TR-491

Remote Object Access Mechanism

by
K. Yoshida

July, 1989

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 Access Mechanism

Kaoru Yoshida
Institute for New Generation Computer Technology (ICOT) [1]

Abstract

As part of the fifth generation project at ICOT, we have been developing an object-oriented logic programming and operating system, SIMPOS, for the personal sequential inference machines: PSI and PSI-II. Currently, about three hundred PSI machines and some other machines have been connected in a local area network and in global area network, and two different protocols: PSI-NET and TCP-IP are supported over the network.

For a distributed system to be built network-transparent most simply and compactly, it should be network-transparent at the base, that is at the method call level for an object-oriented system.

This paper describes the principles and implementation of the *Remote Object Access Mechanism (ROAM)*, which is a general mechanism embedded in SIMPOS to invoke methods to objects on remote PSI machines as well as to objects on the local PSI machine.

ROAM has been implemented as a set of classes. Network-transparent objects can be defined easily only by inheriting a ROAM interface class and customizing some of the inherited methods. ROAM is currently in operation for both of the above two protocols. Using ROAM, a variety of application software has been developed for practical use, one of which, a global file system, is shown as an example.

Keywords: object-oriented system, distributed operating system, network system

[1] 4-28 Mita, 1-chome, Minato-ku, Tokyo 108 JAPAN

1 Introduction

The object-oriented paradigm is powerful and natural for designing and developing large-scale and complex systems. The notion of an object has been introduced into different kinds of languages, including procedural, functional and logic ones, and many object-oriented systems have been developed and used.

As part of the fifth generation computer systems project at ICOT, we have been developing an object-oriented logic programming and operating system, SIMPOS, for the personal sequential inference machines, PSI and PSI-II [2, 3, 4]. The entire system of SIMPOS, including the kernel such as device handlers, are written in an object-oriented and logic programming language ESP [1]. Most of the characteristic properties of SIMPOS are those inherent in ESP.

As a system grows, a network environment becomes necessary to build up a more cooperative environment, so that the users can share their resources and communicate with each other through network. At present, we have connected about three hundred PSI machines and some other kinds of machines, such as VAX, Sun and Symmetry, in a local area network of 10 Mbps and a global network of DDX. Two different protocols: PSI-NET and TCP-IP are supported over the network. Both protocol are avaible for PSI-to-PSI and PSI-to-VAX communications; only TCP-IP is available for the rest.

It is not enough just to provide a network environment. If the user has to use different tools depending on which machine to access, the system will hardly be used. We need a system that seems to be identical no matter where the user logs in or accesses: a network-transparent system.

We sought the possibility of making SIMPOS a network-transparent distributed object-oriented system. As a result, we have embedded in SIMPOS a general mechanism, called the *Remote Object Access Mechanism (ROAM)*, which enables to invoke methods to objects on remote PSI machines as well as to those on the local PSI machine.

Network-transparent objects can be defined only by inheriting a ROAM interface class and customizing some of the inherited methods. ROAM is now in operation for both of the above protocols. The PSI-NET version has been running since June 1987, and the TCP-IP version since June 1989.

This paper describes the principles of ROAM and its implementation in SIMPOS. Before going into detail, the background and requirements behind ROAM are stated in Section 2. The principles and implementation issues are described in Section 3 and Section 4 respectively. The performance of ROAM is presented in Section 5. A variety of practical applications has been developed, of which, a global file system is shown as an example in Section 6.

2 Why is ROAM needed?

We clarify the background and requirements which led up to ROAM in this section.

2.1 Network-transparency at the base

One way to make a network-transparent system may be to provide a remote access function at the application level (in each application program). However, this approach will enlarge (double at worst) the size of the system. In general, a computer system can be regarded as a sphere, the hardware as its core, system software surrounding it, and application software as the outermost layer. An outer layer has a larger surface area, so the amount of software increases accordingly. To make a system simple and compact as much as possible, the system should be network-transparent at the base, at the level of machine architecture.

In those systems based on procedure calls, several protocols for *remote procedure calls* (RPCs), which enables the invocation of procedural calls on remote machines, have been proposed or implemented [5, 10, 17, 21]. Some operating systems embedded an RPC mechanism at the base, and some system description languages absorbed an RPC function in it [6, 7, 9, 11, 16, 18, 19, 20].

We would like to explore this direction in an object-oriented system. For an object-oriented system, computation consists of method calls to objects. Network-transparent access means that methods can be invoked to objects in the same way, independent of whether they are on the local machine or on remote machines. Distributed object-oriented architectures should be able to recognize remote objects and local objects in their data representations and issue either a remote method call or a local method call for an identical method call instruction.

Some experimental systems and languages have been designed to be network-transparent from the machine (or virtual machine) level [8, 14], but no practical system has been developed yet.

Our target machines, PSI and PSI-II, were originally designed to be independent local machines. Possibly the best way to realize network-transparency simply and compactly is to embed a mechanism for it into the kernel of the operating system.

2.2 Problems related to PSI and SIMPOS

The PSI machine provides a logic programming language called KL0 as its machine language. Its programming and operating system, SIMPOS, is entirely written in a high level language, ESP. ESP introduces the notion of an object as a modularization mechanism into KL0. Like ESP, SIMPOS is characterized to be object-oriented and logic-based. The object-oriented and logic-based features of ESP and SIMPOS bring to the user a powerful programming environment, but require us to solve several problems in building a distributed system.

First, ESP is object-oriented. The execution of an ESP program is carried out by calling a method from one object to another. An ESP object is a capsule of slots (states) and methods (operations), which is defined in a class. ESP supports multiple class inheritance and demon method combination. ESP is not only a system description language but also a user's application language. SIMPOS is an open system without any boundary between the system and the user. The user can define any complicated class easily by inheriting system-defined and user-defined classes and by customizing that class.

Object-oriented systems are more dynamic in their execution control than other kinds of systems. The program code for each method call is determined by the destination object which is bound at execution time. Also, any method may be invoked to conceptually small objects as well as to physically large objects. In building a distributed system, this uniformity would make communication granularity smaller than that in other kinds of systems; remote method calls might be issued to such conceptually small objects.

Second, ESP is logic-based. A method is defined by a set of predicates defined in all inherited classes. Each predicate is defined by a set of horn clauses with the same functor and arity. Each clause consists of a head and a body which contains zero or more predicate goals. Variables included in predicates are logical variables. Given a method goal, a corresponding method code is searched from the method table defined by the destination object which is bound to the first parameter of the goal. If the head is successfully unified with the given goal, then the body is executed sequentially. If all goals of the body succeed, the method comes to succeed. When one clause fails, all the parameters are unbound, and an alternative clause is tried, if there is.

