Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment

Mario Tokoro[*]

Department of Computer Science

Keio University

3-14-1 Hiyoshi, Yokohama 223 JAPAN

Tel: +81-44-63-1926 Telefax: +81-44-63-3421 E-mail: mario@keio.ac.jp

June 11, 1990

Abstract

This paper proposes a new computing model called *computational field model* for solving a problem in an object-oriented open distributed environment. In this model, we envisage an open-ended distributed environment as a continuous computational field. We introduce the notion of *mass, distance, gravitational force, repulsive force,* and *inertia* of objects to define a metric space for the field, so that objects migrate to their (sub)optimal (or satisfactory) locations. We also propose a communication model called *assimilation/dissimilation model*, where communication is represented as the assimilation/dissimilation of messenger objects and their migration.

1 Introduction

The more development we see in computer and communication technologies, the more our expectations of computer systems grow. Current demands on computer systems can be summarized as follows: solving larger and more complex problems, realizing more reliable real-time processing, and providing better user interfaces. When predicting how computer systems in the next decade would be like, we must change our view: the computational environment will be distributed, ubiquitous, open-ended, and evolutionary.

[*]also with Sony Computer Science Laboratory Inc., Takanawa Muse Building, 3-14-13 Higashi Gotanda, Shinagawa-ku, Tokyo, 141 JAPAN. Tel: +81-3-448-4380, Telefax: +81-3-448-4273, E-mail: mario@csl.sony.co.jp

Issues then will be shifted from intra-problems to inter-problems. That is to say, we have been trying to solve a large complex problem by means of raising the level of abstraction in describing problems and employing inherent parallelism of the problem. In addition to such endeavors, we should now start to investigate means to satisfy users of different purposes in an open distributed computational environment, while utilizing resources efficiently.

In this paper, we propose a new computing model called *computational field model*, or *CFM* for short. CFM is a computing model for solving many large and complex problems of different purposes in an open-ended distributed environment. We understand that solving a problem implies mutual effects between the computational field and the problem. That is, problems themselves change or affect the environment itself. Or, we can even say that problems are a part of the computing environment. In this sense, we share the same view as described in [Huberman 88].

Our model is based on concurrent objects [Tokoro 88] [Yonezawa 87]. In this model, we envisage an open-ended distributed environment as a continuous computational field. We consider the migration of objects to be a facility that the environment should primarily provide in order to support optimal utilization of resources and users mobility.

We then introduce the notions of *distance* between objects and the *mass* of an object to the model in order to form a metric space. We define *gravitational force*, *repulsive force*, and *inertia*, metaphorically to dynamics. Using these measures, objects migrate to their (sub)optimal (or satisfactory) locations during computation (i.e. over their lifetime). We call this *Mass and Distance-based computing* or *MD-based computing* for short.

The message-passing paradigm for communication doesn't fit well with the computational field model, because the paradigm hids the inherent nature of an open distributed environment. For example, it cannot represent a message in transit and communication delay. Thus, we propose a communication model for the computational field model. It is called *assimilation/dissimilation model*, or *A/D model* for short. In this model, communication is represented as the dissimilation of a messenger object at the sender object, migration, and assimilation of it at the receiver object. Therefore, computation in CFM is represented in a unified manner in terms of objects and their migration.

2 From Parallel to Open Distributed Computing

Parallel computing is one of the important methods for high-speed computing. Parallel computation of a program is achieved in the following two steps:

1. decomposing a problem into subproblems and
2. allocating subproblems to parallel hardware.

It is necessary to consider to balance the load and to reduce overhead for sharing information in order to perform the above steps.

It is difficult in general to determine the allocation of subproblems to processors / computers in advance of execution so that the load of each processor /computer balances for the entire course of execution. Thus, we need dynamic allocation of subproblems to processors / computers.

Decomposing a problem into subproblems and allocating them to parallel hardware, in fact, yield distance between subproblems. Distance manifests in communication delay. And, this prevents an object from knowing the current status of other objects. This leads us to the loss of the unique global view of the system, which is an essential characteristic of distributed systems.

We can now define *distributed computing* as computation with more than one activity (or object) where distance, or communication delay, between activities has to be considered. In other words, *parallel computing* can be defined as a sort of distributed computing where distance, or communication delay, between activities can be ignored.

Our recent computational environment consists of existing servers (i.e., objects or agents). The number and services of servers change from a time to time. Consequently, programming style is changing from writing a whole program in an algorithmic/synthetic style to "try to use them" style. That is, a program is written to make maximal use of existing services at each step in the computation.

