A Hierarchical Mapping Scheme for Mobile Agent Systems

Munehiro Fukuda
Lubomir F. Bic
Michael B. Dillencourt
Fehmina Merchant
Department of Information and Computer Science
University of California, Irvine, CA 92697

Abstract

MESSENGERS *is a system for general-purpose distributed computing based on the principles of mobile agents. Its underlying infrastructure consists of a collection of daemon processes, whose task it is to receive incoming agents, provide the necessary environment for their execution, and send them on to their next destinations as dictated by their behaviors. The structure of the daemon processes and their mapping onto the physical network is the main subject of this paper. Given the autonomous nature of each computation, the mapping problem is very different from those implemented for systems of communicating processes. At the same time, it offers new opportunities for solving the problems of load balancing, dynamic resource utilization, and trade-offs between computation granularity and communication overhead.*

1 Introduction

The terms mobile agents, intelligent messages, autonomous objects, and self-migrating threads have all been used recently to describe systems of self-contained computations capable of moving through an underlying network of computers by issuing explicit migration commands [IEE96, MA997]. The different terminology merely reflects the various application domains where such systems have been employed. This includes, among others, information retrieval (data mining), electronic commerce on the Internet, mobile computing, as well as general-purpose computing. In this paper we concentrate primarily on the latter, where an application is viewed as a collection of native-mode functions whose invocation and data exchange is coordinated by mobile agents.

Among the most important features that make mobile agents systems attractive for the above applications are the following:

- Self-migration is much *easier* to implement than regular passive process migration since each process decides for itself, i.e., at well-defined points, when it wishes to migrate. There is no need to take arbitrary process state snapshots. This, in turn, offers an elegant way for certain types of applications to perform load balancing and dynamic resource utilization.

- Applications are inherently *open-ended* and functionally extensible. Since computations may be moved arbitrarily at runtime, a node is not limited to a fixed set of predefined functions. Rather, each incoming request can carry with it the necessary code for its processing, thus making the node's capabilities virtually unlimited.

- Mobile agents offer better *performance* than message-passing systems for certain types of applications. The main reason is that computations that need to interact may dynamically move to the same node for the duration of their interaction, thus greatly reducing communication latency and overhead.

To give mobile agents their autonomy in moving through the underlying network and perform the necessary tasks in the nodes they visit, a basic infrastructure needs to be established. This consists of a collection of daemon processes, whose task it is to receive incoming agents, provide the necessary environment for their execution, and send them on to their next destinations as dictated by their behaviors. The structure of the daemon processes and their mapping onto the physical network is the main subject of this paper. Given the autonomous nature of each computation, the mapping problem is very different from those implemented for systems of communicating processes. At the same time, it offers new opportunities for solving the problems of load balancing, dynamic

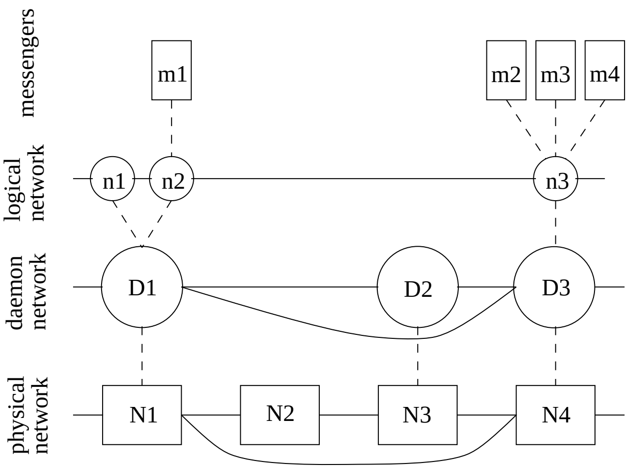


Figure 1: Network layers in MESSENGERS

resource utilization, and trade-offs between computation granularity and communication overhead.

2 MESSENGERS Overview

MESSENGERS [BFD96, FBDM97] is a system that supports the development and use of distributed applications structured as collections of mobile agents, called Messengers[1].

