Intra- and Inter-Object Coordination with MESSENGERS

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

Department of Information and Computer Science
University of California, Irvine

Abstract. MESSENGERS is a paradigm for the programming of distributed systems. It is based on the principles of autonomous messages, called Messengers, which carry their own behavior in the form of a program. This enables them to navigate freely in the underlying computational network, communicate with one another, and invoke pre-compiled node-resident functions. Coordination is facilitated at two distinct levels of abstraction: first, Messengers coordinate the invocation and the exchange of data among the various functions distributed throughout the network in both time and space (intra-object coordination); second, Messengers, each representing a high-level entity, can coordinate their behaviors among themselves (inter-object coordination). These principles, where an application is composed of autonomous, mobile entities whose behaviors may change dynamically and which can coordinate their actions among themselves, offer great flexibility in interacting with and manipulating the application at run time, as well as improved performance. This is illustrated using two concrete examples—a Toxicology simulation from medicine and a study of collective fish behavior from biology.

1 Introduction

In a recent article, Gelernter and Carriero [GC92] argued eloquently for an integration of *computing* with *communication* in programming languages. 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. Consequently, most distributed applications are collections of programs, where explicit communication primitives based on send/receive commands have been embedded for the purpose of coordination. 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.

In this paper we describe MESSENGERS, a coordination paradigm for distributed systems currently being developed by our research group. It is based on the concept of autonomous messages, which we refer to as *Messengers*[1] to differentiate them from ordinary passive messages. A Messenger, instead of carrying just data, contains a process, i.e., a program together with its current status information, including its program counter and local variables. Each node visited by the Messenger resumes the interpretation of the Messenger's program until a navigational command is encountered that causes it to leave the current node (or the Messenger terminates). The program, which may be arbitrarily complex, thus describes the behavior of the Messenger, allowing it to navigate through the network of nodes and to perform arbitrary functions at each of the nodes. Under the MESSENGERS paradigm, a distributed application is viewed as a collection of functions, whose coordination (i.e. invocation and exchange of data) is managed by groups of Messengers propagating autonomously through the computational network.

There is, however, a second aspect of coordination, which occurs at a higher level in certain classes of distributed applications. There, in addition to coordinating the invocation of individual functions that make up a complex behavior, it is sometimes necessary to coordinate the behaviors of the autonomous entities or objects themselves. This is typically the case when the application is described in terms of independent concurrent entities that coordinate their behavior in time and/or space. We refer to this level of coordination as *inter-object* coordination, while the coordination of functions that make up the behavior of an individual object will be called *intra-object* coordination. We will demonstrate that the MESSENGERS paradigm is effective at both levels of coordination.

2 The MESSENGERS Paradigm

MESSENGERS is a paradigm for distributed computing which, similar to other autonomous-message-based approaches [BFD96], departs from the conventional message-passing view by elevating messages to higher-level entities that embody some degree of autonomy or "intelligence" [Bic95, Fuk95b]. A message can be viewed as an object itself, which has its own identity and which can decide at run time where it wishes to propagate next and what tasks it is to perform there. The behavior of each Messenger is described by its program, which it carries along as it moves through the underlying computational network. MESSENGERS exhibits three novel characteristics that differentiate it from conventional message-based paradigms:

- *navigation autonomy* – the ability of each Messenger to navigate through the network, to replicate itself, and to decide what actions to take at each destination. This is achieved by providing navigational statements, which permit the Messenger to go to specific nodes in the network, follow specific

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

links emanating from the node in which it currently resides, or cloning itself to pursue independent paths.
– *intra-object coordination* – the ability of the MESSENGERS paradigm to serve as the coordination layer of software superimposed on other lower-level computations. This is achieved by having each Messenger invoke and control the execution of ordinary C functions residing on various nodes throughout the network; hence each Messenger orchestrates the execution of a group of precompiled components dynamically. This also offers great flexibility in choosing between interpreted code, which is slow but need not be distributed to all nodes a priori, and code compiled into native mode, which is much more efficient but must be made available on all nodes potentially visited by a Messenger
– *inter-object coordination* – the ability of Messengers, each representing a high-level entity in the application domain, to coordinate their respective actions to achieve complex collective behaviors. Spatial coordination is achieved by reading and writing of shared variables set up for this purpose in the nodes they visit. Temporal coordination is supported by special functions that access global virtual time, maintained by the system.

2.1 The MESSENGERS Language

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 workstation. Each link of the logical network has a user-determined name and several (optional) weights. These are integer variables, which Messengers may use for navigation, for example in graph-based applications. Each node has a user-determined name and a system-wide address. It also contains a node variables area, which can be accessed by the visiting Messengers for the purpose of coordination among themselves.

