The Object-Stacking Model for Structuring Object-Based Systems

Yasushi Shinjo Yasushi Kiyoki

Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba, Ibaraki 305, JAPAN
Phone: +81 298 53 5163 Fax: +81 298 53 5206
Email: yas@is.tsukuba.ac.jp, kiyoki@is.tsukuba.ac.jp

Abstract

Object-stacking is a model for structuring object-based systems and a mechanism for integrating multiple servers. This paper describes the object-stacking model and the structure of a distributed operating system based on this model.

In object-stacking, objects are classified into stackable objects and bottom objects. These objects have uniform interfaces. Each stackable object holds the identifiers of other stackable objects or bottom objects as lower objects. Functions of stackable objects are implemented by calling their lower objects. Bottom objects are provided by the system. Complex objects are created by stacking those objects.

In object-stacking, each server provides a single high-level service, such as filtering, caching, masking, or grouping. These servers can be used together by stacking their objects.

1 Introduction

Object-stacking is a model for structuring object-based systems and a mechanism for integrating multiple servers. In this paper, we describe the object-stacking model and the structure of a distributed operating system based on this model. This model is very simple and powerful, and can be used in the wide range of object-based systems and applications.

In distributed operating systems, object-stacking is not only a mechanism for integrating multiple servers but also an implementation method of highly functional servers. For example, a server which provides both a caching service and a replication service is implemented by combining a caching server and a replication server.

Our research of object-stacking has been motivated by the following three technologies:

(1) Object-based distributed operating systems. Most of the modern distributed operating systems are constructed as object-based systems [1,2,4,5,7,12,22]. Object-stacking can be used in these systems. These systems consist of clients, servers and a distributed kernel. Servers manage objects, such as files, processes, or directories, and provide services to clients. The distributed kernel provides physical I/O and IPC (interprocess communication) between clients and servers. Object-stacking requires system objects, object identifiers, and IPC.

Object-stacking is originated in the development of the ReSC distributed operating system which we have proposed [18,19,20,24]. ReSC is an object-based multiprogramming system for parallel/distributed as well as sequential applications. The kernel of ReSC provides a good environment for object-stacking.

Furthermore, it is very active to develop distributed file systems [9,10,21]. One of the main research topics is to find algorithms to maintain cache consistency efficiently. Object-stacking is independent of such algorithms. By using object-stacking, a *generic* replication file server can be developed. In conventional replication servers, a single type of files can be replicated. On the other hand, by using object-stacking, any type of files can be replicated [18].

(2) Implementing a complex function by combining simple functions. Pipe in UNIX is one of the most successful mechanisms for combining commands. Existing simple commands are connected by pipes, and a complex command is created. In object-stacking, the functions of multiple servers are integrated by *stacking* their objects, and as a result, a complex object is created.

(3) Layered programming. Layered programming provides a good method for developing a large system. A large system is divided into highly reusable modules. Similarly, object-stacking can be used for implementing a large server. A large server is divided into highly reusable simple servers.

In Section 2, we describe the object-stacking model. In Section 3, we show the overview of the ReSC system as an environment of object-stacking, and discuss requirements and desired features for constructing

International Workshop on Object-Orientation in Operating Systems (I-WOOOS'92)
Paris, France, September 24-25, 1992.

distributed operating systems. Section 4 presents an example of a server which manages objects for object-stacking. Multiple views and caching are discussed in Section 5. In Section 6, we describe our current status and future direction. In Section 7, we conclude this paper.

2 The object-stacking model

The object-stacking model is based on the following four key concepts:

(1) Objects. In object-stacking, objects are encapsulations of data and exported procedures. In object-based distributed operating systems, each object is managed and protected by its server process. In Figure 1, Server A is a file server, and manages File object a1 and a2. The server receives read and write request messages from clients, executes the procedures, and protects its objects from invalid operations.

In object-stacking, objects are accessed indirectly by using their *identifiers* (object ids). In Figure 1, Client 1 holds the id of Object a1, and sends read and write request messages to this object. These messages are directed to Server A, that is, the server of Object a1. The server interprets the messages.

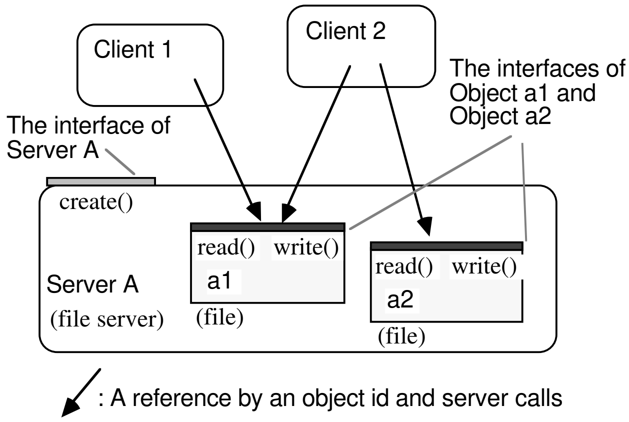

Figure 1: Objects, their server and clients.

