# Dynamic Objects to Support Evolution of Mobile Agents

Yuichiro Tanaka Munehiro Fukuda Naoya Suzuki

Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan
yuichiro@padc.mmpc.is.tsukuba.ac.jp, {fukuda, nas}@is.tsukuba.ac.jp

## Abstract

*The self-modifiability or evolvability is necessary for mobile agents to navigate over network changing unpredictably and continuously. We propose an approach to the evolution of mobile agents based on dynamic objects capable of being attached to, interpreted by, carried with, exchanged among, and structured by the agents. We have implemented such objects on the* MESSENGERS *mobile-agents system, and have coded a distributed task coordination as an application example. This paper presents the implementation and benefits of our approach to the agent evolution.*

## 1 Introduction

Mobile agents have been recognized as an attractive technology to achieve network tasks remotely on behalf of their users [2]. It is, however, impossible to program them as predicting all changes in WAN before their actual deployment to the network. The self-modifiability or evolvability is a promising capability to make the navigation and behavior of mobile agents adaptable to active network environment.

We provide mobile agents with such evolvability by allowing them to dynamically load new objects to themselves. While dynamic linking itself is a familiar technique [3], our objects are unique in a total conjunction of the following capabilities: (1) they are interpretive; (2) they are carried with and detached from a mobile agent; (3) they are exchangeable among mobile agents; (4) they can maintain their live state; and (5) they can construct data structures.

We have extended the feature of MESSENGERS mobile-agents system, where the agents can evolve themselves by utilizing the capabilities of our proposed dynamic objects. We have also coded a distributed task coordination over WAN as an evolvable application example. This paper presents the implementation of agent evolvability in MESSENGERS and demonstrates the benefits of our approach through the task coordination program.

## 2 MESSENGERS Paradigm

MESSENGERS was developed at University of California, Irvine [1]. It consists of a set of Unix daemon processes, each interpreting mobile agents at a different workstation and exchanging them with other daemons. MESSENGERS mobile agents, termed Messengers, dynamically construct and navigate themselves over logical networks.

MESSENGERS are written in a subset of C, involving arbitrary variables except pointers. *Messenger variables* are private to and carried with 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 node. *Network variables* are predefined at each logical node and provide each Messenger with the network information of the current node.

A Messenger script has two types of messengers-unique statements: (1)Navigational statements create(), hop(), and delete() endow the Messenger with mobility, permitting it to create, to move within, and to destroy logical networks; (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.

## 3 Agent Evolvability

### 3.1 Library for Agent Evolution

We extend this original feature to endow Messengers with evolvability that allows them to incorporate and execute new interpretive code in an instance of an abstract data type, termed *Dobj*, as described below.

The instantiation of *Dobj* is available only at run time. Messengers are allowed to use pointers only when accessing new *dobj* instances.[1] They are created dynamically, are self-referenced, are carried with a Messenger and are exchanged between Messengers via pointers. All operations on *dobj*s are achieved through the library functions listed below. (The notation "[arguments]" means that these arguments are necessary only when the function is called from a precompiled C function but from the script level.)

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

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

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

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

If a Messenger associates a *dobj* with a pointer declared as a messenger variable, it serializes and 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 such pointer operations. Further details are given in [4].

A Messenger can evolve itself with *dobj*s by repeating the following sequence of operations: (1) know the size of the new Messenger code best appropriate to achieve the final goal, (2) instantiate a *dobj* to store this code, (3) load the code into the *dobj*, (4) save the current state in another *dobj* and start interpreting the new code, (5) receive a return value and resume the previous state, and (6) release all the *dobj*s if necessary. While interpreting the *dobj*, the Messenger may migrate over network. Therefore, even if it repeats loading code with the same file name, the Messenger may relay a different code at each node it has visited. We have realized this cycle of evolution, using the following library functions:

*M_fsize(filename, filesize):*
Return in *filesize* the file size of *filename*.

*M_fread(dobj, filename):*
Read the *filename* file contents in *dobj*.

*M_fwrite(dobj, filename):*
Write back the *dobj* contents into the *filename* file.

*M_eval(dobj, [arguments], return_value):*
Interpret *dobj* as a code. Pass *arguments* to and receive *return_value* from *dobj*.

*M_return(return_value):*
Return *return_value* to the main code.

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

Figure 1 shows a code example using these functions. The main code detects the file *size* of "new_code" to be incorporated (line 3), creates a *dobj* of *size* bytes (line 4), and loads the contents of "new_code" in *dobj* (line 5). It then evaluates the *dobj* contents with *$arg1* (line 6) and prints out the *val* content received from the *M_eval* (line 7). The "new_code", when interpreted, simply moves the calling Messenger to the *$arg1* node and returns the node name.

```
Main code:
 (1) int size, dobj;
 (2) char $arg1[64], retval[64];
 (3) func(name="M_size"; in="new_code"; out=size);
 (4) func(name="M_dobjCreate"; in=dobj; size, 1);
 (5) func(name="M_fread"; in=dobj; "bytecode");
 (6) func(name="M_eval"; in=dobj, $arg1; out=val);
 (7) func(name="M_print_str"; in=val);

New Code (to be incorporate):
 (8) char $arg1[64];
 (9) hop(node=$arg1);
(10) func(name="M_return"; out=$node);
```

**Figure 1. An example of simple evolution**

The main challenge lies in the *M_eval* function that must still permit the calling Messenger to migrate over network. To guarantee such migration, we have further modified the MESSENGERS daemon as follows.