The MESSENGERS paradigm supports three types of variables, referred to as *messenger, node,* and *network* variables [Fuk95b]. Messenger variables are local to and carried by the Messenger as it propagates through the logical network. Node variables are node-resident and are accessed via a node variable declaration carried by the Messenger. There is only one set of node variables in any given node and they are shared by all Messengers currently running in that node. Network variables describe the logical network, including node names, node addresses, link names, and link weights. All network variables are readable by any Messenger and all except for node addresses may also be modified. Link names and weights may be modified when traversing the corresponding link. Every Messenger program has the following form:

node variables declaration; messenger variables definition; $\overset{\downarrow}{\mathrm{S}_1}$; S_2; ... S_n;

Since node variables are stationary inside a node and thus shared by all Messengers, they are allocated only once by the first Messenger to arrive at that node. Subsequent Messengers then simply access these variables via their node variables declarations, which provide the mapping between the Messenger code and the corresponding node-resident values. The messenger variables definition, on the other hand, denotes private, messenger-carried variables and their data type. Network variables are predefined and referred to without explicit declaration. The declaration and definition formats are similar to those of the C Language and provide a variety of data types excluding pointer, union, and unsigned variables.

The remainder of a Messenger program consists of statements, denoted by S_i. 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 and is saved in a status block, which is also carried by the Messenger as it migrates between different nodes. Each S_i is one of the following three types of statements:

Computation:

A Messenger may perform arbitrary computations using assignment and control statements from the C programming language. These statements are all interpreted by the current node.

1. *assignment*: The Messenger may read and update node 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*: The Messenger may perform all the common control statements supported in C, such as if-then-else, while, do-while and break.

Navigation:

A Messenger may create new logical links and nodes, change or delete existing ones, and move arbitrarily though the network by following links or jumping to specific nodes.

1. *create(node, link, weight, physical)*: Create a *link* along which the Messenger moves, associate *weight* with this *link*, and create *node* on the *physical* node which the Messenger reaches. All parameters are optional. If omitted, the new node/link has no explicit name or weight and is created on a physical node chosen automatically by the system. Hence the last parameter allows the user to explicitly control the logical-to-physical node mapping.
2. *hop(node, link, weight, physical)*: Cause the Messenger to be forwarded to *node* on *physical* node along *link* if the link has the associated *weight*. As with *create*, all parameters are optional, which offers great flexibility in specifying the Messenger's next node destinations. If there are multiple destinations, a copy of the Messenger is propagated to each destination. For instance, *hop(node=a)* propagates the Messenger along all logical links emanating from the current node to all neighboring nodes named "a". If a node address is given as a node parameter, the Messenger jumps directly to the specified node. In addition, the parameters support "wild card" matching,

which results in a powerful multicast mechanism for each Messenger. For instance, *hop(node=*; physical='laguna')* sends a copy of the Messenger to all nodes on the physical node named 'laguna', regardless of any logical links.

3. *delete(node, link, weight, physical)*: Forward the Messenger as with *hop*, but also delete the links traversed by each copy of the Messenger. In addition, if this action erases all links from the departing node and there are no other Messengers currently residing in it, the node is also deleted.

Function Invocation:

This allows the Messenger to load and execute ordinary C functions, precompiled in native mode of and residing on the current physical node.

1. *exec(filename, arguments, [nowait])*: Invoke a precompiled function specified by *filename* as a separate concurrent process and pass *arguments* to it. If *nowait* is specified, the function may continue operating even after the Messenger has left the node (or has terminated).
2. *func(filename, function_name, in_arg, out_arg)*: Invoke a precompiled function specified by *filename* and *function_name* as part of the MESSENGERS interpreter, wait for its completion, and return the results back to the invoking Messenger program. The *in_arg* and *out_arg* are arguments passed to and returned from the function respectively. The invoked function also receives pointers to the node variable area and the memory space holding the messenger variables and thus can manipulate them directly. The loading of the function is triggered dynamically when it is invoked for the first time.

Each Messenger program written in the above high-level language is compiled into an internal machine-like language, which is understood by the MESSENGERS interpreter running in each node. The reason for this compilation is to simplify the task of the interpreter and to reduce the size of messages[Fuk95a].

2.2 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 the interpreter encounters a statement that is navigational or functional, i.e., one that causes the Messenger to be removed from the current node or to be suspended when a precompiled function is invoked. Hence, exclusive operations are guaranteed between any two such statements. This non-preemptive scheduling strategy makes it easy to enforce critical sections but prohibits the use of infinite loops that do not periodically release the processor.

