Implementing Persistent Objects in the Apertos[†]Operating System

Takao Tenma Yasuhiko Yokote Mario Tokoro[‡]

e-mail: {ten,ykt,mario}@csl.sony.co.jp

Sony Computer Science Laboratory Inc.
Takanawa Muse Building,
3-14-13 Higashi-gotanda, Shinagawa-ku,
Tokyo, 141 JAPAN

Abstract

This paper presents a way of providing users with a persistent object running under the Apertos operating system. We present an implementation of persistent objects by using object migration between metaspaces in the reflective object architecture. An Apertos object is stored into stable storage by migrating to a storage metaspace that is an abstraction of object storage. We also present the current status of the implementation.

1 Introduction

Advancing hardware technology enables large distributed computer networks, including portable and mobile computers, such as laptop and notebook computers. In such a computer network, users and objects, such as computers, storage media, data, system software, and other resources, are moving around freely. The computational field model [12] provides a framework for dealing with such large distributed systems.

To provide a useful interface for a large, complex distributed system, a distributed operating system should be able to provide the same computing environment to each user as he/she accesses any computer in the distributed system. Currently, however, a user must adapt his/her computing environment to the local execution environment by himself/herself. For example, we use a workstation at the office, a personal computer at home, and a laptop or notebook computer in a train or airplane. In these situation, we must change our personal computing environment and customize that environment to each hardware, architecture, operating system, and system software.

Furthermore, when we change computers, we must suspend active jobs on the first computer and restart them on the next computer. In the above mentioned example, before we come back home, we first save editing files and stop our applications. Then, we copy and translate saved files from a workstation to a mobile computer. To process the saved files, we start some applications on the mobile computer, in the train, say. When we get home, we must copy the editing files from the mobile computer to our personal computer. Thus, our computing environment is not continuous and we must change the way in which we work. This discontinuity forces us to deal with non-creative tasks, that is, to suspend and to resume our jobs. And this increases the difficulty of our computer system.

Our goal for a persistent object system running under the Apertos operating system is to provide a continuous computing environment to each user, whenever the user logs on any computer in a large computer network that features mobile hosts. We believe this goal can be realized by giving persistence to a user's computing environment.

[†]Formerly called Muse.

[‡]also with Keio University.

Recent distributed file systems, such as AFS and Coda, have succeeded in providing a unique perspective of directory structures for each user [10]. A user of such a distributed file system can obtain the same view whenever the user logs on from any computer that is part of the file system. Argus [7] and Camelot/Avalon [4] provide distributed persistent objects and atomic transaction facilities based on the nested transaction technique [8]. Object-oriented databases [6], such as ORION, GemStone, and ObjectStore, and persistent programming languages with object storages, can give persistence to programming language objects. Object-oriented databases and persistent programming languages provide a means of solving the *impedance mismatch* problem.

The Apertos operating system is an object-oriented operating system. Everything is represented as a concurrent object with a computation power (called an Apertos object). An Apertos object encapsulates its states, the methods that access that state, and the virtual processor that executes its methods. A user's computing environment in the Apertos operating system consists of a group of concurrent objects that communicate with each other. Furthermore, these objects are distributed throughout many computers.

We realized a persistent object system based on the reflective architecture provided by the Apertos operating system. Two metaspaces, mPersistent and mStorage, provide Apertos objects with an execution environment of persistent objects and an abstraction of object storage, respectively. An object migration mechanism gives programmers and users a uniform perspective to control the persistence of Apertos objects.

This paper consists of the following sections. Section 2 discusses the continuity of a computing environment in a distributed operating system. Section 3 discusses the realization of persistent objects on the *Muse Object Architecture* [14]. The Apertos operating system allows users and programmers to work in different abstraction levels throughout the object system. Section 4, shows the current state of implementation for a persistent object system. Finally, in Section 5, future work and concluding remarks are presented.

2 Continuity of Computing Environment

Figure 1 shows a goal for the persistent object system in the Apertos operating system, i.e., to provide a continuous computing environment. A user accesses a large computer network with a terminal computer A, and performs his tasks. The user moves from computer A to computer B taking his/her mobile computer, and continues his/her tasks, on the mobile computer. Computer B, the user connects his mobile computer to a network and continues his/her tasks on computer B or the mobile computer.

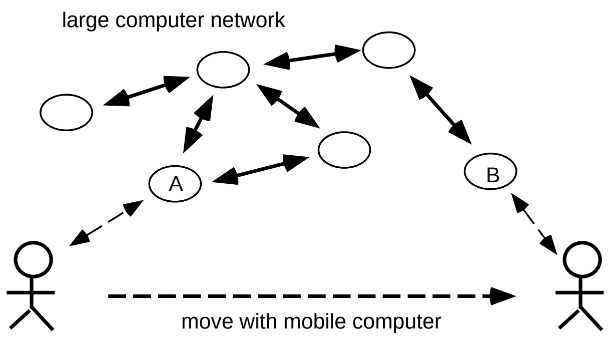

Figure 1: Continuous Computing Environment

To realize the continuous computing environment, many issues must be solved. These are summarized as follows:

