ICOT Technical Report: TR-524

TR-524

The SIMPOS Distributed File System — Its Design and Implementation —

by
K. Yoshida

November, 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

The SIMPOS Distributed File System
– Its Design and Implementation –

Kaoru Yoshida

Institute for New Generation Computer Technology (ICOT)
4-28 Mita, 1-chome, Minato-ku, Tokyo 108 JAPAN

Abstract

The personal sequential inference machines: PSI and PSI-II and its object-oriented logic programming and operating system, SIMPOS, have been developed as part of the fifth generation project at ICOT. At present, more than three hundred PSI machines and some other machines have been connected to each other via LANs and WANs, and have been used not only for research and development but also for actual daily work.

This paper describes the design and implementation of the SIMPOS distributed file system. It was first designed as a local file system; as the need to share resources over the network has arisen, it has gradually evolved into a distributed file system, through revisions and extensions. The goal of the distributed file system was to provide a network-transparent file access environment without loss of compatibility with the existing application software. It has been achieved by adopting a dynamic-object-based model for concurrency-access control, a password/capability-based model for access control, and a remote object access mechanism for communication control. The remote object access mechanism is the one to allow network-transparent method calls to objects. It has been found very beneficial for both system development and application development: the file system could be globalized with minimal development cost, and all the existing software was enabled to access remote files in exactly the same way as local ones, without any modification.

The distributed file system has been in operation since summer 1987 and widely used in the ICOT research center and other research institutes; it has been made available for two different communication protocols: PSI-NET and TCP-IP in spring 1989. To our knowledge, this system would be the first distributed object-oriented system running for practical use in this scale.

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

1 Introduction

As part of the fifth generation project of ICOT, the personal sequential inference machines: PSI [1] and PSI-II and its programming and operating system have been developed [25, 26, 45].

The PSI machines provide as their machine language a sequential logic programming language, KL0, which is almost a superset of Prolog; there is no lower language than KL0, which is available for the programmer on the PSI machines. PSI machines feature their specialty as powerful inference engines and their generality as independent personal workstations. Each PSI machine is managed by an independent operating system, SIMPOS, and is provided with an I/O environment including disk, tape, printer and network devices. A higher-level language, ESP, is provided as an application and system description language. It is a superset of KL0 with an object-oriented extension [12]. The entire SIMPOS, from the user interface to the kernel and device handlers, is written in ESP [13]. The object-orientation of SIMPOS is derived from that of its description language, ESP. SIMPOS was first designed for local use and later provided with a network environment. At present, more than 300 PSI machines and other kinds of machines have been connected to each other via LANs and WANs. They have been used not only for research and development but also for actual daily work such as Japanese text processing. Currently two different communication protocols: PSI-NET and TCP-IP are available over the network; each PSI machine may be connected to the network via at least one of the two protocols.

As the number of users and machines increases and the scale of a system is enlarged and enriched, sharing of resources becomes necessary. The history of distributed file systems started with providing network file servers to share a centralized large scale storage among many small or diskless workstations connected to it via a communication network. Most network file servers developed in the early 1980s, including not only practical ones but also experimental ones, are surveyed and each analyzed from the viewpoint of their ability and mechanisms for atomic update or atomic transaction in [38]. In a strict sense, network file servers belong to a different category from distributed file systems in that there is distinction between machines, whether servers or clients, namely that the machine relationship is not uniform.

Distributed file systems assume a uniformly distributed environment in which each machine has its own independent storage. The first goal of a distributed file system is to achieve network transparency, to enable users to access and share resources without being aware of where they are logging in or where the resources are physically located. In addition, heterogeneity and scalability of the network, user mobility (whether a user may log in anywhere), file mobility, security, consistency and fault tolerance of the file content and compatibility with the existing software ought to be considered in the design. There have been developed several distributed file systems, some of which are already in operation and widely used. A system is

[1]This is the first version of PSI machines and is referred to as PSI-I machines in this paper. The word, *PSI machines*, means both PSI-I and PSI-II machines.

intended for a large scale network composed of thousands of workstations in attempt to realize file mobility to keep the I/O access load balanced over the entire system [32]. The current state of the art in the design and implementation of distributed file systems is stated with fine survey in [20, 35].

Distributed file systems are distinguished from distributed operating systems, in that the former make network-transparent only file resources while the latter deal with not only file resources but also other kinds of resources, mainly memory and CPU, and provide themselves with the substantial mechanisms required for network-transparency [39]. In other words, distributed file systems may be built on non-distributed operating systems.

This paper describes an object-oriented distributed file system which has been developed as part of SIMPOS. The system is characterized with its flexible concurrency control, implicit access control and network-transparent access. It was originally designed as a local file system and has grown to be a distributed file system through revisions and extensions. Here is a brief history of the evolution.

1. *Original: a Semaphore-based Local File System.*

   The first version was designed in 1984 as a local file system whose storage structure is a UNIX-like hierarchical tree composed of expandable files.

   This system assumed that files would be accessed from a single process at the local site. The mutual exclusion between user processes was controlled using the semaphore mechanism that each user process blocks a critical region using lock and unlock primitives.

   According to the semaphore mechanism, lock and unlock primitives are scattered over user processes, so debugging was very hard. Moreover, even if the existing classes were well defined, we would have to pay much attention to any slight modification along inheritance trees not to cause a deadlock for its failure to unlock. As a result, it took a long time until the system gets stable, and still left us some anxiety to the system growing.

2. *Redesign to a Dynamic-Object-Based Local File System.*

   As the number of PSI machines and users increased, the need to share files has arisen. The file system was redesigned in 1985 to embed a resource-sharing mechanism [46].

   Resource-sharing does not mean just letting more than one process access a resouce randomly, but controlling cooperative accesses and competitive accesses between processes. The kernel of the sharing mechanism is mutual exclusion control.

   Another problem in the original design was on the integrated control of local access and network access. Local I/O requests were transacted in the semaphore mechanism, while remote I/O requests were transmitted via message-passing. This control mismatch had complicated the structure and control of the system. To allow resource-sharing over the network and make the system simple and stable, an integrated exclusion control was necessary.

These problems were solved with the notion of a dynamic (or concurrent) object elegantly. An dynamic object is a process which encapsulates a shared resource, as a monitor [15] does, but more simply communicates via message-passing with the outside. Owing to the dynamic object model, all the interactions to shared resources are centralized to a single communication channel to their dynamic objects so debugging should be made much easier.

3. *Extension to a Capability-based Local File System.*

   To safely share resources, merely providing the file system with concurrency control is not sufficient. Even if the PSI users do not intend to destroy others' resources, they might destroy other users' files and directories by mistake. The file system was extended to embed a protection mechanism in 1986.

   For a protection mechanism, a user process is regarded as an offense with a weapon and a resource as a defense with a shield. Our strategy is very simple: a user may hold a set of passwords as the weapon, a resource may hold an access list as the shield, and each directory entry on the way to the resource works as a protection gate, where an access list prescribes what capability (a set of rights [44, 21]) may be given to what password. At every protection gate on the way to a target resource, users show their passwords to obtain a capability to proceed. Thus, a legally-controlled resource-sharing environment was set up in the SIMPOS file system.

   Yet, this system was still a local file system in that remote file access was independently done in network application software above the file system. The file system was not concerned with remote access at all. Each of the I/O related application software, such as editors, had to contain two modules: one for local file access and the other for remote file access.

4. *Extension to a Distributed File System.*

   As the number of application software requiring remote file accesses increases, absorbing the remote file access function in the file system became necessary. The file system was redesigned again in 1987, to provide a remote access mechanism so that users be able to access remote files as they do local ones.

   To enable the network-transparent access at the primitive interface level, we embedded in the file system a remote object access mechanism (ROAM) [47], which enables network-transparent method calls to objects. Every file and directory method was reimplemented using the mechanism. In addition, user names and file pathnames were globalized. As a result, any application software was enabled to access remote files in the same way as local ones without any modification.

Thus, the way to be a distributed file system was step by step. Throughout the revisions and extensions, the primary constraint imposed on us is to keep compatibility with the existing software, so that they will be available without any modification. This problem was solved with the class inheritance mechanism of ESP. All the revisions were absorbed by reconstructing, renewing or newly defining super classes of the existing classes.

The system has been in operation since summer 1987 and made available for the above two different communication protocols in spring 1989.

The rest of the paper is organized as follows: Section 2 clarifies the assumptions and policy taken in the distributed system. The detailed design is described in Section 3. Section 4 examines the design in both functionality and performance, comparing our system with other distributed file systems and distributed object-oriented systems.

2 Requirements

The design of a distributed file system is determined by what kind of properties are required for its usage environment. These properties are tightly related to each other.

Assumptions: We clarify the assumptions and constraints that arose while extending the SIMPOS file system to make it a distributed file system.

- *Heterogeneity.* The PSI network is physically a heterogeneous network in that PSI machines and other kinds of machines are connected to each other. The SIMPOS distributed file system was designed mainly as a homogeneous system to deal only with PSI files, but it provides an object-oriented abstraction to deal with other non-objected-oriented systems' files in almost the same way as PSI files at the primitive interface level. Only the homogeneous aspect is described in this paper.

- *Scalability.* It was assumed that the number of PSI machines joined to the network would increase moderately. In fact, since the SIMPOS distributed file system started running to date, the number of PSI machines tripled, from one hundred to more than three hundred.

- *User Mobility.* The individual PSI users or groups have their own PSI machines to keep their own files in their machines, so they use their machines in their own offices or laboratories, do not move often from one machine after another. User mobility was given high priority.

- *File Mobility.* Since each PSI machine belongs to an individual user, the entire disk space over the network is not assumed to be such a shared space that the storage load should be balanced by implicitly migrating files from one machine to another.

- *Machine Mobility.* It is not so rare to move each PSI machine from one place to another following its user, mainly for room rearrangement and laboratory reorganization. Independence of each machine was of importance.
- *Security.* The PSI network has been used internally by researchers working on the fifth generation project. Security was required to protect files from careless and unintentional destruction, rather than from the invasion of malicious users.
- *Fault Tolerance.* For internal faults within a machine, it is left to the users' responsibility to take a backup of their own disks. The underlying network system was assumed to be reliable enough.

Summarizing the above, the distributed environment we assumed is a cluster of completely independent machines whose connection is very loose, not a totally unified space which regards the CPU and I/O resources of each component machine as part of the entire CPU and I/O resources.

Primary Policy: To keep independence of users, machines and files, we took the following policy in the design:

- *No Centralized Control.* There are no centralized mechanisms to manage users and files; there are no centralized machines; all the machines are treated equally; every machine can be a server as well as a client.

3 Design and Implementation

In this section, we describe in detail the design of the SIMPOS file system. Before going into detail, we summarize characteristics of the description language, ESP, especially its object-oriented features which effected greatly the design and implementation of the file system. Then, the storage management is outlined and the basic mechanisms for concurrency control, access control and communication control are described.