To support time-based coordination, required for example by simulation applications, the interpreters in the system are capable of maintaining global virtual time (GVT). Due to the associated overhead, this feature must explicitly be turned on and off by the user. Once activated, Messengers are able to utilize

GVT via a set of library functions (Section 2.3), which allow arbitrary application functions to be scheduled for future execution based on GVT and other constraints.

Within any given node, interpretation of Messengers continues (optimistically) until the interpreter exchanges Messengers with other interpreters. When a Messenger with an older time-stamp is received, the interpreter locally rolls the execution back to a state corresponding to the received time-stamp. Similar to database transaction processing, this is achieved by restoring an earlier consistent snapshot and reinterpreting all Messengers up to the point of the received time-stamp. In order to reduce the amount of computation required by the local rollback, once GVT is activated, the interpreter restricts the Messenger's migration to neighboring logical nodes only.

Nodes periodically send out messages to other neighbors that allow the system to collectively maintain the most recent stable GVT. This is used by each node to prune the data structures containing rollback information by discarding obsolete snapshots.

The general organization of the interpreter is then as follows. It alternates between two phases: Messenger interpretation and function execution. During the first phase, it makes a complete pass through the ready queue of Messengers and continues interpreting each until one of the following occurs: the Messenger terminates; the Messenger executes a navigational command causing it to move to another node; or the Messenger invokes a compiled function using *func()*. During the second phase, the interpreter calls all functions that have been scheduled as "future events" thus far. The above two phases are repeated until one of the following occurs: both the Messengers ready queue and the function scheduling queue are empty; an interval timer expires; or the local virtual time proceeds beyond a certain threshold. At that point, the interpreter enters an inter-daemon communication phase, which physically exchanges Messengers between workstations using Unix sockets. Thereafter, the interpreter continues with the Messengers interpretation phase as described above.

2.3 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 coordination library is incorporated into the MESSENGERS interpreter as a set of node-resident functions, which may be called from Messenger's *func()* statement explained in Section 2.1 or called within any user-defined function invoked from *func()*. The coordination library's functions fall into two main categories: the first support the control of Messengers at run time; the second support inter-object coordination relative to GVT. This time-based coordination complements the space-based coordination achieved through shared node variables and is of great importance not only to simulation but also to other parallel processing problems, such as distributed termination and barrier synchronization. The following lists the most important library functions:

Functions for Messengers Control:

Since Messengers are created and destroyed at run time, several functions have been provided to support these tasks. The creation of new Messengers is accomplished using the functions *inject()* and *duplicate()*. The former 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 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 function *duplicate()* allows a new Messenger to be created by cloning. It is provided mainly for performance reasons, since it replicates an existing behavior and hence does not involve the file system. The creation of Messengers using this function should not be confused with the 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 *duplicate()*, on the other hand, causes an explicit creation of a new Messengers inside a node.

A Messenger may terminate itself by executing the command *exit* or it may be killed by another Messenger using a function *kill(mid)*, where *mid* is the victim's internal identifier. This is known to its creator (in a way similar to a Unix fork) and may also be passed explicitly to other Messengers. A Messenger can find its own *mid* by calling *getmid()*.

Virtual Time Control:

Maintaining GVT imposes significant overhead on each processor due to the need for additional inter-processor communication and possible rollbacks. Hence the user has the option turn this feature on or off as needed by a given application. The virtual time control functions, *gvt_start()* and *gvt_stop()* are provided for this purpose.

Scheduled Function Invocation:

When GVT is active, any user-defined function may be scheduled to execute in the future relative to the GVT. The function invocation is scheduled as a future event. However, it remains an integral part of the invoking Messenger's behavior. In other words, the invoking Messenger is suspended until the function executes. At that time, the same function may be scheduled again as a future event or control returns to the interpreted Messenger's program.

The scheduling of a future invocation of a function may be based on a specific GVT value, on a predicate, or on a combination of both. The following is a list of the possible options:

1. *Schedule at a specific time:* The function will be invoked when GVT reaches the specified time value. That is, the Messenger schedules its wake-up call at a certain point in time.
2. *Schedule when node variable satisfies predicate:* The function will be invoked when a specified node variable satisfies a given condition, for example, when $N > 0$. That is, the Messenger schedules its wake-up call based on a predicate

over a node variable. This is implemented in the interpreter by checking the appropriate predicate each time it processes the future event queue. When not satisfied, the event remains in the queue. A maximum wait interval (using GVT) may also be specified, at which time the event is canceled. This form of scheduling corresponds to a signaling mechanism between Messengers since the condition can be made true only by some other Messenger.

