ICOT Technical Report: TR-283

TR-283

Remote Object Accessing Mechanism in SIMPOS —Its Design and Application—

by
K. Yoshida

June, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Remote Object Accessing Mechanism in SIMPOS – Its Design and Application –

Kaoru Yoshida
Institute for New Generation Computer Technology (ICOT)

Address: Mita-Kokusai Bldg. 21F., 1-4-28 Mita, Minato-ku, Tokyo 108 JAPAN,
Tel: 03(456)3193
Telex: ICOT J32964
CSNET: yoshida%icot.jp@relay.cs.net
ARPA: yoshida%icot.uucp@eddie.mit.edu
UUCP: ihnp4!kddlab!icot!yoshida

Abstract

This paper describes the implementation of ROAM in SIMPOS.

Remote Object Accessing Mechanism (ROAM) is a general interface to realize network-transparent access to remote objects in the way of remote method call. Remote method calls are based on synchronous communication between proxy and original objects respectively residing on client and server processes and allowed to be nested.

The design and principles of ROAM are detailed and the development of a global file system is shown as its practical large application. Several problems on the design of ROAM are also discussed.

1 Introduction

SIMPOS is a programming and operating system being developed for the personal sequential inference machine PSI [Yokoi84,Nakajima86]. All of the modules constituting SIMPOS, including the kernel layer such as device handlers, are written in an object-oriented logic programming language ESP [Chikayama84]. All the characteristic features of ESP reflect directly to SIMPOS.

An object in ESP is a set of slots (states) and methods (procedures) to be defined in a class. From logic programming point of view, a method is an external predicate to be inferenced with its parameters unified. SIMPOS is an open system without any boundary between the system and the user. With multiple class inheritance and demon method combination, users can define any complicated class on the system.

As for the hardware environment, about one hundred PSI machines and some other machines are being connected together via the local area network 10 Mb Ethernet and the global network DDX.

One of the objectives of building up a network system is to allow users to share resources and communicate with each other over the network and to provide a cooperative environment for their developing applications. Ultimately desired is a network-transparent space which seems to be identical no matter where the user has logged in or the resources are stored. Aiming at the realization of such an environment, several distributed operating systems and description languages have been built up or proposed [Walker83,Cheriton83,Rashid86,Andrew86,Black85,Spur86,Liskov84].

The level to offer the network transparency influences the development of applications and it should be as close as possible to the basic execution level. In procedural systems, a variety of remote procedure call mechanisms which provide the function of making procedure calls over the network have been implemented [Nelson81,Welch86,Satya86,Cheriton84].

In SIMPOS, the basic execution mechanism is the method call to objects.

Remote Object Accessing Mechanism (ROAM) is a basic mechanism to make it possible to make method calls to remote objects over the network. With this mechanism, users can access remote objects in the same way as local objects. There are a couple of distributed object-oriented systems and a language which have been proposed along this purpose [Decouchant86,Anderson86,Black86].

This paper describes the implementation of ROAM in SIMPOS. Firstly, the basic design principles of ROAM are described. As a practical large application of ROAM, the development of a global file system is shown. Finally, ROAM is compared with other related works forcusing on the object migration and global garbage collection which would be substantial problems in designing distributed object-oriented systems.

2 Remote Object Accessing Mechanism (ROAM)

Remote Object Accessing Mechanism (ROAM) is a general interface between applications and communication systems to provide the function of making method calls to remote objects over the network.

The general overview of ROAM is shown in Figure 1. This chapter describes the design principles of ROAM which are categorized as follows:

- object and process reflection
- line management
- global object management
- message management and interpretation
- remote method call
- nested-call control
- object migration
- customization

2.1 Object and Process Reflection

In SIMPOS, the concept of object is separated from that of process. General objects are to encapsulate the data representation and the operation on it, and the execution environment is held by process objects.

When accessing a remote object, the relationship between the object and the process accessing it is directly reflected in the remote node machine.

Original Object and Proxy Object

An object which is originally created is called *original object*. When an original object is accessed from a remote process, an object which works as its agent, called *proxy object*, is created in the remote process.

The role of the proxy object is to pass all the requests of making method calls to its corresponding original object. Basically, it is the original object that should maintain the internal states for its own original functions which are defined in its class. All the internal states that the proxy object concerns, are the ones necessary for passing the requests, which are independent of the original functions.

The original object and its proxy objects are of the same class and have no difference in their object structures except that the slot `object_category` is set the value `original` or `proxy` respectively. Such an object that is accessible over the network is called *global object*.