- **Object Naming**
  An object must have a unique identifier within a large distributed computer network. Difficulties involved in naming are in the migration of objects and the size of the networks. [5] proposes an object naming and addressing scheme without a global view. In this scheme, an object can have an identifier when the object moves around networks. The Apertos operating system uses this scheme for naming Apertos objects.
- **Persistence**
  A computing environment should be saved to and resumed from stable storage. An Apertos object can satisfy the requirements of persistence in our system.
- **Object Group**

Usually, we use multiple applications in multiple windows and access several server processes that exchange information. A group of related objects must be managed in a persistent system to maintain the consistency of a group and provide operations to an entire group such as migration of an object group. V-kernel [2] and ISIS [1] realize a group of concurrent activities, i.e., a *distributed process group,* and provide group operations for maintaining its membership. Group operations include *create, destroy, join*, and *leave.*

- **Mobile Computing**
  In a mobile computing environment, everything can move around networks. So, we must first consider the migration of objects to design a persistent system running under the Apertos operating system.

3 Persistent Objects Realized on the Reflective Architecture

We will discuss the realization of persistent objects on the reflective architecture of the Apertos operating system. This section first overviews the reflective architecture from the viewpoint of constructing an operating system. Then, we discuss reflective computing for changing the computing environment. Object migration between metaspaces is introduced as the basic mechanism of reflective computing. Reflectors and a reflector class hierarchy are provided to create a new metaspace. Finally, we present the realization of persistent objects based on object migration and persistent metaspaces.

3.1 Reflective Architecture

We employ the notation of concurrent object-oriented computing [16] to model an object. [15] introduce a reflection mechanism for the construction of an operating system. Each object encapsulates its state, the methods that access the state, and a virtual processor that executes the methods. Here, a virtual processor of an object can be viewed as a *metaobject.* A metaobject is an object that represents part of the behavior of that object. A metaobject supports, for example, a way communicating with another object, a virtual memory management policy, a means of handling a faulty operation, and a storage management policy. In the model, an object is supported by a group of metaobjects. Here, we call a group of metaobjects a *metaspace.* An object has the execution semantics that are provided by the metaobjects in a metaspace.

Figure 2 shows a model where a white circle represents an object and a gray area represents a metaspace. Since the metaobject composing a metaspace

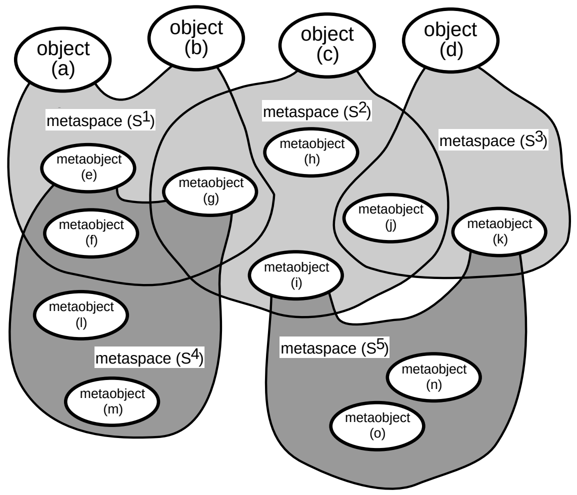

Figure 2: Model of Apertos Object Architecture

is an object, we must introduce a metaspace for a metaobject, so that the object and its metaspace are represented within its *metahierarchy.*

The relationship between an object and metaobjects composing its metaspace is relative. In Figure 2, metaspaces (S^1), (S^2), and (S^3) are composed of metaobjects for objects (a) and (b), object (c), and object (d), respectively. Also, metaspaces (S^4) and (S^5) are composed of metaobjects for metaobjects (e) and (g), and metaobjects (i) and (k), respectively. Here, metaobjects (e) and (g) are members of metaspace (S^1) for objects (a) and (b), i.e., (e) and (g) are metaobjects of (a) and (b), whereas they are objects whose metaspace is (S^4).

Furthermore, since a metaspace consists of metaobjects, a metaobject can be shared between metaspaces. In the figure, metaobject (f) is shared between metaspaces (S^1) and (S^4). Also metaobjects (g) and (j) are shared between metaspaces (S^1) and (S^2), and metaspaces (S^2) and (S^3), respectively. Thus, given an object, we can construct a metahierarchy composed of the hierarchy of metaspaces whose root is an object.

3.2 Reflective Computing in Operating Systems

A reflective architecture provides users and programmers with mechanisms for reflective computing, which we call reflection. Reflective computing is defined as a process for improving object behavior by representing the behavior of an object and by reasoning about the object itself.

An operating system should be constructed by using object-oriented technology, in which everything within a system that should be shared and protected is an object. This encourages modularity, increases reusability and maintainability, and gives users/programmers a single perspective of the system, and also it makes it possible to construct a system that is large in scale and complexity. However, when we consider everything as being an object, we encounter some difficulties. For example, it is difficult to inspect the internals of an object, because an object is protected against access from other objects. Therefore, every object must provide a method of exposing its internals. That will be helpful in the implementation of a debugger. Also, it is difficult to implement an object manager such as an invocation manager and a scheduler, because they need to access metadata such as the representation of a message and a scheduling state of object. The object architecture discussed in the previous subsection can overcome these difficulties. In this sense, an operating system should be based on a reflective architecture.