This feature amplifies the dynamicity of execution control. Each parameter may be bidirectionally bound; it might already be instantiated when the method is invoked or will be after the method is executed. In building a distributed system, this means that parameters must be transmitted bidirectionally, both when sending a method call request and when returning its result.

Summarizing these problems, the basic mechanism for network-transparent access should satisfy the following properties:

General: ESP programs are very dynamic in their execution control. The mechanism should be general enough to be applied to any user-defined classes that contain no information on parameter binding direction.

Customizable: If the binding direction of a parameter is already known when a class is defined, it should be utilized to minimize the communication overhead. The mechanism should be customizable so that it should work efficiently in such predetermined cases

Robust: The mechanism should be robust enough. Even if a target method fails, the mechanism must always succeed without leaving any alternative clauses.

In addition, the following requirements must be satisfied:

Compact, efficient and easy-to-use: The mechanism should be simple and compact for easy development and maintenance, and be efficient in performance. It should also provide a simple interface to the user.

3 Design of Remote Object Access Mechanism

3.1 Principle – Object and Process Reflection –

Assuming the most dynamic cases, the first thing that must be done is to set up a pseudo environment on the local machine, which will reflect the situation on the remote machine where the target object resides.

In SIMPOS, the notion of an object is separated from that of a process. General objects only encapsulate their internal data representation and operations on it. Execution environments are held by process objects.

When accessing an object on a remote machine, the relationship between an object and its owner process must be reflected both on the remote machine and the local machine. This is the principle of ROAM, called *object and process reflection*, and is illustrated in Figure 1.

Original object and proxy object: Suppose that a process tries to access an object on a remote machine. The object on the remote machine is called the *original object*. In contrast, an object, called a *proxy object*, which works as an agent of the original object, is created on the local machine.

When a method is invoked to a proxy object, the proxy object sends a method call request to its original object and accepts its reply. Basically, the internal states required for the original functions are held and maintained by the original object. The proxy object holds only those required for the transmission.

Both the original object and its proxy objects represent some identical entity, called *a global object*. The user need not worry about whether an object is the original or a proxy. It is the role of ROAM to distinguish objects and issue either a remote method call to a proxy object or a local method call to the original object.

Client process and server process: An process which tries to access an object is called *a client process*. If the object is remote, the client process is returned a proxy object. On the remote machine, a process, called a *server process*, is created, which works as a substitute for the client process and owns the original object. For one client process, one server process is created on each remote machine which the client process accesses.

On each machine, there is one process, called the *controller process*, which controls the relationship between a client process and a server process. When a client process tries to access a remote machine for the first time, it sends an initiation request to the controller process on that remote machine. For every initiation request, the controller process spawns a server process, and then lets it communicate with the client process. During communication between the client and a server, there might occur an error that would block up any further communication. When a client process is killed, it sends a termination message to its related controller processes rather than directly to its server processes. When receiving a termination message, the controller process terminates an appropriate server process.

Remote method call: A remote method call (RMC) is performed based on synchronous communication between an original object and each of its proxy objects according to the following procedure, as depicted in Figure 2:

1. When a method is invoked to a proxy object, the proxy object packs the method information into a request message, and then tries to send the message to its original object in a server process. After sending, the proxy object waits for the reply message.

2. When the server process receives a request message, it unpacks the destination object information, and tries to retrieve an original object from the export table according to the information. Then the message is passed to the original object.

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

4. When receiving the reply message, the proxy object unpacks it into a method, which is then unified to the invocation method.

3.2 User Interface – Inherit and Overwrite –

In other distributed languages and systems such as Argus [18] and Eden [6], different abstractions are used for local (or original) and remote (or proxy) objects because of efficient execution.

Our goal is to realize a uniform object model with reasonable performance. ROAM is offered as a set of classes to the user. An original object and its proxy objects belong to the same class and have no difference visible by the user. Any global object class can be defined simply by inheriting a ROAM interface class. This is enough for naive usage. For those who care about performance, customization is available without loss of generality of the function. The following two kinds of customization are possible.

Distinguishing complete objects from incomplete objects: Needless to say, RMCs are more expensive than local method calls (LMCs). Looking at what kinds of methods are invoked, most are small ones like slot access methods. If RMCs are often issued to such small methods, it is hard to expect good performance. If it is already known which slots will be referred to or updated during the execution of the method, they should be put in the request or reply message, so that the number of RMCs will decrease.

Several distributed procedural systems, such as Argus, allow parameters to be passed only by values, not by references. In case of object-oriented systems, however, an object is a reference, so parameters may be references. As a compromise, Emerald [8] supports two kinds of parameter passing: *call-by-object-reference* and *call-by-move*. The former exports only the object reference, the latter transmits the internal states of the object.

We think of mobility as a property of the object itself rather than as how the object is used as a parameter. ROAM introduces the notions of a complete object and an incomplete object. Those proxy objects which have full ability of transmission are called *complete objects*, while those without transmission ability are called *incomplete objects*. An incomplete object is only transmitted as a parameter of an RMC but does not make any RMC to its original object by itself.

Among the internal states of an incomplete object, only those required for executing the original function are transmitted between the original object and the proxy object. The user can overwrite interface methods to freeze and melt these internal states.

Handling the extra field: Customization is also available at the level of message packing/unpacking. Each request message and reply message is provided with a user-definable field at the end, called *the extra field*. The user can overwrite interface methods to handle this field, so that some additional information, including the internal states of incomplete objects, can be transmitted.

The user interface of ROAM is summarized as follows:

1. Inherit a ROAM interface class, which is either `remote_object` or `as_remote_object` The former is for defining complete global objects, while the latter is for defining incomplete global objects. See Figure 8.

2. Define external and local methods. For each method, an external method which calls a global method, `:g_call`, should be defined as an entry, and a local method, `:l_call`, as its body. The global method `:g_call` is a dispatcher to call either a remote method `:r_call` or a local method `:l_call` according to whether the object is proxy or original.

3. If necessary, overwrite user-redefinable methods of the following kinds:

   - a method to create a proxy object,
   - methods to freeze and melt the internal states of incomplete objects, and
   - methods to handle the extra field of request and reply messages.

A sample program is shown in Figure 9.

4 Implementation Issues

This section describes the implementation of ROAM in detail. The ROAM function is categorized into object management, message management, line management and error handling. The first two, object management and message management, are described below.

4.1 Object Management

When an object is referred to from another machine, the object is said to be *exported* from the local machine and *imported* to the remote machine.

In ESP, there are two kinds of objects: class objects and instance objects. ROAM manages these objects as follows:

4.1.1 Identification of Classes and Instances

Class complete name: There is no notion of a meta class in ESP. Instead, the library subsystem in SIMPOS manages all of the class objects. The notion of a package is introduced on top of classes, 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, which is called *a complete class name*. When a class is exported, its complete class name is sent out as class information.