Client Process and Server Process

A process which requests accessing an object is called *client process*. If a target object is remote, a process which works as a substitute of the client process, called *server process*, is created at the remote node,

For one client process, one server process is created per remote node.

There is one process prepared at each node, called *controller process*, which controls the initiation and termination of the server processes at the node. When a client process tries to access a particular remote node for the first time, it requests the controller process at the remote node to create a server process for it.

Remote method calls are synchronous as local method calls are. A client process sends a request message containing a method information and waits for its corresponding reply message containing its result.

When a client process terminates, it requests the related controller processes to kill its server processes.

2.2 Line Management

Each process holds a set of virtual line objects associated with their node name. Since a server process might create its own server process at another remote node, any process can be both an server and an client at the same time. Of the virtual line objects a process holds, one is connected from its client process and the others to its server processes. The virtual line object is an external stream with the synchronization function to enable two processes in different nodes to communicate each other. It is provided by the underlying communication system.

Thus, each virtual line denotes the existence of a server process at the node. Every time a remote method call is made, the calling process inquires to itself whether it holds the corresponding virtual line or not, that is, whether its server process has been created at the remote node or not.

2.3 Global Object Management

An object can be referred from remote nodes either as a arget object or as a parameter of a remote method call. This external reference is called *object exporting* at the resident node and *object importing* at the remote node respectively.

In ESP, there are two kinds of objects: class objects and instance objects. They are managed in the different ways.

2.3.1 Classes

Identification

There is no concept of meta class in ESP. Instead, the library subsystem in SIMPOS manages all of the class objects. The concept of package is supported to realize a multiple class name space. A package is a set of classes. Any class is uniquely identified by a pair of a package name and a class name, called *class complete name*, inside a node. But anothor class might be chosen with the same class complete name at another node.

ROAM regards class objects with the same class complete name as to be identical and leaves it to users to set up the library.

Exporting and Importing

When a class is exported, its class complete name is sent out as its information.

When a class information is received, a class object with the same class complete name is retrieved out of the library at the right node.

2.3.2 Instances

Identification

In SIMPOS, instance objects are not associated with object identifiers that would be implicitly assigned at the creation time. In ROAM, an instance object is assigned *a global object identifier (GOID)* when it is exported and transfers the GOID to its proxy objects. Each GOID consists of the resident node name, exporting time and process number so that it can be uniquely identified over the network.

Each process holds a pair of *an export table* and *an import table*: the former keeps exported objects and the latter imported ones. Each of the exported and imported objects is associated with its GOID.

Exporting and Importing

When an instance object is exported for the first time, a GOID is generated and the object is entered into the export table of its own process. The instance information to be sent out contains its class complete name and GOID.

When an instance information with an unknown GOID is imported, a proxy instance object is created according to the given class complete name and is entered into the import table.

2.4 Message Management

One of the most important functions of ROAM is the message management, especially message interpretation. The interpretation is to pack either method information or result information into a message and to unpack it.

2.4.1 Request and Reply

There are two kinds of messages: one is *the request message* for making a method call and the other *the reply message* for returning its result. Each message is of variable length and free format and consists of the following three fields:

Control field contains the message control information of message type, either request or reply, and global message identifier (GMID) which will be mentioned later.

Standard field contains either the method information of the method name, target object and parameters for the request message or the result information of the execution status and parameters for the reply message.

Extra field is offered as a user-definable field for the customization according to applications, while the above two fields are under the system's control.

2.4.2 Message and Packet

Internally, each message is represented by a set of unit packets. Packets are of fixed lengh and reused; they are taken from the packet pool at the message interpretation stage and released after completing the message transmission. Each packet contains the following two fields:

Data field contains the contents of message.

Control field keeps the packet control information including the packet sequence number.

Externally from the user's point of view, any message is a string of the content fields of its all unit packets without any boundary.

2.4.3 Data Representation

Each data element in the standard field and the extra field is encoded with a tagged data representation. A structured element is represented as a structure of basic elements.

2.5 Remote Method Call

The control sequence of remote method call is synchronous as follows:

1. When a method call is made to a proxy object by a client process, the method information is packed into a request message and sent to the original object on the server process, and then the client process starts waiting for the reply message.

2. When the server process receives a request message, it extracts the destination information out of the message, retrieves the corresponding original object and passes the rest of the message to the object.

3. The original object unpacks the request message into method information, and calls the method to itself. After the method is executed, its execution status and the results of the parameters are packed into a reply message and sent to the proxy object.

4. When the reply message is received by the proxy object, it is unpacked and unified with the initial method pattern.

2.6 Nested-Call Control

The computation process in object-oriented systems is a chain of method calls. In case of remote method calls, it would be a nest of remote method calls — one remote method call might bring forth another remote method call during its execution.