Further, an operating system should provide users/programmers with multiple abstractions, because it will become difficult to satisfy all the requirements of users/programmers as the scale of the system grows. The above model can provide multiple abstractions in the way shown in Figure 3. A new abstraction is given by objects defined below the thick line. In this figure, one thick line provides the upper objects with kernel mechanisms such as object scheduling, invocation management, and memory management. Another thick line adds new services such as persistent objects and realtime scheduling.

By using these multiple abstraction levels, a programmer can realize system level facilities and application level facilities in a uniform manner, that is concurrent object-oriented programming, provided by the object-oriented reflective architecture.

3.3 Object Migration

In an open and mobile computing environment, everything including computer, users, processes, and resources moves around networks for load balancing, availability, reliability, fault-tolerance, and so on. Thus, a distributed operating system should first provide facilities to support migration of objects.

Here, we believe a location of object, i.e., host computer, is one of property of object. Thus, we introduce *object migration* as a basic mechanism for reflective computing and for mobile computing in the Apertos operating system, where an object travels within a metahierarchy.

In Figure 2, for example, we say object (a) of metaspace (S^1) migrates to metaspace (S^2), that is, object (a) changes its metaspace from (S^1) to (S^2). In Figure 3, an object defined above a thick line can migrate to another thick line. In this case, if metaspace (S^1) and (S^2) are in the same host, object (a) does not move to the host computer. Otherwise, if the two metaspaces are in different hosts, the object migrates between the host computers.

It is helpful for users and programmers to define object migration as a basic mechanism for reflective computing. We can describe operating system services within a single framework. For example, an object can migrate to a metaspace that has debugging facilities, that is, an object is debugged. In this case, since a metaobject represents the internals of an object, a debugger can be implemented as a metaobject.

The structure of an operating system facilitates object migration. Since the internals of an object are rep-

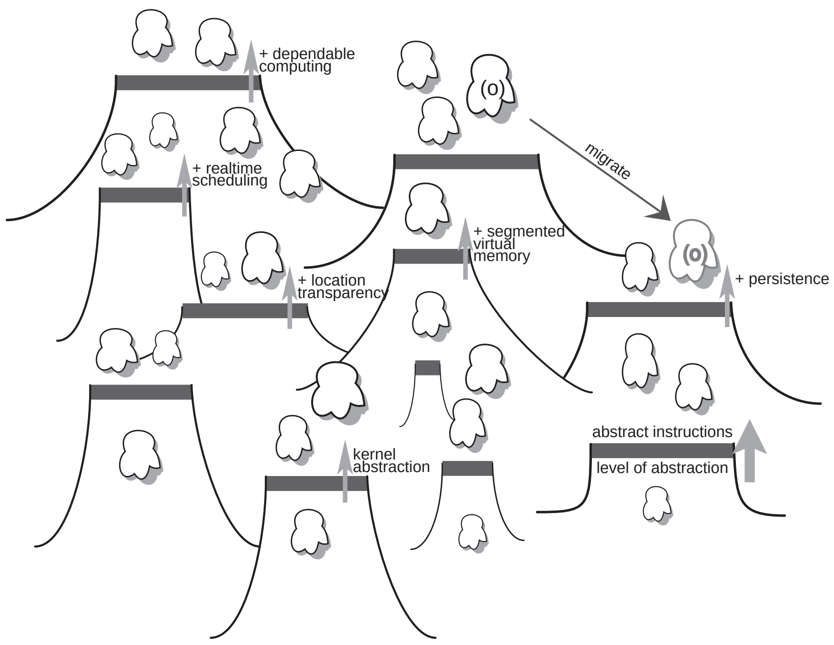

Figure 3: Multiple Abstractions

resented as metaobjects, moving an object is equivalent to transferring metaobjects to a target metaspace. Here, transferring metaobjects can also be considered as object migration. In this way, reflection can be viewed as object migration. An object migrates between metaspaces in the following way.

1. An object sends a migration request to its metaobject. The metaspace to which the object migrates is specified.
2. A metaobject selects a new metaspace and checks the compatibility between the object and the destination metaspace.
3. If the object is compatible with the destination metaspace, the object migrates from the source metaspace to the destination metaspace.

3.4 Reflectors

A reflector is a metaobject that represents a metaspace. That is, it provides an object with metaoperations that expose the mechanisms of reflective computing and defines a group of metaobjects. A metaoperation is an operation in which an object changes its behavior. A reflector is defined within a class hierarchy, that we call the reflector class hierarchy.