ROAM identifies a class object only by its complete class name. Each machine has an independent library, so a different class might be chosen for the same complete class name. Setting up the related libraries is the user's responsibility.

Global object identifier: In the PSI and PSI-II machines, instance objects are not given any identifier at creation. To maintain communication between a proxy object and its original object, both of the objects need to know they are identical. ROAM manages identification of instance objects.

When an instance object is exported, it is assigned *a global object identifier (GOID)*. The GOID and the complete class name of the instance's class is sent out as instance information. Each GOID consists of the resident machine name, exporting time and process number so that it can be uniquely identified throughout the network.

4.1.2 Export and Import Table Management and Global GC:

Export and import table: To retrieve objects from their GOIDs, a pair of an export table and an import table are prepared for each process. When an original instance object is exported for the first time, a GOID is generated and the object is put with the GOID in the export table of its own process. When instance information containing an unknown GOID is imported, a proxy instance object is created according to the given complete class name, and is put with the GOID in the import table.

One of the major problems with the export/import table management is the garbage collection (GC) of the tables, which is what to prepare export and import tables for and when to release their entries. Suppose that we prepare a pair of export and import tables for each machine and support global GC of the tables. A table entry can be released when there is no reference to the object in the entire distributed system. The *weighted reference counting scheme* has been proposed for parallel computer architectures to collect garbage in real time [23, 24]. This scheme

requires the sending of extra messages to maintain the reference count of each table entry. Even with this scheme, global GC must be supported to take care of cyclic chains made over the network, which must stop all the machines at once.

Each machine in our enviroment works independently and adopts the *mark and sweep scheme* for local GC. In addition, most of the object references exist within the same process context. So we decided to adopt the following strategy:

No global GC: ROAM does not support global GC on the export and import tables. No extra message is transmitted for the purpose of releasing object entries in the export and import tables.

One pair per process: A pair of export and import tables is created for each process rather than for each machine, so that they can be maintained locally within the process and released when the process terminates.

4.1.3 Object Propagation

If an RMC functions in the same way as an LMC does, objects may flow out to a process other than the client process or its server processes, for instance, as a parameter of another RMC.

ROAM supports *object propagation*. As an object flow out from one machine to another, one proxy object is created on each remote machine. All of these proxy objects share the same global object identifier (GOID) that is assigned to the original object. Figure 3 illustrates how object propagation is performed. Note that in object propagation while the object body does not move but its references are propagated, in object migration the object body does move. ROAM does not support object migration.

Limitation on object propagation: The *"one pair per process without global GC"* strategy we adopted for the export/import table management limits the ability of object propagation. At the time when an RMC is invoked to the secondary or later proxy objects, which are created by object propagation, the server process holding the original object must be alive.

As mentioned before, ROAM is based on synchronous communication. Under the normal condition, the server processes terminate after the client process does. In the case where a proxy object is propagated as a parameter of another RMC to another object, there is no problem because the client process is supposed to wait for its reply. The problem is when a proxy object is passed to another process by slot setting or stream communication. The exporter process must terminate after waiting for the acknowledgement from the importer process.

In Figure 3, an object, `L21`, is propagated from one machine to another. First, a proxy object, `R11`, is created at node `N1`, corresponding to the original object, `L21`, at node `N1`. They share a GOID, `N2T1P21`. Then another RMC is issued to some object at node `N3` and proxy object `R11` is exported to node `N3` as a parameter of the RMC. At this time, the secondary proxy object, `R31`, is created at node `N3`. During the execution of the RMC, another RMC is issued to the secondary proxy

object, R31, at node N3. This means the RMC is invoked to the original object, L21. Another server process P22 is created at node N2 and the original object is searched from the GOID, N2T1P21. If the original process, P21, is alive at this time, the search succeeds and the method call is successfully issued to the original object, L21. If it is not alive, the search fails and the method call fails. Therefore, the first client process, P11, should be kept alive at this time.

Let us compare our strategy with those taken in other related works.

- *Flamingo* [13] is an RMC interface over an RPC interface; it is specially designed for a window system. Local objects and remote objects are distinguished from each other. Neither global GC nor object propagation is supported. Deleting global objects is the user's responsibility. Each object maintains a one-to-one relationship with its owner process; more than two processes cannot share one object; an object cannot be propagated from one process to another.

- *Distributed Object Manager* [14] is a kernel mechanism built into a Smalltalk-80 system [15] for the purpose of extending it to be a distributed object-oriented system. To access a remote object, it creates *a proxy object* and returns it to the user as ROAM does. This system is designed to support object migration. One object table, corresponding to a pair of an export and import table in ROAM, is created for each node and global GC is supported. Global GC is based on reference counting and consists of two levels: local GC and remote GC. Each object is supplemented with a remote reference count which holds the number of its exports besides the local reference count. This system is an experimental system, in which object access and migration have been simulated between just two virtual nodes in a physical machine.

- *Emerald* [8] is a language designed for describing distributed object-oriented systems and applications. Its prototype system is being implemented. One object table is given per node, but global GC on the table is not supported.

ROAM has a limitation on its object propagation ability: client processes must be synchronous. Even with that limitation, we think the support of object propagation is valuable, since it is useful for such a problem like intermediary copier that initiates the copying of a file on a remote node into another remote node. In terms of efficient and practical implementation, the limitation is no hindrance.

4.2 Message Management

Message management is the function of packing and unpacking method information.

4.2.1 Message Representation

Request and reply: There are two kinds of messages: *request messages* for making method calls and *reply messages* for returning their results, as shown in

Figure 4. Each message is of variable length and free format and consists of three fields: *the control field* for message control information, *the standard field* for method information, and *the extra field* which is provided as a user-definable field for customization.

Message and packet: Each message is a logical entity. It is internally represented as a set of unit packets and seems to be a sequence of packets without any boundary, as shown in Figure 5.

Tagged data representation: Each data element in the standard field and the extra field is encoded by a tagged data representation, as shown in Figure 6. Structured data can be represented and can be nested recursively.

4.2.2 Nested-call Control

Computation of an ESP program is a chain of method calls. This implies that the calling sequence may also be nested in RMCs. During execution of an RMC, another RMC might be issued to some of the parameters contained in the first RMC. Thus, after sending a request message to a server process, the client process might receive another RMC request message before receiving the reply message corresponding to the first request.

To control this nested call sequence, the client process and its server process must maintain a symmetrical relationship to each other in receiving reply messages, as shown in Figure 7.

Global message identification: To maintain the correspondence between a request and its reply, each request message is assigned a *global message identifier (GMID)*, which is transferred to the reply message.

Reply receiving control: After sending a request message, the client process waits for its reply message. If the client process receives a request message before the reply message, it performs a service for the coming request message in the same way as server processes do. After completing this service, the client process resumes waiting for the first reply message.