In order to control this nested call situation, the client process and its server process keep a symmetric relationship to each other in receiving reply messages as shown in Figure 2.

Global Message Identification

To keep the correspondence between requests and replies, each request message is assigned a *global message identifier (GMID)* and the GMID is attached to its corresponding reply message.

Reply Receiving Control

A process starts waiting for the reply message after it has sent a request message. If a request message is received before the reply message, it gives a service for the coming request message in the same way as server processes do. After completing the service, it resumes waiting for the first reply message.

2.7 Object Migration

Since remote method calls function in the same way as local method calls do, objects would be flown out or *roam* to some process other than the client process and its server processes, for instance, as a parameter of another remote method call. This is called *object migration.*

As an object is flown from one node to another, several proxy objects are created at remote nodes. All of these proxy objects share the same global object identifier (GOID) asssined to the original object. Creating proxy objects and retrieving the original object are managed accordinging to their GOIDs as shown in Figure 3.

As methioned before, a pair of export/import tables is kept not per node but per process. Since the original process holding the desired original object might not be alive at the searching time, the search is not assured to be always possible. It would be assured in such a case that a process commits some job on a remote object to another remote process and release the CPU to other processes until the job is finished.

2.8 Error Handling

Most of the problems that make the network system more complicated and different than the local system come from the depth of its layered functions and the variety of its possible errors.

When an error occurs in a multiprocessing enironment, the required information to be left is which method on which object failed in which process for what reason. The general object in SIMPOS is static one only to encapsulate the representation of data and operations but not to keep the execution environment. Supposing that some object might be accessed from more than one process, the execution status should be kept in the process, not in the object.

In the ROAM, each process holds a status object that keeps the execution status of the last method call and the current status is inquired to its own process. Owing to this functionality, it has been made simple and efficient to catch and analyze the error conditions and also possible to leave the access status on shared objects.

2.9 Customization

ROAM supports nested call and object migration so that remote method calls can be chained in the same way as local method calls. The overhead of these functions, however, cannot be ignored. ROAM provide customization facilities for the user to adjust the protocol according to applications and reduce the overhead with.

2.9.1 The Extra Field

Each of request messages and reply messages is provided with a user-definable field at the end, called *extra field.* With the extra field, users can transfer additional information necessary for the method call. Statistically, most methods are found to be such light ones as slot accessing. If it is well known what slots to be accessed in a method, the slot information should better be transferred at the invokation time of the method. In this way, the number of nested calls would be reduced.

2.9.2 Complete Object and Imcomplete Object

An object which has a full set of functions as a proxy object, including the ability of remote method call, is called *complete object*. Introduced here is another called *incomplete object* without this ability, which is referred just as a parameter of some remote method call to other objects and does not makes any remote method call to its original object.

The incomplete object is transferred its internal states for the original functions to the original object while the complete object not. The extra field is also used for storing the frozen internal states of incomplete objects.

2.10 Implementation

The first version of ROAM has been implemented and is currently working. It consists of 16 classes written in about 2200 lines of ESP code, which are categorized into four kinds; remote object kernel classes, message classes, process-related classes and others.

2.11 User Interface

Users can define the class of any global object as follows:

1. Inherit one of the remote object kernel classes.
   ROAM provides two remote object kernel classes; one is the class `remote_object` for defining complete objects and the other the class `as_remote_object` for defining incomplete objects. The class `as_remote_object` is inherited by the class `remote_object`.

2. Define the external and local interface methods.
   ROAM provides two general interface methods for making a method call; one is the global interface method `:g_call` and the other the local interface method `:l_call`. The global interface method `:g_call` internally makes either a remote method call `:r_call` or a local method call `:l_call` according to the object category. Users define the external interface methods to call the global interface method, and the corresponding local interface methods as their bodies.

3. Overwrite user-redefinable methods if needed.
   ROAM allows users to overwrite some of the methods predefined in the remote object kernel classes for their customization purposes. The user-redefinable methods are categorized into three; the proxy object creating method, the internal state freezing/melting methods for incomplete objects and the extra field handling methods.

3 Application – Global File System

As a practical large application using ROAM, the development of a global file system is shown.

Local File System

The previous file system in SIMPOS was a local system which allowed us to access only the local file and directory resources.

Each resource was dynamically expandable in size and was placed in a hierarchical directory tree. For ease of positioning resources, it was possible to set *a current directory* defining the current accessing environment and to define *logical names*, that is, aliases for an absolute name on the directory tree.

The system regarded processes as being cooperative as in [Reid83] and controled the sharing of resources based on some protocol.