3. *Schedule adaptively based on messenger variable:* Unlike node variables considered in the previous case, messenger variables are local to a Messenger and thus can only be changed by that Messenger. Hence a Messenger cannot suspend itself to wait for a messenger variable to satisfy a predicate. Instead, the rate of invocation of the function computing the variable is varied based on certain predicates involving that variable. In general, the interval at which the function is reinvoked is increased (e.g., doubled) at each invocation as long as the predicate is satisfied, and it is decreased while the predicate is false. This permits the rate of function invocation to be continuously and automatically adjusted based on a predicate involving a given messenger variable. The predicate itself may take on three basic forms: (1) it involves only the currently computed value of the messenger variable; that is, as long as the variable M satisfies some predicate $P(M)$, the invocation interval (sampling rate) keeps increasing; (2) it involves the current and the last value of the variable, that is, it considers the estimated first derivative of the variable (its slope) based on its current and previous value; as long as the rate of change (M') of the variable M satisfies some predicate $P(M')$, the invocation interval keeps increasing; (3) it involves the current and the last two values of the variable, that is, it estimates the rate of change of the variable's slope (i.e., the second derivative, M'') over the last two intervals; that is, as long as M'' satisfies some predicate $P(M'')$, the interval keeps increasing.

The following sections illustrate the different types of coordination that are possible within the MESSENGERS paradigm. This includes (1) intra-object coordination achieved by dynamically invoking functions using *func()* or *exec()* as part of a Messenger's behavior; (2) spatial inter-object coordination achieved by reading and writing node-resident variables by different Messengers; and (3) temporal inter-object coordination achieved by utilizing the above GVT-based library functions.

3 Intra-Object Coordination

In this section we illustrate how a distributed application can be structured as a collection of functions, residing on different nodes of a network, where the invocation of these functions and the exchange of data among them is orchestrated by Messengers. We consider an application in Toxicology simulation, which we have been developing in collaboration with UCI's school of Medicine [RA84, FMB+96].

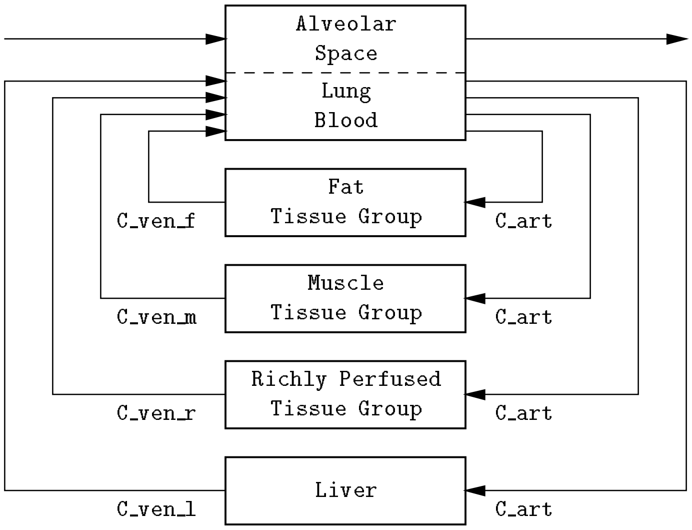

Fig. 1. A Toxicology Model

Simulating biological and physiological processes in living organisms is of great interest to many branches of medicine, biotechnology, and pharmacology. In particular, understanding how chemical substances that are inhaled, ingested, or infused intravenously are distributed through a human or animal body over time, their accumulation in different organs and their metabolism is essential in the development of new drugs and studying the health risks imposed by our changing environment. Research in Toxicology has produced a number of useful models that allow such processes to be simulated on computers. The most advanced models, referred to as pharmaco-kinetic models, attempt to faithfully reproduce the actual physiological processes that take place in a living organism. A model is typically represented as a collection of boxes, each representing a distinct organ or a group of organs having similar physiological characteristics. These are then interconnected by directed links, which represent the flow of fluids, in particular, blood.

Figure 1 shows an example of a simple model. The lung consists of two compartments – the alveolar space represents the area into which outside air is inhaled. Oxygen as well as other chemical compounds are then passed to the lung blood compartment and from there they are carried to all the other organ groups in the body. Each organ group retains some portion of the chemicals based on their size, the volume of incoming blood, and their tissue-specific physiological processes. These values are typically constants and are kept with each organ. The concentration of the chemical in the blood, on the other hand, which also plays an important role in determining how much of the chemical is retained in a given organ, is a variable and must be carried between the organs according to the blood flow. C_{art} in Figure 1 represents the arterial concentration of the chemical being modeled. Each organ changes this concentration based on its specific parameters and hence the outgoing (venal) concentrations C_{ven} are all distinct. They are again combined in the lung, thus completing the basic cycle.