The set of exported procedures of each object is referred to as the *interface* of the object. For example, the interface of a file object is the set of the read and the write procedure. Like the interface of an object, the interface of a server is defined as the set of exported procedures of the server. For example, any server contains an object creation procedure in its interface.

(2) To stack objects. To stack Object b1 on Object a1 means that Object b1 holds the id of Object a1, and Object b1 calls Object a1 to implement its functions. In this call, the server of Object b1 is a client of the server of Object a1. Object b1 *is stackable on* Object a1 if Object a1 has a sufficient interface and functions to implement the functions of Object b1.

Holding object ids does not always mean stacking. For example, although a directory object holds object ids and string names, the directory object is not stacked on the objects which correspond to the object ids. In this case, the directory object does not send messages to the object to implement its function.

If Object b1 is stacked on Object a1, Object b1 is called an *upper object* of Object a1, and Object a1 is called a *lower object* of Object b1. The ids of lower objects are passed to the server of the upper object as arguments to an object creation procedure.

(3) To view a stack of objects as an object. On stacking an object on another object, the lower object may not be a system-provided object. That is, an object can be stacked on top of a stack of objects. We refer to a stack of objects as a *stack object*, the top of a stack as *the top object,* and the bottom of a stack as *the bottom object*.

Only bottom objects do not depend on other objects, and do not hold ids of the other objects. Since bottom objects cannot be implemented by stacking objects, they are provided by the system.

In Figure 2, Object a1 is a bottom object, Object b1 is stacked on Object a1, and Stack Object b1/a1 is constructed. Object b1 holds the id of Object a1, and calls Object a1 to implement its functions. Object c1 is stacked on Stack Object b1/a1, and Stack Object c1/b1/a1 is constructed.

The id of the top object of a stack object is used to identify the stack object. In Figure 2, for example, Stack Object c1/b1/a1 is identified by the id of Object c1. This makes it possible to deal with both stack objects and simple objects uniformly.

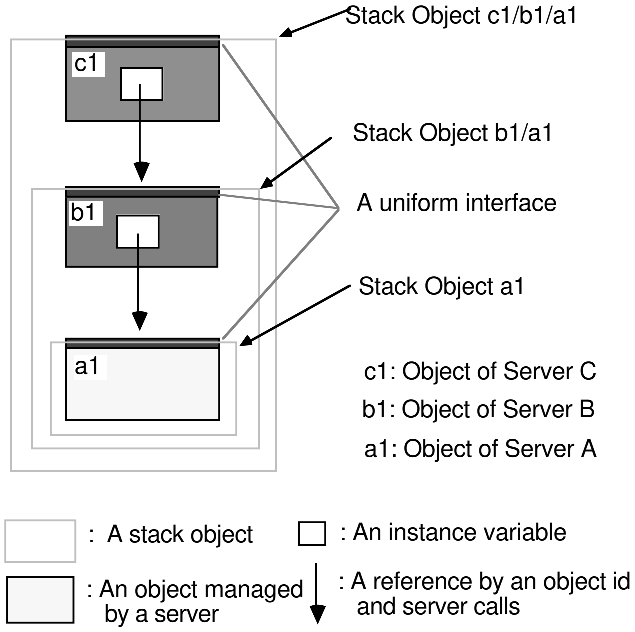

Figure 2: Objects managed by servers and stack objects.

(4) Uniform interfaces. To choose objects flexibly, they must have a uniform interface. If interfaces of objects are not uniform, the objects which can be stacked on an object are restricted.

In object-based distributed operating systems, the interface of file objects is often used as a uniform interface. That is, the object which can accept the read and the write procedure is often stacked on another object which can also accept the read and the write procedure.

If an upper object holds a lower object, and the interface of the lower object is equivalent to that of the upper object, then the upper object is called a *stackable object*. Similarly, a stackable file object means that the object has the file interface and its lower object also has the file interface. A server which manages stackable objects is called a *stackable server*. In Figure 2, Object b1 and Object c1 are stackable objects. Server B and Server C are stackable servers.