5 Evaluation

ROAM has been implemented on top of the session layer of the network hierarchy. The session layer is provided as a virtual line object. The virtual line object is available for two different protocols: the PSI-NET protocol and the TCP-IP protocol. ROAM has been implemented for both protocols. The PSI-NET version and the TCP-IP version are almost the same except for thier message management, because of the difference in the size and format of their physical packets.

Network penalty: For the transport layer and below, a special hardware controller for each protocol is installed between the machine and the Ethernet cable. A network handler process and a network manager process are between the hardware controller and the user process. The round-trip time of transmitting a null packet at the user process level is measured in Table 1. This is called the *network penalty* [16] that and is the pure cost of the underlying network system. The network penalty is about 170 ms for PSI-net and 450 ms for TCP-IP, both for a 1024-byte packet. The PSI-net protocol handles up to 4200 byte as a single physical packet. Since ROAM was originally designed for PSI-net protocol, the unit packet size was set to 4200 byte. TCP-IP protocol, whose packet size is fixed to 1024 byte, was made available later. We took a measurement for two packet sizes: 1024-byte packet and 4200-byte packet for each protocol.

ROAM overhead: ROAM is compact. It consists of 16 classes written in about 2.2K lines of ESP code. Table 2 shows the performance of ROAM measured for the sample program shown in Figure 9. ROAM is available for both PSI and PSI-II machines. This measurement was taken between PSI-II machines whose CPU performance is reported to be about 10 μs for each predicate inference [4]. In the table, the figures without parentheses show the total elapsed time, and those in parentheses show the CPU time spent in the client user process. The ROAM overhead is expressed as follows:

(ROAM overhead) = (RMC elapsed time) - (network penalty) - (LMC elapsed time)

For method `:do(#test, Node)`, the ROAM overhead is 40 ms for PSI-1024 (PSI-net protocol with 1024-byte packet), 45 ms for PSI-4200, and 50 ms for TCP-1024 and TCP-4200. The performance of *Flamingo* is reported to be 90 RMCs/s, which is about 11 ms per RMC. Comparing our 211 ms for PSI-1024 RMC with this figure, the current ROAM is rather slow, mainly because of the heavy network penalty. Further improvement should be made also to ROAM itself.

Packet size: Tables 10 and 11 show how the elapsed time and the CPU time change as the message size grows, where method `:read(#test, FileName, String, Node)` is tested with various sizes of `String`, which is the file size. This test investigates how the packet size influences the performance. These graphs show that the shorter packet is better for short messages, but the longer packet is better for long messages. The unit cost, including process switching and packet sequence checking, is more expensive for the shorter packet, than that for the longer packet. The crossover point for PSI-1024 and PSI-4200 is around 8-Kbyte message size, and around 35-Kbyte message size for TCP-1024 and TCP-4200. The crossover points are determined by the ratio of the ROAM overhead versus the network penalty. Since TCP's network penalty is more expensive, the crossover point appears at such a larger size.

Interface of the session layer: The interface between the underlying network system and ROAM is a buffer rather than a message. Buffers are copied to the

handler's buffer area. As a message is divided into small pieces of packets (or buffers), process switching between the user process and the network manager will increase. Minimizing the amount of process switching and message copying is essential to improving the performance. We are planning to redesign the interface of the network system, so that a whole message is accepted and passed directly to the handler.

Library cache: Most of the time of message interpretation is spent for retrieving class objects from the library and converting symbols to and from their names. Improvement can also be expected by introducing a library cache which would keep a working set of classes.

6 Applications

Using ROAM, a variety of software has been developed for practical use. From them, we take up the implementation of a global file system (GFS) as an example.

The GFS was developed by extending an existing local file system (LFS). LFS manages a hierarchical structure of directories and files both of which are dynamically expandable, and supports sharing control on resources regarding processes as being cooperative as in [22]. For GFS, the directory name space was expanded to be a *global directory tree*. Each resource is given a global name that is the concatenation of the machine node name and its local path name.

The global files and directories are defined as complete global objects. Those related objects which are passed as parameters of the method calls on the resources, such as buffers and position markers, are defined as incomplete global objects.

Modification required for GFS is very small. Only 2.7K lines of ESP code, about 10% of the entire LFS code of 25.5K lines, were added. Such compact implementation is due to the simple ROAM interface by class inheritance and method overwriting. GFS is currently in operation and its performance is acceptable for practical use.

In terms of file and directory access, the higher level applications such as editors are also made global without any modification. The user can access any files and directories from editors and other applications in exactly the same way.

7 Conclusions

This paper presented the principles and implementation of the *Remote Object Accessing Mechanism (ROAM)* in SIMPOS. ROAM enables method calls to remote objects in exactly the same way as to local objects. Although the network penalty due to the network hardware is rather heavy, the overhead of ROAM itself was found to be low enough for practical use.

Through the development of application software using ROAM, we have learned that ROAM is very useful for building distributed object-oriented systems. We were able to extend an existing local file system to a global file system with a very little modification. Together with the global file system, higher level applications were

made network-transparent in file/directory access without requiring any modification. Both the simple interface and the reasonable performance are due to class inheritance and method overwriting. We would like to improve ROAM further, to make it beneficial for a wide range of applications.

Acknowledgments

I would like to express my gratitude to Dr. Shun-ichi Uchida and Dr. Takashi Chikayama for giving her an opportunity to carry out this research. I also thank the staffs of the SIMPOS network and file groups for their great help in implementing ROAM and the file system.

References

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

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

[3] 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, Mar. 1986

[4] H. Nakashima, K. Nakajima, *Hardware Architecture of the Sequential Inference Machine: PSI-II*, Proc. of IEEE Symp. on Logic Programming, Sep. 1987

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

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

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

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

[9] 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*, Communication of the ACM, Mar. 1986

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

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

[12] 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

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

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

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

[16] 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

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

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

[19] 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

[20] 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

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

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

[23] D. I. Bevan, *Distributed Garbage Collection Using Reference Counting*, Proc. of Parallel Architectures and Languages Europe, Jun. 1987

[24] P. Watson and I. Watson, *An Efficient Garbage Collection Scheme for Parallel Computer Architecture*, Proc. of Parallel Architectures and Languages Europe, Jun. 1987