### 3.2 Daemon's Behavior

For a *dobj* code evaluation, the MESSENGERS daemon performs additional operations in the following three phases: (1) the *M_eval* invocation, (2) the migration in *M_eval*, and (3) the *M_eval* return, as illustrated in Figure 2.

1. *M_eval* invocation

   When the daemon loads a new Messenger binary code in a given *dobj*, it also reserves a part of the *dobj* for new *messenger variables* accessed by this code. Upon an *M_eval* invocation, the daemon adds another *dobj* that behaves as a stack to save the current state of the calling Messenger, and thereafter interprets the *dobj* code until it encounters any navigational statements.

2. Migration in *M_eval*

   As described in Section 3.1, the daemon guarantees that a Messenger serializes and carries with it all *dobj*s reachable from its first-level *messenger variables*. Therefore, the daemon temporarily resumes the migrating Messenger's original state before its departure, so that all the *dobj*s are simply visible as a reachable graph from the first-level *messenger variables*, thus serialized and carried together to the destination.

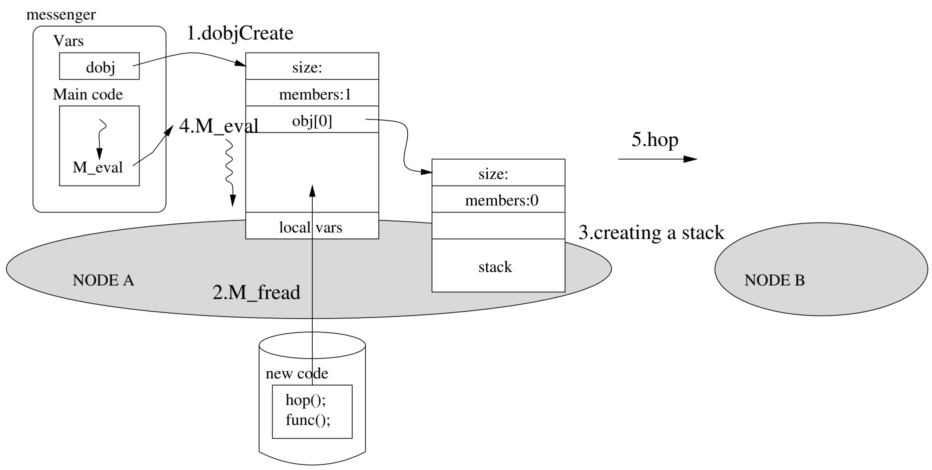


**Figure 2. Daemon's behavior**

3. *M_eval* return

   Upon encountering an *M_return* function, the daemon resumes the previous state of the calling Messenger and discards only the *dobj* that has worked as a stack. The evaluated *dobj* itself remains linked to the Messenger until it is explicitly deleted or moved to a node.

The key idea in the *M_eval* implementation is to prepare another *dobj* as a stack while not distinguishing it from the other *dobj*s at migration time. This prevents the daemon from introducing a brand-new stack mechanism only for evolution.

Even after interpreted through *M_eval*, *dobj*s maintain the latest updates on their variables unless it is deleted by *M_dobjDelete*. This persistency resembles that of static variables in a dynamically linked C library or static class variables in Java. However, *dobj*s are unique in being carried with and exchanged between agents. In summary, *dobj*s are interpreted, mobile, exchangeable, persistent, and structured, which all satisfy the capabilities we have proposed.

# 4 Task Coordination by Evolvable Agents

Using a distributed task coordination over WAN, we demonstrate how evolvable Messengers tolerate unpredictable changes in WAN and achieve their given tasks. Figure 3 shows the script of such an evolvable Messenger, called *CoordAg*. It coordinates a user job execution on a remote server and reports its results to the client user.

More precisely, *CoordAg* receives from a user a set of a job name, arguments given to the job, and resource requests, each maintained into *job*, *args*, and *rsl* Messenger variables (line 5). *CoordAg* records its home address (line 6), migrates to a specific server, (i.e., "yeehaw" in line 7), and obtains a copy of *svExec dobj* from the server (line 8). This *dobj* includes another Messenger script, *Exec* to execute a user job using reserved server resource (lines 13-31). *CoordAg* simply evaluates this *dobj* and returns to the client with its results (lines 9-12).

The *Exec* script actually takes care of a user job execution. It calls the *reserve_rsc* function to hold server resource as specified in the *rsl* variable (line 18). If the resource is reserved successfully, it launches an independent process to carry out a given user job and saves its computation results in the *result dobj* (lines 19-24). Otherwise, *Exec* forwards the calling Messenger, (i.e., *CoordAg*) to another server, "howdy" where it retries the same sequence of job execution (lines 25-29).