The MESSENGERS system involves four levels, as shown in Figure 1. The lowest level is the *physical network* (a LAN or WAN), which constitutes the underlying computational nodes. Figure 1 shows a portion of this network consisting of four nodes, $N1$ through $N4$, and their interconnections. Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process running a MESSENGERS language interpreter. At most one daemon process is running on any one physical node but the daemon network topology is arbitrary. That is, two daemon nodes, for example, $D1$ and $D2$, may be connected while the underlying physical nodes, $N1$ and $N3$, do not a have a direct link. This is implemented at the daemon level as virtual links using network node addresses. The *logical network* is an application-specific computation network created on top of the daemon network. As shown in Figure 1, any number of logical nodes (including none) may be mapped onto a daemon node. The topology of the logical network is again arbitrary, that is, logical links do not have to coincide with daemon links. The highest level in the hierarchy are the Messengers themselves, which may navigate freely by following existing logical links or directly hopping to other logical nodes. As shown in Figure 1, any number of Messengers (including none) may reside concurrently in any logical node.

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C and are compiled into a form of byte code for more efficient transport and parsing. Each script is carried in its entirety by the Messenger as it propagates through the network and is replicated each time the Messenger needs to follow more than one logical link. This requires a programming style different from the commonly used approaches to distributed computing. In particular, the user does not supply any node programs to execute on the various network nodes. Rather, all node programs and their communications are part of the underlying infrastructure of daemons. The application itself consists of only the logical network and the corresponding Messengers navigating this network. The programmer's point of view is that of a *navigator*, sitting in the "driver's seat" of a Messenger, and guiding it on its way through the computation.

Messengers navigate through the logical network based on their own internal program and state. They are also capable of cloning themselves, both implicitly and explicitly, to follow multiple links or to perform different subtasks. This is accomplished by explicit *navigational* statements, which also permit the creation or destruction of logical links and/or nodes. A number of optional parameters may be specified as part of the navigational statements, including the specification of particular nodes, links, or link weights. Wild cards may also be used for partial matching. The Messenger is replicated and propagated to all destinations that match the navigational specification.

Messengers may also perform arbitrary computations in the nodes they visit. This can take two forms. First, the object's internal program may contain *computational* statements, which are interpreted, and which permit arbitrary arithmetic, logic, and control operations to be performed. Second, the Messengers may invoke ordinary C functions as part of their behavior or spawn complete programs as separate concurrent Unix processes. The C code is executed in native mode of the current physical node.

The basic principles and programming aspects of MESSENGERS have been presented in earlier papers [Bic95, BFD96, FBD97]. Hence in the remainder of this paper we will focus on the mapping problem, which extends MESSENGERS' conceptual view to also include issues of performance on a distributed archi-

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

tecture.

2.1 The Mapping Problem

Given the four levels of Figure 1, the mapping problem in MESSENGERS is much more complex than a single-level logical-to-physical mapping common to message-passing systems. It involves the following hierarchy of mappings, each of which presents different challenges but also new opportunities to solve important problems:

- *Messenger-to-Logical-Node Mapping.*
- *Logical-Node-to-Daemon-Node Mapping.*
- *Daemon-Node-to-Physical-Node Mapping.*

2.1.1 Messenger-to-Logical-Node Mapping

Each Messenger decides for itself which logical node it executes on at any point in time. If these decisions are based on some properties of the logical node, this can be used effectively for load balancing in certain types of applications. In particular, a node's area of responsibility may be changed, which automatically forces Messengers to hop to other nodes.

This is applicable to certain classes of problems, notably, individual-based simulations, where Messengers, each representing an autonomous entity, navigate through a 2- or 3-dimensional virtual space. To illustrate this type of application, consider a biological model of simulating a school of fish [HW92]. The purpose of this model is to reproduce "collective intelligence", i.e., the ability of a group of individuals to perform certain functions without any individual taking on the role of a leader. Rather, the group behavior is the results of local interactions among neighboring fish.

Assume that the simulated body of water is a two-dimensional space partitioned into cells along the x and y axes. Each cell, implemented as a logical node and connected to its four nearest neighbors, is assigned to a workstation and hence all fish Messengers currently located in that cell operate on the corresponding workstation.

