Mobile Agents, DSM, Coordination, and Self-Migrating Threads: A Common Framework

LUBOMIR F. BIC and MICHAEL B. DILLENCOURT
Department of Computer Science
University of California, Irvine
Irvine, CA 92617
USA
bic@ics.uci.edu http://www.ics.uci.edu/~bic

Abstract: - We compare four paradigms that have recently been the subject of considerable attention: mobile agents, distributed shared memory (DSM), coordination paradigms, and self-migrating threads. We place these paradigms in a common framework consisting of three layers—the computational model, its implementation on a physical architecture, and the interface to the system's environment—to demonstrate that self-migrating threads subsume the other three paradigms in terms of their capabilities to organize and coordinate computation, map the concurrent activities onto a multicomputer architecture, and provide an interface for interaction with their environments on the underlying host computers.

Key-Words: - Self-migrating threads, mobile agents, DSM, coordination

1 Introduction

Mobile agents, distributed shared memory (DSM), coordination paradigms, and self-migrating threads represent four lines of research that have each gained considerable attention in recent years. Mobile agents provide autonomous service entities capable of roaming communication networks in search of information and services. DSM aims at providing an abstraction for distributed memory computers such that applications could be written using shared memory programming paradigms. Coordination paradigms also focus on providing structured abstractions of the data or information space but, in addition, provide new conceptual models for expressing concurrency and coordination among the activities operating on the structured logical space. While these three research areas appear to be unrelated, there are similarities among them that are best understood by examining them in a common framework together with self-migrating threads. Self-migrating threads navigate through a logical space, based on their own internal program and state, and collectively solve a global problem through their individual efforts. The self-migrating threads draw heavily on ideas from the other three areas.

2 Mobile Agents

Mobile agents are self-contained entities that can navigate autonomously through the underlying network and perform a variety of tasks in the nodes they visit. Figure 1(a) captures the essence of most mobile agent systems, which focus on the following major aspects:

- The computational model underlying mobile agents system is similar to a multithreaded environment, where individual threads consist of a program and a state, and communicate with one another via shared or distributed memory mechanisms. The main extension to this model is navigation. The computational model provides special commands or other linguistic constructs that enable agents to relocate themselves or their clones to other physical nodes in the network and to continue executing in the new environment.
- To serve a useful function, a mobile agent must be able to interact with the environment of the host on which it currently resides. This is accomplished by providing an interface to the host's operating system, which permits the agent to access data and/or invoke services available on the current host.
- To permit autonomous navigation, a layer of software consisting of daemons is superimposed on the underlying physical network. The task of each daemon process is to receive agents, interpret their behavior, and send them on to other daemons as necessary. The daemons themselves have no intelligence; all functionality is carried as part of the mobile agents. The daemons use existing physical links to communicate with one another. Hence the mapping of resulting daemon network onto the physical network is trivial; the former is a subset of the latter as determined by the user.