Some organs, notably the liver, also carry out metabolic functions, which may change some portion of the original chemical compounds into others chemicals. For example, styrene, which is inhaled from the air is metabolized into styrene oxide and hence the model must be able to account for multiple chemicals circulating through the body and continuously changing from one to another.

All of the above processes, i.e. the metabolism and the retention of chemicals in organs, are described by sets of differential equations. 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 conventional simulation languages 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, precompiled as C functions, 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.

Advantages

There are two primary advantages of the MESSENGERS-based implementation over other approaches. The first is the ability to manipulate the model dynamically at run time; the second is performance due to parallelism and the ability to perform discrete-event rather than time-driven simulation.

Dynamic Model Manipulation: Most existing simulation and modeling tools require the entire experiment to be set up *a priori*; only the final output may be used as feedback for the next experiment. Hence it is difficult to make incremental progress by backtracking to an earlier point and continuing the experiment in a different direction. MESSENGERS, on the other hand, permits not only parameters to be changed at run time but also the functional composition of the model itself. This is because a given simulation model is a collection of functions whose invocation is triggered by Messengers circulating through the model. Thus each simulation run is composed dynamically from the node-resident functions, depending on the programs carried by the Messengers. This gives the user great flexibility in interacting with the model—it allows both the definition and the state of a simulation experiment to be modified at run time by terminating existing Messengers and injecting new ones as necessary. Hence intermediate results observed from the current simulation run may be used as immediate feedback to steer the ongoing experiment.

Assume for example that the user wishes to modify a given organ's behavior, e.g., the liver in Figure 1, to compute its venous concentration using not only the kinetic flow equation but to also include metabolism. This would be achieved by terminating the Messenger passing through that organ and replacing it with one that included a call to the appropriate metabolic equations as part of its computation. With existing simulation control language approaches, the decision to add a metabolic step to an organ would require a major revision and recompilation of the entire model.

Performance: One of the main advantages of a MESSENGERS implementations is the inherent resulting parallelism. In the above model, each organ is logically a separate computational node and thus may be mapped onto a distinct computer, subject to communication overhead, which must be weighed against the gains from parallel execution.

Another major performance improvement can be achieved by a paradigm shift from a time-driven to an event-driven simulation, which is much easier to do with MESSENGERS than with conventional approaches. Most existing pharmaco-kinetic models are time-driven. That is, the differential equations are discretized and their solutions are derived by an integration process that cycles through all organs and recomputes the changes at each constant time step Δt, chosen as one of the simulation parameters. With MESSENGERS, it is easy to reproduce this time-driven behavior. In the above example, the lung node can serve as the time keeper. Initially, a wave of Messengers is sent along the arterial links. Each of these, having triggered and completed all the necessary computations in its respective node, returns along the venous links back to the lung node where it deposits its concentration value. The last Messenger to arrive triggers the next cycle of the simulation, which combines the individual concentrations, computes the next Δt, and causes the next wave of Messengers to leave the lung node. Hence this implementation mimics exactly the original time-driven simulation.

The main difficulty with a time-driven simulation is the choice of an appropriate Δt increment, which represents a trade-off between the model's fidelity and speed. Most time-driven simulations assume a constant Δt for the entire run. Better results may be obtained by varying Δt in response to the rate of change in the computed variables to automatically adjust the fidelity of the model. This improved version, however, is still very wasteful of computational resources. The reason is that the rate of change in the different parts of the model is not uniformly distributed yet a time-driven implementation requires that all nodes recompute their state at every step. As a result, even nodes where minimal or no changes occurred must still perform the same amount of computation as nodes where changes are significant.

The solution to this problem is to make the simulation event- rather than time-driven, which is accomplished by sending Messengers between nodes only when certain critical variables change by a given 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 to account for the most rapidly changing variables) but reflects the actual current changes. Using the MESSENGERS paradigm, it is simple to make a transition form a time-driven to an even-driven simulation, since the basic concept of Messengers circulating through the system is present in both cases. That is, each Messenger uses the scheduling function (time- or value-based) presented in Section 2.3 to determine its next step. In a conventional implementation, on the other hand, the fundamental structure must be changed to replace the global cycle inherent to time-driven simulation by a continuous event time line characteristic of event-driven simulations.