The MESSENGERS script for each fish is shown in Figure 2. Each fish is programmed to "live" for a number of steps corresponding to the simulation run (line 2). At each step, the fish computes its new position (line 3) by calling a native-code C function *upd_pos*. Lines 4–8 accomplish the migration of the fish; when its y-coordinate falls outside of the current node's area of responsibility, the fish transfers itself to the corresponding "southern" or "northern" neighbor using a *hop* statement. Similarly, it may hop "east" or "west" based on its x-coordinate. Hence the fish move not only within the simulated space but actually migrate among processors in the network.

```
(1) fish() {
(2)  for (i = 0; i < sim_run; i++)
(3)    func(name = "upd_pos"; ...)
(4)    if (my_y < min_y)
(5)       hop(link = SOUTH);
(6)    elseif (my_y > max_y)
(7)       hop(link = NORTH);
(8)  /* analogous code for EAST/WEST */
(9) }
```

Figure 2: Fish behavior in school simulation

For the purposes of load balancing, each logical node represents an area of responsibility within which Messengers move by recomputing their respective positions. When the new position falls outside of a node's area of responsibility, the Messenger automatically hops to the appropriate node. This can be exploited for load balancing by dynamically adjusting the decomposition of the simulated 2- or 3-D space. When a node's area of responsibility decreases, some of its Messengers will automatically be forced to hop to neighboring nodes, thus decreasing its load. There is a variety of possible spatial decompositions and information exchange strategies. The main point, however, is that load balancing automatically exploits the autonomous navigation capabilities of Messengers to coax them away from heavily loaded nodes to achieve better overall load distribution.

2.1.2 Logical-Node-to-Daemon-Node Mapping

The daemon network provides a virtual network whose topology is freely configurable by the user. The logical network is then mapped to this intermediate daemon network as follows. Since deriving an optimal explicit mapping is not only very difficult but oftentimes unnecessary, MESSENGERS aims at an *implicit* mapping scheme that gives the user sufficient control over the amount of *parallelism* versus the *communication overhead*, but, at the same time, does not require an exact assignment to daemon nodes or even the knowledge of their exact topology.

The logical network is created by individual Messengers incrementally at runtime. At system startup, a single logical node, named *init*, is created on every daemon node. Any Messenger may be injected (from the shell or by another Messenger) into any of the *init* nodes. From these it may start creating new logical nodes and links on the current or any other daemon.

Dynamic incremental creation is necessary to support the full range of dynamic computation problems, which require the creation of structures whose size and shape are unknown *a priori*. The creation of new nodes and/or links is a form of navigation, expressed using the following single *create* statement[2]:

$$create(n; l; d; N; L; D)$$

where each triple (n, l, d) specifies a *new* logical node. This is created on the daemon node specified by the triple (N, L, D).

The new node n to be created can be named or unnamed, and can be connected to the current node c by the link l, which could be named or unnamed. The link can also have a direction specified in d. The target daemon node N may be specified by name or by the symbol "$*$," which denotes a "wild card", i.e., a symbol that matches any name. N must be connected to the current daemon by a link whose name matches the given variable or constant L and direction D; "$*$" again matches any name.

The semantics of *create* are then as follows: For each destination specification (N, L, D) determine the set of daemon nodes that match the specification. That is, (1) the daemon's name or address must match the specified name or address and (2) the daemon must be connected to the current daemon via links that match the specified link name and direction. From this (possibly empty) set of potential destinations, choose *one* and create the logical node n on it.

The choice of the daemon from the set of possible daemons is key to the implicit mapping strategy, which consists of two-parts. First, it must be decided whether the new node is to be created locally or on another daemon. If the latter, the daemon must choose among possibly many potential target daemons. The first decision is controlled by a parameter, called *creation threshold value*. This is set by the user at systems initialization and controls the number of logical nodes that can be created on any given daemon. That is, a create operation will create the new logical node locally on the current daemon until the threshold value is exceeded. At that time, the Messenger is forced to another daemon where the logical node is created and the creation counter at the source daemon is reset to zero.

