs themselves as the computational engine, where each node is an independent logical processor, capable of receiving, processing, and emitting messages traveling along the graph's edges. Computation in such a network then starts by 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. Even though the creation of the spanning tree is asynchronous and its shape is non-deterministic (depending of the current traffic explorer messages encounter), messages are not autonomous in the sense of carrying their own behavior. The original paper suggests that decisions regarding the propagation of messages be performed by application-specific node programs. There is, however, nothing that would preclude the carrying of the necessary navigational information on the messages themselves and hence they could be implemented as completely autonomous objects. Furthermore, it is very natural for the programmer to think in terms of waves of autonomous messages. Hence this paradigm, together with Dijkstra's diffusing computations [DiSc80] can be viewed as the foundation for later approaches, where messages propagating autonomously through networks are the primary vehicle for solving problems.

2.4. 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 [Bic85, Bic87, BiLe87]. BPEM is a computational model designed to facilitate the parallel processing of knowledge, represented in the form of semantic nets [Woo75] or cognitive maps [Zha92]. 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* 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. To answer a query then corresponds to the problem of finding a match for a given template in the underlying knowledge net such that each free variable is bound to a node label in

* 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.

the knowledge net. The answer, if one exists, is the set of bindings for the free variables.

To illustrate the principles, consider a simple knowledge network that records the parent relationships between individuals. That is, a directed link labeled "parent" from node n to node m represents the fact that n is the parent of m:

$$n \xrightarrow{parent} m$$

The knowledge net is thus a collection of labeled nodes interconnected via the appropriate "parent" links. Answering queries, such as "who are the parents of m", can then be represented as finding matches in this network to the corresponding template, in this case, a directed link labeled "parent" from any node (i.e. a free variable X) to the node m:

$$X \xrightarrow{parent} m$$

Along these lines, arbitrarily complex queries can be constructed as network templates. For example, the query "who are the grandparents of m" would be expressed as the following template, where X would be bound to parents and Y to grandparents of m.

$$Y \xrightarrow{parent} X \xrightarrow{parent} m$$

In addition to viewing the problem of knowledge extraction as a non-procedural template matching process, BPEM provides a way to perform the template matching in an asynchronous distributed manner using autonomous objects. The template, which could be of arbitrary topology, is first converted into a tree and injected into the net in the form of a autonomous object, where each network node is an interpreter of the network pattern. The essential steps of the interpreter in each node are the following: for each received message, see if the root of the template matches the node's content. If not, discard it; else, propagate each subtree along all matching edges. When a leaf node is matched, propagate a success message to the original parent node. Hence the object propagates and replicates itself along multiple paths through the underlying knowledge net until a matching pattern is found or the knowledge net is exhausted.

To illustrate this distributed interpreter, consider again the above grandparent query. This template would be placed on a message and injected into the node m of the underlying knowledge network. This node propagates it along all incoming edges labeled "parent". The receiving nodes bind themselves to the variable X and

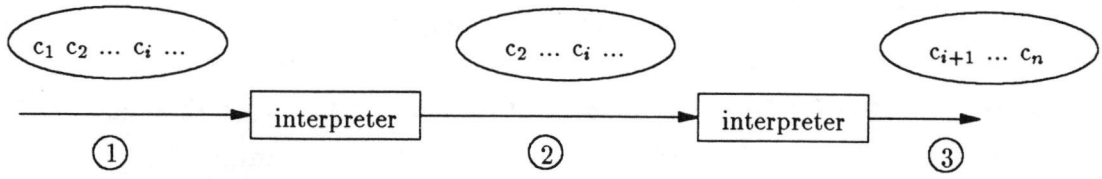

Figure 1
Autonomous Objects

propagate the remainder of the template, again along all incoming edges labeled "parent". All receiving nodes, if any, then constitute the final answer to the original grandparent query.

2.5. WAVE

Another representative of the Autonomous Objects paradigm is WAVE [SAP88, BOR92, SABO94], 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. Hence the interpretation is incremental in both time and space. This is repeated until the given program terminates, causing the corresponding WAVE program instance to cease to exist.

Figure 1 illustrates the above basic principles. It shows two network nodes, each running the WAVE interpreter as one of its applications. A WAVE program is received by the first node (marked as step 1 in the figure) where the current command c_1 is assumed to be navigational, thus causing it to be forwarded to another node (step 2). Assuming the next command, c_2, is computational, the

receiving node would interpret it and go on to the next command c_3. This continues until another navigational command, say c_i, is encountered, which causes the WAVE program to again propagate to one or more neighboring nodes, each of which would continue interpreting the program at command c_{i+1}.

WAVE features a very elaborate set of commands to control both computation and navigation, which have been tested on a number of different applications. As part of the computation, WAVE programs may also create 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 [BOR92].

2.6. Telescript

Telescript is 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 [WAY94] 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 highly distributed computing. Each attempts, in some way, to "colonize" the underlying network and to use it as a resource to speedup some computational task. Telescript, on the other hand, attempts to provide a communication medium independent of all underlying network protocols or transport services. It is targeted to client-server type applications in large geographically distributed networks, notably, the Internet. Instead

of having to establish a traditional "session", where low-level data messages are exchanged between the client and the server, Telescript permits the creation of an autonomous object by the client, which is released into the net in the search of a suitable service. What makes this approach independent of network protocols is that Telescript objects are interpreted and hence any "Telescript-sensitive" node, i.e., one that contains the Telescript interpreter, is able to communicate with any other such node.

Similar to Remote Evaluation, Telescript also aims at reducing message traffic over the network. Without Telescript or Remote Evaluation, each client must engage in a low-level message exchange with a (remote) server using the existing network protocols. An autonomous object, on the other hand, is dispatched only once to the server site, where it performs the necessary interactions locally or possibly dispatches its own subobjects to other sites. When the task is completed, it reports the answer to the original client. Hence only a single "round trip", traveled by the object, is necessary between the client and the server.

2.7. 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, GOL86, RIDA89]. 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 [RIDA89] 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.