In such a programming style, it is impossible to know in advance of execution what kinds of services are available at a certain time. In addition, in order to know the available services in the course of executing a program, an object has to use time and computational power. Nevertheless, the result returned to the object might not be correct, since the state of the system could change before it takes the planned action. This is the significance of the notion of *Open Systems* [Hewitt 84]. Although this is an unavoidable drawback from the conventional viewpoint of programming methodology, we should affirmatively utilize this characteristic for efficiency and robustness of a system.

Let us now define *open distributed environment.* An open distributed environment is a computational environment, where services, processing capacity, and connection topology of computing elements are changing from time to time. An open distributed environment is a multi-user environment, where a number of various tasks with different purposes are put in a real-time fashion. An open distributed environment supports mobile users in the following three ways:

- It provides a user with the same environment (for accessing, programming, execution, etc.) regardless of accessing locations.

- It provides a portable computer which can be used in the two modes: *attached mode* (or on-line mode) when it is connected to the network and *detached mode* (or off-line mode) when it is disconnected from the network. In the attached mode it is used as an off-line computer that provides the user with a limited functionality. When it is attached to the network, the work that has been done while being detached from the network will be incorporated into the user's total environment.

- It also provides a mobile computer which is connected to the network even while it is moving.

Open distributed computing can now be defined as computation in an open distributed environment.

We foresee that such an open distributed environment will be obtainable in the next decade by the advancement of computer and communication technologies. Thus, we need to establish a new computing model for it.

3 Object Migration

Object-orientation is a technique of modularization in programming and execution, where modularization is performed in analogy to objects in the real world. In the conventional sense, an object possesses a set of procedures which correspond to computable requests and a local storage to keep its state. Interaction among objects is performed in the form of message passing. An object can communicate with *any* other objects through messages when it knows the addresses (or id's) of the objects and its interface protocol at the execution time. Thus, computation is modeled as objects and message passing among the objects. Object-oriented computing can be understood as the departure from the microscopic view of computing where computation proceeds by executing an algorithm of a procedure to the macroscopic view where computation proceeds as mutual effects among objects.

In contrast to conventional notion of object, a *concurrent object* [Tokoro 88] [Yonezawa 87] possesses a virtual processor in addition to its local storage and a set of procedures. By incorporating a processor in an object, we can eliminate the notion of the *locus of execution* or *allocation of a processor to an object* from object-oriented computing. Thus, we can employ concurrent objects as a simpler unit for concurrent and distributed computing. Hewitt has been advocating this notion as the theory of Actor [Hewitt 73]. Orient84/K [Tokoro 84], ConcurrentSmalltalk [Yokote 86] [Yokote 87], and ABCL [Yonezawa 86] also employ this notion. As a result, those languages have the following very important advantages:

- Objects in the real world exist in parallel and execute in parallel. By using concurrent objects, it becomes very easy and natural to model computation in analogy to the real world.

- The allocation of processors to objects becomes an implementation issue rather than language issue. Therefore, a program becomes independent from the executing system architecture (i.e., shared/distributed memory system, the number of processors in the system, and so forth).

In an open distributed environment, it is important to efficiently utilize existing objects through relocating such existing objects, as well as the new objects which are created for a task, to their optimal locations taking cost and effect of migration into account. Such migration of objects is necessary for so-called load balancing in general, but more specifically, for supporting the mobility of users and achieving robustness of the system against error and faults. That is to say, object migration is a fundamental facility that the environment should provide.

4 Computational Field Model

The optimal location of an object can be determined by calculating cost and expected effect of migration on the network of various computers. However, since the open distributed environment is *open*, the topology of the system is changing, new computing functions are being added, and old computing functions are being replaced from time to time. It is *distributed* so that there is no unique view of the system. Therefore, there is no notion of *optimality*. Only we can do is to try to find a satisfactory sub-optimal location within a reasonable time and cost.

On the other hand, the members of computing elements connected to network increases and the mesh of networks becomes finer. Thus, it becomes reasonable for us to consider the network of computers to form a continuous computational field. As a result, calculation of sub-optimal locations for object migration can be easily done.

The significance of introducing the notion of CFM to open distributed computing is that we can raise the level of abstraction by one level so that we have a clearer view of the system. That is to say, we can ignore details such as the topology and capacity of networks and the kinds and performance of connected computers.