Global File System

The local file system has been extended to a global file system using ROAM to allow us to access any file and directory resource over the network at the method call level.

The global file system retains all of the above characteristic features of the local file system. Only the difference visible to the user is that the name space of resources is expanded to be *the global directory tree* consisting of the local directory trees as subdirectory trees. Each resource is given an absolute name (the node name concatenated with the local pathname).

Implementation

The file and directory resources are defined as complete objects. The related objects which are be passed as parameters of the method calls on the resources, such as buffers and position markers, are defined as incomplete objects.

The amount of the modification required to realize the global file system was 2.7K lines of ESP code that is about 10 % of the entire system of 25.5K lines.

4 Evaluation

The first version of ROAM has been implemented on the PSI machine. The performance of the current PSI is about 30 μs per one predicate inference as reported in [Nakajima86].

From the process' point of view, the communication is levelized into three; the user-user process level for ROAM, the manager-manager process level and the handler-handler process level in the underlying communication system which functions for the session layer. For the functions of the transport layer and below, a hardware controller called LIA/LIB interfaces between the communication system and the physical Ethernet cable. The round-trip time of transmitting a null packet is measured at each level and is shown in Table 1. This is the so-called *network penalty* [Cheriton83] that means the pure cost of the communication system.

In order to grasp the overhead of ROAM itself, *primitive methods* which take no parameter have been executed and their elapsed times are listed in Table 2. (The definition of the primitive methods is shown in Appendix 1.) Contained in the elapsed time are the network penalty obtained in Table 1, the local call time and the overhead from ROAM. For the primitive method, the local call time is 30μs and the ROAM overhead shows the least cost required for the message interpretation.

As for the global file system, several representative methods have been executed remotely and locally and their elapsed times are listed in Table 3.

The Table 4 shows the execution times of several methods offerred in SIMPOS, which have been utilized in ROAM, especially for the message interpretation.

Comparing the current version of ROAM with the other related systems in performance, it is evaluated to be slow. For instance, *Flamingo* is reported its performance to be 90 RMCs/sec, that is, about 11 msec/RMC, while ROAM takes 640 to 680 msec/RMC.

The network penalty at the lowest handler-handler level is attributed to the hardware configuration adopting the LIA/LIB controller. The factors which are loading the overhead above this level, that is, in the communication system and ROAM are as follows:

packet size The packet size is set to be 4200 bytes, but most of the methods are measured to require less than 1400 bytes. By changing the packet size to be 1400 bytes, about 260 msec will be reduced.

message copy ROAM was implemented on the existing communication system which had been designed for a general purpose. The communication system accepts buffers, not messages and copy these buffers to the handler's buffer area again. As a message is divided into small pieces of packets as buffers, the number of process switching between the user process and the network manager will increase. In order to minimize the number of process switching and message copying, we are designing another communication system specially for ROAM, that will accept a message and pass it directly to the handler.

library cache The message interpretation requires 100 to 140 msec at least. Most of this time is spent for retrieving class objects from the library and converting atoms to/from strings as shown in Table 4. We are planning to introduce some library information cache to keep the working set of classes with some limitation given on its usage.

5 Discussions

Through this experience with ROAM and the global file system, we have been faced with several problems.

One of the major problems left unsolved is that we gave some limitation on object migration because of not supporting the global GC (garbage collection). Namely, a pair of export/import tables is not given per node but per process and is maintained locally in the process according to its execution environment. Since these tables are released at the process termination, the remote method call to some migrated proxy object is possible only if the process holding its original object is alive at that time.

There are several reasons why we did not support the global GC.

Firstly, we have been wondering whether it would be practical or not to support the global GC over such a loosely connected system via the local area network like our PSI's. Suppose that we would have to pay for the overhead of transmitting some extra information for updating the global reference count of each global object. It must be feasible if it would be on tightly connected multiprocessor systems.

Secondly, the ROAM has been implemented over the existing local system whose GC strategy is so-called an exhausted GC based on *the mark and sweep method* not on *the reference counting method*. Would we adopt the reference counting method for global objects, every object migration should be declared in an explicit way that is not assumed for local objects.

Forcusing on this problem, we would like to compare ROAM with other related works.

Flamingo [Anderson86] is an RMC interface over an RPC interface, which is specially designed for a window system. This system assumes the usage in its applications and does not support either the global GC or the object migration. Local objects and remote objects are distingished from each other. Deleting global objects is left to the user's responsibility. Any object keeps one-to-one relationship with its owner process, that is, is fixed to one process and cannot migrate. In other word, more than one process cannot share one object.