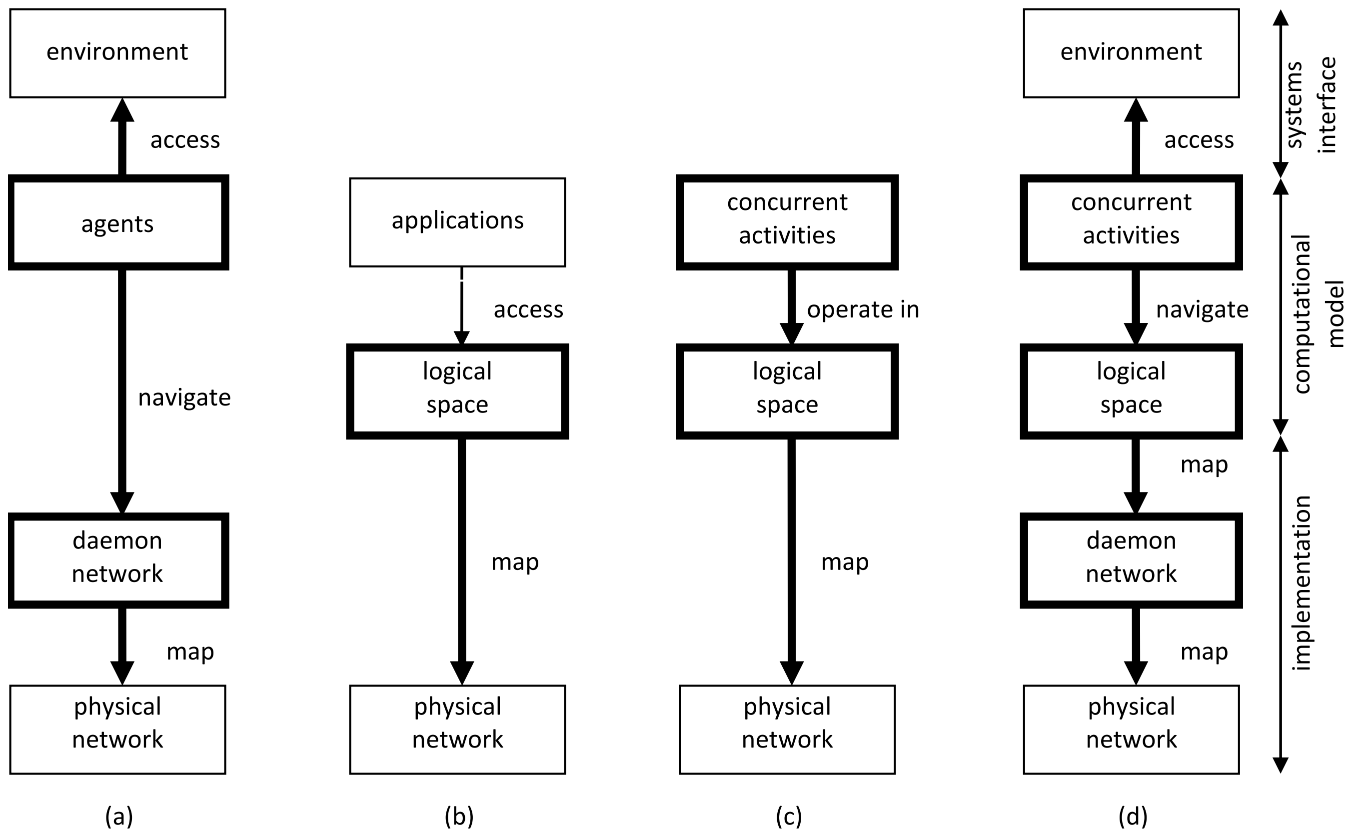

Fig. 1: Comparison Framework. (a) Mobile agents. (b) DSM. (c) Coordination paradigms. (d) Self-migrating threads

A number of mobile agent projects have been carried out in recent years [1,2,3]. Most focus on "intelligent" agents, i.e., those that can serve as personal assistants, roaming the Internet and perform arbitrarily complex services on behalf of their users. One of the first proposals was Telescript [4], which was centered on the design of a special-purpose language for expressing agents' behaviors, including their ability to move themselves around the Internet. More recent approaches rely on existing languages, such as Java (used by IBM Aglets [5]) or Tcl/Tk (used by Agent Tcl [6] and Tacoma [7].)

Another focus of mobile agent research has been on intelligent communication. This is best represented by the Messenger projects of the U. Geneva [8]. Mobile agents are viewed as "prototypic" mobile agents on top of which more complex "intelligent" agents can be built. The objective of this project is to replace traditional messages and communication protocols by mobile agents.

There are a number of advantages of mobile agents over traditional approaches. First, they offer a more natural metaphor for both users and programmers in that they replace the traditional client/server or send/receive points of view by self-contained activities that encapsulate both communication and remote computing. A second advantage is the inherently open-ended nature of mobile agents, which permits new functionality to be introduced at runtime as needed. Finally, mobile agents can significantly reduce message traffic in client/server type applications. Instead of engaging in a bandwidth and latency intensive message exchange with a server, the client may dispatch an agent to the server site, which performs all the necessary interactions locally. 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.

The heavy lines of Figure 1(a) indicate the emphasis of mobile-agent system on the agents' navigational capabilities, the daemon infrastructure, including its mapping to the physical network, and the agents' interface to the host's environment.