The top of the hierarchy is m^{Common}[1], an abstract class that provides succeeding reflectors with common facilities such as object migration and descriptors that designate metaobjects representing objects.

$$\begin{tikzcd}
 & & m^{\mathrm{Common}} & & \\
 & m^{\mathrm{Base}} \arrow[ur] & m^{\mathrm{Zero}} \arrow[u] & m^{\mathrm{Storage}} \arrow[ul] & m^{\mathrm{Class}} \arrow[ull] \\
m^{\mathrm{Persistent}} \arrow[ur] & m^{\mathrm{Drive}} \arrow[u] & m^{\mathrm{System}} \arrow[ul] & &
\end{tikzcd}$$

Figure 4: Part of Reflector Class Hierarchy

There are several reflectors defined as subclasses of m^{Common} as shown in Figure 4. Reflectors and their *reflector class hierarchy* describe protocols for reflection programming. m^{Zero} is a reflector that represents a metaspace for reflector programming. m^{Base} is a reflector that provides a metaspace for concurrent

[1] In this paper, we use notation $m^{\mathrm{Reflector}}$ to indicate a reflector.

objects. mDrive is a reflector that provides metaoperations for interrupt handler programming of device drivers. mSystem is a reflector that supports system objects in kernel mode. mClass is a reflector that represents a metaspace for class objects which are static and immutable templates for objects [15], so that it provides metaoperations to manage classes such as relocating code segments and inspecting the internals of a class. Two reflectors mStorage and mPersistent realize the persistence of Apertos objects, as described in the next subsection. We can define a new reflector that has new metaoperations as a subclass of an existing reflector.

3.5 Metaspaces for Persistent Objects

Persistence is one of property of object, and the realization of persistent objects needs the internal structure of objects. Thus, we design a persistent system as a metaspace and a reflector in the object architecture. In this system, an Apertos object can acquire persistence properties by migrating to the metaspace for persistence dynamically. Also, object storage is designed as a metaspace and a reflector for providing a uniform perspective about the location of an Apertos object such as host computers, memory, disks, tapes, and other storage media. In the rest of this subsection, we describe two metaspaces, mStorage and mPersistent.

3.5.1 mStorage

An *object storage* manages objects in a stable storage system and controls accesses to the objects. Figure 5 shows mStorage, is an object storage in the Apertos operating system. In this figure, a dashed arrow represents a relationship between an object and its reflector. mZero is a reflector of mBase, and mBase is a reflector of object (A). When Apertos object (A) migrates to the mStorage metaspace, object (A) is stored into stable storage, a disk device object in the mStorage metaspace. So, there is no *load* or *save* operation in mStorage level. Object migration is a unique interface to change properties of objects. In this figure, an object migration from mBase metaspace to mStorage metaspace corresponds to a *save* operation. An object migration from mStorage metaspace to mBase metaspace corresponds to a *load* operation.

Of course, in a lower abstraction level, *save* and *load* operations are provided, and the programmer can override such operations to customize the object storage. This illustrates the advantage of the reflective architecture from the viewpoint of uniformity.

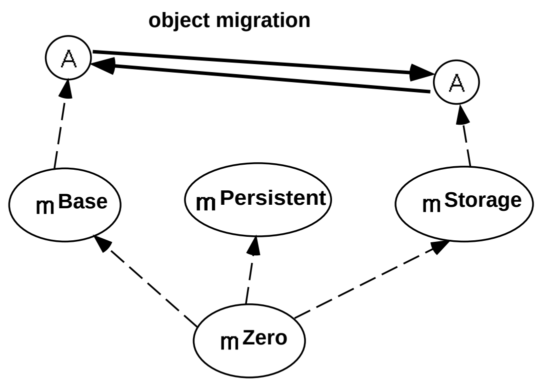

Figure 5: mStorage

3.5.2 mPersistent

Figure 6 shows mPersistent, a reflector to support persistent objects. When Apertos object (A) migrates to mPersistent metaspace, first a replica of that object (A2) is created in the mStorage metaspace. Changes in Apertos object (A) are propagated to replica (A2) to enable the recovery of object (A).

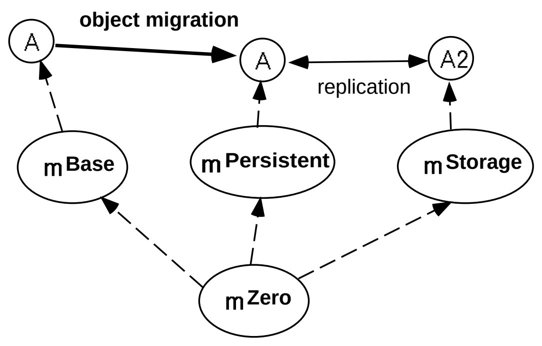

Figure 6: mPersistent

A group of related objects, that exchange messages, requires to maintain the consistency between replica of objects in the group. The system should prevent lost messages and orphan messages [11]. To archive these goals, mPersistent manages an object group and creates a distributed object group. Objects, belonging to the same group and which are on the same host, are on the same mPersistent reflector. mPersistent reflectors on different hosts create a distributed object group by using techniques in V-kernel [2] and ISIS [1].

mPersistent updates replicas of objects, that belong to a same group, at one instant to assume consistency between the replicas. So, communications between an object on mPersistent and an object on another metaspace invoke a group operation for maintaining membership of the group. For example, in Figure 7, when object (B) on mBase sends a message to object (A) on mPersistent, object (B) migrates from mBase to mPersistent. Finally, object (A) and object (B) become to join a same group maintained by the mPersistent.