ROAM has been designed as an extentional layer over the basic execution level of the method call because such a primitive modification on the structure of objects or the code of method call was impossible for the current stage. Thus, neither the bare structure of object recognizes whether it is local or global, nor the built-in predicate *method_call* handles the remote method call itself. There are some system and languge which have been designed from the kernel level.

Distributed Object Manager [Decouchant86] is a kernel mechanism built in a Smalltalk-80 system for the purpose of extending it to a distributed object-oriented system. For accessing a remote object, this system creates *a proxy object* and returns it to the user as ROAM does. This system, however, is designed straight to support the object migration completely. One object table, corresponding to the pair of export/import table in ROAM, is managed per node and the global GC is supported. The global GC is based on the reference counting method and consists of two levels; the local GC and the remote GC. Each object is supplemented with a remote reference count which keeps the number of its exporting in addition to the conventional local reference count.

Emerald [Black86] is a languge designed for constructing distibuted object-oriented systems and applications. One object table is given per node as the above system and the object migration is supported with the following optimizations in efficiency. Objects are categorized into *global objects*, *local objects* and *direct objects* for built-in types. The former two kinds are almost the same as in ROAM meaning whether the object reference can be exported or not. Also supported are two kinds of the parameter passing semantics; *call-by-object-reference* and *call-by-move*. The former is to export only the object reference, the latter to transfer the internal states of the object. This is similar to the differnce between the complete objects and incomplete objects in ROAM.

As a feasible extension to make the object migration sound in ROAM, we would like to take an explicit declaration strategy such that each global object would be declared to be either *mobile* or *fixed* meaning its possibility of migration and the mobile objects would be entried in other *global object table* prepared per node. This extention is one of our future's work.

6 Conclusions

This paper presented the design and implementation of *Remote Object Accessing Mechanism (ROAM)* in SIMPOS and its application to the development of a global file system.

ROAM enabled method calls to be made to remote objects completely in the same way as to local objects. The interface of ROAM was simple by the class inheritance and the method overwriting that it enabled us to develop distributed applications compactly as much as easily. Using ROAM, we could extend the existing local file system to the global file system with a small amount of modifications. Through this experience, we found that ROAM is very useful and indispensable to the development of distributed applications.

As for the performance, the current version of ROAM was found to have several problems concerning on the packet size, the message copy and the library access. We are planning to improve them in the next version.

The designing of ROAM brought us some question of what a distributed object-oriented system would be. The complete function of object migration holds only with the global GC supported. We gave some limitation on the ability of object migration because of not supporting the global GC. We are planning to extend the current design to assure the searching of mobile objects assuming the explicit declaration strategy.

With ROAM, SIMPOS has taken the first step toward a distributed object-oriented system.

Acknowledgements

The author would like to express her special thanks to Tadashi Mano, Masahiko Tateishi, Hirotaka Fukui, Masahiro Hoshi and Hiroshi Ohsaki who helped in the design, implementation and evaluation of ROAM. Also thanks go to Prof. Richard F. Rashid and Dr. Hideyuki Tokuda both from Carnegie Mellon University and Prof. Mario Tokoro from Keio University for their helpful comments.

References

[Chikayama84] T. Chikayama, *ESP Reference Manual*, Technical Report TR-044, ICOT 1984

[Yokoi84] T. Yokoi, *Sequential Inference Machine: SIM -Its Programming and Operating System*, Proc. of FGCS'84, Tokyo 1984

[Nakajima86] K. Nakajima, H. Nakashima, M. Yokota, K. Taki, S. Uchida, H. Nishikawa, A. Yamamoto and M.Mitsui, *Evaluation of PSI Micro-Interpreter*, Proc. of IEEE COMPCON-spring'86, March 1986

[Nelson81] B. J. Nelson, *Remote Procedure Call*, Technical Report CSL-81-9, Xerox 1981

[Almes83] G. T. Almes, *The Evolution of the Eden Invokation Mechanism*, Technical Report 83-01-03, Dep. of Computer Science, University of Washington, Jan. 1983

[Black85] A. P. Black, *Supporting Distributed Applications: Experience with EDEN*, Proc. of the 10th ACM Symp. on Operating Systems Principles, Dec. 1985

[Black86] A. P. Black, N. Hutchinson, E. Jul and H. Levy, *Object Structure in the Emerald System*, Proc. of ACM OOPSLA'86, Oct. 1986

[Andrew86] J. H. Morris, M. Satyanarayanan and M. H. Corner, J. H. Howard, D. S. H. Rosenthal and F. D. Smith, *Andrew: A Distributed Personal Computing Environment*, Communiation of the ACM 29, 3, Mar. 1986