2.8. A Classification

In this section we classify the various paradigms surveyed in the previous sections along two orthogonal axes. The first axis captures the notion of *autonomy* of messages, that is, to what degree 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 autonomy: *none, potential*, and *inherent*. The first contains paradigms where messages contain no navigational information (other then their immediate destination); the second includes systems where messages could contain some navigational information but this capability has not been addressed explicitly by the developers; finally, the third category contains paradigms where the fundamental operational principles are based on messages representing completely autonomous objects with their own behaviors.

The second axis captures the ability of a paradigm to serve as a *coordination* language for programs resident on various nodes throughout the network. We distinguish four levels along that axis: *none, process-based, function-based (resident)*, and *function-based (carried)*. The first category includes paradigms not intended for coordination, i.e., those where messages have no ability to invoke any compiled functions or interact with other processes; the second includes those where functions already compiled and resident on the receiving node may be invoked by spawning a new process for their execution, or where the autonomous object can interact with other processes on the remote host; the third category include paradigms where autonomous objects can invoke and execute precompiled functions as part of the currently running process; finally, the fourth includes paradigms where the autonomous objects can actually carry a compiled program and invoke it at the receiving node.

Figure 2 shows how the paradigms surveyed in the previous section can be categorized based on the two criteria. Along the autonomy axis, RPCs and method invocations in (distributed) object-based systems fall into the first category, where a message contains only the name of the procedure/method to be invoked, together with the necessary parameters. In terms of coordination capabilities, they fall into

COORDINATION \ AUTONOMY	none	potential	inherent
none		Echo Algorithms	BPEM
process			WAVE
function (resident)	RPC Methods		Telescript Intelligent Email
function (carried)		Remote Evaluation	MESSENGERS

Figure 2
Classification

the third category, since their main purpose is to invoke remote functions efficiently as part of the current (server) process. Echo Algorithms and Remote Evaluation both have the potential for autonomy in message passing, even though this aspect has not explicitly been addressed. In the case of Echo Algorithms, 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 can 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 that 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. While Remote Evaluation can (and is intended to) be used to perform efficiently the invocation of pre-compiled functions carried throughout a network, Echo Algorithms have no coordination 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.

The remaining four paradigms, BPEM, WAVE, Telescript and Intelligent Email, are highly unconventional in their basic philosophy. In all four cases, the emphasis is on completely autonomous objects 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. 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. However, due to their ability to invoke Unix shell processes, WAVE programs also possess a fair amount of coordination capability, but only at the process level. Telescript and Intelligent Email aim at providing autonomous agents capable of navigating the Internet. The main emphasis here is on coordination in that 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. Computational capabilities are secondary, employed only as part of the overall coordination process. However, in addition to their communication capabilities, both also provide mechanisms for invoking node-resident functions. Finally, the MESSENGERS paradigm, which will be described in Section 3, falls into the fourth category of coordination paradigms, which permit an efficient execution of functions carried by the autonomous objects.

We wish to emphasize that the concept of *autonomy* as used in this paper should not be confused with process or object *migration*, which is orthogonal to the former. While autonomy refers to the ability of *messages* to navigate, migration refers to the ability of the *sending/receiving programs* to change location. There are a number of systems that support object migration (without message autonomy). For example, Emerald [JLHB88] allows arbitrary migration of fine-grain objects, including objects passed as parameters to remote sites. The purpose it to reduce communication overhead during the remote operation. However, unlike Remote Evaluation, the object being moved has no 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. Another example of migrating entities are Worm-based programs [ShHu82], where the application consists of a certain number of communicating components, which try to find free network nodes to use as computing resources. The distinguishing feature of a Worm is that it regenerates any of its components automatically and dynamically if that component is killed.

This allows the Worm to "colonize" a subnetwork for its own purposes and to migrate through the network as the availability of computing nodes changes.

As already mentioned above, Worm programs support the migration of the communicating nodes rather than any autonomy of messages traveling through the network. Program migration and message autonomy could, however, coexist in the same system and effectively complement each other. In particular, a paradigm, such as WAVE or MESSENGERS, that requires interpreters to be running on all participating network nodes, could be built on top of a Worm-program. The latter would establish the necessary network of interpreters and maintain their availability by recreating those that have been killed on other nodes in the network. The Autonomous Objects paradigm would then operate within this dynamically maintained network of interpreters.

3. MESSENGERS

In this section we present a unifying view of the various paradigms presented in Section 2, by describing a system, called MESSENGERS, currently under development at the University of California, Irvine. MESSENGERS is a general Autonomous Objects paradigm, which incorporates the main features of the individual models of Section 2 into a common framework. It is intended for the composition and coordination of concurrent activities in a distributed environment using autonomous objects called *Messengers.**

3.1. The Language of MESSENGERS

For the purposes of this section we assume that nodes and links of the computational network have unique labels. Both nodes and links are logical entities mapped onto the physical network. Multiple logical nodes can be mapped onto the same physical node, in which case the interpreter is multiplexed among these logical nodes, thus making them independent of one another and running concurrently. Logical links are addresses used by the logical nodes to communicate with one another. The language distinguishes three types of variables: *node*, *link*, and *messenger* variables, which are associated with nodes, links, and Messengers, respectively. The first two types are stationary (i.e. node- and link-resident) while the third is carried by the Messenger as it propagates through the network. At

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

any point in time, a Messenger has access to its own (messenger) variables, the node-resident variables of the node currently interpreting the Messenger, and the link variables of the link along which the Messenger arrived at the current node.

Every Messenger program has the following form:

$$\downarrow$$
$$c_1\ c_2\ \ldots\ c_i\ \ldots\ c_n;\ \text{messenger variables; functions}$$

Each c_i is either a computational or a navigational command. The arrow is used to indicate the current command, 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. Finally, "functions" denotes a (possibly empty) set of precompiled functions carried by the Messenger for the purpose of remote evaluation.