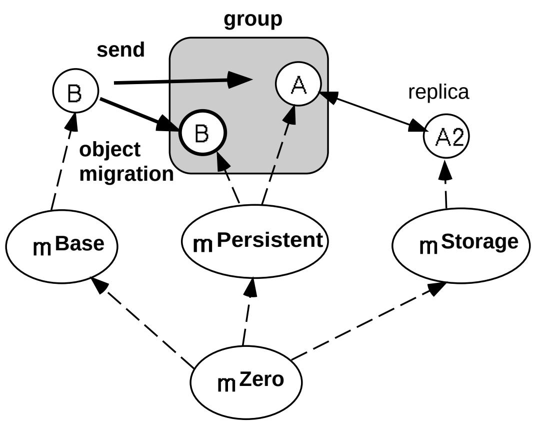

Figure 7: Object Group in mPersistent

4 Implementation

The Apertos operating system is being implemented on Sony NWS-1400, PWS-1500, and NWS-1700 workstations. The kernel and reflectors mZero, mBase, mDrive, and mStorage are already running. Metaobjects such as managing virtual memory, storage, disk devices, network devices, SCSI controllers, and console devices are also available.

The system is implemented using the AT&T C++ programming language [3]. Libraries for programming in the Apertos system are provided. Group operations between hosts are now being implemented.

Figure 8 shows an overview of the persistent object system. A white circle is an Apertos object, while a gray circle is a metaspace. the persistence of Apertos objects is realized by two reflectors, mPersistent and mStorage, and the metaobjects in their metaspaces.

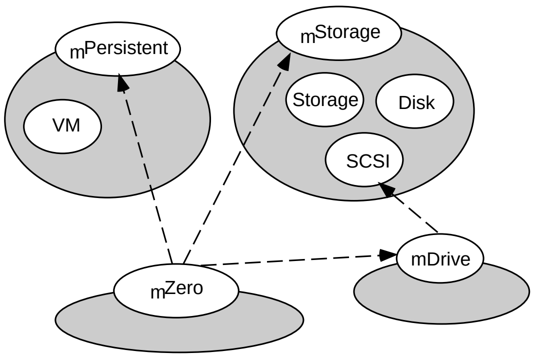

Figure 8: Implementation of Persistent Object System

In this section, we first describe the internals of Apertos objects. Then, we describe the reflectors and objects necessary to realize the persistent Apertos object. The prototype system and its measurements are also given.

4.1 Internals of Apertos Object

The Apertos operating system boots from a boot image into which several Apertos system objects and Apertos object classes are packed. An Apertos object is created from a class, except for some basic system objects.

4.1.1 Class Format

An Apertos class consists of metainformation and multiple bodies. Metainformation contains three types of information, (1) reflector information, (2) type information, and (3) body information.

Reflector information is used for initializing metadata in the reflector when it creates an object. This includes:

- system stack size,
- user stack size,
- scheduling priority,
- CPU priority (interrupt mask level),
- execution mode (user model or kernel mode), and
- name of default reflector.

Type information specifies the interface of the class. This is used to send a message from other languages, such as the shell language. The contents of the type information are as follows.

- symbol name of class
- version number
- number of methods
- method interfaces, i.e., selector id, selector name, return type, and argument names and their types

A class can have multiple different bodies. *Body information* specifies one of class body.

- type of body: There are several types of class body, such as binary, source code, byte code, or interpreter code. Currently, only binary code is used.
- CPU type: The CPU type is specified for a binary body.
- format: For a binary body, the file format of the body, such as AOUT format or COFF format, is specified.
- version number: A version number of the body is specified.

4.1.2 Object Format

An object in virtual memory consists of a set of segments and two metadata, *Context* and *Descriptor*.

A *Context* is an abstraction of the CPU and is managed by *MetaCore* (a micro-kernel). A context has a register set, program counter, and so on.

A *Descriptor* is created and managed by a reflector of the object. The contents of the descriptor depend on a reflector. A descriptor of m^{Base} has the following data.

- scheduling status
- memory segment information
- scheduling priority
- CPU priority (interrupt mask level)
- message queue
- context identifier
- object identifier
- class identifier

For an m^{Base} object, an object can have any of the following six types of scheduling status.

1. DORMANT: an object waits for a message.
2. READY: an object is ready to be run.
3. RUNNING: an object is running.
4. WAIT: an object waits for a reply to a *Call* message.
5. METAWAIT: an object waits for a reply to a meta call from its reflector.
6. SUSPEND: an object is suspended.

A driver object on m^{Drive} has one more status IOWAIT that waits for an interrupt. The m^{Zero} object has four status, DORMANT, SUSPENDED, READY, and RUNNING.

The body of an Apertos object is a set of segments. There are three types of segments,

- text segment,
- data segment, and
- stack segment.

By using metainformation in descriptor, a reflector migrates and controls Apertos objects.

4.1.3 Instantiation of an Object

To install several related objects in one address space, an object management system can allocate an object into one slot (small part of an address space). The size of the slot is fixed in an address space. A reflector object instantiates an Apertos object from a class, using the following steps.