3.1 Object-Orientation

ESP is a sequential object-oriented logic programming language based on Prolog [12]. It contains logic features and object-oriented features together. For the former, unification is used as a parameter passing mechanism, and backtracking as a control structure. For the latter, data encapsulation and multiple class inheritance mechanism are supported as follows:

Data Encapsulation: ESP objects are *static objects* in that they only encapsulate a set of internal states stored in *slots* and offer, as their interface protocol, a set of *methods* to manipulate the internal states, whereas *dynamic objects* are supposed to hold an activation record as a unit of concurrent execution. Thus, no notion or mechanism for concurrency is contained in ESP itself.

It is SIMPOS which introduces dynamic objects. A class, *process*, is provided to create process instances, each of which holds an activation record. The kernel of SIMPOS schedules the process instances based on a non-preemptive scheduling method, not on a time-slicing method.

Note that, according to the strict definition of object-orientation that there should be nothing other than objects, the object-orientation of ESP is not complete, since primitive data such as integers and symbols are not objects.

Multiple Class Inheritance: Each object is an instance of a class. Each class can inherit a set of methods and slots from more than one class, which is called its *super* class. The multiple class inheritance mechanism of ESP is powerful like Flavors [24].

An ESP method is a predicate defined for an object, such as

```
:add_data(Accessor, String) :-
     :put_data(Accessor, String),
     :add(Accessor, immediate) ;
```

where the first argument (`Accessor`) means an object. A method is identified by the combination of its message name and the number of arguments, such as `:add_data/2`. When this method is invoked, two goals: `:put_data/2` and `:add/2` are executed sequentially.

For one method identifier, three kinds of predicates: *principal* predicates, *before demon* predicates and *after demon* predicates, may be defined in a class and its super classes and make a logical consequence composed of AND-combinations and OR-combinations. This method inheritance mechanism enables incremental programming as well as generic programming.

Those date stored in slots are side effects, to which destructive assignment is allowed. Two kinds of slots are provided: one is private slots which are visible only in their class, and the other is public slots which can be shared by child classes.

In the file system, we have made much use of the method inheritance mechanism. For the following basic mechanisms, many classes are defined and inherited by file and directory classes. For example, class `directory` inherits forty classes. The multiple class inheritance mechanism has greatly contributed to reduce the development cost of the file system.

3.2 Storage Management

The structure of a local file system is similar to that of the UNIX [2] file system [23].

[2]UNIX is a trademark of Bell laboratories.

- *Single Size Blocks.* A disk volume is physically partitioned into 1K byte pages, and logically managed with four pages as an allocation block [3].

  This volume management is much simpler than the UNIX 4.2 BSD file system which supports cylinder groups and two different sizes of blocks in a volume [23].

- *Expandable Files.* Those which are stored in it are dynamically expandable files whose allocation size is not specified at creation.

- *Various File Types.* Three types of files are managed: *binary files* which are text files, *table files* which are fixed-length record files, and *permanent object files* which are, like typed files in Apollo [19], frozen images of objects in main memory.

- *Tree-Structured Name Space.* Each file is given a numerical identifier for its physical data like an i-node number in UNIX, and is linked with more than one symbolic name in a tree-structured name (directory) space whose top is called the *root directory*.

  A file name under a directory consists of an *identifier*, an *extension*, and a *version*, such as "`humm.txt.3`". A file is uniquely addressed in the local file system by its pathname, such as "`>sys>user>gee>humm.txt.3`", where the first "`>`" stands for the local root directory.

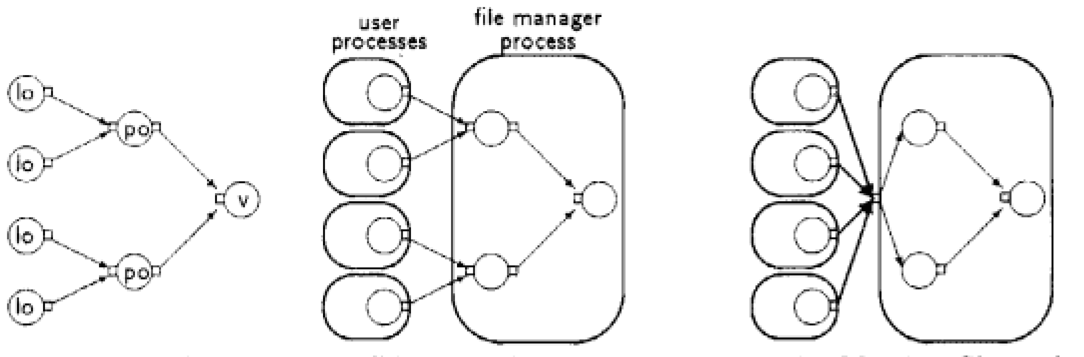
- *Versioning.* The version ("3" in the above example) is assigned by the file system and incremented every time a file is newly created. If only an identifier and an extension are specified in opening a file, its latest version is retrieved. Version control is a matter of name binding, not automatic logging. The file content of each version is mutable, unlike immutable files in Cedar [37]. Different versions are preserved until the user explicitly purge them.

- *World.* The notion of a *world*, which is a set of aliases, is introduced. An alias may be defined for any pathname. For example, by defining an alias, "`snoopy`", for the directory, "`>sys>user>gee`", the file, "`humm.txt`", can be referred to as "`snoopy:humm.txt`".

- *On-Close Writing Policy.* The data transfer between the main memory and I/O devices is done via an I/O bus memory. A partition of fixed length is allocated to the disk device handler, and used as a cache for disk I/O. The cache is controlled by the device handler based on a simple LRU management algorithm.

  Input data is transferred from the disk unit to the cache and copied to a user's buffer. Output data in a user's buffer is transferred to the cache. Updates on

[3]The block size was one page in the original design, and was changed to four pages later, because, according to our statistic results, about 50% of files are smaller than 4K bytes in most PSI machines.



(a) Ideal (b) Fusing Processes (c) Merging Channels

Figure 1: Optimizations of a dynamic object model

administrative information, such as the i-node table and free block map, are accumulated in the main memory and written to the cache every time a file is closed. At the end of the closing, all the data in the cache is written back to the disk.

3.3 Global Naming

Among those changes introduced for the SIMPOS file system to be a distributed file system, the only one noticeable by PSI users is that user names and file/directory names are globalized.

User Name: Users are locally identified by their loin names by the user management system at their own site. There is no notion of a user group. There is no group for a user to belong to or for a resource to belong to, either. However, a similar function can be realized with passwords. With a common password, users can be grouped. With different passwords, a single user can belong to several groups.

File/Directory PathName: In each machine, files and directories are addressed with their pathnames from the root directory, such as "`>sys>user>gee>humm.txt`", which are called *local pathnames.*

Global Name as Combination of Host Name and Local Name: User names and file/directory pathnames are globalized just by combining their host name and their local name, as $< HostName >::< LocalName >$, where a host name identifies a machine not a geographical position.

3.4 Concurrency Control

To control concurrent accesses to a shared resource, we separate a *logical access,* which is a sharer, from a *physical data,* which is a shared resource. The former is called a *logical object* and the latter a *physical object.* For every resource to be

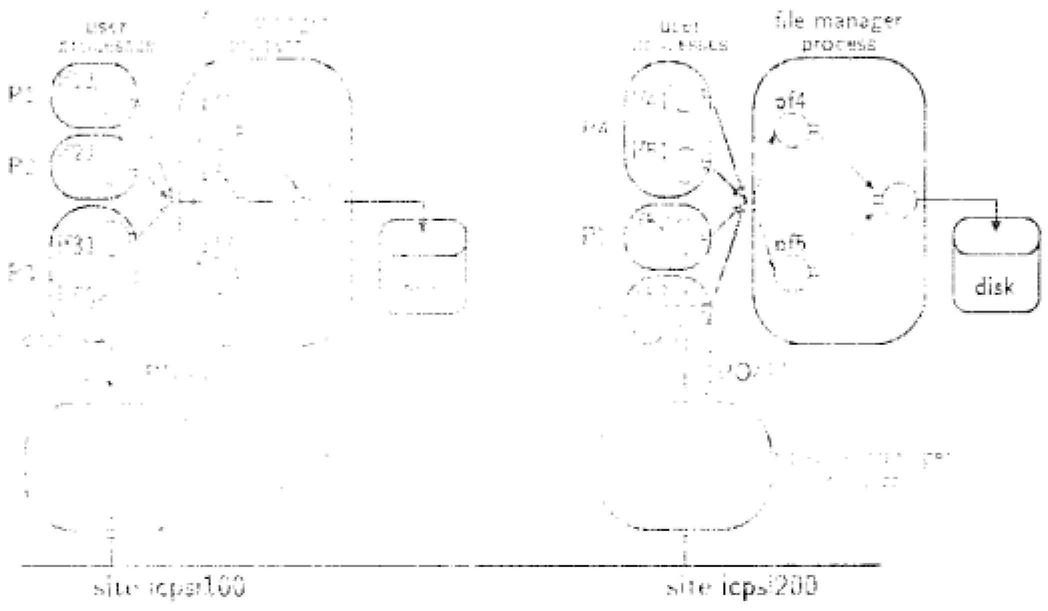


Figure 2: An overview of the SIMPOS distributed file system

shared, including a file, a directory and a volume, there exist one or more logical objects and a physical object.

Ideally, each physical object should be a dynamic object as shown in Figure 1-a. PSI-I machines, however, limits the number of process creations only up to 63. Because of this limitation and also for higher performance, this model has been optimized as follows: all dynamic objects are fused into one process, called a *file manager process* (Figure 1-b); further their communication channels are merged into one (Figure 1-c). The resulting model is similar to the model of scheduling lightweight processes (or *threads*) under a full-fledged process, that is commonly used by most distributed object-oriented systems in their implementations on top of UNIX.

Figure 2 shows a view over the network. The symbols: lfxx, pfx, Pxx stand for logical file objects, physical file objects and user processes. At site icpsi100, processes P1, P2 and P3 are accessing local files pf1, pf2 and pf3 via logical files lf11, lf21 and lf31. Process P3 is also accessing a remote file pf5 at site icpsi200, via a logical file lf53' at the local site, and further lf53 at the remote site. At site icpsi200, three processes P4, P5 and P6 are sharing a single physical file pf5. Process P4 is accessing pf4, too.

3.4.1 Interface Primitives Based on Asynchronous-Communication

Session: A file access starts with invoking a `open` or `make` method of a file interface class object, such as `#binary_file` and `#table_file`, to obtain a logical file, and ends with invoking a *close* method to the logical file object. The duration is called a *session*. During the session, each file access operation is done by invoking an interface method, such as `read`, `write` and `add`.

For example, the following is a program to make a new file, named "`>sys>user>gee>humm.txt`", and append a string "`To be, `" to the file:

```
:make(#binary_file, File, ''>sys>user>gee>humm.txt''),
:add_data(File, ''To be, ''),
:close(File),
```

Asynchronous communication: The first role of a logical object is communication, whose network-transparency is described later.