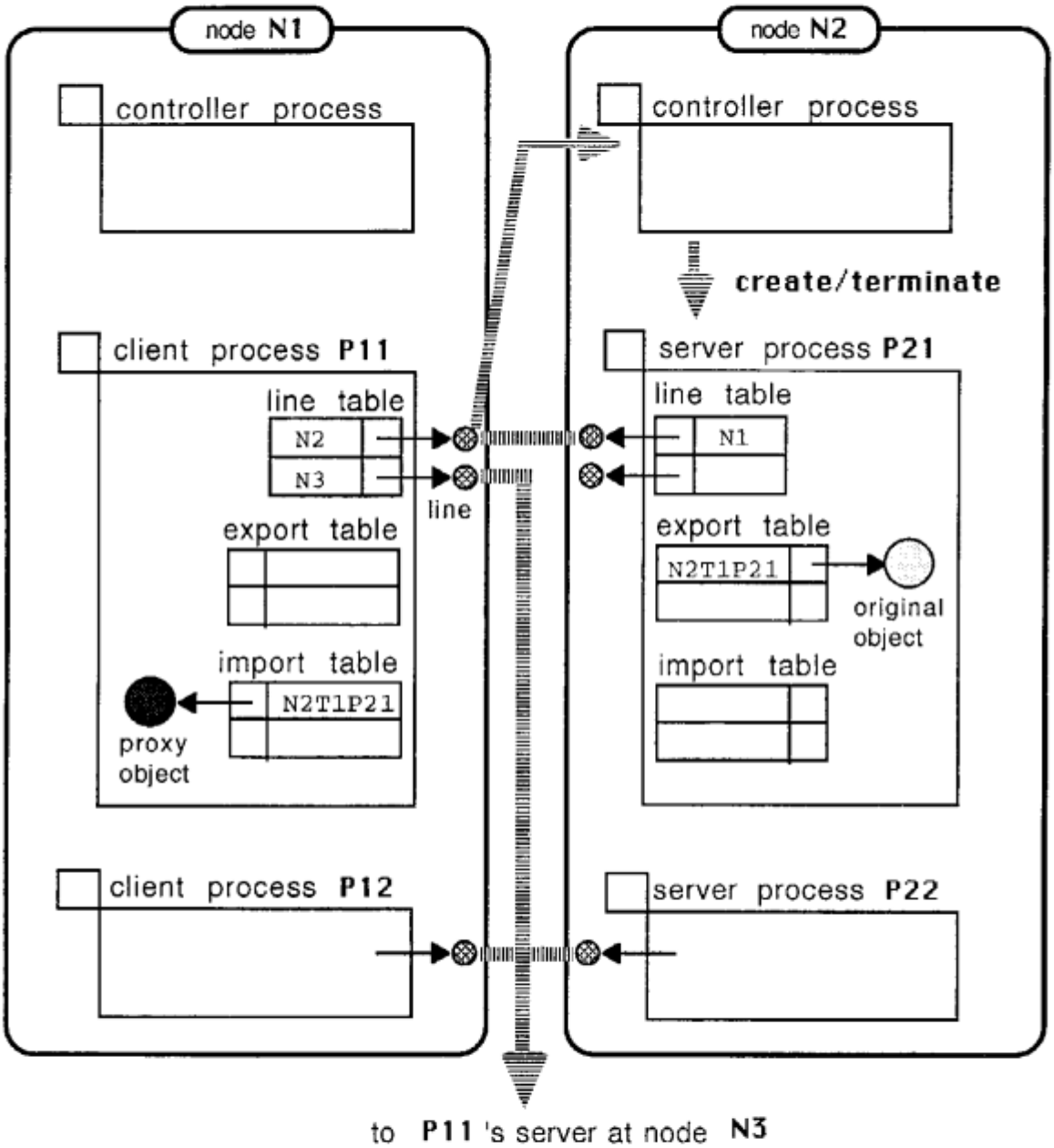


Figure 1: Structure of ROAM

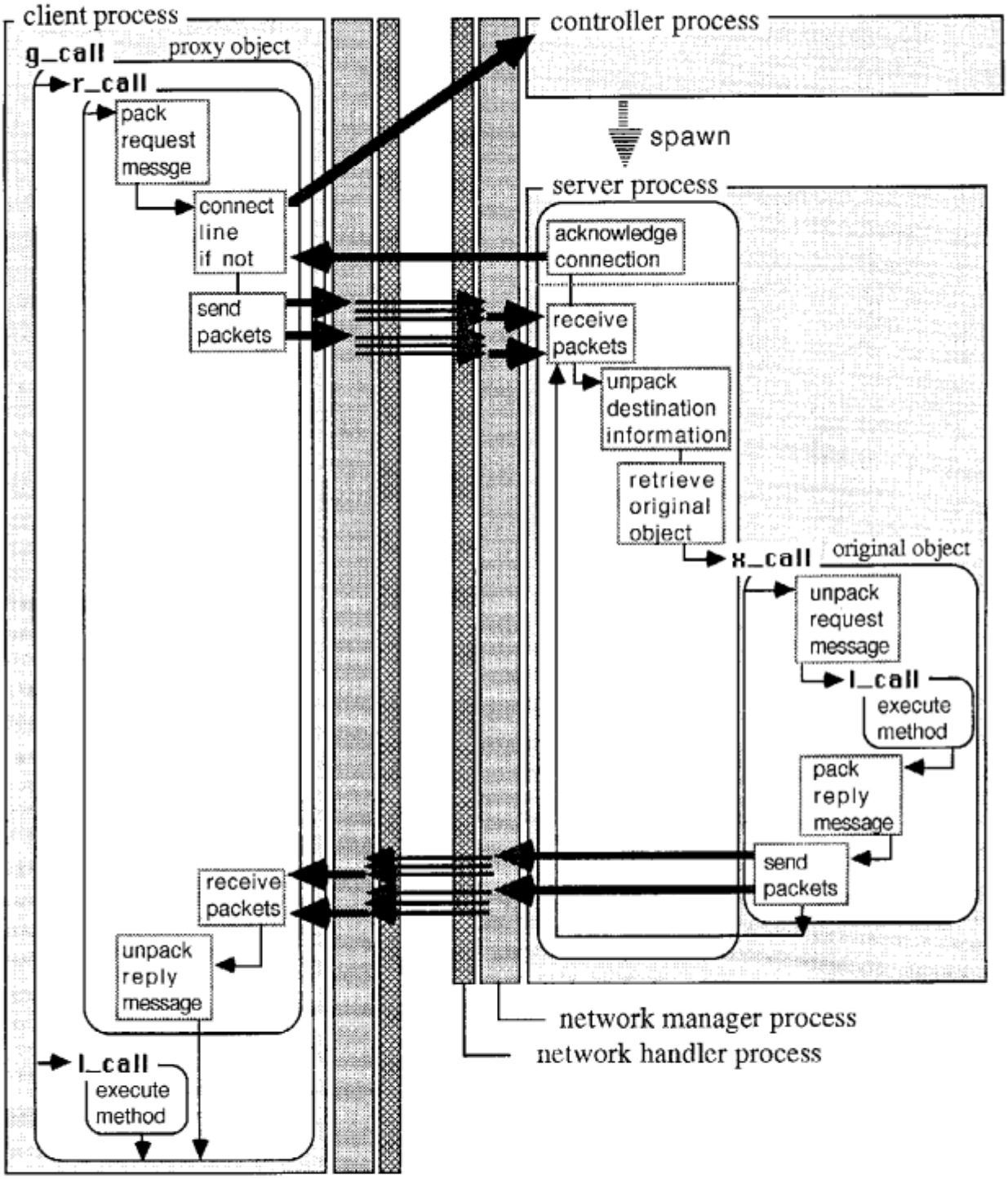