1. To obtain class information, the reflector sends a message to the m^{Class} reflector.
2. The reflector gets an object identifier for the newly created object by sending a message to namer object.
3. The reflector allocates memory space to the object by sending messages to the *VM* objects that manages virtual memory of the system. An object has its own memory space that is specified as an address space identifier and slot identifier.

An Apertos object management system divides one address space into several slots. When an object executes in user mode, the *VM* object tries to allocate one slot for the object from an existing address space. But, if there is no free slot, a new address space is created and a slot is obtained from the newly created address space. When an object executes in kernel mode, the object manager allocates a slot from the kernel space.

4. The VM manager allocates segments, such as a text segment, data segment, and stack segment, in the assigned slot.
5. The reflector creates a new descriptor to schedule and manage the created object.
6. Finally, the reflector schedules the created object.

4.1.4 Migration of an Object

An Apertos object can migrate from one reflector to another reflector, using the following steps. We assume that object (O) in reflector (X) tries to migrate from reflector (X) to reflector (Y).

1. Object (O) calls *Migrate (target reflector (Y))* to its reflector (X).
2. Reflector (X) checks the compatibility between object (O) and the reflector (Y). If they aren't compatible, the reflector returns an error code to object (X).
3. Reflector (X) sends a message to (Y) and (Y) creates a descriptor for the object (O).
4. The contents of the descriptor for (O) in reflector (X) are transformed into the corresponding descriptor in reflector (Y).
5. Segments of object (O) managed in reflector (X) are transformed into reflector (Y).
6. The descriptor in (X) is deleted.
7. Reflector (Y) schedules object (O).

These steps relate to migration between reflectors for objects in virtual memory. The steps constituting migration between virtual memory and stable storage are described in the next two subsections.

4.2 m^{Storage} Metaspace

The m^{Storage} reflector provides an abstraction of object storage. The current m^{Storage} metaspace consists of *Storage*, *Disk*, and *SCSI* system objects. The *Storage* object is described in the next subsection. The *Disk* object provides a raw device interface with the disk device. The *SCSI* object provides an interface with the SCSI controller.

The m^{Storage} reflector supports the following operations:

- The *Call* operation sends a message to an object in stable storage. If the target object has not migrated in a metaspace that supports objects in virtual memory, such as m^{Base}, the reflector moves the target object from this space to m^{Base}. Then, the reflector delivers the message to the migrating object. Otherwise, if the target object has already migrated to a metaspace for objects in virtual memory, the reflector delivers the message to the loaded object.
- The *Send* operation performs the same operation as *Call* except that m^{Storage} delivers a *Send* message instead of *Call* message.
- The *New* operation creates a new object in storage.
- The *Delete* operation removes an existing object from storage.
- The *Find* operation returns a map from an object to its reflector.
- The *Migrate* operation moves an object to another metaspace. When a target metaspace is a metaspace for objects in virtual memory, the object is loaded.
- The *MigrateIn* operation from another reflector moves an object on the caller reflector to the m^{Storage} metaspace, and the object is saved into the stable storage.

m^{Storage} provides a more generic interface of object storages. A programmer can create other customized storage metaspaces with different disk allocation policies, device types, media types, and so on.

The following steps detail object migration from m^{Base} to m^{Storage}.

1. Four free storage identifiers are allocated to the object. A storage identifier corresponds to an *i-node number* in the UNIX[2] file system.
2. A new descriptor for mStorage for the migrating object is created by transforming a descriptor of the mBase reflector.
3. The new descriptor, containing the storage identifiers for all segments, is saved to disk. This contains *Context* and the contents of the descriptor in the mBase reflector. The contents of the descriptor and the number of saved segments depend on the status of the saved object. If the object is DORMANT, the CPU structure and register sets in context, the message queue, and a stack segment are not saved. If the object is READY, the message queue should be saved. METAWAIT status requires that the CPU structure and message queue are saved. An object in WAIT status or SUSPEND status can not migrate to mStorage. It will be saved to storage by waiting for the state change. But, this is not implemented.
4. All segments are saved into the storage object. In the current implementation, different storage identifiers are assigned to each segment.
5. The old descriptor in mBase is deleted.

On the other hand, an Apertos object migrates from mStorage to mBase, in the following steps.

1. In this case, we assume an object (X) on mBase sends a message to an object on mStorage.
2. mStorage sends a *MigrateIn* message to the mBase reflector to load object X.
3. By using an i-node entry to designate an Apertos object, is described in Section 4.4 in detail, a descriptor of object (X) in the mStorage is read from the disk.
4. All segments are loaded from the disk.
5. A new descriptor for object (X) is created in mBase by transforming a descriptor of the mStorage reflector.
6. The mBase reflector creates a new context for object (X) and sets values for the context.
7. The mBase reflector schedules object (X).

[2]UNIX is a registered trademark of AT&T Bell Laboratories.

4.3 mPersistent Metaspace

The mPersistent reflector provides facilities for persistence and management of object groups. The metaspace consists of a *VM* object for managing virtual memory.

