Introducing Dynamic Data Structure into Mobile Agents

Munehiro Fukuda Naoya Suzuki
Institute of Information Sciences and Electronics
University of Tsukuba, Ibaraki 305-8573, JAPAN

Lubomir F. Bic
Department of Information and Computer Science
University of California, Irvine, CA 92697, U.S.A.

Abstract *One of the performance drawbacks of mobile agents is copying overhead when agents move data between working places or exchange it with other agents. Thus several agent systems use message passing when moving large amounts of data. As an alternative, we provide mobile agents with an abstract data type (ADT) whose instances are created dynamically, are self-referenced, and are exchanged between agents and their current working places through references. This ADT functionality makes each agent capable of not only carrying dynamic data structures over a network but also exchanging data with other agents without any copying operations. We present the ADT implementation in* MESSENGERS *and study the performance using two different applications.*

Keywords: Mobile Agents; Abstract Data Type; Dynamic Data.

1 Introduction

Mobile agents have gained much popularity because of their navigational autonomy that enables them to carry out remote network tasks on behalf of their clients. Since the underlying philosophy of mobile agents is to dispatch the execution to remote servers, rather than to send data, efficient data transfer using mobile agents has not been paid much attention. When moving data to another working place or exchanging it with other agents, the data is usually copied between shared space provided by the node and the agent's private space, which results in excessive data-copying overhead. In addition, unnecessary communication overhead is incurred if agents cannot adjust their size dynamically. Thus, several agent systems, such as Agent TCL and IBM Aglets, provide message-passing facilities as an option to move large amounts of data [3].

From the programming point of view, it is desirable for mobile agents to encapsulate and carry all the data they need, without the assistance of message-passing facilities. Several examples of parallel computations that benefit from data encapsulated within mobile agents were presented in[5]. Our focus is therefore to allow agents to incorporate portions of shared data areas within their private data areas and to carry them along during subsequent migrations without excessive data-copying overhead.

For this purpose, we provide mobile agents with an abstract data type (ADT) whose instances are created dynamically, are self-referenced, and are exchanged between an agent and its current working place through references. This makes each agent capable of constructing dynamic data structures as ADT instances, exchanging them with its working place through references, and thus reducing data-copying and communication overhead when moving them from one place to another. We have implemented this ADT functionality in MESSENGERS-S, our interpretive version of a mobile-agents system[2], and measured performance improvements in matrix multiplication and dynamic queue manipulation applications using four or more workstations.

There are several related works on transferring dynamic data portions: MESSENGERS-C, our compiled version of mobile-agents system[7] makes each agent capable of carrying virtual arrays, whose size can be changed dynamically. However, it does not address the overhead of copying data between an agent and its working place. Some object-oriented mobile agents, such as Telescript[3] and Voyager[6], allow an agent to copy to a new location all its own objects (except the transient parts) to avoid the problem of dangling pointers. Voyager provides a dynamic aggregation that allows an agent to dynamically attach to itself secondary objects, called facets. However, facets cannot be nested or removed, and thus do not allow complicated data structures to be carried. The dangling pointer problem has been well investigated and partially addressed in thread migrations[4], but it is difficult to make two or more migrating threads share heap portions via pointers. Therefore, our proposed ADT functionality is unique in its capability of constructing and exchanging dynamic data structures among agents and their working places through references.

In this paper, we present the functionality, the challenging issues, and the implementation of the proposed ADT in MESSENGERS-S, and demonstrate performance improvement in matrix multiplication and queue manipulation applications.

2 Principles of Operation

2.1 Overview of MESSENGERS

The MESSENGERS system is a mobile-agents system, implemented as a collection of interpreter daemons instantiated on all physical nodes participating in the computation. A daemon's task is to continuously receive mobile agents, termed *Messengers*, arriving from other daemons, interpret their script programs, referred to as Messenger scripts, and send them on to their next destinations as dictated by their behaviors.