Figure 2: Execution thread of ROAM

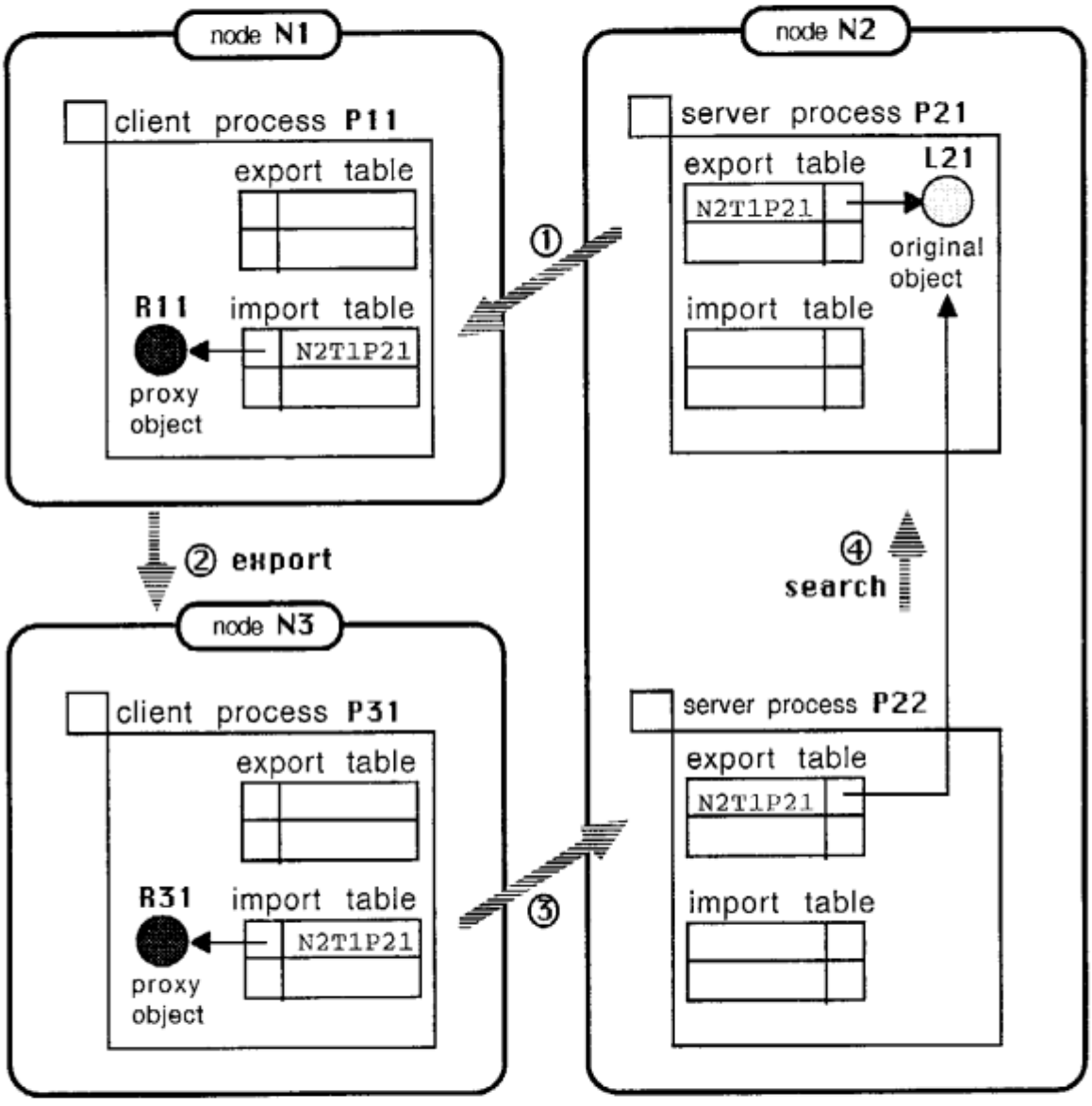
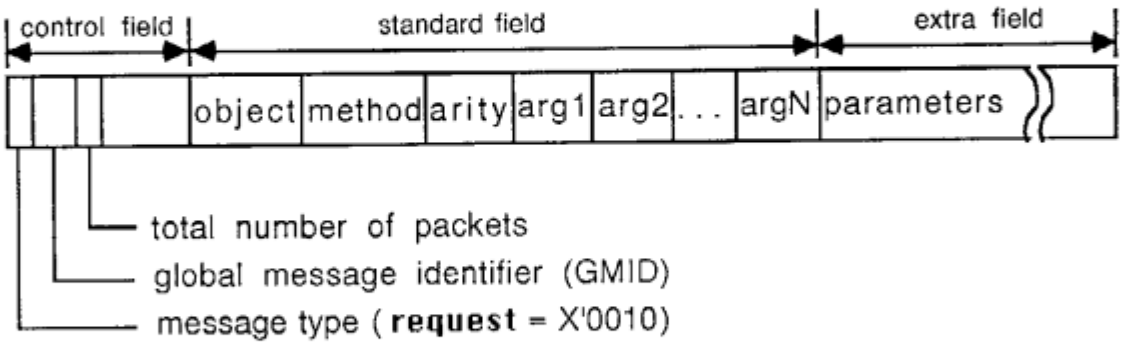