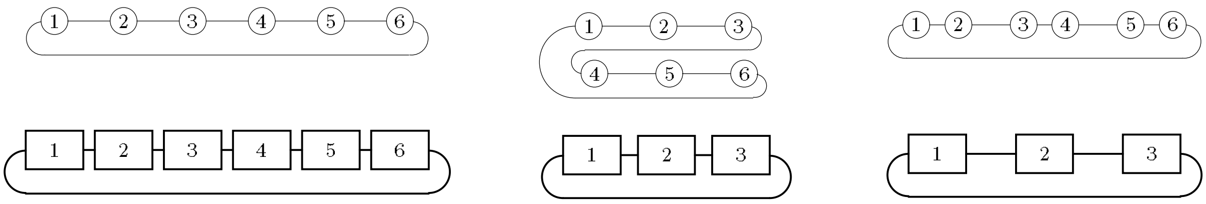
The second decision is based on the history of creations maintained by each daemon node. Initially, when more than one potential target daemon meets the selection specification, one is chosen at random. For each subsequent remote creation, a different daemon is chosen. When all possibilities have been exhausted, the sequence is repeated, thus choosing subsequent target daemons in a round robin manner.

Based on the preceding discussion, the complete mapping strategy currently implemented in MESSENGERS can be summarized as follows. Prior to systems initialization, the user describes a daemon network of arbitrary size and topology in a configuration file. The mapping of the logical network being constructed by individual Messengers at runtime onto this daemon network is controlled through the following parameters:

- The *number of nodes* in the daemon network. This determines the maximum *degree of parallelism.*
- A *creation threshold value*. This determines the number of logical nodes that can be created by any Messenger on any given daemon. Hence it controls the *clustering* of the computation.
- The *topology* of the daemon network. The fanout and connectivity of each daemon's links control the *placement* of the logical nodes relative to each other.

The order in which the logical nodes are created by the application, in conjunction with the above parameters, then determines the actual mapping of logical nodes onto daemon nodes. To illustrate this graphically, consider the mapping of a simple static one-dimensional structure, such as a ring, onto the physical network. Assuming the n logical nodes are created in ascending order starting from 1, we can produce a 1-to-1 mapping by selecting n daemons, interconnecting them into a ring, and specifying the creation threshold as 1. This forces each new logical node to be created on a new daemon, resulting in the mapping shown in Figure 3a. Figures 3b and c show how granularity can be increased by reducing the size of the daemon network. Figures 3b is the result of reducing the number of daemons by half while keeping the creation threshold still at one. The logical ring wraps around the daemon network twice. In Figures 3c, the threshold value has been doubled, thus resulting in a clustering of two logical nodes per daemon. Note that each of the mappings was accomplished by modifying the daemon topology/size or the creation threshold values, all of which are defined outside of the actual application program. Hence it is possible to adapt the application to different physical environments *without reprogramming or recompilation.*

[2]The actual *create* statement is more complicated, permitting multiple nodes/links to be created during the same operation. This simplified form, however, is sufficient to illustrate the mapping problem.



(a) (b) (c)

Figure 3: Mapping of a ring structure

The above example can easily be extended to multi-dimensional structures, permitting the user to control the clustering of individual elements, columns, rows, or planes of such structures. It can also be applied to the mapping of non-rectangular structures, such as trees, whose size and/or shape may only be known at runtime.

The main point here is that the mapping scheme is implicit, i.e., the user does not need to know anything about the physical network, yet it allows granularity and parallelism to be controlled at the programming level.

2.1.3 Daemon-Node-to-Physical-Node Mapping

The mapping of daemon nodes onto physical nodes needs to satisfy two main properties: (1) the communication distances between daemon nodes should be minimized; (2) if the network is shared by other users/processes, the mapping must select the currently least loaded nodes.

MESSENGERS is currently running on an Ethernet-based LAN, which is "fully connected" and hence condition (1) is trivially satisfied. For a non-fully connected network, including a ring network or an inter-network of multiple LANs, many of the existing mapping strategies [LA87, CA93, YBN93, BS95] are applicable since the daemon network topology is static. This section focuses primarily on the second issue, which requires dynamic resource control.

The current MESSENGERS implementation supports two forms of daemon-to-physical mapping. The first is *explicit* and *static*; the user must select the exact set of physical nodes (workstations) that are to participate in the computation by listing them in a configuration file. This has the disadvantage that the user needs to be aware of the current network status. In particular, any disconnection or reconnection of a workstation or change in their current load require a manual change to the configuration file and thus makes it difficult to take advantage of the currently least loaded workstations. The second scheme, while still static, is *implicit*. The user only needs to provide a list of potential workstations on which the current daemon network is to be mapped. This may contain all the workstations the user has access to or any subset thereof. The initialization routine then examines the current loads of all listed workstations and assigns those with the currently smallest load to the daemons.