4 Inter-Object Coordination

In this section we consider a class of applications structured as collections of complex interacting entities. Typical examples of such applications are interactive battle simulations [Com94], particle-level simulations in physics [HE88], traffic modeling [Res94], and various individual-based simulation models in biology or ecology [HDP88, Vil92, HW92]. Related to the latter is also the study of collective behavior in AI, which investigates the mechanisms that result in complex and highly coordinated behaviors of groups of individuals, such as a school of fish or an ant colony, while each individual has only a very limited local knowledge of the "problem" and a set of simple rules to follow [Res94]. Finally, advanced graphics and animation applications, especially those involving large numbers of individuals, such as the animation of a flock of birds, have also taken advantage of individual-based modeling [Rey87].

To illustrate the type of problems we are concerned with in this section and to show how they are solved within the MESSENGERS paradigm, we consider 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. Figure 2 illustrates the model components graphically. Each individual fish, shown as the black icon in the center of the illustration, has a certain radius (R3) of "visibility". Within this circle, excluding the dead angle area directly behind the fish, it first chooses a small number of neighbors, which

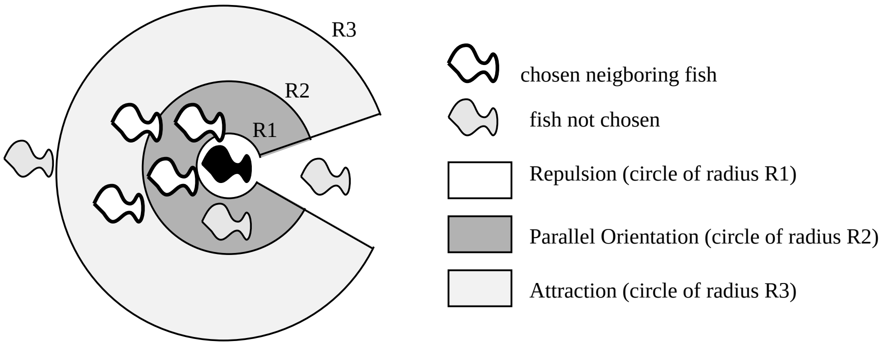

Fig. 2. Ranges of the Basic Behavior of a Fish

it will use to recompute its own vector of motion, i.e., its speed and orientation. In the model at hand, the number of neighbors is four and they are chosen and prioritized according to certain rules. In particular, neighbors that are closest and those that are in front are preferred. In Figure 2, the four white fish icons indicate the chosen neighbors. Based on each of its neighbor's distance, speed, and orientation, the fish then determines a possible correction in its own speed and/or orientation. The following cases are distinguished:

1. *Repulsion* – if the neighbor fish is too close (within the circle of radius R1), the fish tries to avoid a collision by turing away.
2. *Parallel orientation* – if the neighbor fish is in the preferred range, i.e., between R1 and R2, the fish swims in the same direction and at the same speed as its neighbor.
3. *Attraction* – if the neighbor fish is too far away, i.e., between R2 and R3, the fish swims in the direction of its neighbor.

The individual speed and orientation corrections resulting from considering each of its neighbor are then combined, for example by computing their averages, to determine the new speed and orientation.

The above model, which is based on actual observations of fish behavior, is an example of collective intelligence: the graceful and highly complex movements of the entire school are the result of local interactions among individual neighboring fish. Similar models have been formulated to investigate other phenomena of collective behavior. For example, by defining rules for inter-species interaction, it is possible to study the various evasive maneuvers used by fish schools to avoid predators.

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 3 shows a partitioning using a homogeneous grid where each grid cell is mapped

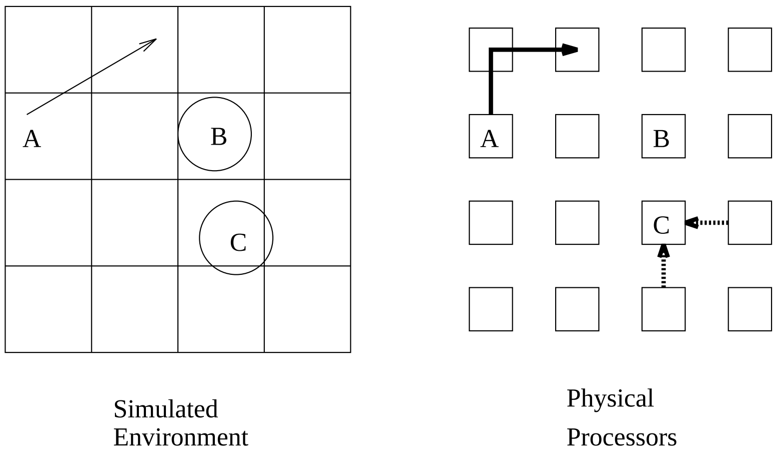

Fig. 3. Logical and Physical Object Migration

onto one physical node. In our present implementation, the physical network is a LAN of workstations, implying a fully connected topology. Hence processors only need to keep track of their logical neighbors in the simulated space.