The mPersistent reflector supports the following operations:

- The *Call* operation invokes a method defined in the target object. This activates the internal scheduler that determines an object to be activated next. If the target is not ready to accept the request, it is stored into the queue maintained in mStorage. As described in the previous section, a receiver object migrates to mPersistent for joining a group.
- The *Send* operation performs the same operation as a *Call* operation, except that the object that initiates the *Send* operation continues execution. A receiver object migrates to mPersistent.
- The *Reply* operation delivers the result back to the sender object. It activates the internal scheduler to find the object to be activated next. If a request is pending for the object initiating the Reply operation, it is scheduled for processing.
- The *Migrate* operation moves an object to another metaspace. The object leaves the group managed by the reflector.
- The *Checkpoint* operation takes frozen images of objects on mPersistent, that is, mPersistent creates or updates a replica in mStorage. A descriptor is saved when the state of the object changes. Only modified pages in the segments of the object are saved.

4.4 Storage Object

An Apertos object is stored into the storage object. The storage object is based on the Sprite log-structured file system [9]. The log-structured file system writes sequentially all modifications to disk in a log to speed up both write operations and crash recovery. The Sprite log-structured file system introduces a *segment*, which is a small part of the disk, to overcome fragmentation problems and decrease log cleaning cost. Figure 9 shows an example of using segments in a log-structured file system. A log is written to a

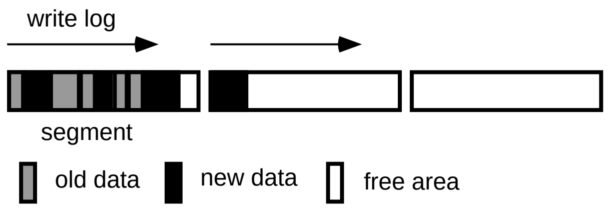

Figure 9: Segments in Log-Structured File System

segment sequentially. Once a segment is filled, the next log is written into the next free segment. The *Storage* object makes a full segment free by a cleaning operation performed during the night. Also, when the number of free segments becomes less than the threshold, a cleaning operation is invoked. The cleaning operation moves all logs in a source segment to another destination segment to free up the source segment.

The *Storage* object in the Apertos operating system also uses segments, each each segment having a generation number. The generation of segments is introduced to minimize cleaning costs by localizing related objects. Younger segments are cleaned frequently, while older segments are not cleaned so frequently. Figure 10 shows a generational cleaning operation in a segment-based log-structured file system. During the cleaning operation, a log having an old ver-

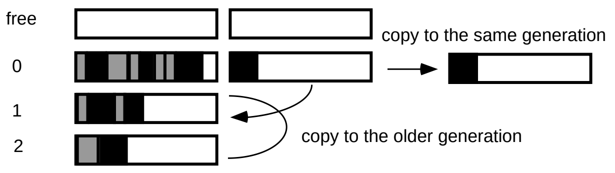

Figure 10: Cleaning Segments in Log-Structured File System

sion number in the source segment remains untouched, that is, the log is cleaned. A log with a younger version number is copied from the source segment to another segment. If an object for a log is not modified during the threshold period, the log is copied into an older segment. Otherwise, the log is copied to a target segment having an identical or younger generation number.

In the current version of the storage object, the size of a segment is over 1 megabyte, and the size is a multiple of the cylinder size. The maximum generation number is 10. To prevent the *read-fragmentation* problem, which is a problem that an object is split into multiple logs and the cost of a read operation increases, an entire object is written into one log. Since the current Apertos operating system has no huge object, this policy increases the efficiency of read operations. For a large object whose size is bigger than that of a segment, a large segment is created by merging several segments.

The Storage object supports the following operations:

- The *Connect* operation connects a *Disk* object to the *Storage* object. This reads the UNIX disklabel in the target disk and sets the disk parameters. The current storage object uses one disk partition to enable the UNIX operating system and Apertos operating system to coexist on the same disk.
- The *Disconnect* operation disconnects the connected *Disk* object from the *Storage* object.
- The *Newfs* operation writes to free segments on the disk.
- The *Scan* operation reads all disk logs, and creates an i-node table in memory. All i-nodes are kept in memory to enable efficient disk access. The size of an i-node entry is not so large. It is currently 14 bytes; 4 bytes for the object identifier of the saved descriptor, 4 bytes for the block address on disk, 4 bytes for the version number, 1 byte for the types of i-node entry, and 1 bytes for flags.
- The *Create* operation creates an object index in the storage object.
- The *Read* operation reads an object on disk.
- The *Write* operation writes an entire object to disk.
- The *Clean* operation invokes a cleaning operation. This is called at night or when the free area of the disk space is exhausted.

4.5 Preliminary Measurements

The results have been obtained on a Sony PWS-1550 workstation that has a 25MHz MC68030 CPU with a minimum 4MB of physical memory. It features an SCSI-1 controller. Objects are transferred in DMA non synchronous transfer mode.

[13] shows the costs of kernel and reflector operations. This paper shows the costs of the persistent system.

Table 1 shows the results of our measurement of object migration between metaspaces on the same host. The object is 10K bytes, the biggest object in the prototype system.