The MESSENGERS system involves three levels of networks. The lowest level is the *physical network*, which constitutes the underlying computational nodes. Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process running a MESSENGERS language interpreter. The *logical network* is an application-specific computation network created on top of the daemon network. 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 and it may start creating new logical nodes and links on the current or any other daemon.

Messenger scripts are written in a subset of C, involving arbitrary variables except pointers. They distinguish three types of variables. *Messenger variables* are private to and carried by each Messenger as it propagates through the logical network. *Node variables* are resident in nodes of the logical network and shared by all Messengers visiting the same logical nodes. *Network variables* are predefined at each logical node and give each Messenger access to the network information local to the current node.

A Messenger script has two types of MESSENGERS-unique statements as well as standard C computational statements: (1) Navigational statements *create()*, *delete()*, and *hop()* endow the Messenger with mobility, permitting it to create and destroy logical nodes and/or links, and to move within the logical network; (2) Function invocation statements permit the dynamic loading and invocation of precompiled C functions to be executed in native mode. The *exec()* statement spawns a separate concurrent process for the invoked function, while the *func()* statement invokes the function as part of the current Messenger's behavior and returns its results to it.

The MESSENGERS library functions are called through the *func()* statement or via a C function, to support various controls on Messengers such as their creation, destruction, communication, and synchronization.

With the above features, Messengers are not capable of constructing, carrying, and exchanging dynamic data structures with others. We extend Messengers to endow them with such a capability, which allows them to create, handle, and destroy objects of an abstract data type, termed *Dobj*, as described below.

2.2 Programming with an Abstract Data Type

Dobj is an abstract data type provided to MESSENGERS. Its structure is defined as follows:

```
typedef struct {
  char *data;   // a user-defined data
  int size;     // the data size
  int members;  // # children
  Dobj *owner;  // a pointer to parent
  Dobj *obj[1]; // pointers to children
} Dobj;
```

When instantiated, a *dobj* object[1] allocates a user-defined region of *size* bytes, pointed to by **data*. This region must not include any pointer variables. It also contains an **obj[members]* array to maintain the references to other *dobj*s, instantiated as its children. The children store the reference to their parent in their **owner* field. This parent-children relationship is maintained automatically by the system. The *Dobj* instantiation is available only at runtime. In other words, Messengers can only declare *Dobj* pointers inside their scripts but cannot create instances at compile time. Thus new objects must be associated with their pointers upon their instantiations. All operations on *Dobj* are achieved through the library functions listed in Figure 1. (The notation "[arguments]" means that these arguments are only necessary when the function is called from a precompiled C function but not from the script level.)

If a Messenger associates a *dobj* with a pointer declared as a *messenger variable*, it carries with it this *dobj* and all its children during any subsequent network migration. However, if the Messenger associates this *dobj* with a pointer declared as a *node variable*, then the *dobj* remains in the current node. Therefore, a Messenger can simply carry a whole data structure to another node by moving a reference to this structure from a node variable to a Messenger variable before its migration.

Figure 2 gives a simple illustration of the *Dobj* features. Assuming that two Messengers are migrating from different nodes to their parent in a binary tree, their script lets one of them merge the other's *Dobj* queue into its own queue. More specifically, the first Messenger arriving at this parent node moves its *Dobj* queue from its *q1* messenger variable to the *q2* node variable (line 9) and terminates itself (line 10). The second Messenger detects a queue in *q2* and thus calls *the merge()* function (line 12), which finds *q1*'s tail (line 18) and links the *q2* to it (line 19). Thereafter, if the current node is not the root, this Messenger carries all the *dobj*s in its queue up to the next *parent* node (line 14) and repeats the same merge sequence. Otherwise, it skips the navigation (line 13) and stores the final result in the root's *q2* (line 9).