Any command, c_i, constituting the Messenger's program could be one of the following:

Computation:

1. *define and initialize a node variable*: the Messenger defines and initializes a new variable resident in the current node
2. *define and initialize a link variable*: the Messenger defines and initializes a new variable associated with the link along which it arrived at the current node
3. *define and initialize a messenger variable*: this defines and initializes a new variable local to the Messenger; that is, the variable does not stay with the current node or link but is carried along with the Messenger as it propagates to other nodes
4. *read node address*: this allows the Messenger to read the address of the current node
5. *arithmetic or logic statement*: the Messenger may perform arbitrary arithmetic and logic operations using any node, link, or messenger variables
6. *control statement*: the Messenger may perform statements such as if-then-else, while, or repeat–until as part of its computation

Navigation:

7. *move_to_node(S)*: this causes the Messenger to be forwarded to node S in the network; that is, the Messenger is routed directly to the destination node without following any logical links

8. *replicate_along_link(L)*: this causes the Messenger to be replicated and a copy sent along all links labeled L; if L is omitted, the Messenger is sent along all links; when all replicas have been sent, the original copy in the current node ceases to exist

9. *send_along_new_link(L)*: this is similar to the previous command, however, the sending of the Messenger causes the creation of a new link along which the Messenger is sent; it is received by a new node created at the end of the newly created link, which continues interpreting the Messenger; additional parameters specify how the new node and link are to be mapped onto the physical network

10. *terminate*: this causes the interpreter to discard the current Messenger

Interaction:

11. *call*: this invokes a precompiled function, waits for its completion, and returns the results back to the invoking Messenger's program; the invoked function could be one already residing on the current node or it could actually be carried by the Messenger

12. *spawn*: this also invokes a precompiled function but as a separate process, which runs concurrently with the invoking Messenger and may continue operating even after the Messenger has left the node (or has terminated)

Commands 1–6 are strictly computational, allowing Messengers to define and use arbitrary variables, which can then either be left in the current node or link, or carried along to other nodes. The next 4 commands are navigational, allowing the Messenger to specify where it wishes to propagate within the network. This also includes termination, which is viewed as the Messenger leaving the network.* Command 9 also embodies the concepts of mapping of the logical onto the physical

* Termination could also be viewed as a computational control command. We chose to view it as navigation, since, from the interpreter's point of view, the Messenger is leaving the current node and hence causes the interpreter to relinquish control as with the other navigational commands.

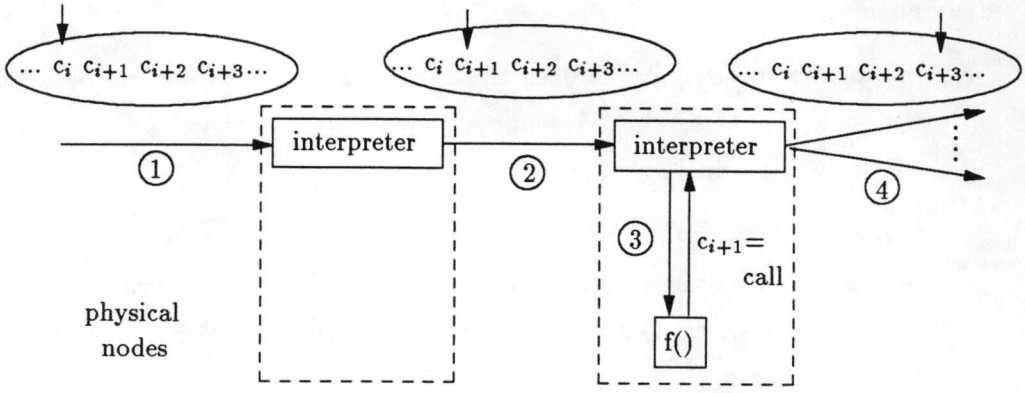

Figure 3

Autonomous Object Invokes Node-Resident Program f()

network, since this allows new nodes and links be created, thus expanding the logical network. As indicated above, additional parameters used with the command determine the mapping. In particular, the new node could be mapped locally (i.e., on the current computer); remotely (on another computer); or on a computer with specific properties. The link could be mapped according to the following criteria: maximum bandwidth; shortest latency; communication cost. Finally, the last two commands are related to communication with the environment, allowing the Messenger to invoke node-resident functions as subprograms or as concurrent processes.

The interpreter for the above Messenger language executing in each node then has the following basic structure:

```
repeat forever
  receive Messenger
  repeat
    interpret current command
    increment current command pointer
  until command = navigational
```

For each new Messenger, the interpreter continues processing its commands until it encounters one that is navigational, i.e., one that causes the Messenger to be removed from the current node. This also includes the terminate command, which does not propagate the Messenger but discards it.

Figure 3 illustrates these basic principles graphically. It shows two network nodes, each running the Messenger interpreter as one of its applications. A

Messenger is received by the first node (marked as step 1 in the figure) where the current command c_i is assumed to be navigational, thus causing the Messenger to be forwarded to the next node (step 2). The next command, c_{i+1}, is assumed to be a call statement, causing the invocation of some node-resident program f() (step 3). Upon returning from the function call, the interpreter carries out the next command, c_{i+2}, which we again assume to be navigational. It causes a replica of the Messenger to be sent to all neighbors of the current node (step 4), with the current command pointer at command c_{i+3}. Note that steps 3 and 4 are sequential since the propagation awaits the completion of the function. If, however, a "spawn" command was used instead of "call", the two steps would proceed concurrently.

3.2. Comparison with other Paradigms

The above abstract Messenger language embodies many of the concepts developed as part of the paradigms presented in Section 2. For example, it is easy to see how a program corresponding to a simple RPC would be constructed: it would consist of a function call, surrounded by two navigational commands—one to transport the Messenger to the server site and the second to carry back the results. Remote Evaluation can be handled using a similar program. The main distinction is that the function to be evaluated is carried along by the Messenger.

Most of the capabilities of the other paradigms, notably Echo Algorithms, BPEM, Telescript, and WAVE can also be subsumed naturally, depending on the specific commands provided when implementing the above abstract Messenger language. WAVE is closest to the above Messenger language and, in terms of its basic operational principles, MESSENGERS is just an abstraction of WAVE. There are, however, several important distinctions between the two. First, WAVE does not allow for compiled functions to be carried as part of Messengers. Furthermore, WAVE must create a new Unix process for every compiled function it invokes, which, due to the excessive overhead, greatly limits its usefulness to serve as a coordination language for distributed programs. With MESSENGER, functions may be executed directly by being loaded and invoked as part of the interpreter process. This light-weight capability is essential to make MESSENGERS useful for coordination of distributed activities, such as the distributed simulation application described in Section 4.2.