Communication between a logical file and its physical file is asynchronous, which consists of the following primitive operations: `submit` to (non-blocking) send a *request*, and `wait` to wait for a *reply*, and `probe` to probe if a *reply* has returned. The correspondance between a request and a reply is made by their specifying a common message identifier. For every I/O operation, a synchronous method and an asynchronous method are prepared, such as `:add` and `:submit_add`. A synchronous one is expanded to consecutive `submit` and `wait` primitive operations.

The asynchronous communication feature raises independence of a user process and the file manager process, since a user process can run during the submit and the wait, while the file manager is suspended. This feature has been effectively used for double-buffering. Compared to the performance with synchronous communication, about 20 % improvement could be gained.

Data Caching: The second role of a logical object is data-caching.

The I/O bus memory between the main memory and disk unit is a cache of fixed length and invisible to users. Instead, every I/O operation asks two arguments: a *buffer* holding a buffer area of variable length, and a *marker* holding an access position, to be explicitly specified. Note that `read` and `write` are random access operations to read and write data at the specified position, while `add` is sequential access operation whose access position is kept by the physical file.

This interface level is fully user-controllable but too low for end-users. A higher level interface is offered to them: a logical file has a pair of a buffer and a marker, which is called an *accessor*, in it, and also allows to attach other external accessors to it. With this mechanism, most of the users do not need to handle the low level buffer.

For each request message, references to a buffer and a marker, not their contents, are passed from a logical file to a physical file. In invoking synchronous operations in a row, the internal accessor can be used from one operation to another. However, in case that more than one asynchronous operation is invoked in a pipelined manner, the content of the accessor is not guaranteed, since different request messages might share an identical accessor. This problem can be solved by preparing external accessors for overlapping operations as follows:

```
:open(#binary_file, File, ''>sys>user>gee>humm.txt'', shared, immediate),
:accessor(File, Acc),              % attach an external accessor
:put_data(File, ''or ''),          % put ''or '' in internal accessor
:submit_add(File, ID1, suppress),
```

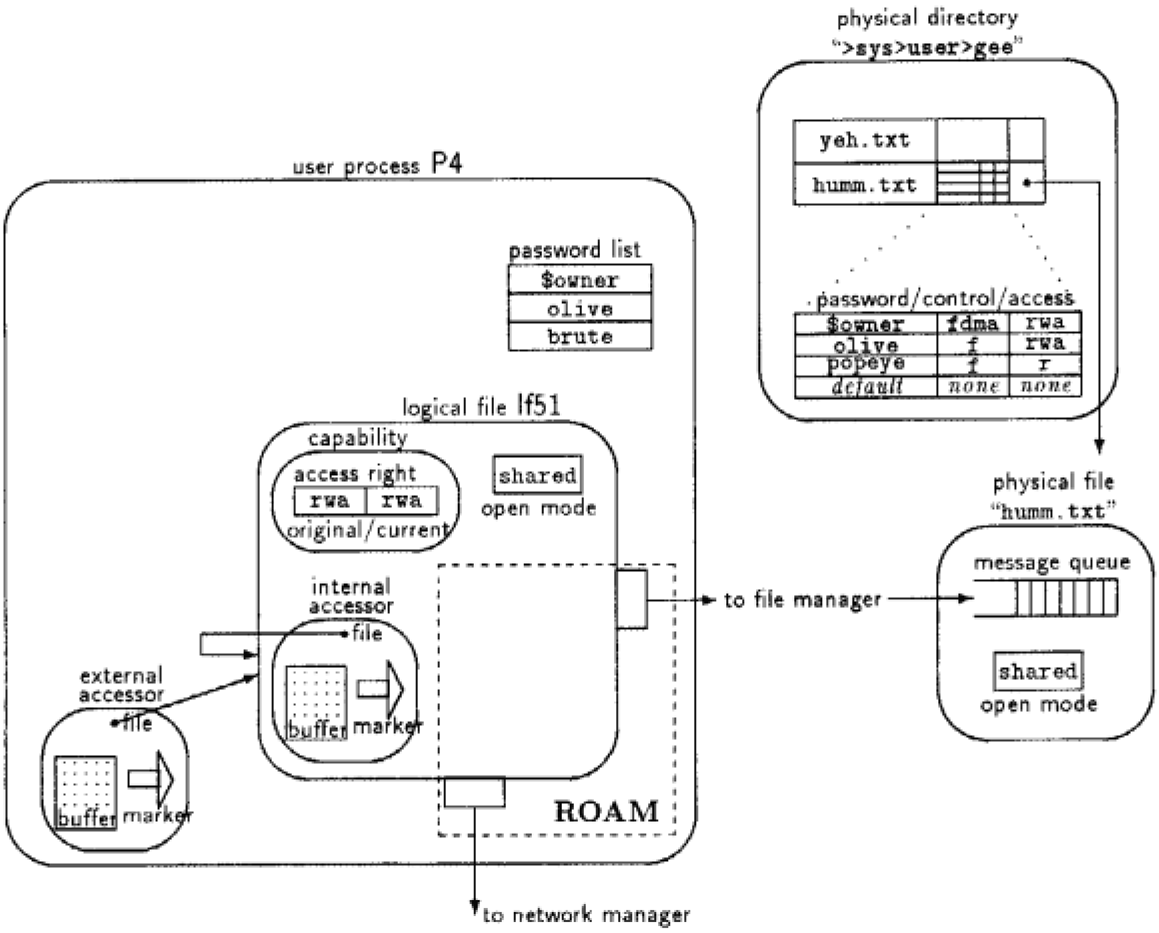


Figure 3: Structural details around logical file lf51

```
      :put_data(Acc, ''not to be.''), % put ''not to be.'' in external accessor
      :submit_add(Acc, ID2, suppress),
   :wait(File, ID1),
   :wait(Acc, ID2),
```

In a remote access, each accessor can be used as a cache to the primary data at the remote site. Since the buffer size is flexible, the whole file copy is possible as well as partial reads and writes.

Figure 3 shows the above situation in which a user process opened a file addressed "`>sys>user>gee>humm.txt`", and obtained a logical file, named lf51 in Figure 2.

3.4.2 Concurrency Rule

Sequentiality and concurrency of file accesses are controlled every time a file is tried to be opened and closed. When a file open is allowed, a user process is returned a logical file.

Open Mode: The following four kinds of logical open modes are provided:

Exclusive (Single Reader and Single Writer): A logical file is allowed to open when its physical file is closed. During the session, both reads and writes are allowed.

Input (Multiple Readers and No Writer): A logical file is allowed to open when its physical file is closed or open in the input mode. During the session, reads are allowed, but writes are prohibited.

Output (Multiple Readers and Single Writer): A logical file is allowed to open when its physical file is closed or open in the shared mode. During the session, this logical file is allowed both reads and writes, but suppresses all write operations from other shared-mode logical files.

Shared (Multiple Readers and Multiple Writers): A logical file is allowed to open when its physical file is closed or open in the shared mode or the output mode. During the session, both reads and writes are allowed but the above mentioned suppressions of write requests might occur.

For each physical file, one logical open may suppress other logical open or write requests against its open mode; their suppressions are released when the logical file is closed.

Suppose that another process, P5, at site icpsi200 opened the same file in the output mode concurrently with the above process, P4, and issued two output operations to the obtained logical file, lf52, as follows:

```
:open(#binary_file, File, ``>sys>user>gee>humm.txt'', output, immediate),
:add_data(File, ``That is ''),
:add_data(File, ``the question! ''),
:close(File),
```

Since lf52 is open in the output mode, the two output operations are guaranteed to be continuous. In contrast, since lf51 is open in the shared mode, after lf52 opens, all the output requests, such as `:submit_add`, from lf51 will be suppressed. Depending on the time when lf52 is opened, the file content will result in either "`To be, or not to be. That is the question! `", "`To be, or That is the question! not to be. `", or "`To be, That is the question! or not to be. `".

Request Mode: Even when a request has to be suppressed, two choices are left to the user: withdrawing the request or waiting until the suppression is released. They are called *request modes*, and specified `suppress` and `immediate` at opening/making a file and submitting every request.

3.4.3 Message Management

The suppression and release of requests are controlled by physical objects under a file manager process.

Each physical object has a message queue which keeps all the suppressed requests in their arrival order. A file manager is just a scheduler of physical objects. All the messages from logical objects are sent to the file manager. Each message carries the information of its sender (logical object) and destination (physical object). Every time a file manager receives a message from a logical object, it delivers the message to the corresponding physical object.

Message Priority: Each message is given a priority of two levels: *normal* and *express*.

- *Express Messages.* When receiving an express message, the physical object immediately execute it.
- *Normal Messages.* When receiving a normal message, the physical object takes the following actions, according to the service order rule mentioned below:
  1. If the request is acceptable, it immediately executes the request.
  2. If the request must be suppressed and the specified request mode is `suppress`, that is to wait, it enqueues the request to its own message queue.
  3. If the request must be suppressed and the specified request mode is `immediate`, that is to withdraw, it immediately returns a reply message notifying the withdrawal.

  The service order of normal requests issued to a physical object is determined not only by its current open mode but by the arrival order of requests enqueued in its message queue. If there are already suppressed requests, even if a currently coming request is acceptable under the current open mode, it is also suppressed.

Express messages have been effectively used when a user process is aborted. When a user process is being aborted, each of those logical files which belong to the user process issues an *abort* message to the file manager. When a physical file receives an abort message, it clean up all suppressed requests issued from the logical file and to release all related suppressions caused by the logical file open. We learned that, for a stream-based control model, message priority is of great importance.

3.5 Access Control

The primary assumption made in designing the protection mechanism is that most of the PSI users are conscientious. Rather than excluding malicious invasions, protecting resources from their careless or unintentional destruction was the purpose of the protection mechanism. Hence, we aimed at making the protection mechanism transparent, so that those conscientious users can access resources without being aware of the existence of the protection mechanism.

The protection mechanism is divided into two phases,

- a *capability-granting phase* to grant to a user process a logical object with a capability given according to the passwords of the user process, and
- a *capability-checking phase* to check the capability of a logical object every time an operation is issued to the logical object.

and in both phases all the access control is implicitly done without any interaction with the user.

3.5.1 Capability-Granting

The basic idea underlying our capability-granting mechanism is to regard each directory entry as a protection gate, a password as a key to open the gate, and each user process as a holder of passwords. A protection gate may accept more than one password from a user process, it opens differently depending on what password has been applied.

User Process Side: Each user is given a right according to their rank and passwords. When a user tries to open or newly create a file or retrieve a directory by invoking `open`, `make`, or `retrieve` method, the user information is carried over though the capability-granting phase. For remote access, the user information is implicitly sent at initiating the connection by a remote object access mechanism. Details are mentioned later.

- *Password Holder.* A user has a *password holder* containing up to ten passwords, each of which is a string of at most sixteen characters. In addition to these explicitly defined passwords, every user is implicitly given a *default password* mentioned later.