Figure 3: Object propagation

Request message



Reply message

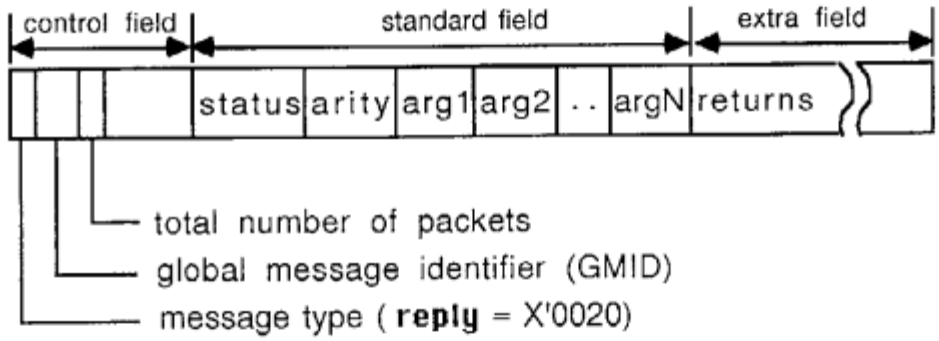


Figure 4: Message representation

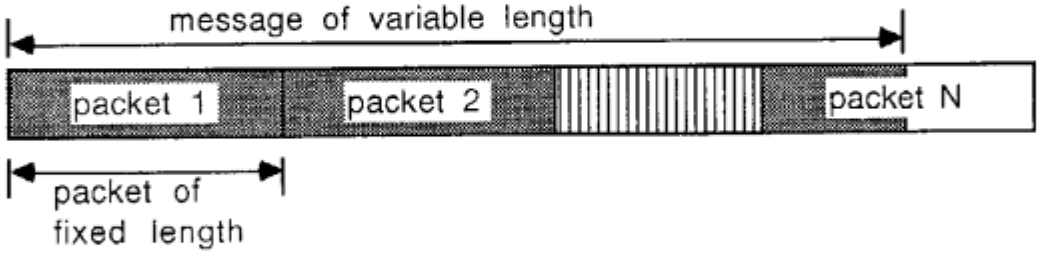


Figure 5: Packet representation

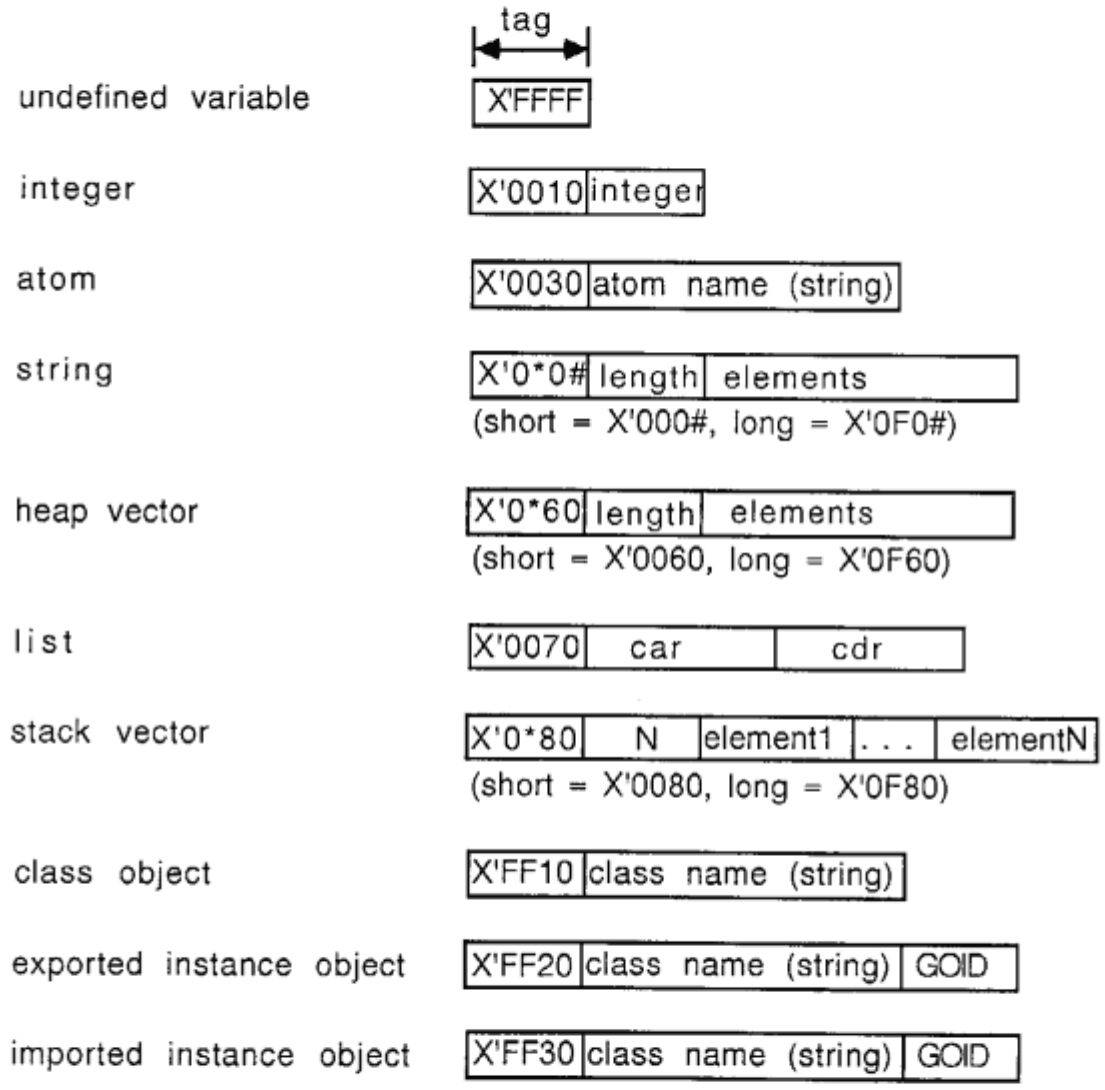


Figure 6: Data representation

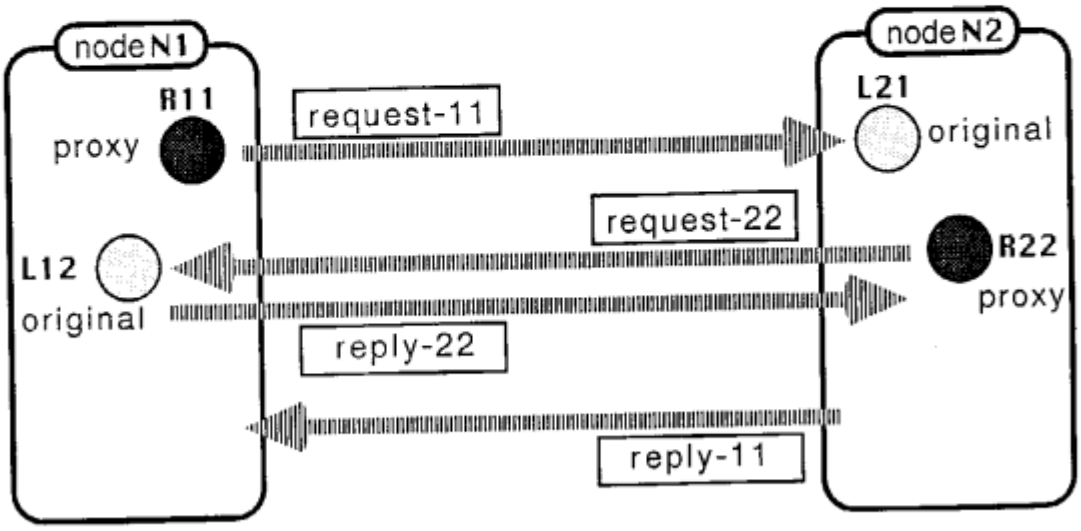


Figure 7: Nested-call control

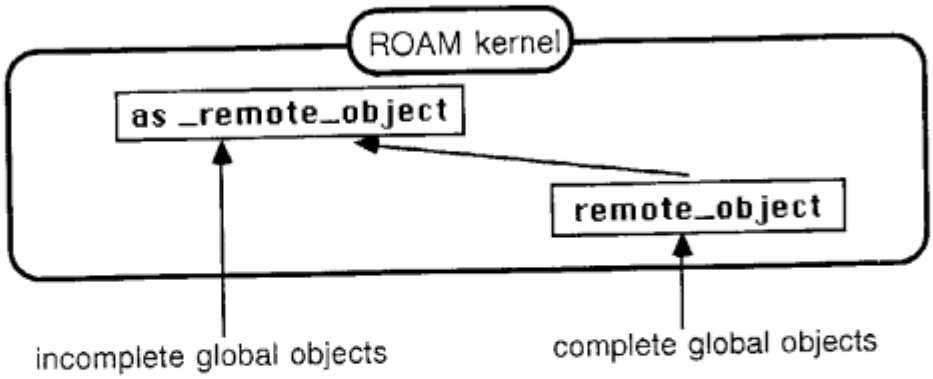


Figure 8: ROAM interface classes

```
class test has
    nature    remote_object ;          % inherit a ROAM kernel class
                                       % external class methods
    :do(Class, Node) :-
        :g_call(Class, do, Node, Node) ;
    :make(Class, Instance, Node) :-
    :g_call(Class, make, Instance, Node, Node) ;
    :read(Class, FileName, String, Node) :-
    :g_call(Class, read, FileName, String, Node, Node) ;
                                       % local class methods
    :l_call(Class, do, Node) :- !;
    :l_call(Class, make, Instance, Node) :- !,
        :new(Class, Instance) ;
    :l_call(Class, read, FileName, String, Node) :- !,
        :open(#binary_file, File, FileName),
        :size(File, Size),
        :new_buffer(File, Size/2, 16),
        :read(File),
        :get_data(File, String, _),
        :close(File) ;
  instance
                                       % external instance methods
    :do(Instance) :-
        :g_call(Instance, do, ) ;
                                       % local instance methods