A second important distinction between WAVE and MESSENGERS is that, unlike WAVE, the MESSENGERS interpreter does not permit multiple active Messengers to coexist in the same logical node. This is because it does not relinquish control until encountering a navigational command, which removes the current Messenger from the node and frees the interpreter to continue with the next Messenger. That is, the sequence of commands between the current and the next navigational command is indivisible. The main advantage of not multiplexing the interpreter among different Messengers is that no explicit constructs to enforce synchronization among Messengers are needed. At the same time, this does not restrict the expressive capabilities of this paradigm since a Messenger could give up control explicitly (by a command analogous to sending itself to the same node), thus allowing the interpreter to receive and interpret other Messengers in the meantime.

WAVE, while permitting multiple concurrent objects in a node, does not provide any synchronization primitives. That is, cooperating WAVE programs cannot block on a condition but can only coordinate their interactions through a form a busy-waiting on shared node variables, which makes the programming of many applications very awkward. This problem could be solved at the expense of a more complex state management. Each WAVE program would have to be promoted to an actual process, with its own running, ready, or blocked state. The base language could then be extended to include explicit synchronization constructs, such as semaphores, out of which more complex constructs could be built. Not allowing programs to interleave arbitrarily, as is the case with Messengers, eliminates this problem.

The final distinction between WAVE and MESSENGERS is their user interface. WAVE features a highly condensed syntax with a large number of special-purpose operators for navigation, computing, and communication. This permits the construction of arbitrarily complex self-contained objects but requires the learning a new language that is dissimilar to any other programming language in both syntax and semantics. Messengers, on the other hand, use the familiar c syntax, with a very few special constructs to express navigation. Unlike WAVE, the Messenger language is intended primarily for coordination, while most of the actual computation should be performed by compiled node-resident functions. Hence its emphasis is on simplicity in orchestrating the operations of distributed applications consisting of large numbers of independent functions as their basic building blocks.

4. Capabilities and Applications

The purpose of this section is to illustrate the capabilities of Autonomous Objects paradigms, such as MESSENGERS, by describing solutions to problems from a variety of application areas, some of which would be much more difficult to solve using the conventional Communicating Objects paradigm.

4.1. Network Control—Computing in Unknown or Dynamic Topologies

The ability to operate in networks without any knowledge of their topology or even in networks whose topology may be changing dynamically while computation is in progress, is very important to a number of modern application domains, such as telecommunication. A typical example is a mobile telephone network, where large numbers of mobile units are not only constantly being turned on and off but also roam in space and need to be tracked by a static network of sender/receiver stations. Using an Autonomous Objects paradigm, it is possible to construct arbitrary control structures superimposed over the physical network. The purpose of such a logical structure is twofold: (1) to hide the underlying networking details, thus providing a more convenient abstract computing environment for an application, or (2) to establish a given logical structure, such as a spanning tree or a ring, for the purposes of network control; e.g., for distributing data or commands to all nodes or for collecting status information.

To illustrate how MESSENGERS can be used for this purpose, we consider the well-known graph-theoretic problem of finding a shortest-path spanning tree in an undirected network with a given node S as its root. Normally, this problem assumes a fixed weight associated with every link, which is used to compute the length (i.e., cost) of each path in the tree. We consider an important variation of the problem, where the link weight is not a fixed constant but corresponds to the current propagation delay on that link. Hence the problem is to construct a "shortest-communication" spanning tree in the current network topology. For simplicity, we assume that the logical network topology is identical to the physical network topology, i.e., a MESSENGERS interpreter is running in every physical node and each can communicate with all its direct neighbors.

The following Messenger program finds the spanning tree as defined above. The program is based on the principles of Echo Algorithms—it corresponds to the first phase of every Echo Algorithm, which dynamically constructs a spanning tree

by asynchronously spreading explorer messages into all nodes. The program is written using the constructs outlined in Section 3.

```
(1) move_to_node(S)
(2) repeat
(3)     M_PREDECESSOR = read_node_address()
(4)     replicate_along_links(ALL)
(5)     if N_VISITED == FALSE
(6)     then N_VISITED = TRUE
(7)          N_PREDECESSOR = M_PREDECESSOR
(8)     else terminate
(9) until terminate
```

The program utilizes three variables. N_VISITED is a node variable that records whether the current node has already been visited by a Messenger. M_PREDECESSOR is a messenger variable that carries the address of the node from which the current Messenger arrived. N_PREDECESSOR is a node variable that is set to M_PREDECESSOR if the current Messenger is the first to arrive at the node, i.e., if it traveled along the fastest path from node S.

Note that the program is the behavioral description of a Messenger. Hence the code (compiled into an internal representation to minimize its length) is carried by the Messenger as it propagates through the network. Initially, the Messenger is injected into the starting node S (line 1). This node interprets the repeat statement (line 2), where the next command (line 3) reads the address of the current node into the variable M_PREDECESSOR. The next command (line 4) then causes the entire Messenger to be replicated to all neighbors of S. A copy of the M_PREDECESSOR variable is carried by each Messenger.

Each receiving node continues interpreting the next command (line 5), which determines if the node has already been visited before. If not, the variable N_VISITED is set to TRUE to indicate that it now has been visited and the node from which the Messenger arrived is remembered in N_PREDECESSOR (line 7). The collection of all N_PREDECESSOR values thus record an incrementally constructed trace through the net, which will become the final spanning tree when all Messengers terminate. If the condition on line 5 fails, the current branch dies (line 8). Otherwise, the repeat statement causes the Messenger to remember the address of the current node (line 3) and to again replicate itself to all neighbors (line 4), with its current command pointer advanced to the next command (line 5). This is

repeated until failure occurs (in line 8), causing the repeat statement to terminate (line 9). When all nodes have been visited, the spanning tree is complete.