We call such a model *computational field model*, or *CFM* for short. In this model, solving a problem is envisaged as a mutual effect between the computational field and the problem. By using CFM, we can first observe the nature and behavior of a problem macroscopically and find the method to control the problem so as to maximally utilize the given resources.

One important notion in CFM is the *principle of locality*. Effect of any event is local. This is due to the distance between objects in the field, and communication delay in the field. Thus, we define a metric space for the field and the method of computing in the field in the next two subsections.

4.1 Metric Space

In distributed computing, it is very important to keep the communication delay between objects short. Communication delay is a function of geographical distance, communication bandwidth, and other communication overhead. Let us introduce a metric space and define *distance* between objects as geographical distance between the objects. This is reasonable since ultimate communication delay between objects is determined by geographical distance.

In open systems where computation proceeds utilizing existing objects (servers), we should use closer objects if the same services are provided, and we should ask objects to move closer for higher performance. However, an open system is a multiuser system. Thus, an object which is used by n users (or n objects) should be placed at (or migrated to) a location where those users can efficiently use it. In order to decide such an ideal location, it is rational to define *gravitational force* between objects by the frequency and size of information change for communication between the two objects.

If we want to define the locations of objects only by gravitational forces, all the objects get together at one single point. In such a case, although communication delay is minimum, the execution speed of each object is reduced, because all the loads are gathered into one computer. Thus, we should introduce *repulsive force* between two objects, which can be defined by the product of the size of the objects over the distance to the n-th power between the objects.

Assume that at a certain time in the course of computing the objects are all placed at their optimum locations. Also assume that we can know their optimum locations at the next time. It is not always true that migration should take place. This is because we have to pay a cost for migration.

The cost for migration is a function of the distance and the *mass* of the migrating object. The mass can be defined by the size of the object. Repulsive force can now be redefined using the mass of an object. We may also have to take the *inertia* of an object into consideration, which is interpreted as the overhead for migration. According to the above consideration, the location of the objects at the next time should be determined by the cost and the effect of migration.

4.2 MD-based Computing

We now define *MD-based computing* as a method of computing in CFM where computation proceeds by trying to maximally utilize existing objects and determining the optimal location of objects at the next timing taking cost and effect of migration into account. Determination of the optimal locations is performed by using the notions of distance, mass, gravitational force, repulsive force, and inertia. Thus, MD-based computing is a computational method in which each user (or each object) locally gives their best effort to achieve satisfactory allocation of objects. Conflict between users for their satisfactory object allocations should be solved by negotiation between users to find sub-satisfactory object allocations for them. This is viewing computation as transforming a part of the huge computational field into its adequate shape. That is to say, in MD-based computing, solving a problem is considered as mutual effects between the computational field and the problem.

Figure 1 illustrates MD-based computing in CFM. There are two tasks (task A and task B) being executed in a open distributed computational field. If you want to start a new task, you will put the task to the field through your interface computer. The load of the computer becomes very high so that repulsive forces appear among the objects consisting the task. At the same time, gravitational forces appear between these objects and some existing objects on the computational field, as these objects communicate with the existing objects. Thus, the task starts to diffuse, that is, objects consisting the task migrate. Some of the existing objects may move to their new locations, being attracted by the new task. Negotiations among objects may be necessary for finding satisfactory locations. Moreover, if you move, some objects may follow you. Please note that all the existing computations in the computational field form the environment for the new task, and the new task changes the environment.

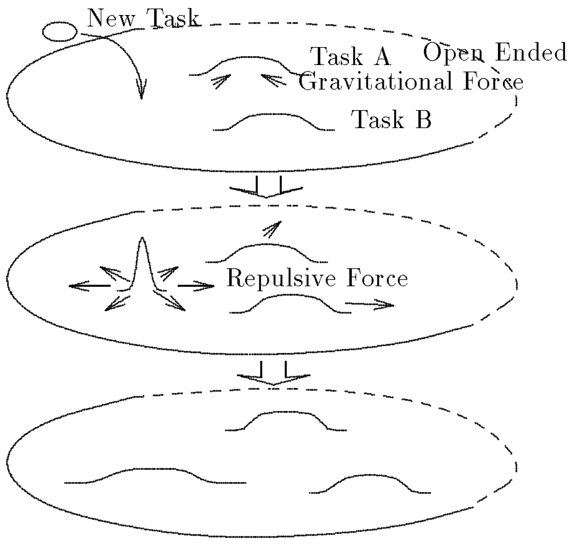

Figure 1: MD-based computing in a computational field

5 Communication in CFM

In most object models, communication is performed in a special way: it uses a special chunk of information called *message* to convey information from an object to another, and the delivery of a message is performed as a hidden, system's function. In such a communication model, there is no state representing a message in transit. We consider, however, representing a message in transit to be essential to describe computation in the computational field model. This is because, since we introduced the notion of distance, communication delay manifests in the model, and consequently we need to represent a message in transit. We also would like to represent that a message actively chases the destination object.