[1] We use *dobj* when referring to an instance of the proposed abstract data type *Dobj*

*dojbCreate(Dobj [*]*dobj_p, [Dobj *owner], int size, int members):* Create in *dobj_p* a new object with a user-define region in *size* and *members* of children.

*dobjDelete(Dobj [*]*dobj_p):* Delete the object pointed by *dobj_p* and its successors.

*dobjMove(Dobj [*]*src, Dobj [*]*dest, [Dobj *new_owner]):* Move the object from *src* to *dest*.

*dobjCopy(Dobj [*]*src, Dobj [*]*dest, [Dobj *new_owner]):* Copy the object from *src* to *dest*.

Figure 1: *Dobj* library functions

3 Implementation

In this section, we will discuss the *Dobj* implementation issues, the actual maintenance mechanism, and its pros and cons.

```
node{
 Dobj* q2; // not empty at a leaf
 int root; // 1 at the root, else 0
}

messenger {
  Dobj* q1;
  while (1) {
    if (q2) {
      func(name="dobjMove"; in=q1,q2)
      exit;
    } else {
      func(name="merge"; in=q1,q2);
      if (root) continue;
      hop(node="parent");
} } }

merge(Dobj** q1, Dobj** q2) {
  Dobj* d;
  for(d=*q1;d->obj[0]!=0;d=d->obj[0]);
  dobjMove(q2, &(o->obj[0]), o);
}
```

Figure 2: A code example using *Dobj*

3.1 Implementation Issues

Dangling Pointer Problem: This is the most serious problem we need to be concerned with when introducing dynamic objects into mobile agents. This is because the use of pointers generally depends on the semantics of the user program. The problem is also related to garbage collection, which must free all dynamic objects no longer referenced by any pointers. During migration, a mobile agent needs to detect the pointers still in use, salvage all dynamic objects reachable from those pointers, and free all other objects.

The *Dobj* specification addresses such pointer problems as follows: permitting pointers only in *Dobj*'s *obj[]* array and *owner* field makes it easy for the MESSENGERS daemon to salvage and reconstruct *Dobj* data structures upon a migration; the use of *Dobj* library functions restricts the use of pointers to only legal operations; finally, all garbage collections can be implemented inside these functions.

Reference Replication Problem: Replicated references to the same object may allow an agent and its working place to share the object. This blurs the agent-place boundary, and brings up a few semantic ambiguities in moving objects: the agent must decide whether to carry or leave such a shared object during a migration; if it is to be carried, then the agent must decide whether to replicate it or simply remove it from the source place; the agent must also decide how to deal with an object that already exists on the destination node.

To avoid these problems, we introduce the idea of ownership into *Dobj*. Each *dobj* must be owned by another *dobj*, a Messenger, or a node. Only the owner can store its child *dobj*s' references in its *obj[]* array. If the owner is a Messenger or a node, it maintains those references in its *Dobj* pointers, and each child *dobj* has its owner's reference in its *owner* field. Therefore, trees are represented as ownership relationships (i.e., dynamic data structure) formed using *dobj*s. Moving or copying a *dobj* to another owner means moving or copying a subtree starting from this *dobj*. Although there are now no replicated references between *dobj*s, Messengers, or nodes, and thus no more ambiguous Messenger-node boundaries, temporary reference replications are still necessary to change data structures, as discussed below.

Temporary references: A temporary pointer must be used to search for a given *dobj* in a data structure and to modify its data. However, it is an unnecessary burden to keep the content of a temporary pointer whenever it is carried with a Messenger. Furthermore, such a pointer becomes meaningless when it refers to a certain *dobj* in a source node and the corresponding *dobj* is not found in a destination.

This dilemma can be resolved by allowing precompiled C functions to declare and use *Dobj* pointers as their local variables. Therefore, searches for and data modifications of *dobj*s are permissible inside of C functions but not on a script level, while the basic operations such as creation, deletion, move, and copy are available on both levels. Since Messengers migrate with script-level navigational statements, such local *Dobj* pointers are discarded before migrations, and thus need not be preserved.