  A user process is given at its initiation a copy of the password holder of the user who initiated the process, and the copy may be modified during the process life. Hence, those processes initiated by the user may have different password holders.

  A user process must show its own password holder every time it goes through a protection gate on the way to reach a target file or directory.

- *User Rank.* Users are categorized into two: *super users* and *general users*, according to their privilege. Super users are those who have a privilege to skip the access list check step in the capability-granting phase and to get full permission.

  At registration, each user is given a *permanent rank* which is leveled into four: `super_U` (privileged over the entire network universe), `super_A` (privileged in the resident network area), `super_N` (privileged at the resident site) and `general` (non-privileged). Super users are temporarily ranked down to general ones when they try to access out of their resident region: `super_N` users on accessing out of the resident site and `super_N` users on accessing out of the resident area. Also, super users may temporarily level themselves down to general ones. The temporary rank, either **super** or **general**, is called a *user mode*.

Directory Entry Side: For each directory entry, its *network publicity* and an *access list* are specified to protect itself.

- *Network Publicity.* Either `g` (global) to allow remote users as well as local users to access the directory entry, or `l` (local) to allow only local users to access the directory entry. If the network publicity of the root directory is set to `l`, the files and directories under the root directory are accessible only by the local users but no remote users.

- *Access List.* An access list is tuples of $< Password, ControlRight, AccessRight >$.

  Three kinds of passwords can be specified: one *meta password*, **$owner**, representing the globally unique login name of the resource owner, up to three user-defined passwords, and one *default password* given to everybody. Owing the meta password function, users can access their own resources without being aware of the existence of the protection mechanism.

  The *Control Right* is a meta-level right to handle the directory entry itself, which is composed of four permissions: `f` (to find the directory entry), `d` (to delete and rename the directory entry), `m` (to modify the right of the directory entry) and `a` (to append passwords of the directory entry).

  The *Access Right* is a right to access the physical file or directory object pointed from the directory entry, which is composed of three permissions: `r` (to read the file; to look up the directory), `w` (to overwrite data to the file; to delete directory entries from the directory) and `a` (to append data to the file; to register new directory entries to the directory).

Capability-Granting Algorithm: The following three-level checks are implicitly done for every directory entry on the way to the target object:

1. *Publicity Check.* Check if the network publicity to the directory entry accepts the user. If so, proceed to next.

2. *User Rank Check.* Check if the user rank is **super**. If so, grant a full capability and skip next.

3. *Access List Check.* Check the password holder of the user process against the access list of the directory entry and return a logical object for the resource that the directory entry addresses. The logical object is granted a total sum of the capabilities given for the accepted passwords, which is called a *original capability*.

   If this check is successful, proceed to the next directory entry until it reaches the target object.

When all checks result in success, the user process is returned a logical object for the target. As for a logical file, its original capability might be limited according to its open mode. For example, write operations are restricted in the input open mode, so w and a must be inhibited. The limited capability is called a *current capability.*

Note that in the actual implementation, a logical object is not created for each directory entry, by having all the checks done by physical directory objects under the file manager process.

3.5.2 Capability-Checking

The third role of a logical object is capability-checking.

Every time a primitive operation is invoked to a logical object, the logical object checks if it can execute the operation or not, according to its own current capability. If allowed, it really executes the operation.

Figure 3 shows a situation just after process P4 has reached a directory entry for a file ">sys>user>gee>humm.txt" which is owned by some other user. Since the directory entry accepts a password, olive, the user process has obtained a logical file, lf51, with rwa as the original capability. The open mode was shared, so the current capability is the same as the original capability, which will allow to invoke the subsequent add operations.

3.6 Communication Control

The communication between a logical object and its physical object is network-transparent. Once a user process obtains a logical object, there is no distinction visible to the user process whether its physical object is local or remote.

Let us execute the following operations in process P3 at site icpsi100,

```
:open(#binary_file, File, ''icpsi200::>sys>user>gee>humm.txt'',
                                                exclusive, suppress),
:read_data(File, String),
:close(File),
```

then we will obtain a logical file, File, shown as lf53', and the file content in String.

This network-transparent access has been realized by basing the SIMPOS distributed file system on a remote object access mechanism (ROAM) [47].

3.6.1 Remote Object Access Mechanism (ROAM)

The ROAM is a mechanism which has been developed on top of the session layer of the SIMPOS network system, to enable the invocation of method calls to remote objects as well as to local objects.

Principle: The ROAM is based on the principle of *object and process reflection*, that is to keep a symmetric relation between an object and its owner process at the remote site as well as at the local site. When a process, say a *client process*, tries to access an object at a remote site, a corresponding process, called a *server process*, is created at the remote site. For each client process, one server process is created at every site which the client process accesses. While the server process accesses the target object, say an *original object*, at the remote site, an corresponding object, called a *proxy object*, is created at the local site and given to the client process. Original objects and proxy objects have no difference in their structure. They belong to the same class and recognize themselves whether they are original or proxy. Original objects execute actual operations, while proxy objects forward every operation request to their original objects and do not any actual operation themselves.

In Figure 2, process `P6` is a server process for the client process, `P3`, and `lf53'` is a proxy object for the original object, `lf53`, The relation between `P3` and `lf53'` is symmetric to that between `P6` and `lf53`.

Function: The ROAM has been designed for a general purpose not specific to the file system. It provides the following general functions:

- *Message Management.* For each method call, a request message is sent from a client to its server and a reply message back from the server to the client. The message management is to package and unpackage the request and reply messages of a method call. A message is packed in tagged representation into a sequence of communication packets of fixed length, so a message may be of any length and those data and objects contained as its arguments may be of any type, any class and any binding direction: either input or output. Also, it is allowed to send extra information in addition to the method information in each message.

- *Line Management.* When a client process tries to access a remote site for the first time, a server process is created at the remote site and a virtual circuit is connected between the client and the server. The virtual circuit is cut off when either of the client or server is terminated. The user information is implicitly sent from the client to the server at the initiation of a virtual circuit.

- *Object Management.* The export and import of objects, to make correspondance between original objects and proxy objects, is managed. It is allowed to propagate an object reference from one site to another and to make a nest of remote method calls, that is to invoke other remote method calls back and forth between the server process and the client process during executing one remote method call.

Interface: The ROAM offers a very simple interface based on class inheritance. Network-transparent objects can be defined as follows:

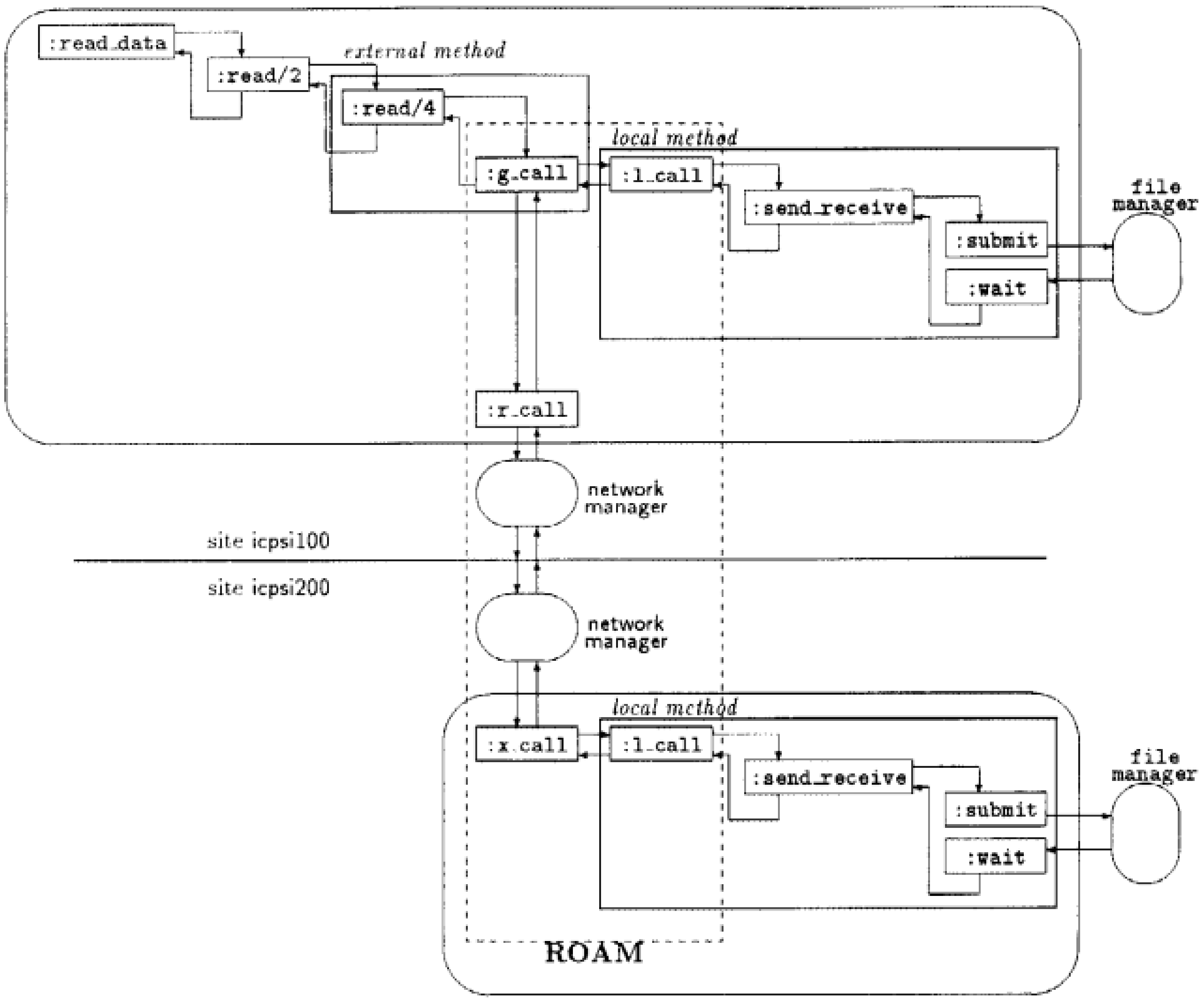


Figure 4: Calling sequence of method `:read_data`

1. Inherit one of the two ROAM kernel classes:

   - Inherit class `remote_object` for defining a *complete global object* which may be a destination object of a remote method call.
   - Inherit class `as_remote_object` for defining an *incomplete global object* which is only carried as an argument of remote method calls to other objects. For each incomplete global object, a proxy object and an original object make a pair as well as a complete global object, but only local method calls are invoked to them at their own site. To synchronize their contents with each other, the internal states of an incomplete global object may be frozen and delivered as extra information in request and reply messages of the remote method calls.

2. For each method, define two kinds of methods using the offered ROAM interface methods:

- an *external method* which is an entry of the method entry. This method must be defined to execute a global method call, `:g_call`, which will issue either a remote method call (`:r_call`) or a local method call (`:l_call`) depending on whether the object is a proxy or an original.