Note that the above program performs its task in a fully distributed manner, without making any assumptions about the topology of the given network. The resulting spanning tree is non-deterministic, reflecting the current communication load on each of the links of the underlying network and thus guarantees the minimum amount of time to visit all nodes [CHA82].

A similar program has been written in WAVE to construct the shortest-path spanning tree in a network where the weights are recorded explicitly as link labels [SABO94]. The main distinction between this algorithm and the above shortest-communication version is that, with the former, a branch does not terminate in a node if it has already been visited but only when the path recorded thus far is shorter than the one found by the current WAVE program. If the latter is shorter, the node variables are updated to record the new path and the WAVE program continues propagating. One drawback of the WAVE solution is that it does not have link variables and hence the link weights are recorded as textual labels associated with every link. With MESSENGERS, explicit link variables may be defined, which can then record information, such as weights, independently of the link labels.

Other programs, where autonomous objects navigate their way in *a priori* unknown topologies, can be written for other well-known graph-theory problems, such as the traveling salesman problem, shortest paths, maximum cliques, transitive closure, network radius and diameter, cycles, articulation points, or various graph sort problems [SABO94, CHA82].

4.2. 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 offer a much more natural paradigm within which such applications can be implemented.

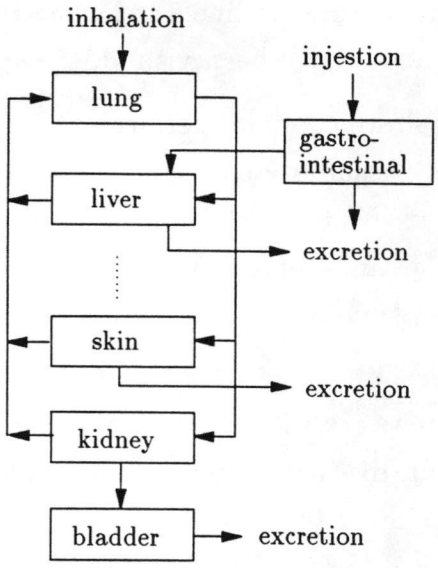

Figure 4
Toxicological Model

4.2.1. Biomedical Simulations

Simulating biological and physiological processes in living organisms is of great interest to many branches of medicine, biotechnology, pharmacology, as well as the military. In collaboration with UCI's Medical School, 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. Figure 4 shows a simple 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 in Fortran 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 pass 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 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.

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 a grid. 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 prescribed and triggered by the arrival of a Messenger. Hence the user may easily change the computation while it is in progress by modifying the Messengers circulating though the model.

4.2.2. DIS

The objective of the Distributed Interactive Simulation (DIS) initiative is to establish standards that will permit the creation of synthetic environments to conduct large-scale interactive battle simulations involving human personnel, aircraft and other simulators operated by human pilots/drivers, simulated aircraft/vehicles (called Computer Generated Forces), simulated unmanned objects, such as missiles, geographical features, and other important phenomena [LOR92]. The proposed standard is based on defining a common unit of discourse, called a Protocol Data Unit (PDU), which encodes the necessary information about possible events. All dynamic objects are then required to inform each other of their movements and other events they cause by broadcasting that information using PDUs.

One of the main problems with this philosophy is that there is no correlation between the simulated space and the underlying distributed network. Each participant simply gets a copy of the entire simulated space (geographical database) and is responsible for maintaining all updates locally. The simulated objects do not migrate through the physical network, only through the local simulated space. Consequently, logical proximity is unrelated to physical proximity; e.g., two planes flying in close formation could each be running on machines located on different continents, or vice versa. The unfortunate implication is that PDUs from any given object must be broadcast to the entire physical net, even though only a small number of participants in the object's logical vicinity may potentially be interested in

this information. (Multicasting is being explored by the DIS community but no satisfactory solutions have yet been found.)

Although DIS is now a de facto standard and its basic philosophy is not likely to change, there are other applications with similar characteristics. At the last DIS conference, we demonstrated that the principles of autonomous objects offer a more natural formalism to satisfy the requirements of such open-ended simulations due to their inherent mobility in the net [BBCS94]. The simulated space is divided into a grid and each subregion is mapped onto a different physical node. Autonomous objects, each representing a distinct simulated object, such as an aircraft or ship, then move through the simulated space according to their predefined behaviors or in response to an operator's directions. Once an object reaches the boundary of a subregion, it is handed off to the appropriate neighboring region. Hence objects move not only in the simulated space but actually *migrate* physically through the network. The important implication is that logical proximity of objects is related to their physical proximity and thus information about a given object does not have to be broadcast to the entire net.

Many open-ended applications, such as DIS, also require the introduction of new objects or new behaviors of existing objects at run time. Under existing message passing paradigms, this is difficult to achieve, since the set of functionalities (behaviors) for each object is defined and compiled in advance. To introduce a new message type requires the definition of new functions to handle such messages. This problem can be captured very succinctly by considering the generic structure of a mobile object, such as a simulated aircraft, which is to propagate through a network of monitoring stations. The program skeleton describing the object's behavior is described as follows:

```
repeat
  f(...)
  next_node = g(node_variables, my_variables)
  forward to next_node
until h(node_variables, my_variables)
```

The object must traverse a series of logical nodes representing monitoring stations. In each station, it performs some local computation, represented by the function f and then determines where to go next, i.e., the value of the variable next_node, which is computed by applying the function g to local variables

found in the current station node (node_variables) and variables it carries along (my_variables) as it travels through the net. Once the destination is determined, the object is transferred to the next station. In this way it keeps moving through the net until the termination condition, computed by the function h, is satisfied, at which time it ceases to exist.

The main point here is that the three functions, f, g, and h, which determine the object's overall behavior, are unspecified. They could be different for each object, thus defining objects with different behaviors. Furthermore, they may not all be known at the time the simulation process is started; rather, the user should be able to introduce new object types on the fly by specifying the functions f, g, and h as needed in a given situation.