Each individual fish is then implemented as a Messenger, which carries its own behavior. In general, it could be programmed to "live" for a certain period of time, to interact with other members of the same species or with predators, to multiply, etc. In the application at hand, the objective is to study the collective school behavior. Hence the behavior of each individual fish is to swim indefinitely in the simulated space and to continuously adjust its speed and orientation by interacting with its nearest neighbors as dictated by the biological model. For this purpose, each Messenger representing a fish carries its current position (coordinates within the simulated environment) and motion vector at a given point t in the simulated time. Based on these and the relevant information about its nearest neighbors, it computes its new position and vector at time $t + \Delta$. It then schedules the reinvocation of its behavior function to occur at time $t + \Delta$ using a time-based scheduling function (Section 2.3), which suspends it until that time. When it is awakened the function computes its new position and vector, and again suspends itself until the next Δ interval has passed. The node interpreter services all events in the order of simulated time and hence all fish advance concurrently as the interpreter is multiplexed among the individual Messenger programs.

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 2.1), which causes it to transfer into the processor responsible for the corresponding grid cell. Figure 3 illustrates the movement of a fish (A) in the logical space and the corresponding Messenger migration through the physical nodes.

To implement the necessary coordination among individual fish we need to consider two cases. As prescribed by the biological model, each fish examines its neighborhood in a given radius R3. When the entire circle is within a single grid cell (as is for example the case with fish B in Figure 3), the fish can look for its nearest neighbors within its own processor by examining node variables describing the presence of fish within that node. If, however, the circle extends outside the grid cell, it is necessary to communicate with one or more neighboring processors. For example, fish C in Figure 3 requires communication with two other neighbors. This is implemented by exchanging the contents of the boundary strips (of width R3) of each processor with the corresponding neighbor. This exchange may be triggered in either a demand-driven manner, i.e., by the arrival of a fish in a boundary strip, which requires that the corresponding neighbor's boundary strip be examined, or in a data-driven manner, i.e., by automatically reporting any changes in the boundary strips to the corresponding neighbors.

Advantages

Analogous to the Toxicology application of Section 3, there are two primary advantages of the MESSENGERS-based implementation of the fish schooling simulation over other approaches. The first is the ability to describe new behaviors at run time; the second is performance due to parallelism, reduced communication and the ability to perform discrete-event rather than time-driven simulation.

Dynamic behaviors: With most existing simulations, the different types of entities that will be participating in a simulation must be set up *a priori*. To introduce a new entity, the simulation programs have to be modified, recompiled, and restarted. But to be most effective, the user should not only have the ability to introduce entities with known behaviors (developed at compile-time) but should also be able to create entities with new behaviors while the simulation is in progress. For instance, in a school of fish model, the user may wish to introduce new types of fish (e.g. a predator), or modify existing behaviors.

One of the advantages of the MESSENGERS' philosophy is that it is inherently open-ended in that 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 can therefore be created and injected into the simulation space on the fly as the simulation is progressing. Similarly, existing behaviors may be changed by replacing Messengers on the fly.

Performance: To exploit parallelism, applications of the type described in this section need to deal with two important mapping issues: (1) how to divide the simulated environment (e.g. the ocean) over the processors, and (2) how to distribute the mobile objects (e.g. the fish). To reduce communication, it is imperative that the two be closely correlated. 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 3. If all fish in a particular logical grid cell are always 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 R3) 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 R3 relative to the grid cell size.

In a conventional message-based implementation of the problem, guaranteeing that the logical environment (including all mobile entities) 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. With MESSENGERS, on the other hand, the correlation requirement is satisfied automatically. Each Messenger representing a fish simply performs a *hop* statement when it detects that its new coordinates are outside the current grid cell. This causes it to migrate to the corresponding new processor. Since Messengers are inherently mobile in the physical space, this can be performed very efficiently.

In addition to the above dramatic reduction in inter-processor communication due to strict logical-to-physical space correlation, MESSENGERS also facilitate an easy transition from a time-driven to an event-driven simulation. The reason, which is analogous to that given for the Toxicology simulation in Section 3, is that each individual Messenger controls its own behavior. In a time-driven implementation it would recompute its position and schedule itself to wake up after a fixed time interval Δ using a time-based scheduling function. It is however just as simple to vary this interval by having each fish compute an appropriate Δ value at each step. For example, it could adjust Δ based on its current speed and/or the number of other fish in its neighborhood. This automatically adjusts the fidelity of the computation by invoking the relevant functions more frequently when necessary while not wasting unnecessary computational resources when changes are minimal. Making such a transition from a time-driven to an event-driven simulation in a conventional implementation is very difficult, since each is based on a fundamentally different organization.