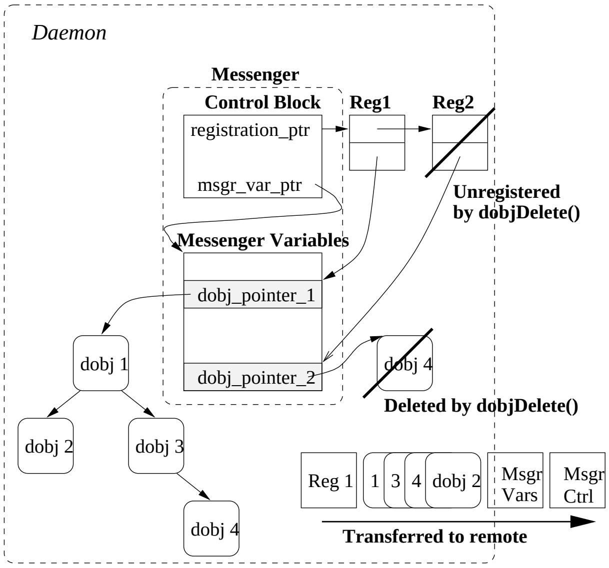


Figure 3: *Dobj* maintenance in MESSENGERS daemon

3.2 *Dobj* Maintenance Mechanism

Dobj maintenance is achieved by the four library functions introduced in Section 2.2 and the MESSENGERS daemon.

The library functions not only provide basic operations on *dobj*s, but also maintain the inter-*dobj*s ownership and perform garbage collection of *dobj*s deleted, replaced or overwritten. When a *dobj* pointer in a Messenger or a node is passed to a library function to associate a new instance with it, the function registers the pointer's position and attaches the registration to the control block of the Messenger or the node. For instance in Figure 3, *dobj 1* is associated with *dobj pointer 1*, whose position is then registered in *Reg 1*. The dissociation of an instance from a *dobj* pointer performs the reverse operations. Again in Figure 3, *dobjDelete()* deletes *dobj 4*, nullifies *dobj pointer 2*, and unregisters its position from *Reg 2*.

The MESSENGERS daemon salvages all *dobj*s traversable through the ownership chain starting from each of *Dobj* pointers in a Messenger that is leaving for another daemon. It only needs to focus on the pointers in use, all of whose registrations have been attached to this Messenger's control block. This *dobj*s salvation, transfer, and reconstruction are achieved as follows: (1) since the ownership relationship forms a tree, its *dobj*s are recursively traversed from the children to their parent, (in the order of *dobjs 2-4-3-1* in Figure 3); (2) those *dobj*s and the corresponding pointer registrations are sent with the Messenger in FIFO order to the destination; (3) the destination daemon first receives the child *dobj*s on its internal stack, pops them when receiving their parent, reconstructs their ownership relationship, and again keeps this parent on the stack as a child; (4) it repeats the sequences 1-3 until it receives the root *dobj* and the pointer registration; finally (5) the daemon locates the *dobj* pointer in the Messenger and updates its content with the root reference.

3.3 Pros and Cons

As briefly mentioned in Section 1, the *Dobj* concept has three advantages: (1) no additional copying operations required when moving data portions to another *node*; (2) no excessive static allocation of space for a Messenger to carry data of unpredictable size; and (3) dynamic data structures made available in a Messenger.

However, we must also consider the possible drawbacks of this approach. First, instead of the costs of copying, the daemon must pay the price of saving and reconstructing *dobj*s during migrations. It must also deal with multiple objects, whose actual data portion is relatively small in comparison with their total size. Second, *Dobj* predefines several variables, such as *size*, *owner*, and *obj[]*, in addition to the actual data, thus resulting in increasing the total memory requirements. Third, if a Messenger migrates before completing any *dobj* modification, it must search the *dobj*s tree every time it invokes a C function. This is because all temporary pointers in C become invalid upon a function return, followed by migration.