Autonomous objects offer great flexibility in developing such open-ended applications, since the common agreed-upon basis of understanding is not an encoding of facts but a complete language, which is universally understandable at any node containing the basic interpreter. This allows many decisions to be postponed until run time. In [MBBC95] we have formulated several primitive Autonomous Object programs (using WAVE) for guiding simulated objects through a spatial database. These include primitives to follow a given line segment already existing in the spatial database (e.g. a road), to move in a certain direction for a certain distance through the simulated 2-dimensional space, or to replicate and spread a wave of autonomous objects into an area within a certain radius. The first two primitives then can be combined into more complex behaviors, allowing objects to navigate through the simulated space. The third primitive may be used, among other things, for orientation. That is, the object can actively "look" for other object, events, or geographical features, by sending out waves of "sensing" objects, which spread into the observer's vicinity and report back information they have been programmed to detect. This form of active sensing of the environment can be invoked any time during the object's lifetime and thus can guide its navigation through the space.

4.3. AI Search Problems and General Problem Solving

There are two classes of AI problems that can benefit greatly from the distributed nature of autonomous objects. The first has to do with searching for matching patterns in graphs while the second actually creates graphs dynamically as it searches for solutions.

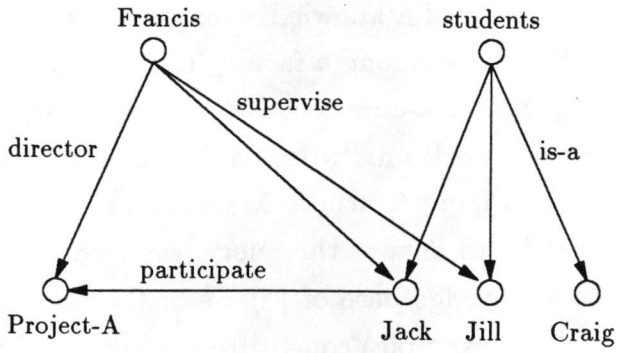

Figure 5
A Semantic Network

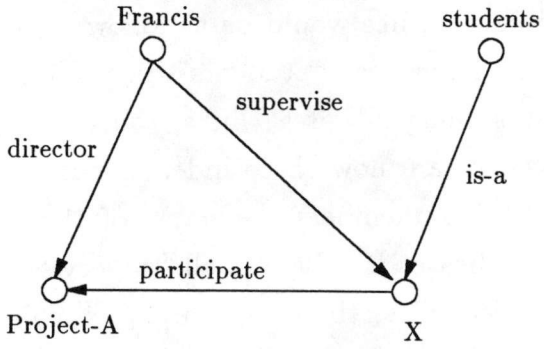

Figure 6
Sample Query Template

4.3.1. Graph Matching Problems

Knowledge is frequently recorded in the form of directed graphs, where nodes represent concepts or actions and edges represent relationships. Semantic nets [Woo75] or cognitive maps [ZHA92] are common forms of such knowledge representation. Many problems can then be formulated as searches for given patterns in the underlying knowledge nets. In a traditional implementation, the nets are viewed as passive repositories of knowledge, where programs are the agents that search for and process the recorded information. With autonomous objects, we can view the knowledge structure as an active network where each node is an interpreter and the search for matching patterns is performed by autonomous objects navigating through the net. These principles, developed originally as part of the BPEM model, have already been introduced in Section 2.4. We now present a more complex example involving a non-trivial search pattern.

Consider a portion of a knowledge network shown in Figure 5, which records the various relationships among a faculty member (Francis), his project (Project-A), and three students. Assume now that we wish to answer a query such as "which of Francis' students work on Project-A?". This can be captured by a network template shown in Figure 6, where X represents a free variable to be bound to the answer. That is, to answer the query, we need to find a match for the query template in the knowledge base of Figure 5; the underlying nodes that match X ("Jack" in the above example) constitute the answer.

To find the answer using autonomous objects, the template of Figure 6 is transformed into a sequence of navigational steps that mimic a sequential scan of the pattern and test for matches as the objects propagate through the knowledge net. One possible sequence would be as follows. First, the autonomous object is sent to one of the nodes, say "students", in the knowledge net (Figure 5). From there it replicates along all "is-a" links, thus reaching the nodes "Jack", "Jill", and "Craig". There are now three independent autonomous objects proceeding concurrently. Each remembers the name of the current node (to later detect the cycle) and replicates itself along all "supervise" links. Since "Craig" has no such link, the object dies; the others, upon reaching a node, check if this node is named "Francis" and, if so, propagate themselves along the "director" link. Upon a successful match against the "Project-A" node, they continue along the "participate" link. If the node reached at the end of that link matches the name of the node each autonomous object remembered earlier, the cycle has been closed, indicating a successful match of the entire pattern.

4.3.2. Problem Solving

To illustrate the use of autonomous objects in dynamically building a general problem search tree, consider the "farmer-goat-wolf-cabbage" problem, which is representative of many similar problems for AI. It can be stated simply as follows:

A farmer with a goat, a wolf, and a cabbage is standing on the left bank of a river and wishes to cross it in a boat, which can hold only the farmer and one of its three possessions at a time. The task is to find a series of river crossings such that, at no time, is the goat left alone with the cabbage or with the wolf.

The algorithm given below (which has originally been formulated in WAVE [SABo94]) illustrates informally how an autonomous object program could be

written to solve this problem. Let L represent the set of possessions the farmer can currently choose from at the left river bank. Initially, L={w,g,c,@} where w, g, and c represent the wolf, goat, and cabbage, respectively, and @ means "empty", symbolizing that the farmer may choose to take the boat across empty. Similarly, R represents the possessions the farmer may choose from on the right bank; initially, R={@}. The algorithm is then as follows:

```
replicate_along_new_link()
L={w,g,c,@}
R={@}
repeat
  for each element e in L do:
     if w,g or g,c are in L-e then terminate
     else replicate_along_new_link()
          L=L-e
          R=R+e
     for each element e in R do:
        if w,g or g,c are in R-e then terminate
        else replicate_along_new_link()
             replicate program along this link
             L=L+e
             R=R-e
until L={@}
```

From its starting point (which could be any node in the network), the autonomous object first creates a new link and propagates itself along that link to a new node. In this node it creates the two initial lists L and R. From there, the object then dynamically creates a breadth-first parallel search tree as follows. It considers each element e of L for possible transfer to the right bank. If g is left on L with either w or c, this branch terminates. Otherwise, a new link and a new node are created and the entire program is replicated along that link, thus representing the movement of the particular element e to the right bank. At the receiving end of the link, the newly created node moves the element e from L to R. After that, an analogous process is repeated in each of these nodes with the elements on R (representing the transfer from the right to the left bank), trying to move them to L without leaving w,g or g,c together. Eventually, one or more paths result in a configuration where L is empty while R contains all possessions.