[Satya86] M. Satyanarayanan and E. Siegel, *MultiPRC: A Parallel Remote Procedure Call Mechanism*, Technical Report CMU-CS-86-139, Carnegie Mellon University, August 1986

[Rashid86] R. F. Rashid, *From RIG to Accent to Mach: An Evolution of a Network Operating System*, Proc. of FJCC, Nov. 1986

[Jones86] M. B. Jones and R. F. Rashid, *Mach and Matchmaker: Kernel Language Support for Object-Oriented Distributed System*, Proc. of ACM OOPSLA'86, Oct. 1986

[Anderson86] D. B. Anderson, *Experience with Flamingo: A Distributed, Object-Oriented User Interface System*, Proc. of ACM OOPSLA'86, Oct. 1986

[Decouchant86] D. Decouchant, *Design of a Distributed Object Manager for the Smalltalk-80 System*, Proc. of ACM OOPSLA'86, Oct. 1986

[Goldberg83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesly, Reading, 1983

[Cheriton83] D. R. Cheriton and W. Zwaenepoel, *The Distributed V Kernel and its Performance for Diskless Workstation*, Proc. of the 9th ACM Symp. on Operating Systems Principles, Nov. 1983

[Cheriton84] D. R. Cheriton and W. Zwaenepoel, *The V Kernel: A software base for distributed systems*, IEEE Software vol.1, 2, Apr. 1984

[Liskov84] B. Liskov, *Overview of the Argus Language and System*, Programming Methodology Group Memo 40, MIT Lab. for Computer Science, Feb. 1984

[Walker83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, *The LOCUS Distributed Operating System*, Proc. of the 9th ACM Symp. on Operating Systems Principles, Nov. 1983

[Spur86] M. Hill, S. Eggers, J. Larus, G. Taylor, G. Adams, B. K. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendleton, S. Ritchie, D. Wood, B. Zorn, P. Hilfinger, D. Hodges, R. Katz, J. Ousterhout and D. Patterson, *Design Decisions in SPUR*, IEEE COMPUTER 19, 11, Nov. 1986

[Welch86] B. R. Welch, *The Sprite Remote Procedure Call System*, Technical Report UCS/CSD 86/3-2, University of California Berkerly, June 1986

[Reid83] L. G. Reid and P. L. Karlton, *A File System Supporting Coorperation between Programs*, Proc. of the 9th ACM Symp. on Operating Systems Principles, Nov. 1983