    :l_call(Instance, do, ) :- !;

end.
```

Figure 9: Sample program

Table 1: Performance of the session layer (ms)

	PSI-net protocol		*TCP/IP protocol*	
physical packet size (byte)	1400		1024	
interface buffer size (byte)	1024 (1 pkt)	4200 (3 pkt)	1024 (1 pkt)	4200 (5 pkt)
round-trip time	171 (31)	428 (36)	450 (82)	1165 (284)
initialization	595 (144)		1140 (190)	

Table 2: Performance of ROAM (ms)

Method	*RMC on PSI-net*				*RMC on TCP/IP*				*LMC*	
	1024		4200		1024		4200			
:do(#test, N)	211	(40)	475	(51)	503	(101)	1214	(311)	2	(0.17)
:make(#test, I, N)	220	(46)	484	(57)	512	(107)	1224	(313)	2	(1.88)
:do(I)	207	(38)	471	(49)	495	(99)	1211	(307)	1	(0.06)
:read(#test, FN, S, N)										
file size = 600 byte	342	(43)	608	(54)	620	(105)	1347	(312)	128	(44)
file size = 1200 byte	469	(60)	611	(55)	775	(167)	1351	(313)	130	(44)
file size = 2400 byte	597	(77)	617	(56)	922	(228)	1357	(314)	134	(45)
file size = 4800 byte	857	(112)	886	(84)	1206	(350)	1883	(558)	140	(46)
file size = 9600 byte	1500	(198)	1164	(116)	1983	(657)	2423	(806)	153	(46)
file size = 19200 byte	2667	(357)	1728	(182)	3323	(1221)	3500	(1303)	179	(48)
file size = 28800 byte	3967	(531)	2544	(271)	4925	(1866)	5060	(2042)	206	(50)
file size = 38400 byte	5160	(687)	3122	(337)	6225	(2420)	6106	(2549)	231	(51)

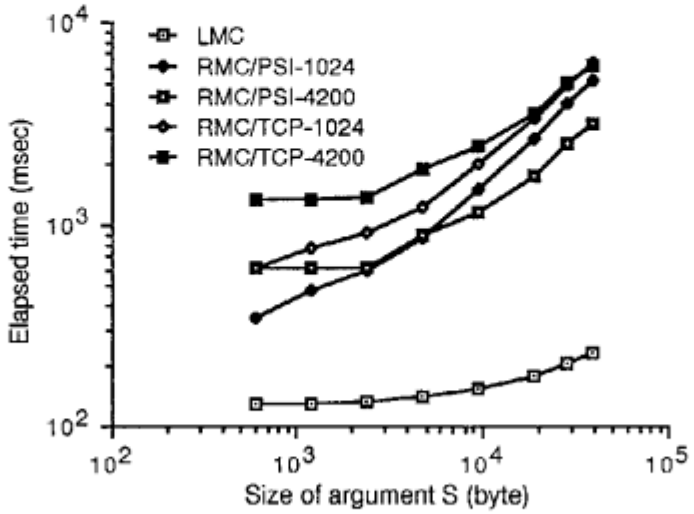


Figure 10: Elapsed time of `:read(#test, FN, S, N)`

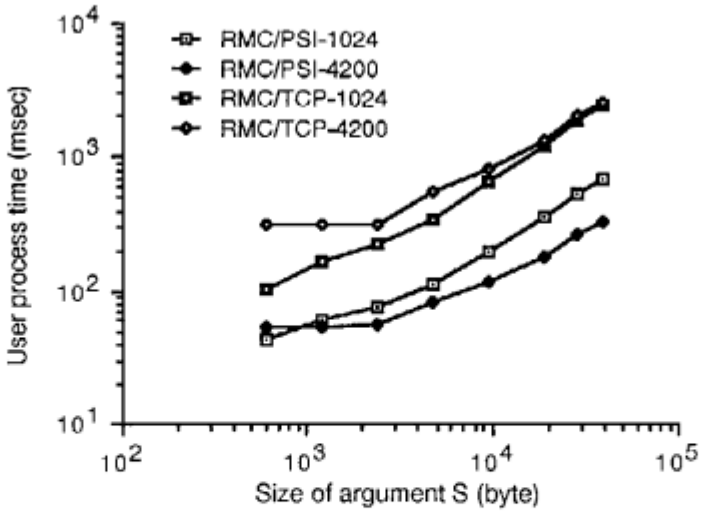


Figure 11: User process time of `:read(#test, FN, S, N)`