Figure 7 illustrates the search tree. Starting from the root node, only the movement of g does not violate the two restrictions and hence a new node with $L = \{w, c, @\}$ and $R = \{g, @\}$ is created. From there, two possible paths are possible. One of these ferries g back to L, thus resulting in the original configuration. This

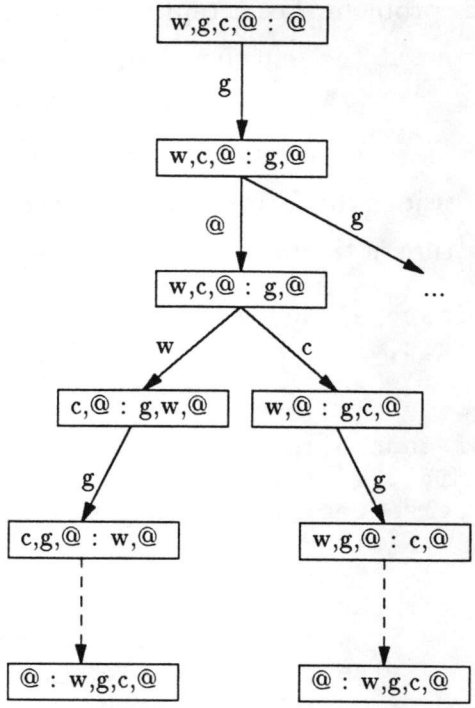

Figure 7
Problem Search Tree

path, which does not lead to any new solution, is not expanded further in the figure.† The successful path, which moves the boat empty to L results in a new node, where L and R are unchanged.

The next step allows the movement of either w or c from L to R. The two successful branches then continue in parallel, each exploring all possible moves, until a solution is found. The two possible sequences of successful moves are $g; @; w; g; c; @; g$ and $g; @; c; g; w; @; g$.

The main advantage of the above autonomous objects solution is that it can adapt automatically to any underlying network topology, thus exploiting parallelism available at the time of execution. This can be achieved by letting each autonomous object decide, based on the current network status, whether to create the next tree node locally or on another computer.

† This path can easily be optimized away by an additional test in the program. This was left out from the algorithm to keep it as simple as possible.

4.4. Other Advantages of Autonomous Objects

In addition to the capabilities presented in the preceding sections, there are several other areas where the Autonomous Objects paradigm offers greater flexibility than other approaches to distributed computing. Among these are robustness, security, and accountability.

Robustness is the resilience to failure and the ability to recover from failure so that operation can resume in the least disruptive manner. The latter depends greatly on the system's flexibility, since no design will be able to predict all possible failure scenarios. The Autonomous Objects paradigm by its very nature provides for great flexibility and, consequently, robustness, of a system in that it does not "hard-wire" any behavior into its nodes but rather constantly responds to messages as they arrive.

To illustrate the point, consider the problem of node failure in a network. One possible strategy to deal with a single node failure is the following: each logical node, initially, determines its surrounding topology, i.e., the addresses of all its neighbors. It then designates one of its neighbors (running on a different physical node) as its "guardian" to whom it sends the information about its surrounding topology. This guardian then periodically tests the availability of its "ward". If the latter is destroyed or damaged, the guardian recreates it and all its links on a different physical node using the previously supplied information.

The above recovery protocol can obviously be written in any conventional language. However, once it has been implemented and deployed, no modification to the scope or strategy of the recovery algorithm is possible. For example, to change the strategy such that it could deal with the concurrent failure of two nodes would require each node to obtain a larger view of its surrounding topology and to designate more than one possible guardians. Hence, given a conventional programming paradigm, any such change would require new programs to be written, compiled, and distributed to all nodes in the net. With autonomous objects, on the other hand, the behavior is carried on a message. Hence, extending the recovery protocol sketched above would only require the development of a new "program" released into the network on one or more messages. There is no need to modify any nodes or to even know how many nodes there are and in what configuration. Hence it is possible to establish (logical) network structures with a

built-in capability of self-regeneration or replication, where the level of resiliency can be modified arbitrarily at any time during the system's operation.

Security and Accountability. Autonomous objects are similar to viruses or worms in that they spread autonomously through a network. A fundamental difference, however, is that objects are interpreted while viruses and worms are compiled. Viruses typically attach themselves to legitimate (compiled) programs and thus become very difficult to detect. Worms actively look for idle computer nodes, which they "invade" by spawning remote processes on them. Autonomous objects, on the other hand, are under complete control of the interpreters, which decide what privileges they will be granted, which resources, including CPU time, they may utilize, and how their creators will be billed for the services rendered. Many of these issues, including billing and controlling "runaway" objects by restricting the amounts of resources they are allowed to utilize throughout their lifetime, have been studied extensively as part of Telescript [WHI94]. Hence autonomous objects offer a much safer way to provide public services in a highly distributed environment.

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 autonomy in navigating through the network. We have demonstrated the capabilities of this approach by presenting a spectrum of problems, formulated originally using the various instances of the general Autonomous Objects paradigm, notably, Echo Algorithms and WAVE.

In addition to autonomy, we have also considered the ability of the paradigm to serve as a coordination language for applications consisting of node-resident compiled programs distributed throughout the network. The need for such languages has been argued for eloquently by Gelernter and Carriero in a recent article [GECA92]. 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.

We feel that the MESSENGERS paradigm, which incorporates the ability to efficiently invoke functions on remote nodes into the general autonomous objects philosophy, 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 the toxicology simulation models, which are collections of sequential distributed functions threaded together at run time through waves of autonomous objects that cycle through the computational nodes. The same principles can, of course, be applied to applications in other areas. Our current work includes the identification of useful navigational and coordination constructs that would allow us to design a language aimed specifically at the construction of distributed program ensembles.