3 Distributed Shared Memory

Distributed shared memory (DSM) systems provide the *illusion* of a common shared memory on a multi-computer, where each node only has a private local memory and can communicate with other nodes via a network or a switch. Figure 1(b) captures the essence of

most DSM systems, indicating the main emphasis of this line of research:

- The shared space provided by a DSM system is a passive component, which is accessed by the various applications running on the system. The organization and structure of the shared logical space is what distinguishes different DSM approaches. These represent the trade-offs between the system's expressiveness and the resulting performance.
- The implementation provides the mapping of the logical space onto the physical architecture. Its complexity depends on the size of the semantic gap that must be bridged.

One of first approaches to providing DSM was based on paging [9]. Similar to a paged-based virtual memory in a single-processor system, the virtual shared space is partitioned into fixed-size pages. However, instead of moving them between primary and secondary memory as needed, they are moved between different processors. A number of implementations have been proposed to keep track of the migrating pages to facilitate performance while ensuring that memory consistency is not violated.

The above approach to DSM guarantees sequential memory consistency, which is the most convenient from the programming point of view but also the most costly to implement. Other approaches have taken a more restricted view of what a DSM is to gain better performance [10]. These restrictions require the memory consistency model to be weakened. For example, a causally consistent DSM guarantees that different processes see only causally-related accesses to the shared variables in the same order, while causally-unrelated accesses may be observed in a different order. In addition, the view of the shared memory may change. That is, instead of providing a one-dimensional flat sequence of data locations, thus mimicking the view of physical RAM, the shared portion may be restricted to only certain variables or data structures. In this case, special synchronization primitives are typically provided, with the understanding that the consistency of the shared data is guaranteed only in conjunction with these primitives. For example, release and entry consistency guarantee a consistent view of shared data only when a critical section is exited or entered, respectively.

Regardless of the particular scheme or implementation, the focus of all DSM-based schemes is the logical space organization and its mapping onto the underlying physical architecture, as shown by the heavy lines of Figure 1(b).

4 Coordination Paradigms

Coordination paradigms are closely related to DSM and it is difficult to draw a clear line between the two research thrusts. We characterize coordination paradigms as approaches that go significantly beyond the scope of DSM by addressing not only the aspect of space but also integrate its operational aspects into a common model:

- Like DSM, coordination paradigms provide the abstraction of a logical space, which consists of data and possibly functions, and which is structured specifically to facilitate the development of distributed applications. Unlike DSM, it is not always the data that is brought transparently to the current processes or thread as needed. Rather, a coordination paradigm may support the ability of an activity to relocate itself to another (physical or logical) domain to gain access to some data.
- In addition to the above *spatial* aspect, coordination paradigms also incorporate a *temporal* aspect by providing specific mechanisms or constructs to operate on the logical space, thus coordinating the concurrent activities comprising the computation. These, in general, are closely integrated with the logical space. They typically include mechanisms for controlling synchronization, communication, and creation/destruction of the computational activities required to orchestrate the operation of a complex system. Hence, from the programming point of view, coordination paradigms may be viewed as extensions of the DSM concept.

The two abstract layers, which are the main focus of all coordination paradigms (as indicated by heavy lines of Figure 1(c)), are then mapped onto the underlying computational structure—a network or a multiprocessor. The mapping, however, is typically outside of the scope of the coordination paradigm.

A large number of coordination paradigms have been proposed and developed in recent years, which can be subdivided into several broad categories. One approach to coordination utilizes *channel-based* communication between processes. Processes communicate directly with each other by reading from and writing to *ports*. Ports of processes are connected to ports of other processes via *channels*. This approach leads to a clean separation of computation and coordination functions. An example of the channel-based approach is the IWIM model [11].

Another approach to coordination is *medium-based* coordination. At a very abstract level, all medium-based approaches to coordination work on the same principle. There is a common medium or state space, shared by the processes. Processes can modify the state space, and

these modifications affect the behavior of other processes. Computation is performed by the processes, and coordination is achieved through the shared state space.

One of the most prominent examples of the medium-based approach is Gamma [12], based on a chemical reaction metaphor. The state space is a multiset of objects. Gamma programs consist of matched (reaction conditions, action) pairs. Execution proceeds by replacing a collection of objects that satisfy a reaction condition by the result of applying the corresponding action. As programs are executed, they may cause multiset transformations that create the reaction conditions necessary to allow other programs to execute.