  In case of a proxy object, when a request message arrives at the remote site, its server process issues an external call, `:x_call`, to its original object, which will issue a local method call, `:l_call`, there.
- a *local method* which defines the body of an actual operation that an original object should execute.

Figure 4 shows the calling sequence of a local method call and a remote method call with an example of method `:read_data`.

3.6.2 Application to the file system

We embedded the ROAM in the logical objects and their related objects. Appendix A shows some of those methods which have been referred to in this paper.

Although we could regard every object as a complete global object with the ROAM, we took the following strategy for higher performance:

- Only logical files and directories were implemented as complete global objects, by inheriting class `remote_object` and defining external methods and local methods. (See class `binary_file` and its supers.)
- Those related objects, such as buffers and markers, were implemented as incomplete global objects, by inheriting class `as_remote_object` and defining what attributes should be delivered as extra information in which kinds of messages. Their internal states were delivered in request and reply messages of file I/O operations, only when necessary. For example, the content of a data buffer was delivered in the request messages of output operations and the reply messages of input operations. (See class `buffer` and `remote_binary_file`.)

4 Evaluation

In this section, we compare the SIMPOS distributed file system others from two aspects: one from a distributed file system and the other from a distributed object-oriented system, to examine its functionality and performance.

4.1 As a Distributed File System

There are a number of distributed file systems already in operation, many of which are based on UNIX. Table 1 summarizes a comparison of the SIMPOS file system with the following representative distributed operating systems and file systems:

- Apollo Domain [19], a distributed operating system for a ring network of Apollo workstations, marketed by Apollo Computer, Inc. It primarily features a network-wide single-level storage management which regards all storages over the network as a single virtual memory. Communication takes place as part of the virtual memory management.

- Locus [41], a UNIX-compatible distributed operating system kernel, developed at UCLA. It aims at a high degree of network-transparency of file access and process execution. For reliability, file replication and nested atomic transactions are supported.

- V [9, 10, 11], a message-based distributed operating system kernel, developed at Stanford University. It provides efficient general-purpose IPC primitives based on message-passing, which is tuned for remote file access.

- Sun NFS (the Network File System) [31], a network file system interface on UNIX, marketed by SUN Microsystems, Inc.

- Andrew [32, 16, 36], a distributed computing environment for a large scale network composed of thousands of UNIX workstations, being developed at CMU.

- Sprite [43, 28], a network operating system being developed for SPUR, a high-performance multiprocessor workstation, being developed at UCB.

Naming Control: Pathnames in SIMPOS are not location transparent, in that site information is emdedded in them, as in Ibis [40] and the Cedar File System [37]. In UNIX United [?], whose primary goal is to keep complete UNIX semantics, the top "`/`" still is regarded as a local root directory and "`/../`" as a parent directory of the local root directory. A site name is not explicitly specified, but pathnames give some hint about storage site, so it is not location transparent, either.

Many UNIX-based systems, such as Locus, Sprite, NFS and Andrew, support *remote link* to mount an remote entire file system or a subtree onto a local directory. Remote mount is convenient, but consistency and security of the mounting is left to the responsibility of system administrators. The mount information must be copied or cached at each client site. The possible scale is limited. Pointing out this problem, rather than the remote mount mechanism, a global name service is provided in V [11].

In SIMPOS, there is neither remote mount nor global name service. Instead, the world mechanism played an important role. With the world mechanism, aliases can be defined for remote sites, directories and files as well as local ones. For example, if an alias, "`snoopy`", is defined as "`icpsi200::>sys>user>gee`", the file, "`humm.txt`", can be referred to as "`snoopy:humm.txt`" at any remote site. Even though name resolution is statically done, the world mechanism is very useful. By defining aliases for those which are often accessed, we did not feel so much problem on the lack transparency.

Concurrency Control: When to make updates by one accessor visible to other accessors is called *consistency semantics*. Many systems are taking one of the following semantics:

- *UNIX semantics*, in which any update is reflected spontaneously so that all shares keep a single access position.
- *time-dependent semantics*, in which updates are reflected in some delay and each sharer may access at an independent position.
- *session semantics*, in which updates are invisible until a file is closed.

For Andrew, scalability is a dominant factor. To raise performance, the session semantics and the *whole file copy* strategy are taken. When a remote file is opened, the whole file content is transferred and updates during the session are sent back at the closing time. This strategy is disadvantageous in such a case as updating a large scale database; imagine if the whole database of 100M byte long must be copied just to update a single record of 100 byte long! Hence, the file size is limited to a few mega bytes long.

In Locus, write-sharing of UNIX semantics is supported using a token mechanism that is to give each client a token which marks which copy of a resource is valid: access to a shared resource requires the token. Client-server communication to maintain tokens limits the performance.

SIMPOS supports all three kinds of semantics. Input, output and exclusive opens realize session semantics, read and write operations in shared opens imply time-dependent semantics, and add operations in shared opens keep UNIX semantics. These open modes has been very effectively used in various applications software. The combination of output and shared opens enables time-dependent single-writer multiple-reader semantics; it has been used by debugging tools in which one process generate a log and another monitors. Input and exclusive opens guarantee session semantics; they have been used in database retrieval and update. Moreover, since a single read operation can deal with the whole data or a part data of a file, the user, who is supposed to know most about their own application, can control how much to copy to save wasteful communication. For the UNIX semantics, the current position is kept only by the physical file at the server not by logical files at client sites, so there is no communication overhead for the maintenance unlike in Locus.

Access Control: Andrew pays great attention to security and protection, assuming malicious users over the large scale network. An access list may be defined per directory, so all entries under a directory are given the same protection. Entries of an access list are user or group identifiers and users may be grouped hierarchically like in Grapevine [8].

In SIMPOS, an access list may be defined per directory entry, so each directory entry may be given a different protection; the protection is not per file, so for the same physical file a different protection can be given for each possible path to it: fine control is possible. There is no notion of user group in SIMPOS; instead, the password holder mechanism could play a part of user-grouping without any group database management.

Communication Control: In Apollo and Locus, which are both distributed operating systems, network communication is done inside the kernel. Apollo primarily features a network-wide single-level store management which regards all the disk storages over the network as a single virtual memory. Remote disk and memory access is done as part of the demand paging function of the virtual memory management. In Locus, all interprocess communications are done inside system calls, including those on file access and process creation, using a lower level protocol specialized for file access. In V, message-based general-purpose IPC primitives are provided by the kernel and an I/O protocol is implemented on top of that.

The rest of the systems use RPC mechanisms for network communication. The function of RMC (remote method call) mechanisms, such as ROAM, is very similar to that of RPC (remote procedure call) mechanisms [27, 33, 34, 42]. Such layer that realizes a network-transparent procedure or method call is called the *stub layer* and its interface program, such as `:g_call` and `:l_call`, is called a *stub* [27].

For the development a distributed file system, basing it on a stub layer, whether it is a RMC mechanism or a RPC mechanism, surprisingly reduces the development cost. Benefits experienced in the Alpine file system developed on top of the Cedar programming environment using Cedar RPC are reported in [7].

Similarly, the ROAM benefited us at the following points:

- Network-transparent access could be implemented with very little effort. Only 2.7K lines of ESP code were added, which is about 10% against the entire code of the previous local file system.

- The maintenance of the entire file system has been simplified, mainly because of the code compactness.

- Later extensions on the file and directory interface, such as adding new I/O primitive operations and modifying the existing ones could be very easily done, just by adding or modifying their external interface methods and local interface methods. We could be completely free from details about network communication; no additional communication protocol were needed to be designed.

- When the distributed file system was designed, in 1987, the ROAM was available for the PSI-NET communication protocol. Later in 1989, the ROAM was made available for the TCP/IP communication protocol, too. A remarkable thing is that the file system ran for both of the different protocols without any effort!

  At present, each PSI machine has connected at least one of the two network controllers to support these protocols. Within the ROAM embedded in each logical object, which protocol is available between the client node and the server node is checked, and communication packets are transmitted according to the chosen protocol. In case that both protocols are available, the PSI-NET protocol is selected.

In the implementations of the above RPC mechanisms: we can see the following differences than the ROAM:

- (UNIX) process switching is costly. To raise utilization of the server site, one server thread per client is created, rather than one server process per client, and a cluster of threads is executed under a single or a few processes at the site.
- If a process of the cluster of server threads keeping internal states on communication is downed, the damage would be large. To keep clients and server stateless concerning communication, the datagram mode is adopted, rather than the virtual circuit mode, as the underlying communication mechanism.

These implementation techniques are closely related with the difference between a method call and a procedure call.

The major difference between a method call and a procedure call is whether their semantics are determined dynamically at execution time or statically at compile time. A method code is dynamically looked up from the method table of a destination object, while a procedure code is statically bound. In a method call, each argument may be an object reference of any class or data of any type; their class or type is unknown until the execution time. In a procedure call, the data type of each argument is specified in the program, so RPC mechanisms provide stub translators which generate stub programs statically. RMC mechanisms have to dynamically check data types and object classes, and to manage the export, import and garbage collection of objects. The object management differentiates RMC mechanisms from RPC mechanisms.

For the object management, we have to choose a stateful client-server model. To keep it reliable, we selected the virtual-circuit mode. To simplify garbage collection of objects, the ROAM assigns one export/import table to each process, not to each site. When these processes are terminated, the table is collected by the underlying garbage collector. Thus, the design of the ROAM is very simplified without losing reliability.

4.2 As a Distributed Object-Oriented System

There have been developed several RMC mechanisms and distributed object-oriented systems, categorized as follows, most of which are still at the stage of being experimental systems, not have been operational as practical systems yet. Table 2 summarizes a comparison of SIMPOS with these systems.

1. *Independent Distributed Object-Oriented Systems*
    - Eden [1, 2, 5], an experimental distributed object-oriented system, developed at University of Washington.
    - Emerald [6, 18], an experimental distributed object-oriented language system, being developed at University of Washington.

2. *Family of Extended Smalltalk*

    - DOM(Distributed Object Manager) [14], an experimental distributed Smalltalk system, being developed at Universite de Grenoble.
    - DS(Distributed Smalltalk) [4], an experimental distributed Smalltalk system, being developed at University of Washington.

   Another model similar to DS is proposed in [22].

3. *RMC Mechanism on Object-Oriented System*

    - ROAM on SIMPOS. Since the ROAM kernel class, `remote_object`, is not inherited by class `class` which is inherited by any other class like class `Object` in Smalltalk, SIMPOS is non-distributed object-oriented systems.

4. *RMC Interface on Non-Object-Oriented System*

    - Matchmaker on Mach [17], an RMC stub generator for multi-lingual interprocess communication on Mach, developed at CMU.

5. *Distributed Typed Data System*

    - Apollo, introduced above. The secondary feature of Apollo is its object-based storage system (OSS), which regards each file as a frozen image of abstract typed data. This approach is completely different from the others above in that objects are only abstract typed data, without the ability to communicate.