References

[AME87] P. AMERICA, POOL-T: A Parallel Object-Oriented Language. In *Object-Oriented Concurrent Programming*, Akinori Yonezawa and Mario Tokoro, Ed., The MIT Press, 1987, pp. 199-220.

[BST89] BAL, H.E., STEINER, J.G., TANENBAUM, A.S. Programming Languages for Distributed Computing Systems. *ACM Surveys 21*, 3 (1989).

[BIC85] BIC, L. Processing of Semantic Nets on Dataflow Architectures. *Artificial Intelligence. 27* (1985).

[BIC87] BIC, L. Data-Driven Processing of Semantic Nets. In *ParallelComputation and Computers for Artificial Intelligence*, J. Kowalik, Ed., Kluwer Publ., 1987.

[BBCS94] BIC, L., BORST, P., CORBIN, M., SAPATY, P. The WAVE Control Protocol for Distributed Interactive Simulation. *11th Workshop on Standards for the Interoperability of Distributed Simulation* (Sept, 1994), Orlando, FL.

[BILE87] BIC, L., LEE, C. A Data-Driven Model for a Subset of Logic Programming. *ACM TOPLAS 9*, 4 (Oct, 1987).

[BINE84] BIRRELL, A.D., NELSON, B.J. Implementing Remote Procedure Calls. *ACM Tranc. Computer Systems 2* (1984).

[BLL88] BERSHAD, B.N., LAZOWSKA, E.D., AND LEVY, H.M. PRESTO: A System for Object-Oriented Parallel Programming. *Software-Practice and Experience 18*, 8 (August, 1988), pp. 713-732.

[BOR92] BORST, P. The First Implementation of the WAVE Systems for UNIX and TCP/IP Computer Networks. *Technical Report 18/92* (Dec, 1992), Univeristy of Karlsruhe.

[CHA82] CHANG, E.J.H Echo algorithms: depth parallel operations on general graphs. *IEEE Trans. SE SE-8 (4)* (1982).

[DACH88] DALLY, W.J., CHIEN, A.A. Object-Oriented Concurrent Programming in CST. *The Third Conference on Hypercube, Concurrent Computers and Applications 1* (Jan., 1988), pp. 434-439, Pasadena, CA.

[DISC80] DIJKSTRA, E.W, SCHOLTEN, C.S. Termination Detection in Diffusing Computations. *Inf. Process. Letters 16,*, 5 (Aug., 1980).

[ECGS92] VON EICKEN, T., CULLER, D.E., GOLDSTEIN, S.C., SCHAUSER, K.E. Active Messages: a Mechanism for Intergrated Communication and Computation. *19th Int'l Sump. Computer Architecture* (May, 1992), Gold Coast, Aus.

[GEE91] GEESINK, L.H. The Coordination of Distributed Active Messages in a Dynamic Network Topology. *The Computer Journal 34*, 6 (1991).

[GERO88] GEHANI, N.H., ROOME, W.D. Concurrent C++: Concurrent Programming with Class(es). *Software—Practice and Experience 18(12)* (Dec., 1988), pp. 1157–1177.

[GECA92] GELERNTER, D., CARRIERO, N. Coordination Languages and their Significance. *Comm. ACM 35*, 2 (Feb., 1992).

[GOL86] GOLD, E. Envoys in electronic mail systems. *ACM SIGOIS Conf. Office Automation Systems*, SIGOIS Bulletin 7, (2,3) (1986).

[JLHB88] JUL, E., LEVY, H., HUTCHINSON, N., BLACK, A. Fine-Grain Mobility in the Emerald System. *ACM Trans. Computer Sys. 6*, 1 (Feb., 1988).

[LOR92] LORAL SYSTEMS COMPANY Strawman Distributed Interactive Simulation Architecture Description. *ADST/WDL/TR-92003010 1 and 2* (1992), Training Devices, Naval Training Systems Center, Orlando, FL.

[MBBC95] MERCHANT, F. BIC, L., BORST, P., CORBIN, M., DILLENCOURT, M., FUKUDA, M., SAPATY, P. Simulating Autonomous Objects in a Spatial Database. *9th European Simulation Multiconf.* (June, 1995), Prague, Czech Rep..

[RIDA89] RICHARDSON, P.W., DANIELSEN, T. Intelligent Messages or When messages come alive. In *Network Information Processing Systems*, (K. Boyanov and R. Angelinov, Eds.), North-Holland, 1998.

[SAP88] SAPATY, P. WAVE-1: A New Ideology of Parallel and Distributed Processing on Graphs and Networks. *Future Generations Computer Systems 4(1)* (1988).

[SABO94] SAPATY, P.S., BORST, P.M An Overview of the WAVE Language and System for Distributed Processing of Opne Networks. *Technical Report* (1994), University of Surrey, UK.

[SHHU82] SHOCH, J.F., HUPP, J.A. The "Worm" Programs—Early Experience with Distributed Computation. *Comm. ACM 25*, 3 (March, 1982).

[StGi90] Stamos, J.W., Gifford, D.K. Remote Evaluation. *ACM Trans. Programming Lang. and Systems 12*, 4 (Oct, 1990).

[TuMa94] Tucker, L.W., Mainwaring, A. CMMD: Active messages on the CM-5. *Parallel Computing 20* (1994), Elsevier Science Publ..

[Way94] Wayner, P. Agents Away. *BYTE* (May, 1994).

[Whi94] White, J.E. Telescript Technology. *Technical Report* (1994), General Magic, Inc., Mountain View, CA 94040.

[Woo75] Woods, W.A. What's in a Link: Foundations for Semantic Networks. In *Representation and Understanding*, Bobrow & Collins, Ed., Academic Press, 1975.

[YoTo87] Yokote, Y., Tokoro, M. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, Akinori Yonezawa and Mario Tokoro, Ed., The MIT Press, 1987, pp. 129-158.

[Zha92] Zhang, W-R. et al. A Cognitive-Map-Based Approach to the Coordination of Distributed Cooperative Agents. *IEEE Tran. Systems, Man, and Cybernetics 22*, 1 (Jan/Feb, 1992).