It is important to find the necessary conditions under which the use of *Dobj*s becomes advantageous. For this purpose, we measured the performance of matrix multiplication and queue manipulation applications by varying the number of *dobj*s and their data size.

4 Performance

We conducted performance evaluations on a network of ten 170 MHz SPARCstations (96MB memory each) connected by a 100Mbps collision-free Hitachi switch.

4.1 Matrix Multiplication

We used a block-oriented algorithm [1], which partitions two input matrices A and B, and the resulting matrix C into $s \times s$ rectangular blocks. Assuming that coordinates $[i, j]$ address each block, each triplet of $A[i, j]$, $B[i, j]$, and $C[i, j]$ is assigned to processor $[i, j]$. Then, the algorithm repeats the following three distinct phases s times:

1. *Distribution:* During each iteration k, all blocks $A[i, j]$ satisfying $j = (i + k)\ mod\ s$ are multi-cast to all processors in the same row.

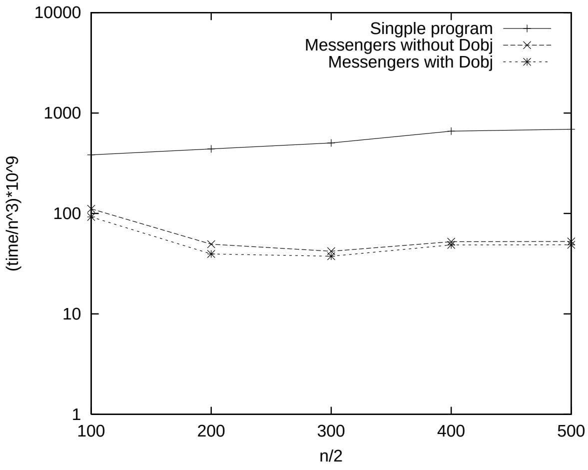
2 *Block multiplication:* For the local blocks of A, B, and C, each processor multiples A with B and adds the result to C.

3 *Rotation of B:* Each processor moves its block B to its neighbor in the same column using a circular shift.

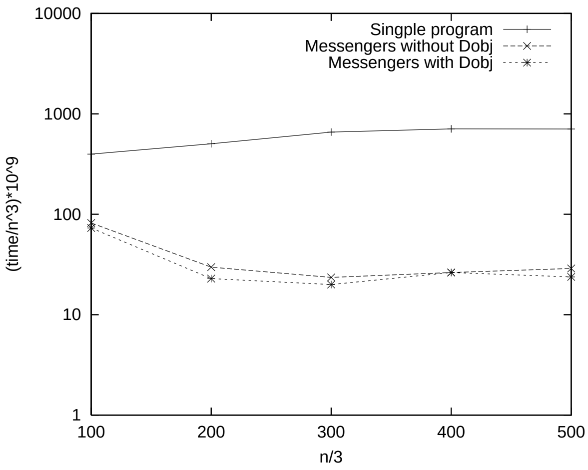
We coded three different versions of this program: a block-oriented sequential program, MESSENGERS in the original specification, and the MESSENGERS using *Dobj*. Figure 4 shows the results when using 2×2 and 3×3 processors. The results show that the speedup by the two versions of MESSENGERS is almost linear over the sequential program for large blocks. Furthermore, the *Dobj* version achieved more than 20% speedup over the non-*Dobj* version for smaller block sizes. This is because the *Dobj* version does not copy the blocks between a node and a Messenger, so that the blocks were able to fit in the cache memory.

4.2 Queue Manipulation

To assess *dobj*s' manipulation overhead, we generated a queue of *dobj*s, each containing



(a)



(b)

Figure 4: Matrix multiplication results (a) on 4 processors (b) on 9 processors

a random number and dummy data, and reordered them using the merge-sort algorithm. We first divided the queue recursively until each sub-queue was assigned to a different processor. We sorted it locally and merged each pair of sub-queues recursively, as shown in the code of Figure 2.