RMC Implementation Strategy: Both extended Smalltalk's, DOM and DS, adopt a similar proxy-original model: when an object is exported, a proxy object, which is an instance of class `Proxy`, is automatically created; class `Proxy` overrides a method, `doesNotUnderstand`, which is a method to catch all undefined messages, to forward every received message to its original. A slight difference is that DOM's proxies are implemented as part of the distributed object manager which is on the system side, while DS's proxies are full-fledged Smalltalk objects. In their model, all objects are treated uniformly by the same proxy objects which have no knowledge about their original object and work in the same manner for any kind of methods.

In contrast, ROAM is also based a proxy object model, but different from the above in that ROAM's proxy objects have knowledge about their original objects. The proxy class, `remote_object`, is inherited by global object classes, so that proxy objects belong to the same class of their original objects. For each method, an external entry and a local entry are specified. Thus, the ROAM's proxy-original model is less universal and less elegant than the above model, but more realistic for practical use over the network in scale of hundreds of workstations.

Basically, one remote method call may cause additional remote method calls. For example, in the above file system, if we applied straightforwardly the proxy-original model to any objects, including buffer and marker objects, a huge number of fine grained messages would be transmitted. Making it worse would be that the binding direction of each argument of an ESP method is not specified; with the universal mechanism, all arguments would have to be carried back and forth. We wanted to get from performance-sensitive programmers information about arguments as much as possible, which will help in reducing communications. Furthermore, to reduce communication traffic, it was necessary for a proxy to cache a copy of a part of the internal states of its original, so that the proxy could solve some problem by itself without interacting with its original every time. As a result of accumulating these techniques, the SIMPOS file system could raise tolerable performance along with high reliability for practical use as a distributed object-oriented system.

Eden takes a similar strategy to ours, which is based on stub generation; the EPL (Eden Programming Language) compiler generates stub routines which is linked with user programs.

In Emerald, objects are categorized into three, according to their object dependency: global, local and directory objects. Global objects can move freely; local objects move along with other global objects; directory objects are directly represented in other objects. The compiler generates a code to check the *resident* bit and, if it it off, trap the kernel.

For a practical system, object dependency is a very important subject to be studied, in order to reduce communication traffic and raise reasonable performance. In contrast to that ROAM introduced the notion of object category of complete global objects and incomplete global objects, Emerald, which aimes at the scale of 100 workstations, allowed to specify a parameter-passing-mode, either call-by-object-reference, call-by-move and call-by-visit, for each argument. Call-by-object-reference is normal; it delivers an object reference only. Call-by-move and call-by-visit are in common to deliver the data representation of an argument object with the invocation message and to relocate the argument object at the destination site. The argument object remains at the destination site in call-by-move, and returns to the source in call-by-visit.

Object Migration: Object migration is supported in Eden, Emerald, DOM, DS, while it is not in ROAM, since the proxy-object relationship is build only for global objects in SIMPOS.

Global Object Management and GC Strategy: In DOM, DS and Emerald, an object management table is prepared for each site, and the global garbage collection scheme is reference counting in DOM and DS, and mark-and-sweep in Emerald.

In ROAM, an export/import table is given to each process, not to each site and is cleaned up when the process is terminated. Since global GC is not supported, no communication for external reference count management is needed. The life of an object is guaranteed during the synchronous session between client and server processes. In the SIMPOS file system, all cases have been covered in this usage.

Permanent Object Management: In Eden, there is no independent file system; instead, a filing function is absorbed in the object model. Objects are autonomous dynamic objects, which can save and retrieve their *checkpoints* into/from the secondary storage. When a file is opened or a directory is retrieved, a corresponding object (process) is created. The behavior of an Eden's file or directory object is very similar to that of a SIMPOS's file or directory object, but file migration and nested transactions are supported in Eden. Their protection scheme is very similar to ours, too: each directory entry contains a capability to access its physical object; when accessing a directory entry via a directory object, a process is given a capability.

4.3 Performance

Table 3 shows the performance measured on the SIMPOS file system.

Through the revisions and extensions, the SIMPOS file system has been greatly improved in performance as well as enriched in functionality. Compared to the original design, for example, a make operation became three times faster and read and write operations more than twice as fast. As yet, we have not obtained the desired performance that can be competitive with the results of others, such as reported in [28, 16].

The cost of opening and closing a file is 16 ms for a local file and 922 ms for a remote file in SIMPOS, while it is 2.17 ms and 8.11 ms in the Sprite file system which is the top in performance [28]. SIMPOS is 50 to 100 times slower. The read throughput is 420 byte/sec for local and 14 byte/sec for remote in SIMPOS, while it is 3269 byte/sec and 224 byte/sec in Sprite. SIMPOS is about 10 times slower.

As for the underlying network layer, the 1K byte round-trip time is 171 ms in the session layer of the PSI-NET communication protocol in SIMPOS, while it is 2.2 ms in the V kernel [9], and 5.8 ms in Sprite RPC [42].

As for the stub layer, the current ROAM uses 4200 byte physical packets for a conventional reason for the PSI-NET protocol, and a new version using 1024 long physical packets is now being tested. The cost of a null RMC, such as instance method `:do(Instance)`, which carries no parameter and immediately returns at the server site, is 471 ms with the current ROAM and 207 ms with a new ROAM in SIMPOS, while a null RPC is 32.0 ms in RPC2 [33] in Andrew. and 32.5 ms in its extension, MultiRPC [34]. Thus, the overhead of the network and stub layers of SIMPOS is at least 5 times heavier.

Recall that the difference between an RMC and an RPC. We should refer to the performance of other distributed object-oriented systems in literature. In Emerald, a null RMC is 27.9 ms, an RMC with a constant parameter is 33.0 ms, and an RMC with a remote object reference parameter is 61.8 ms, in contrast to 0.019 ms for a LMC, which is almost the same as our LMC cost. In DS, a null RMC is 136.9 ms and an RMC with a constant parameter is 140.2 ms while a LMC is 0.130 ms which is very expensive. As for the single RMC performance, our system is slower than the others. One difference in implementation techniques is that our system, which has been developed for practical use, employs reliable virtual circuits, while Emerald and DS, which are small scale experimental systems, uses unreliable datagrams on UNIX.

Here, we should remind that the total performance is not determined merely by the single RMC performance, but also by the the number of RMC invocations. With their universal proxy-original model and global GC, a vast number of messages will be transmitted, though their total performance measurements are not in literature yet.

The slowness is not derived from the CPU performance: the performance of the PSI-II machine is about 30μs per one predicate inference [25]; process switching costs 1 ms, symbol retrieval 0.35 ms, and class retrieval 3.24 ms. These cost are comparable to the others. Problems are analyzed as follows and their improvements will be our future work:

- *Disk I/O Architecture.* Data transfer between the main memory and the disk storage is done via a cache on the I/O bus memory. To reach the cache controlled by the disk device handler, process switching occurs four times, from a user process to the file manager then to the device handler and back. At least 4 ms has to be spent for every I/O operation. Caching data and control information on main memory as much as possible should be considered.

- *Network Interface.* The physical packet size is fixed to 4200 byte long now, but most method calls are found to be shorter than 1K byte. Changing the physical packet to 1K byte long will save about 260 ms.

- *ROAM Overhead.* In packaging and unpackaging messages of ROAM, symbol manipulation and class information retrieval are costly. Caching technique should be introduced for these database access. The reimplementation of the ROAM is now under way.

5 Conclusion

In this paper, we have described the design and implementation of the SIMPOS file system, looking at the history of revisions and extensions from a local file system to a distributed file system. The system has been in operation since summer 1987 and widely used to support daily work in the ICOT research center and other reseach institutes.

The design stood on rather optimistic assumptions in user mobility, file mobility, security and fault tolerance, but from the last two year experience, we have found the functionality and reliability of the system sufficient enough for the operation over the medium scale network of 300 workstations. Flexible concurrency control contributed to expand the variety of applications; implicit password/capability-based access control provided a protection veil invisible to our well-mannered users; simple name extension and the embedding an RMC mechanism on the basis enabled to use the existing application software without any modification. The only remaining problem is performance; only with the currently undergoing reimplementation of ROAM, a great performance improvement can be expected.

In the overall design of a distributed file system, communication control is a central issue. Adopting the remote object access mechanism, ROAM, gave a great influence on the reduction of the total development cost. Similar effects are seen in the development of other distributed file systems using RPC mechanisms.

More characteristic to our system is its object orientation. To our knowledge, the SIMPOS file system would be the first a distributed object-oriented system running for practical use in this scale. The experimental distributed Smalltalk systems adopt an universal proxy-object model that the same proxy object is created for any object, which uniformly forwards every message to its original object. This scheme is very simple, but leaves no room for optimization. Since a remote method call may cause additional remote method calls, the number of issued messages would be tremendously vast. It is very doubtful to us if this scheme is applicable for a practical large-scale system. For the development of a distributed system, how to reduce the communication traffic is a key to raise high performance.

The ROAM is a first-order proxy-original model that allows proxy objects to have knowledge about their original objects; the invocation of ROAM have to be specified for each method, but several optimizations are possible. In the implementation of the file system, the object categorization of complete global objects, which are treated as independent dynamic objects, and incomplete global objects, which are treated as dependent passive data, played an important role to reduce the number of messages.

We believe that making much use of the information on object dependency that users already know should be the most promising way to make a distributed object-oriented system practical.

Acknowledgments

This research has been conducted as part of the fifth generation project. I would like to express my gratitude to Dr. Kazuhiro Fuchi, Dr. Shun-ichi Uchida and Dr. Takashi Chikayama for their support and encouragement.

Many people engaged in the development of the SIMPOS file system, through the history from the original local file system to the distributed file system. I would like to thank all these people for their development effort. My special thanks are sent to Tadashi Mano, Masahiko Tateichi, Tatsuo Kato, Yoshiro Azuma, Masahiro Hoshi and Hirotaka Fukui for their sustained effort to raise the system to be a distributed file system.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe: *The Eden System: A Technical Review*, IEEE Transactions on Software Engineering, Vol. 11, No. 1, Jan. 1983.

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

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

[4] J. K. Bennett: *The Design and Implementation of Distributed Smalltalk*, Proc. of ACM OOPSLA'87, Oct. 1987.

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

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

[7] M. R. Brown, K. N. Kolling and E. A. Taft: *The Alpine File System*, ACM Transactions on Computer Systems, Vol. 3., No. 4, Nov. 1985.

[8] A. Birrell, R. Levin, R. Needham and M. Schroeder: *Grapevine: An Exercise in Distributed Computing*, Proc. of the 8th Symp. on Operating Systems Principles, Dec. 1981.

[9] 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.

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

[11] D. R. Cheriton and T. P. Mann: *Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance*, ACM Transactions on Computer Systems, Vol.7, No.2, May 1989.

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

[13] T. Chikayama: *Programming in ESP - Experience with SIMPOS* , ICOT Technical Report TR-285, 1987.