5 Conclusions

We have presented a novel paradigm, called MESSENGERS, which is based on the concept of autonomous messages, each carrying a complete program to describe its behavior. These programs are interpreted by daemons running in each physical node of a network, which allows Messengers to navigate through the network, to communicate with one another, and to invoke precompiled node-resident functions. The MESSENGERS paradigm is related to other approaches that allow behaviors to be carried on messages. Among these are WAVE [Sap88], Java (Sun Microsystems), Telescript (General Magic), and various versions of "intelligent email". A survey and a classification of these systems based on their abilities to navigate autonomously and to coordinate the invocation of node-resident functions may be found in [BFD96].

In this paper we have focused on MESSENGERS' capabilities as a coordination paradigm, which can be exploited at two different levels of abstraction. First, each Messenger's behavior depends not only on the interpretive program it carries but also on the functions it may invoke and utilize when visiting a node. Hence the behavior of any given Messenger is determined by the composition of functions it invokes at run time. This capability, which we called intra-object coordination, allows us to view an application as a collection of distributed node-resident functions, whose invocation and data exchange is coordinated by the Messengers. Alternately, we can view an application as a collection of autonomous objects, each represented by a dynamically composed Messenger, which need to coordinate their actions among themselves in time and/or space. We refer to this level as inter-object coordination. We have shown, using concrete examples, that there are important classes of applications that can benefit greatly from the inherent flexibility and improved performance resulting from the MESSENGERS concept.

MESSENGERS is currently operational on a LAN of Sun-4 workstations interconnected by an Ethernet. Our current research efforts concentrate on the development of the two applications discussed in this paper and on further improvement of the MESSENGERS system itself, which includes the development of a graphics-based interface for the construction and control of simulation models, and on automatic load balancing to further improve performance. For additional information, the reader is referred to our WWW page:
http://www.ics.uci.edu/~bic/messengers.html.

References

[BFD96] L. Bic, M. Fukuda, and M. Dillencourt. Distributed Computing using Autonomous Objects. *IEEE Computer*, (to appear) 1996.

[Bic95] L. Bic. Distributed Computing using Autonomous Objects. *5th IEEE CS Workshop on Future Trends of Distributed Computing Systems, Cheju Island, Korea*, August 1995.

[Com94] DIS Steering Committee. The DIS Vision: A Map to the Future of Distributed Simulation. *Institute for Simulation and Training*, 1994.

[FMB+96] M. Fukuda, K. L. Morse, L. Bic, M. Dillencourt, E. Lee, and D. Menzel. A Novel Approach to Toxicology Simulation based on Autonomous Objects. *SCS Western MultiConference, Simulation in the Medical Sciences, San Diego*, Jan 1996.

[Fuk95a] M. Fukuda. MESSENGERS: Intermediate Code Specification. *Report MSGR-04, University of California, Irvine*, 1995.

[Fuk95b] M. Fukuda. MESSENGERS: Language Specification and System Interface. *Report MSGR-01, University of California, Irvine*, 1995.

[GC92] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[HDP88] M. Huston, D. DeAngelis, and W. Post. New Computer Models Unify Ecological Theory. *BioScience*, 38(10):682–691, November 1988.

[HE88] R.W. Hockney and J.W. Eastwood. *Computer Simulations using Particles.* IOP Publishing Ltd, Bristol, Great Britain, 1988.

[HW92] A. Huth and C. Wissel. The Simulation of the Movement of Fish Schools. *Journal of Theoretical Biology*, 156:365–385, 1992.

[RA84] J. C. Ramsey and M. E. Andeersen. A physiologically based description of the inhalation pharmacokinetics of styrene in rats and humans. *Toxicology and Applied Pharmacology*, 73:159–175, 1984.

[Res94] M. Resnick. Changing the Centralized Mind. *Technology Review*, pages 33–40, July 1994.

[Rey87] C.W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*, 21(4):25–34, July 1987.

[Sap88] P. Sapaty. WAVE-1: A New Ideology of Parallel and Distributed Processing on Graphs and Networks. *Future Generations Computer Systems*, 4(1), 1988.

[Vil92] F. Villa. New Computer Architectures as Tools for Ecological Thought. *Trends in Ecology and Evolution (TREE)*, 7(6):179–183, June 1992.