Another well-known example of coordination through a shared state space is the Linda system [13]. The state space is a pool of data called a *tuple space*. Processes may insert, read, and remove tuples from the tuple space using various primitives. They may also spawn new activities that leave new tuples in the tuple upon their termination. Processes select tuples associatively, by issuing requests for tuples that match certain templates.

In the Linda model, the state space is shared by all processes. PoliS [14] is an enhancement to the basic model intended to simplify the design of distributed systems. PoliS allows multiple named tuple spaces, called *places*, where each tuple belongs to exactly one tuple space. The execution threads in PoliS are autonomous active tuples, called *agents*. Because agents are tuples, an agent belongs to exactly one tuple space. An agent can read tuples inside its own tuple space and can write tuples to any tuple space. These simple operations provide a uniform approach to spawning new activity, the migration of such activities, and the exchange of information among them.

Despite the significant differences among the various coordination paradigms, they all share the common characteristics captured by the two highlighted layers of Figure 1(c).

5 Self-Migrating Threads

MESSENGERS [15,16,17,18] is a system based on the principles of self-migrating threads, called *Messengers*. The system distinguishes three separate levels of networks. The *physical network* is the underlying computational resource. The *daemon network* is a collection of Unix processes, whose task is to interpret the behavior of the self-migrating threads. The *logical network* is an application-specific computation network created at run time on top of the daemon network. Multiple logical network nodes may be created on the same daemon network nodes, thus running on the same physical node, and they may be interconnected by logical links into an arbitrary topology.

The self-migrating threads 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 self-migrating thread is replicated and propagated to all destinations that match the navigational specification.

Self-migrating threads 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 permit arbitrary arithmetic, logic, and control operations to be performed. Second, the objects may invoke ordinary C functions as part of their behavior or spawn complete programs as separate concurrent Unix processes. The system also supports implicit mapping of the logical network onto the daemon network.

6 Conclusion

Figure 1(d) illustrates graphically the capabilities of self-migrating threads vis-a-vis the other three lines of research discussed earlier. This shows that self-migrating threads incorporate aspects of mobile agents, DSM, and coordination paradigms by explicitly addressing all levels of the proposed framework:

- Like coordination paradigms, self-migrating threads provide mechanisms for organizing the data and activities in space. This consists of a logical network, which contains local variables accessible by activities operating in the current logical node.
- Self-migrating threads also support temporal coordination. Computations are capable of moving autonomously through the logical network, accessing data and functions in each node, and coordinating their activities with one another. This form of navigation is reminiscent of mobile agents but is performed at the logical rather than the daemon level and thus is an integral part of coordination.
- Like mobile agents, self-migrating threads use a daemon network that provides an abstraction of the underlying physical network. This permits activities to migrate between physical nodes and hence is used to implement navigation in the logical network. That is, whenever an activity moves between two logical

nodes mapped onto separate physical nodes, the corresponding daemons accomplish the transfer.

- The daemons also play a crucial role in the mapping of the logical organization to the physical network, thus explicitly addressing the mapping problem that is the main focus of DSM systems. The logical network is the shared space potentially accessible by any Messenger. It is however not a simple linear array of cells. Rather, all variables are segregated into logical nodes and connected by logical links. Whenever a Messenger hops from one node to another, the set of shared global variables it is able to access at that moment changes.
- Like mobile agents, self-migrating threads also support an interface to the system's environment, which allows the current activities to access data and services provided by the host's operating system. In particular, a Messenger can dynamically load, link, and execute a C function in native mode, spawn a new concurrent UNIX process to run any C program accessible on the current host, and it can also communicate (via a shared buffer or a message queue) with any currently running UNIX process on the same host.

The above characteristics make self-migrating treads into a powerful programming paradigm that offers many advantages over message-passing and other computational paradigms. Specifically, self-migrating threads offer a new style of programming, referred to as Navigational Programming [19], which provide the ability to form mobile pipelines [20], to speed up a sequential computation by distributing its underlying data [21], and to support incremental parallelization of sequential programs [22,23].

References:

[1] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus, Mobile Agents: Motivations and State-of-the-Art Systems, *Technical Report TR2000-365*, Dartmouth College, Hanover, New Hampshire, April 2000.