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

[15] C. A. R. Hoare: *Monitors: An Operating System Structuring Concept*, Communication of the ACM, Vol.17, No.10, Oct. 1974.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West: *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, Vol.6, No.1, Feb. 1988.

[17] 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.

[18] E. Jul, H. Levy, N. Hutchinson and A. Black: *Fine-Grained Mobility in the Emerald System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.

[19] P. H. Levine: *The Apollo DOMAIN Distributed File System*, Y. Paker, J.-P. Banatre, M. Bozyigit (editors), NATO ASI Series: *Theory and Practice of Distributed Operating Systems*, Springer-Verlag, 1987.

[20] E. Levy and A. Silbershatz: *Distributed File Systems: Concepts and Examples*, Technical Report TR-89-04, Dept. of Computer Science, The University of Texas at Austin, March 1989.

[21] A. K. Jones, R. J. Chansler, I. E. Durham, K. Schwans, and S. Vegdahl: *StarOS, a Multiprocessor Operating System for the Support of Task Forces*, Proc. of the 7th Symp. on Operating Systems Principles, Dec. 1979.

[22] *Transparent Forwarding: First Steps*, Proc. of ACM OOPSLA'87, Oct. 1987.

[23] M. K. McKusick, W. Joy, S. Leffler and R. S. Fabry: *A Fast File System for UNIX*, ACM Transactions on Computer Systems, Vol.2, No.3, August 1984.

[24] D. A. Moon: *Object-Oriented Programming with Flavors*, Proc. of ACM OOPSLA'86, Oct. 1986.

[25] 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.

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

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

[28] M. N. Nelson, B. B. Welch and J. K Ousterhout: *Caching in the Sprite Network File System*, ACM Transactions on Computer Systems, Vol.6, No.1, Feb. 1988.

[29] B. Randel: *Recursively Structured Distributed Computer Systems*, Proc. of the IEEE 3rd Symp. on Reliability in Distributed Software and Database Systems, Oct. 1983.

[30] 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.

[31] R. Sandberg, D. Goldberg, S. Keiman, D. Walsh, B. Lyone: *Design and Impentation of the Sun Network File system*, Proc. of Usenix 1985 Summer Conference, Jun. 1985.

[32] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West: *The ITC Distributed File System: Principles and Design*, Proc. of the 10th Symp. on Operating Systems Principles, Oct. 1985.

[33] M. Satyanarayanan: *RPC2 User Manual*, Technical Report CMU-CS-84-038, Carnegie Mellon University, 1986.

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

[35] M. Satyanarayanan: *A Survey of Distributed File Systems*, Technical Report CMU-CS-89-116, Feb. 1989.

[36] M. Satyanarayanan: *Integrating Security in a Large Distributed System*, ACM Transactions on Computing Systems, Vol.7, No.3, Aug. 1989.

[37] M. D. Schroeder, D. K. Gifford and R. M. Needham: *A Caching File System For Programmer's Workstation*, Proc. of the 10th Symp. on Operating Systems Principles, Dec. 1985.

[38] L. Svobodova: *File Servers for Network-Based Distributed Systems*, ACM Computing Surveys, Vol.16, No.4, Dec. 1984.

[39] A. S. Tanenbaum and R. Von Renesse: *Distributed Operating Systems*, ACM Computing Surveys, Vol.17, No.4, Dec. 1985.

[40] W. F. Tichy and Z. Ruan: *Towards a Distributed File System*, Proc. of Usenix 1984 Summer Conference, 1984.

[41] 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.

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

[43] B. R. Welch and J. Ousterhout: *Prefix tables: A simple mechanism for locating files in a distributed file system*, Proc. of IEEE 6th Intl. Conf. on Distributed Computing Systems, May 1986.

[44] W. A. Wulf, R. Levin, and S. P. Harbison: *Hydra/c.mmp: An Experimental Computer System*, McGraw-Hill, 1981.

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

[46] K. Yoshida, T. Mano, S. Saito, M. Komatsu and T. Chikayama: *Implementation of a Large Scale System in Object-Orientation – Development of the SIMPOS File System* , (in Japanese) ICOT Technical Report TR-108; JSSS Workshop on Object-Oriented Computing '86, March 1986.

[47] K. Yoshida: *Remote Object Access Mechanism*, submitted to Journal of Information Processing, JIPS, June 1989.

Table 1: Comparison of distributed file and operating systems

System	SIMPOS	Apollo	Locus	V	Sprite	NFS	Andrew
Position	Object-oriented logic-based operating system (+ RMC mechanism)	Distributed object-based operating system with a single level store management.	Distributed operating system kernel, upper compatible with UNIX, aimed at network-transparency and reliability.	Distributed operating system kernel, aimed at efficient message-passing and remote file access.	Network operating system (Sprite kernel + RPC mechanism)	Network file system interface (UNIX kernel + RPC mechanism)	Distributed computing environment (UNIX kernel + RPC mechanism)
Scale, Components	Medium-scale LAN of 300 PSI workstations	Ring network of Apollo workstations	LAN of about 20 PDP-11 and VAX machines	LAN of 35 Sun-2 and Micro Vax-II workstations, most of which are diskless.	LAN of Sun-2 and Sun-3 workstations, most of which are diskless.	LAN of UNIX (4.2 BSD) workstations	Large-scale LAN of thousands of IBM-RTs, Sun-2s and VAX-750s, each running on UNIX (4.2 BSD)
Naming scheme	*Host::Local*, Cluster of local hierarchical trees.	Hierarchical tree, location independent	Single hierarchical tree built up of disjoint subtrees, called *logical filegroups*	Single hierarchical tree layered into three levels	Single UNIX tree with prefix tables	Single UNIX tree with global name space	Each personal tree separated into a private name space and a shared name space.
Location transparency	Not supported.	Single-level storage management. Objects are identified by network-wide unique identifiers.	Remote mount of a logical filegroup (a UNIX file system).	Three level naming service. No remote mount.	Remote mount of a domain (a UNIX file system).	Remote mount of a subdirectory of an exported file system.	Remote mount of a volume (a UNIX quota).
Communication Primitives	ROAM (virtual-circuit).	Demand paging function of virtual memory management.	Specialized page-level file access protocol in the kernel.	Message-based general-purpose IPC primitives in the kernel and Verex I/O protocol on top of that .	Sprite RPC (unreliable datagram).	RPC/XDR on UDP/IP (unreliable datagram).	RPC2 (unreliable datagram).
Client versus Server	Each machine can act both. One server process per client.		Each machine can act both.	Usually clients are diskless.	Usually clients are diskless. One server thread per client.	Each machine can act both. One server process per client	Each machine can act both. One server thread per client.
Security	Authentication but no encryption in ROAM						Authentication and encryption in RPC2
Protection	Access list per directory entry, capability-based checking	Access list per file	Standard UNIX protection (access list per file)		Standard UNIX protection (access list per file)	Standard UNIX protection (access list per file)	Access list per directory
Client's cache	In memory, variable length (local cache: in I/O bus memory, fixed length)	In memory, variable length	In memory, fixed length		In memory, variable length	In memory, fixed length	On disk
Writing policy	Write-through, on close (local cache)	Delayed or on unlocked	On close		30 sec delay	On close or 30 second delay	On close
Open Modes	Exclusive, input(read sharing), output(read sharing write exclusive), shared (read/write sharing)	Exclusive (serialization using lock/unlock)	Exclusive, write sharing using tokens		Exclusive, write sharing	Exclusive	Exclusive
Consistency semantics	UNIX semantics, timing-dependent semantics, session semantics		UNIX semantics		UNIX semantics	Time-dependent semantics	Session semantics (whole file copy)
Performance(ms)							
null RPC	471 (207) (:do/i)			2.2	5.8		32.0
1K byte round-trip	171 (session)			11.6 (=5.8x2)	15.0 (=7.5x2)		

Table 2: Comparison of distributed object-oriented systems and interfaces

System	SIMPOS	Eden	Emerald	DOM	DS	Matchmaker
Position	Non-distributed object-oriented operating system, supplemented with RMC mechanism.	Distributed object-oriented system, whose objects are dynamic.	Distributed object-oriented language system, whose objects are dynamic.	Extended Smalltalk, distributed Smalltalk system	Extended Smalltalk, distributed object-oriented system	RMC stub generator language as multi lingual interprocess communication interface on non-object-oriented operating system
Language, Object Dynamicity, Class Inheritance	ESP, static, multiple class inheritance.	EPL, dynamic, no class, no inheritance.	Emerald, dynamic, no class, no inheritance.	Smalltalk, static, multiple class inheritance.	Smalltalk, static, multiple class inheritance.	Matchmaker
Scale, Components	Medium-scale LAN of 300 PSI workstations	LAN of Sun-2 workstations, on top of UNIX	LAN of 4 Micro Vax workstations, on top of UNIX 4.2BSD Aimed at the scale of 100 workstations.	simulated on a Sun-2 workstation, on top of UNIX 4.2BSD	LAN of 2 Sun-2 workstations, on top of UNIX 4.2BSD	on Mach
Network Layer	Virtual circuit	UDP datagrams (reliable)	UDP datagrams (reliable)		UDP datagrams (reliable)	
RMC Implementation Strategy	First-order proxy-original model: a global object class inherits class `remote_object` whose interface method `:g_call` forwards a message to the original object. Proxy and original objects are of the same class. Category of complete and incomplete global objects.	Stub routines are generated and liked with user programs by the compiler.	Category of global, local and direct objects. For invokation to a global object, a compiled code to check the *resident* bit and if not trap the kernel is generated. Parameter-passing modes: call-by-object-reference, call-by-move, call-by-visit supported for optimization.	Universal proxy-original model: proxy objects are instances of class `Proxy` in which method `doesNotUnderstand` forwards every message to their original objects.	Universal proxy-original model: proxy objects are instances of class `Proxy` in which method `doesNotUnderstand` forwards every message to their original objects.	Ports, which are communication channels with capabilities to the Mach kernel, are regarded as objects. Generated stub programs for sending/receiving a message to/from a port are executed by a thread. Multiple threads run under a single server process existing at each site.
Object Migration	Not supported.	Supported.	Supported.	Supported.	Supported.	
Class Compatibility	Not guaranteed. User's responsibility.	No class. Code is carried with an object.	No class. Code is carried with an object.		Not guaranteed. Class compatibility check prior to migration.	
Global Object Management and GC	Object management table per process. Cleaning up original/proxy objects in the table on process termination. Local GC based on mark-and-sweep. Global GC not supported.		Object management table per site. Local and global GCs both based on parallel mark-and-sweep.	Object management table per site. Global GC based on reference counting supported.	One object management table per site. Local GC based on mark-and-sweep. Global GC based on reference counting.	
Global Object Identification	Locally direct pointer. Globally unique identifier (= [node#,creationtime]) given to an object at its first exportation.	Globally unique identifier.	Locally direct pointer. Globally unique identifier *OID* and an *forwarding address* (=[node,timestamp]).	Globally unique identifier (=[node,virtual-pointer]).	Not globally unique identifier.	
Permanent Object Management	Independent file system. Capability-based protection.	Only autonomous objects, saving/retrieving their *checkpoints* into/from storage. Capability-based protection. File migration, file replication and nested transactions are supported.	Hierarchical directory system supported.			
Performance(ms)						
LMC	0.010	> 1	0.016		0.130	
RMC (null)	471 (207)		27.9		136.9	
RMC (1 val)			33.0 (call-by-move)		140.2	
RMC (1 ref)	484		61.8 (remote ref)			