A dynamic mapping facility is currently being developed to take advantage of the changing loads at run time. When the performance of a workstation decreases below some threshold (due to other unrelated processes running on that workstation), it will attempt to find an alternate workstation and relocate the entire daemon process (including all its logical nodes and Messengers) to that new target. This is similar to object migration [JLHB88, RTL+91] or process migration at the operating systems level, which, in its full generality, requires taking a complete process snapshot at an arbitrary point in time and reinstating it on a different machine. Fortunately, MESSENGERS daemons are generic interpreters and thus can be programmed to effectively *migrate themselves* without much help from the system. The basic idea is to provide a primitive to move a logical node from one daemon to another. This can be implemented using a simple message exchange between the source and target daemon, during which the internal data structures representing the chosen logical node and any Messengers currently executing in that node are transferred to the target.

Given such a primitive, a daemon migration then proceeds as follows: a new daemon process is created on the selected target machine. The source daemon, having completed its current interpretation cycle, transfers all its logical nodes to the target daemon using the above primitive. This transfer is atomic since the source daemon ceases interpreting or receiving any new Messengers. Once the transfer of all logical nodes is completed and all links are updated correspondingly, the source daemon is destroyed while the new daemon resumes it operation with the next interpretation cycle.

The main point here is that self-migration can also be applied at this level to achieve dynamic resource utilization. This is much easier to implement than passive migration since it does not require the taking of process snapshots at arbitrary points.

3 Conclusions

This paper presented a framework for the mapping problem in MESSENGERS. Due to the self-migrating features of the computations and the layered organization of the infrastructure, this presents special problems, requiring different solutions at each level. The hierarchical mapping framework illustrates that systems based on self-migrating computations offer new opportunities for solving important problems in distributed computing, including load balancing, dynamic resource utilization, and implicit granularity control.

MESSENGERS is operational on Sun SPARCstations under UNIX (SunOS and Solaris). Communication among daemons, each implemented as a single UNIX process, is accomplished via sockets and hence MESSENGERS can run over the Internet or locally within a LAN. All system software is available in source form for research purposes: `http://www.ics.uci.edu/~bic/messengers`.

References

[BFD96] L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8), Aug. 1996.

[Bic95] L.F. Bic. Distributed computing using autonomous objects. In *5th IEEE CS Workshop on Future Trends of Distributed Computing Systems*, Aug. 1995.

[BS95] F. Berman and B. Stramm. Mapping function-parallel programs with the Prep-P automatic mapping preprocessor. In B. Shirazi, A. Hurson, and K. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, 1995.

[CA93] V. Chaudhary and J.K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 4(3):328–46, March 1993.

[FBD97] M. Fukuda, L.F. Bic, and M.B. Dillencourt. Messages versus messengers in distributed programming. In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS-97)*, Baltimore, MD, May 1997.

[FBDM97] M. Fukuda, L.F. Bic, M. Dillencourt, and F. Merchant. Distributed coordination with messengers. *Science of Computer Programming*, (to appear), 1997. Special Issue on Coordination Models, Languages, Applications.

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

[IEE96] IEE. *IEE Colloquium on Intelligent Agents and Their Applications*, No.1996/101, London, UK, Apr. 1996.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grain mobility in the Emerald system. *ACM Trans. Computer Systems*, 6(1), Feb. 1988.

[LA87] S-Y. Lee and J.K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. Computers*, 36, Apr. 1987.

[MA997] *MA '97: First International Conference on Mobile Agents*, Springer-Verlag, Lecture Notes in Computer Science, Berlin, Germany, Apr. 1997.

[RTL+91] R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software—Practice & Experience*, 21(1):91–118, Jan. 1991.

[YBN93] J. Yang, L.F. Bic, and A. Nicolau. A mapping strategy for MIMD computers. *Int'l Journal of High Speed Computing*, 5(1), 1993.