[2] D. Kotz, R. Gray, and D. Rus, Future Directions for Mobile Agent Research, *IEEE Distributed Systems Online*, **3**(8) (2002).

[3] D. Shiao, Mobile Agents: A New Model of Intelligent Distributed Computing, *IBM Developer Works*, China. 2004.

[4] The Telescript Reference Manual, Tech Report, General Magic Inc., Mountain View, CA 940404, June 1996.

[5] G. Cabri, L. Ferrari, L. Leonardi, R. Quitadamo, Strong Agent Mobility for Aglets based on the IBM JikesRVM, *Tech Report,* Universita di Modena e Reggio Emilia, 2003.

[6] R. S. Gray, Agent Tcl: A flexible and secure mobile-agent system, *Proceedings of the Fourth Annual Tcl/Tk Workshop* (TCL 96), Monterey, CA, 1996

[7] D. Johansen, R. van Renesse and F. B. Schneider, An Introduction to the TACOMA Distributed System, *Technical Report 05-23*, Department of Computer Science, University of Tromso, 1995.

[8] G. Di Marzo, M. Muhugusa, C. Tschudin and J. Harms, The Messenger Paradigm and its Impact on Distributed Systems, *ICC'95 Workshop on Intelligent Computer Communications*, 1995.

[9] K. Li, A Shared Virtual Memory System for Parallel Computing, *Proc. of the 1988 Int'l Conf. on Parallel Processing*, 1988.

[10] G. Antoniu and L. Bouge. DSM-PM2, A portable implementation platform for multithreaded DSM consistency protocols, *Lecture Notes in Computer Science*, 2026:55, 2001.

[11] F. Arbab, The IWIM Model for Coordination of Concurrent Activities, in *Coordination Languages and Models*, Cesena, Italy, 1996.

[12] J-P. Banatre and D. Le Metayer, Programming by Multiset Transformation, *Comm. CACM,* 36(1):98-111, 1993.

[13] N. Carriero and D. Gelernter, Linda in Context, *Comm. CACM,* 32(4), 1989.

[14] P. Ciancarini, Distributed Programming with Logic Tuple Spaces, *New Generation Computing*, 12(3):251-284, 1994.

[15] L. Bic, M. Fukuda, and M. Dillencourt, Distributed Computing using Autonomous Objects, *IEEE Computer*, 29(8), 1996.

[16] M. Fukuda, L. Bic, M. Dillencourt, F. Merchant, MESSENGERS: Distributed Programming Using Mobile Agents, *Transaction of the Society for Design and Process Science* (SDPS), Vol. 5, No. 4, 2001

[17] M. Fukuda, L. Bic, M. Dillencourt, and F. Merchant, Distributed Coordination with MESSENGERS, *Science of Computer Programming*, 31(2), 1998

[18] R. Utter, Diaktoros: Full State Migration with Mobile Agents, *PhD Thesis*, Dept. of Information and Computer Science, University of California, Irvine, 2006.

[19] L. Pan, L. Bic, M. Dillencourt, and M. K. Lai, NavP Versus SPMD: Two Views of Distributed Computation, *Int'l Conf. on Parallel and*

Distributed Computing and Systems (PDCS 2003), Marina del Ray, CA, November 2003

[20] L. Pan, M. K. Lai, M. Dillencourt, and L. Bic, Mobile Pipelines: Parallelizing Left-Looking Algorithms Using Navigational Programming, *12th IEEE Int'l Conf. on High Performance Computing* (HiPC-2005), Goa, India, December 2005.

[21] L. Pan, L. Bic, M. Dillencourt, and M. K. Lai, Distributed Sequential Computing, *Advances in Computation: Theory and Practice*, Vol. 16, Nova Science Publishers, Inc., New York, 2004.

[22] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M.. Dillencourt, and L. Bic, Incremental Parallelization Using Navigational Programming: A Case Study, *International Conference on Parallel Processing* (ICPP-2005), Oslo, Norway, June 2005.

[23] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M.. Dillencourt, L. Bic, and L. Yang, Toward Incremental Parallelization, *IEICE Trans. Inf. & Syst.*, Vol. E89-D, No. 2, pp. 390-398, Feb. 2006.