We coded two different MESSENGERS implementations: one with and the other without the use of *Dobj*. The latter simply defines the queue as a static array. Figure 5 shows the relationship between the number of *dobjs* and their data sizes that are necessary to outperform the

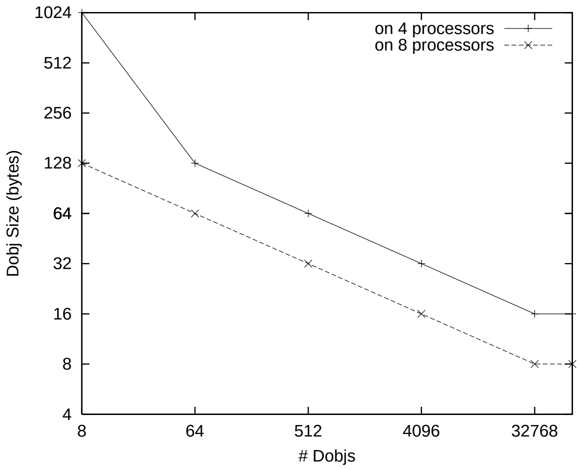


Figure 5: Threshold between MESSENGERS with and without the use of *Dobj* in queue manipulation

MESSENGERS version without *Dobj*. The two different curves correspond to four and eight processors. The area above the curves represents the conditions under which using *Dobj* is advantageous. At one extreme, the queue manipulation requires more than 32768 of very small objects. At the other other extreme, the data size of each *dobj* must be at least 1 Kbyte (on four processors) or at least 128 bytes (on eight processors) to make the maintenance overhead of *dobj*s smaller than the communication and data-copying overhead in the original MESSENGERS version.

5 Conclusions

We introduced the abstract data type *Dobj* into MESSENGERS, and made its agents capable of exchanging dynamic *Dobj* instances with their working places via references. The instances are all maintained and garbage-collected by library functions, with the help of the underlying daemon processes. This relieves the application programmer from direct pointer manipulations. While reducing the data-copying and communication overhead, the use of *Dobj* introduces other forms of overhead, notably, the cost of salvaging *dobj* instances and transferring additional predefined variables during migrations. However, our performance results demonstrated that *dobj*s are beneficial in matrix multiplication due to a more efficient utilization of cache memory. In the queue manipulation, *dobj*s are beneficial with large numbers of queue elements or large element sizes.

Currently, the *Dobj* library includes only four operations: creation, destruction, move and copy. More sophisticated functions are needed to allow the construction of dynamic structures with less coding effort. Such extensions will be the focus of our future work.

References

[1] A. Geist, et al. *PVM: Parallel Virtual machine. A User's Guide and Tutorial for Networked parallel Computing.* MIT Press, Cambridge, MA, 1994.

[2] L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, Vol.29(No.8):55–61, Aug. 1996.

[3] W.R. Cockayne and M. Zyda. *Mobile Agents.* Manning, Greenwich, CT, 1998.

[4] D. Cronk, M. Haines, and P. Mehrotra. Thread migration in the presnse of pointers. In *Proc. of the 30th Hawaii Int'l Conf. on Systems Sciences - HICCS'97*, pages 292–298, January 1997.

[5] M. Fukuda, L.F. Bic, M.B. Dillencourt, and J.M. Cahill. Messages versus messengers in distributed programming. *Journal of Parallel and Distributed Computing*, to appear in 1999.

[6] Object Space Inc. Voyager, core technology 2.0. User guide, Object Space Inc., Dallas, Texas, 1997.

[7] C. Wicke, L.F. Bic, M.B. Dillencourt, and M. Fukuda. Automatic state capture of self-migrating computations in MESSENGERS. In *Proc. of the 2nd Int'l Workshop on Mobile Agents - MA'98*, pages 68–79, Stuttgart, September 1998.