A message can be considered as a special object. Thus, communication can be thought of as moving of an object from one place to another. Since object migration is provided as a primary facility in the computational field model, we utilize this for the delivery of messages.

Based on the above motivation, we propose a model of communication in the computational field model called *assimilation/dissimilation model* or *A/D model* for short. In this model, communication is composed of the following three steps:

1. A sender object assimilates a messenger object at the location of the sender object. The messenger object is a *catabolite* having a message and the receiver's information.
2. The messenger object migrates to the location of the receiver object.
3. The receiver object assimilate the messenger object into it when it arrives.

The first and third steps are synchronous actions, while the second step incurs asynchrony.

We define the following primitives for this communication model:

- *dissimilate* creates a messenger object with a message and the receiver's information in the caller object and dissimilates it.
- *position* returns the position of an object. This is used to locate the receiver. The answer may not show the true (correct) position but is a hint due to the nature of an open distributed environment.
- *migrate* lets itself migrate to the location of a specified object (i.e., the receiver object). We can specify a delivery service class and timeout duration. The object migrates to the receiver object, asking the location of the receiver object by *position*.
- *assimilate* assimilates a messenger object at the same location into itself so that information contained in the messenger object can be accessed. We can specify

timeout duration. It returns Boolean *true* if succeeded. If failed because of some reasons such as time out it returns Boolean *false*.

The main features of this model are that interaction between objects is performed by *assimilation* and *dissimilation*, analogous to anabolism and catabolism in biology, and that an object actively migrates to its destination by itself.

By using this model, various communication can be described. For example, unidirectional asynchronous communication from object *A* to *B* is represented as follows: *A* dissimilates a messenger object *M1* and let *M1* migrate to the position of *B*. *M1* waits for *B*'s assimilation. *B* assimilates the messenger object so that it can access the information contained in the messenger object (see Figure 2). Unidirectional synchronous communication from *A* to *B* is represented by dissimilating *M2* by *B* and letting *M2* migrate back to *A* for acknowledgment after *B* assimilates *M1*, while letting *A* wait for the acknowledgment(see Figure 3).

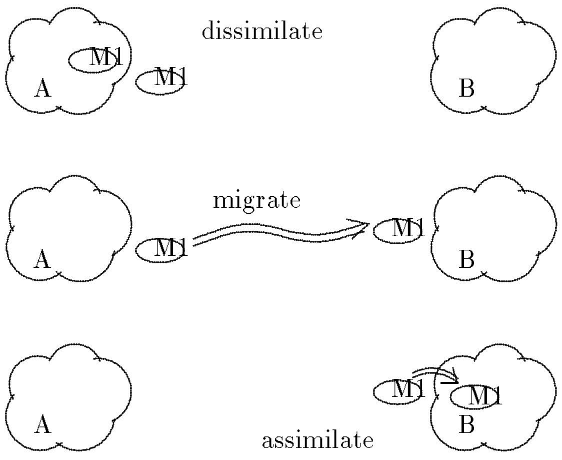

Figure 2: Unidirectional asynchronous communication

6 Conclusion

In this paper, we proposed a new computing model called *computational field model* for solving a problem in an open distributed environment. In this model, we view an open distributed computing environment as a continuous computational field, as opposed to conventional view of a computing system being a discrete computational field. We defined the computational field as a metric space, introducing *mass, distance, gravitational force, repulsive force,* and *inertia*, and proposed the method of *MD-based computing* to give an idea of how to perform computation in the computational field model. We envisage solving a problem as mutual effects between the computational field and the problem.