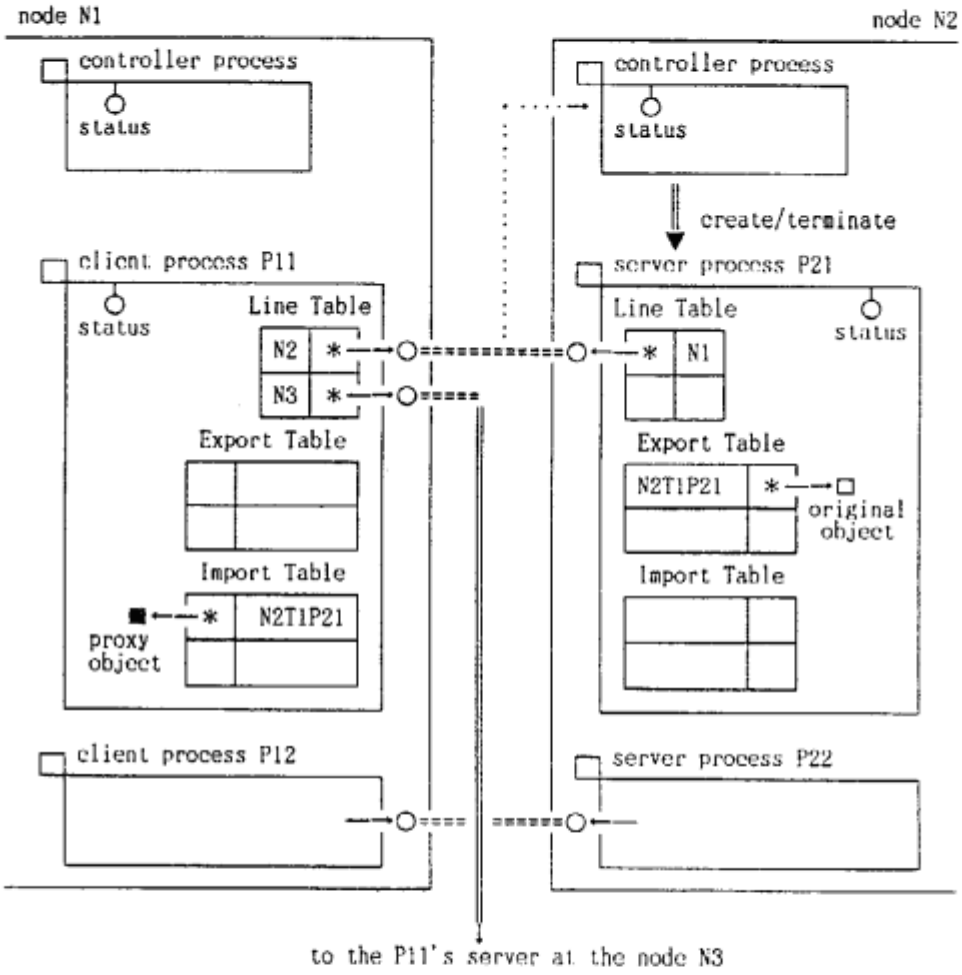


Figure 1: Remote Object Accessing Mechanism (ROAM)

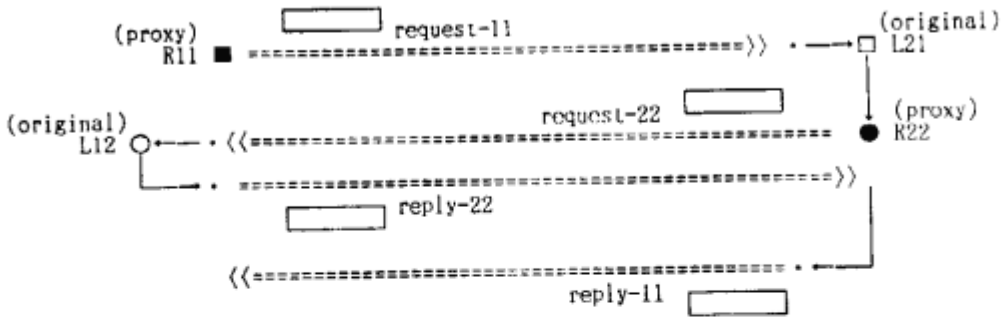


Figure 2: Nested-Call Control

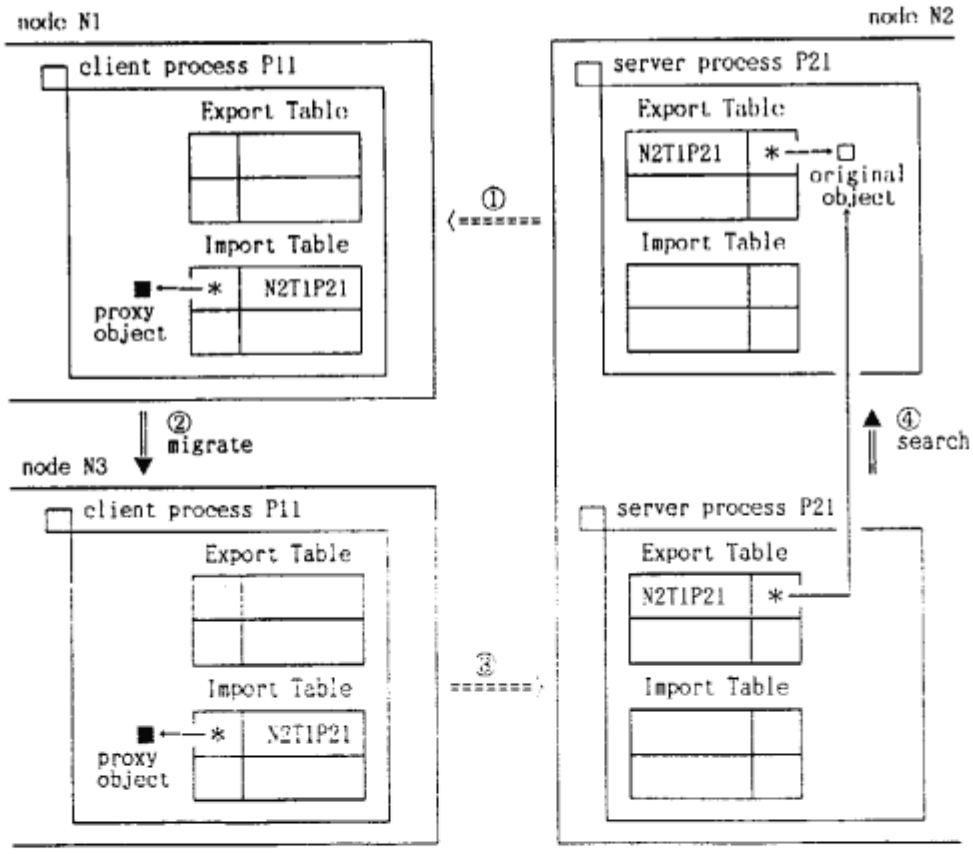


Figure 3: Object Migration

Table 1: Performance of the Communication System

(msec)