Table 3: Performance of the SIMPOS file system (ms)

Method	*Remote Call*				*Local Call*	*comment*
	PSI-NET		*TCP/IP*			
	1024	4200	1024	4200		
Basic cost						
simplest local method call					0.01	
process switch					1	
:`get_atom` (symbol retrieval)					0.35	
:`get_class_object` (class retrieval)					3.25	
Session Layer						
round-trip time	171	428	450	1165		
initialization (connection)		595		1140		
ROAM						
`:do(#test, Node)`	211	475	503	1214	2	null call
`:make(#test, Instance, Node)`	220	484	512	1224	2	create instance
`:do(Instance)`	207	471	495	1211	1	null call
File Methods (class `binary_file`)						
`:make/c` (create and open file)		599		1285	114	
`:open/c` (directory info on memory)		569		1235	76	
`:open/i` (after retrieved)		463		1168	7	
`:close/i` (just after sync)		459		1122	9	
`:read/i` (data on cache)		510		1201	12	1K byte
		747		1719	19	4K byte
		7079		14499	238	100K byte
`:write/i` (data on cache)		508		1169	13	1K byte
		776		1714	19	4K byte
		7672		14191	238	100K byte
Directory Methods (class `directory`)						
`:retrieve/c`		1027		2389	69	
`:make/c` (create directory)		981		2348	47	
`:delete/i`		550		1254	59	
`:undelete/i`		541		1237	58	
`:find/i`		553		1266	40	
`:expunge/i`		492		1182	24	100 files

A Class Definitions

```
-----------------------------------------------------
class  binary_file  has
   nature                  % inheritance declaration %
       remote_binary_file,
       binary_file_accessor,
       local_binary_file,
       p_binary_file,
       logical_file,
       with_retrieval ;

            ...
instance
%===================================================%
% External Methods                                  %
%===================================================%
:read(Lfile, Buffer, Marker, RequestMode) :- !,
    :g_call(Lfile, read, {Buffer, Marker, RequestMode}) ;

:add(Lfile, Buffer, Marker, RequestMode) :- !,
    :g_call(Lfile, add, {Buffer, Marker, RequestMode}) ;

:accessor(Lfile, Accessor) :- !,
    :create(#binary_file_accessor, Accessor, File) ;

:submit_add(Lfile, Buffer, Marker,
                            MsgID, RequestMode) :- !,
    :g_call(Lfile, submit_add,
            {Buffer, Marker, MsgID, RequestMode}) ;

:wait(Lfile, Buffer, Marker, MsgID) :- !,
    :g_call(Lfile, wait, {Buffer, Marker, MsgID}) ;

            ...
end.
-----------------------------------------------------
class  local_binary_file  has
instance
%===================================================%
% Local Methods invoked to an original object       %
%===================================================%
:l_call(Lfile, read, {Buffer, Marker, RequestMode}) :- !,
            ... ,
    :send_receive(Lfile, input, do_read,
                  {Buffer, Marker}, RequestMode), ... ;

:l_call(Lfile, add, {Buffer, Marker, RequestMode}) :- !,
            ... ,
    :send_receive(Lfile, output, do_add,
                  {Buffer, Marker}, RequestMode), ... ;

:l_call(Lfile, submit_add,
       {Buffer, Marker, MsgID, RequestMode}) :- !,
            ...,
    :submit(Lfile, output, do_add,
                {Buffer, Marker}, RequestMode, MsgID) ;

:l_call(Lfile, wait, {Buffer, Marker, MsgID}) :- !,
            ... ,
    :wait(Lfile, MsgID, {Buffer, Marker}), ... ;

            ...
%===================================================%
% Internal Methods executed under file manager      %
%===================================================%
:do_add(Lfile, Buffer, Marker) :- !,
    :interface(Lfile, Pfile, AccessStatus), ... ,
    :add(Pfile, Buffer, Marker, AccessStatus), ... ;

:do_read(Lfile, Buffer, Marker) :- !,
    :interface(Lfile, Pfile, AccessStatus), ... ,
    :read(Pfile, Buffer, Marker, AccessStatus), ... ;

            ...
end.
-----------------------------------------------------
```

```
-----------------------------------------------------
class  logical_file  has
   nature
       file_accessor,
       local_logical_file,
       p_logical_file,
       logical_object,
       as_agent_of_physical_file,
       as_resource,
       as_file,
       with_file_capability ;

%===================================================%
% External Methods                                  %
%===================================================%
:open(Class, Lfile, Pathname, OpenMde, RequestMode) :- !,
    :remote_or_local(#pathname, Pathname,
                     Mode, _, FullPathname, _),
    :g_call(Class, open,
     {Lfile, FullPathname, OpenMode, RequestMode}, Mode) ;
            ...
end.
-----------------------------------------------------
class local_logical_file has

%===================================================%
% Local Methods invoked to an original object       %
%===================================================%
:l_call(Class, open, Lfile, Pathname,
                            OpenMode, RequestMode}) :- !,
    :retrieve(#directory, Lfile, Pathname, FullPathname),
            ... ,
    :set_full_pathname(Lfile, FullPathname),
    :open(Lfile, OpenMode, RequestMode) ;

            ...

instance
:l_call(Lfile, open, {OpenMode, RequestMode}) :- !,
    :mode_type(Lfile, OpenMode, OpType), ... ,
    :send_receive(Lfile, OpType, do_open,
                         {OpenMode}, RequestMode),
            ... ;

            ...
%===================================================%
% Internal Methods executed under file manager      %
%===================================================%
:do_open(Lfile, OpenMode) :- !,
            ... ,
    :interface(Lfile, Pfile, Status),
    :open(Pfile, OpenMode, Lfile, Status), ... ;

            ...
end.
-----------------------------------------------------
class  logical_object has
   nature
       as_agent_of_physical_object,
       with_capability,
       with_internal_communication,
       with_access_status,
       remote_object ;           %% inherit ROAM %%

            ...

end.
-----------------------------------------------------
```

```
------------------------------------------------------
class  binary_file_accessor  has
    nature   file_accessor ;

instance
%====================================================%
% Higher Level Interface Methods                     %
%====================================================%
:read_data(Accessor, String) :- !,
    :read_data(Accessor, Buffer, immediate) ;
:read_data(Accessor, Buffer, RequestMode) :- !,
    :read(Accessor, RequestMode),
    :get_data(Accessor, Buffer) ;

:read(Accessor, RequestMode) :- !,
    :buffer(Accessor, Buffer),
    :file_marker(Accessor, Marker),
    :read(Accessor!file, Buffer, Marker, RequestMode) ;

:add_data(Accessor, String) :- !,
    :add_data(Accessor, String, immediate) ;
:add_data(Accessor, String, RequestMode) :- !,
    :put_data(Accessor, String),
    :add(Accessor, RequestMode) ;

:add(Accessor) :- !,
    :add(Accessor, immediate) ;
:add(Accessor, RequestMode) :- !, ... ,
    :file_marker(Accessor, Marker),
    :add(Accessor!file, Accessor!buffer, Marker,
                                        RequestMode) ;

:submit_add(Accessor, MsgID) :- !,
    :submit_add(Accessor, MsgID, suppress) ;
:submit_add(Accessor, MsgID, RequestMode) :- !, ... ,
    :submit_add(Accessor!file, Accessor!buffer,
            Accessor!file_marker, MsgID, RequestMode) ;

:wait(Accessor, MsgID) :- !,
    :wait(Accessor!file, Accessor!buffer,
          Accessor!file_marker, MsgID) ;

          ...
end.
------------------------------------------------------
class  file_accessor  has

:create(Class, Accessor, File) :- !,
        unbound(Accessor),
        :new(Class, Accessor),
        :set_accessor(Accessor, File) ;

          ...
instance
    attribute   file, file_marker, buffer ;

:put_data(Accessor, StringOrVector) :- !, ... ,
    :put_data(Accessor!buffer, StringOrVector), ... ;

:get_data(Accessor, StringOrVector) :- !, ... ,
    :get_data(Accessor!buffer, StringOrVector), ... ;

          ...
end.
------------------------------------------------------
```

```
------------------------------------------------------
class  remote_binary_file  has

          ...
instance
%====================================================%
% Specification of Extra Information Delivery        %
%====================================================%
:set_parameters(Robj, read,
            {Buffer, Marker, RequestMode}, Packet) :- !,
    :set_buffer_marker(Robj, request_input,
                          Buffer, Marker, Packet) ;

:get_parameters(Lobj, read,
            {Buffer, Marker, RequestMode}, Packet) :- !,
    :get_buffer_marker(Lobj, request_input,
                          Buffer, Marker, Packet) ;

:set_returns(Lobj, read,
            {Buffer, Marker, RequestMode}, Packet) :- !,
    :set_buffer_marker(Lobj, reply_input,
                          Buffer, Marker, Packet) ;

:get_returns(Robj, read,
            {Buffer, Marker, RequestMode}, Packet) :- !,
    :get_buffer_marker(Robj, reply_input,
                          Buffer, Marker, Packet) ;

          ...
%
:set_buffer_marker(Obj, Type, Buffer, Marker, Packet) :- !,
    :get_attributes(Buffer, Type, BufferAttributes),
    :set_data_element(Packet, BufferAttributes), ... ;

:get_buffer_marker(Obj, Type, Buffer, Marker, Packet) :- !,
    :get_data_element(Packet, BufferAttributes),
    :set_attributes(Buffer, Type, BufferAttributes), ... ;
          ...
end.
------------------------------------------------------
class  buffer  has
   nature
       as_remote_object ;    %% inherit ROAM %%
          ...

instance
%====================================================%
% Specification of Delivered Attributes              %
%====================================================%
:get_attributes(Buffer, request_input,
                             [ByteSize, ElementSize]):- !,
    :get_io_buffer(Buffer, ElementSize, ByteSize) ;

:set_attributes(Buffer, request_input,
                             [ByteSize, ElementSize]):- !,
    :set_io_buffer(Buffer, ElementSize, ByteSize) ;

:get_attributes(Buffer, reply_input,
                      [Buffer!data_size, ReadBuffer]):- !,
    :get_data(Buffer, ReadBuffer) ;

:set_attributes(Buffer, reply_input,
                               [DataSize, ReadBuffer]):- !,
    :copy_data_buffer(Buffer, DataSize, ReadBuffer) ;

          ...
end.
------------------------------------------------------
```