Table 1: Costs of Object Migration

primitive	*cost (in msec)*
mBase $\leftrightarrow$ mBase	0.84
mBase $\rightarrow$ mPersistent	99
mBase $\leftarrow$ mPersistent	0.84
mBase $\rightarrow$ mStorage	817
mBase $\leftarrow$ mStorage	249

In addition to the cost of object migration between mBase and mBase, an object migration from mBase to mPersistent needs a cost to create a replica of the migrate object, as well as a cost for maintenance for an object group. This data does not include the cost of managing an object group. Object migration from mBase to mStorage needs the cost of a *Write* operation to a storage object and a *Delete* operation on mBase. Object migration from mStorage to mBase needs the cost of a *Read* operation of mStorage and a *New* operation of mBase.

Table 2 shows the cost of the operations provided by a Storage object. We used a 100-megabyte SCSI disk device with 85 segments on the disk device.

Table 2: Cost of Storage Operations

operation	*cost (in msec)*
Connect	1.1
Disconnect	13.9
Newfs	4,402.4
Scan (initial state)	5,840.6
Create (1 object)	15.1
Read (1 object)	97.6
Write (1 object)	83.0
Clean (initial state)	193.4

5 Conclusion

We have shown a persistent object system in the Apertos operating system. The reflective architecture of the operating system and object migration realizes a persistent object system. With this architecture, an Apertos object becomes persistent by migrating in an mPersistent metaspace. The mStorage metaspace provides an abstraction of the object storage, and *load* and *save* operations are replaced with object migration. The users/programmers of Apertos operating system can treat changing properties of objects, including location of objects, scheduling policy, memory management policy, and persistence, within a single interface, i.e., object migration. Adaptation of existing operating system facilities is possible in meta programming level. These shows the advantages of reflective architecture in the construction of an operating system.

A prototype system of the Apertos operating system is currently operating. In the prototype system, a storage system based on a log-structured file system is implemented.

Our major goal of the research is to provide a continuous computing environment in the Apertos distributed operating system. To realize this goal, the following issues remain to be solved.

- Mobile computers can not offer large computation power, large disks, or large memory. Thus, it is difficult for an entire computing environment to migrate to a small computer such as laptop and notebook types. The operating system should support two operations: (1) to separate part of a computing environment from an entire computing environment, and (2) to install a separated part of the computing environment into an original system.

- A local execution environment depends heavily on the underlying hardware configuration and system software. Thus, a personal computing environment should adapt itself to the underlying execution environment to continue the computation. It is difficult to provide the same computing environment on all computers. Thus, the system has to provide a logically continuous computing environment rather than a full-compatible computing environment. For example, editing a program does not need the same editing tool. *Emacs* and *vi* may be used interchangeably to continue editing text files.

The prototype implementation of the Apertos operating system is available to anyone for nonprofit purpose.[3]

Acknowledgments

We thank the members of Sony Computer Science Laboratory Inc. Several discussions with these people were helpful to us in designing the structure of the system. Further, we thank Mr. Nobuhisa Fujinami and Dr. Shinji Kono of Sony Computer Science Laboratory Inc. who gave us many useful pointers for improving this paper. We also thank Mr. Atsushi Mitsuzawa and Mr. Kenichi Murata of Keio University who helped us to implement the system.

References

[1] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 47–76, February 1987.

[2] David R. Cheriton. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 77–107, May 1985.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[4] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann Publishers, Inc., 1991.

[5] Nobuhisa Fujinami and Yasuhiko Yokote. Naming and Addressing of Objects without Unique Identifiers. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992. also appeared in SCSL-TR-92-004 of Sony Computer Science Laboratory Inc.

[6] Won Kim. *Introduction to Object-Oriented Databases.* The MIT Press, 1990.

[7] Barbara Liskov. DISTRIBUTED PROGRAMMING IN ARGUS. *Communications of the ACM*, Vol. 31, No. 3,, March 1988.

[8] J. Eliot B. Moss. *Nested Transactions.* The MIT Press, 1985.

[9] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 1–15, October 1991.

[10] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, Vol. 23, No. 5,, May 1990.

[11] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 3, pp. 204–226, Aug. 1985.

[12] Mario Tokoro. Toward Operating Systems for the 2000's. In *Proceedings of the Workshop on OS for the 90th and Beyond July 1991, Kaiserslautern, Germany. Also appears in a volume of LNCS*, July 1991. also appeared as Technical Report SCSL-TR-91-005.

[13] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992.* ACM Press, October 1992. also appeared in SCSL-TR-92-014 of Sony Computer Science

[3]Information about the distribution of the Apertos operating system is available through anonymous FTP from `scslwide.sony.co.jp`.

Laboratory Inc.

[14] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. *ACM Operating Systems Review*, Vol. 25, No. 2, pp. 22–46, April 1991. also appeared in SCSL-TR-91-002 of Sony Computer Science Laboratory Inc.

[15] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of ECOOP'89 European Conference on Object-Oriented Programming*, July 1989. also appeared in SCSL-TR-89-001 of Sony Computer Science Laboratory Inc.

[16] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.