Needless to say, the *CoordAg* Messenger could be programmed in a self-contained manner. In other words, it may not farm out the details of resource reservation and job execution to *Exec*. However, what if the agent fails in resource reservation? It should be coded to carry a complete itinerary of all possible sites with it and migrate one site to another until the required resource is obtained. This is not only a problem. The agent must also avoid faulty, busy, and out-of-service hosts. Furthermore, anticipating inevitable faults, it needs to check-point and recover on-going distributed computation. Such problems in WAN are numerous, unpredictable, and thus extremely difficult to be addressed by the static behavior of mobile agents.

By dynamically linking *Exec* to it, the *CoordAg* Messenger can concentrate on the main sequence of task coordination: job dispatch, resource reservation, job execution, and report to a client. The incorporated *Exec* script actually enables the *CoordAg* Messenger to reroute or forward itself to a well-functioning server when detecting exceptions and faults including a resource scarcity problem. In addition to agent rerouting and rescheduling, this coding style of *Co-*

```
CoordAg:
 char $arg1[64], $arg2[64], $arg3[64];
 char job[64], args[64], rsl[64], server;
 dobj myExec, myResult; double home; int id;
 job=$arg1; args=$arg2; rsl=$arg3;
 home=$address;
 hop(node=0; physical="yeehaw");
 func(name="M_dobjCopy";in=svExec;out=myExec);
 func(name="M_eval";
      in=myExec, job, args, rsl; out=result);
 hop(node=home);
 func(name="M_print_str"; in=result);

Exec:
 char $arg1[64], $arg2[64], $arg3[64];
 char job[64], args[64], rsl[64];
 dobj myExec, result; int permit;
 job=$arg1; args=$arg2; rsl=$arg3;
 func(name="reserve_rsc"; in=rsl; out=permit);
 if (permit) {
   job = job + ">& file";
   exec(file="job"; in=arg2);
   func(name="M_size"; in="file"; out=size);
   func(name="M_dobjCreate";in=result,size,0);
   func(name="M_fread"; in=result, "file");
 } else {
   hop(node=0; physical="howdy";
   func(name="M_dobjCopy";in=svExec;out=myExec);
   func(name="M_eval";
        in=myExec, job, args, rsl; out=result);
 }
 func(name="M_return"; in=result);
```

**Figure 3. Distributed documentation**

*ordAg* and *Exec* also brings the following three advantages:

1. Check-pointing:

   When a *CoordAg* Messenger intends to visit an error-prone server, the current server temporarily saves a copy of *CoordAg* in case it needs to resume it upon an abnormal termination. The *CoordAg* Messenger also automatically includes a new *dobj* to spawn a background Messenger that returns to the current server to cancel the temporary copy if a job execution completes successfully.

2. Load balancing

   *CoordAg* should be forwarded to an idle server during its user job execution, when the current server is unexpectedly overloaded. In such a case, a server embeds this *CoordAg* Messenger into a new *dobj* and dispatches a carrier Messenger to move it to an idle server, where the *CoordAg* Messenger is resumed.

3. Runtime software update

   *Exec* itself at each server can be replaced with a newer version brought by a *CoordAg* or another carrier Messenger. When that happens, the server may further forward such a carrier Messenger to its neighbors. This can be extended to automatic software update or distribution in a heartbeat fashion.

## 5 Conclusion

We have realized the evolution of mobile agents by allowing them to dynamically attach interpretive objects to themselves at run time. The objects are not only additional codes but also the ones capable of being carried with, exchanged among, structured by, and detached from mobile agents. We have demonstrated that such objects assist mobile agents in rescheduling their itinerary, check-pointing their on-going computation, improving per-site load balancing, and updating server functionality.

As well as better functionality, our approach has potential to improve the performance of mobile agents by permitting them to discard objects unnecessary for their future navigation. One of our next works is to evaluate this performance improvement. Another future work is to justify the applicability of our proposed agent evolvability. For this purpose, we are planning to implement our idea on Java-based mobile agents, with the support of dynamic Java classes [6]. Furthermore, we are constructing a practical environment for distributed task coordination using evolvable agents [5].

## References


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

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

[3] M. Franz. Dynamic linking of software components. *IEEE Computer*, Vol.30(No.3):74–81, March 1997.

[4] M. Fukuda, N. Suzuki, and L. F. Bic. Introducing dynamic data structure to mobile agents. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'99*, pages 1854–1860, Las Vegas, NV, 1999.

[5] M. Fukuda, Y. Tanaka, L. M. Campos, and S. Kobayashi. Inter-cluster job coordination using mobile agents. In *3rd Annual International Workshop on Active Middleware Services*, San Francisco, CA, to appear on August 6th 2001.

[6] S. Malabarba, R. Pandey, J. Gragg, and E. Barr. Runtime support for dynamic java classes. http://pdclab.cs.ucdavis.edu, Parallel and Distributed Computing Laboratory, CS Dept., UC Davis, Davis, CA, 1999.