Process Level	*Round-Trip Time*	
	Packet Size (byte)	
	1400	4200
handler level	180	
network manager level		
user process level	283	545*
initialization		1100
(connect line and create server)		

* : *Network Penalty* to be referred in Table 2

Table 2: Performance of ROAM

(msec)

Primitive Method	*Total Elapsed Time*	*ROAM Overhead**
primitive class method		
`:do(Class, Node)`	685	140
primitive instance method		
`:do(Instance)`	645	100

* : ROAM Overhead = Total Elapsed Time - Network Penalty - Local Call Time (30μs)

Table 3: Performance of Global File System

(msec)

Method		*Remote Call*	*Local Call*
File Methods			
`:make/c2` (create and open)		1055	167
`:write/i2` buffer size =	1024	1197	27
	40960	4817	399
`:read`	1024	1160	25
	40960	4621	179
`:size/i1` (inquire size)		788	10
`:close/i0`		2197	1446
Directory Methods			
`:make/c2` (create)		1835	180
`:expunge/i0`		755	13
`:find/i2`		879	19
`:delete/i2`		934	20
`:undelete/i2`		930	20
`:add/i2`		1134	25

Table 4: Performance of Related Functions

(msec)

Class/Method	*Elapsed Time*
class `library`	
`:get_class_object/c2`	20 ~ 90
class `symbolizer`	
`:get_atom_string/c2`	2
`:get_atom/c2`	1
`:enter_atom/c2`	5

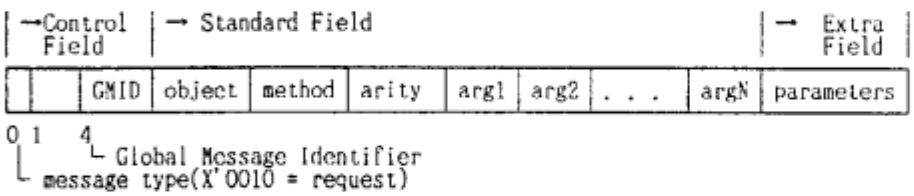
Appendix 1 Definition of Primitive Methods (in ESP)

```
class  test  has
   nature                                                    % inheritance %
     remote_object ;
                                                             % external class methods %
   :do(Class, Node) :-
   :make(Class, Instance, Node) :-
     :g_call(Class, make, { Instance } , Node) ;
                                                             % local class methods %
   :l_call(Class, do, { } ) :- !;
   :l_call(Class, make, { Instance } ) :- !,
     :new(Class, Instance) ;

 instance
                                                             % external instance method %
   :do(Instance) :-
     :g_call(Instance, do, { } ) ;
                                                             % local instance method %
   :l_call(Instance, do, { } ) :- !;

end.
```

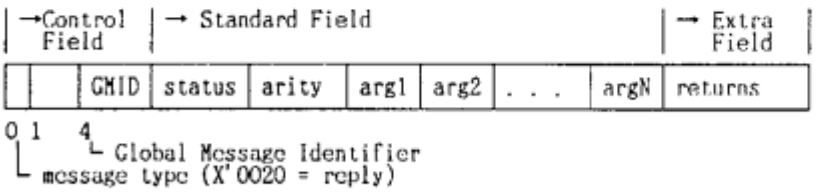
Appendix 2. Message and Packet

(1) Message

① Request Message



② Reply Message

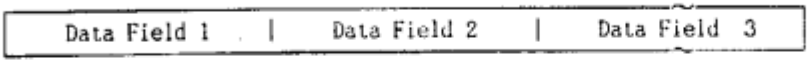


(2) Data Representation

	Tag				
Undefined Variable	X'FFFF				
Integer	X'0010	Integer			
Atom	X'0030	String			
String	X'0*0#	Length	Data		
	(short X'000# , long X'0F0#)				
heap vector	X'0*60	Length	Data		
	(short X'0060 , long X'0F60)				
list	X'0070	Car	Cdr		
stack vector	X'0*80	N	Element1	. . .	ElementN
	(short X'0080 , long X'0F80)				
class object	X'FF10	Class Name String			
exported instance object	X'FF20	Class Name String	GOID		
imported instance object	X'FF30	Class Name String	GOID		

(3) Packet

① External View of Packets



② Internal Representation of Packet