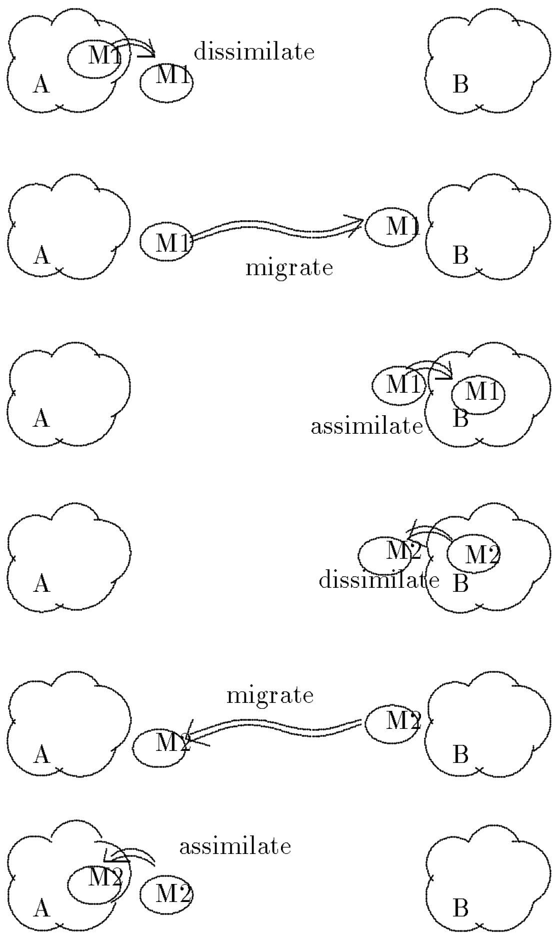

Figure 3: Unidirectional synchronous communication

We also proposed a new communication model called *assimilation/dissimilation model* in which communication in a computational field is represented as the *dissimilation, migration*, and *assimilation* of objects. Thus, computation in an open distributed environment is represented in a unified manner in terms of *objects* and their *migration.* The difference between object migration in MD-based computing and that in communication is as follows: in MD-based computing, objects migrate in a passive fashon by gravitaional and repulsive forces, while in communication, objects migrate in an active fashion in their own right.

Based on the notion of the computational field model and MD-based computing, we have been developing a new operating system called MUSE [Yokote 89] which supports object migration as its primary function. We are also developing object models and programming languages called MUSIC [Watari 90], MyAO [Minohara 89], and Orient90 to describe wide variety of application programs.

Acknowledgment

The author wishes to thank Carl E. Hewitt, Kohei Honda, Shinji Kono, Tatsuo Minohara, Chisato Numaoka, Minoru Uehara, and Yasuhiko Yokote for their critical comments and discussion.

References

[Hewitt 73] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, August 1973.

[Hewitt 84] Carl Hewitt and Peter de Jong. Open Systems. In J. Mylopoulos and J. W. Schmidt M. L. Brodie, editors, *On Conceptual Modeling*, Springer-Verlag, 1984.

[Huberman 88] B. A. Huberman, editor. *The Ecology of Computation.* North Holland, 1988.

[Minohara 89] Tatsuo Minohara and Mario Tokoro. An Object Oriented Database Programming Language Model. In *Advanced Database System Symposium'89*, Information Processing Society of Japan, December 1989.

[Tokoro 84] Mario Tokoro and Yutaka Ishikawa. Object-Oriented Approach to Knowledge Systems. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, November 1984.

[Tokoro 88] Mario Tokoro. Issues in Object-Oriented Concurrent Computing. In *Proceedings of 4th Conference of Japan Society for Software Science and Technology*, September 1988. (in Japanese).

[Watari 90] Shigeru Watari, Ei-ichi Osawa, Yasuaki Honda, and Mike Reeve. *Towards Music: A Description Language for the Muse Object Model.* Technical Report SCSL-TM-90-001, Sony Computer Science Laboratory Inc., February 1990.

[Yokote 86] Yasuhiko Yokote and Mario Tokoro. Design and Implementation of ConcurrentSmalltalk. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1986*, ACM, September–October 1986.

[Yokote 87] Yasuhiko Yokote and Mario Tokoro. Concurrent Programming in ConcurrentSmalltalk. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pp.129–158, The MIT Press, 1987.

[Yokote 89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of European Conference on Object-Oriented Programming*, July 1989. also appeared in SCSL-TR-89-001 of Sony Computer Science Laboratory Inc.

[Yonezawa 86] A. Yonezawa, J-P. Briot, and E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1986*, ACM, September–October 1986.

[Yonezawa 87] Akinori Yonezawa and Mario Tokoro. Object-Oriented Concurrent Programming: An Introduction. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pp.1–7, The MIT Press, 1987.