2.1 ZFS, CFS and StdFS

In this subsection, we show an example of object-stacking by using the following three file servers (Figure 3):

- **StdFS** (Standard File Server). Each object takes no lower objects. Data of a file object is stored in disk blocks.
- **ZFS** (Compression File Server). Each object takes a lower object for storing compressed data. When the server receives a write request to an object from a client, the server compresses the data in the request, and sends a write request with the compressed data to the lower object of this object. When the server receives a read request to an object from a client, the server sends a read request to the lower object of this object. The server uncompresses the data which is returned by the lower object, and returns the uncompressed data to the client.
- **CFS** (Encryption File Server). CFS is very similar to ZFS, except that CFS performs encryption or decryption instead of compression or uncompression.

In ZFS, each object usually holds an object of StdFS as a lower object, that is, a file of StdFS is used to store the compressed data. As a result, a compressed file (1 in Figure 3) is created. If an object of ZFS holds an object of CFS instead of StdFS, that is, if a file of CFS is used to store the compressed data, a compressed encrypted file (2 in Figure 3) is created.

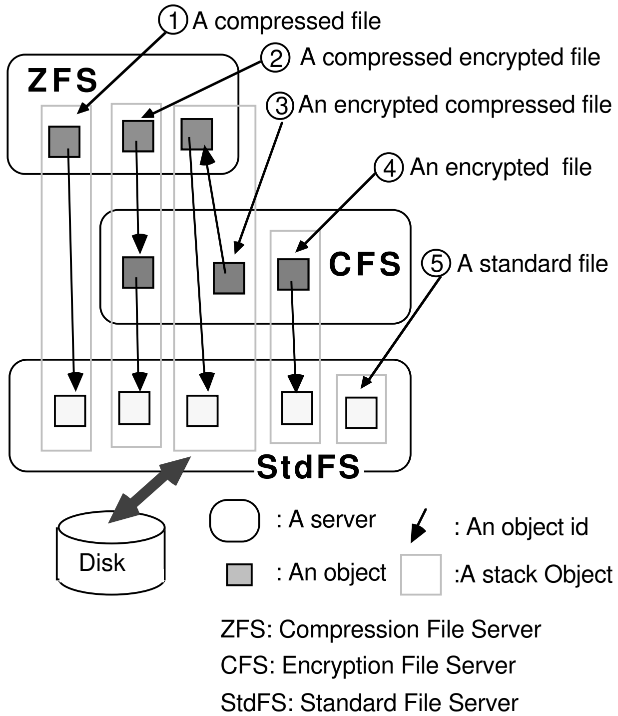

Figure 3: Compression, Encryption, and Standard File Server.

In object-stacking, the order of stackable objects is not fixed. In this example, it is possible to stack objects in the order ZFS/CFS/StdFS (2 in Figure 3) as well as the order CFS/ZFS/StdFS (3 in Figure 3). (In the case of compression and encryption, the order affects compression ratio. To compress and then encrypt is better than to encrypt and then compress.)

2.2 Filters in UNIX and object-stacking

In UNIX, many commands can be connected by pipes and used together. Such commands are called *filters*. For each filter in UNIX, its corresponding file server can be built. For example, from the command line of UNIX:

```
% tr A-Z a-z <from >to
```

the filter server which translates upper case characters to lower case ones is built. When this server receives a read request to an object from a client, the server sends a read request to its lower object. The server translates the data returned by the lower object to lower case ones, and returns the translated data to the client.

This filter file server provides read-only files because the write operation cannot be defined. ZFS and CFS, which are described in Section 2.1, are also filter file servers. They provide read-write (readable and writable) files. In general, if a filter command does not have any inverse command, its corresponding filter file server provides read-only or write-only files. (Write-only files are found in the mail system and the logging system.)

In the framework of object-stacking, a set of commands connected by pipes in UNIX can be regarded as a stack of process objects. Unlike a stack of file objects, a stack of process objects is *volatile*, and a new stack of process objects is created every time the commands are invoked. On the other hand, a stack of filter servers' file objects is *stable*, and no new object is created even though it is accessed multiple times. Therefore, caching techniques can be used to improve performance.

2.3 Server-stacking

The concept of stacking servers has been proposed [9,16]. We refer to this concept as *server-stacking*. In object-stacking, when a new object is created in a server, the ids of its lower objects are passed to the server as arguments to an object creation procedure. In server-stacking, when a server starts up, the id of the lower server is passed to the server as an argument to the server's program. If servers are stackable, that is, if they hold other lower servers and the interface of the lower servers is equivalent to their interface, the configuration and order of the servers can be chosen flexibly

In server-stacking, the concept of objects may not exist, and services may be provided through procedure calls. If the concept of objects exits, every stack of objects would have a uniform structure, as shown in Figure 4-a. For example, if CFS is stacked on StdFS and ZFS is stacked on CFS (cf. Section 2.1), every file would be a compressed encrypted file.

Server-stacking is a special and protected case of object-stacking. First, this is because the server itself can be regarded as an object. Secondly, stacks in object-stacking may have a uniform structure as a result.

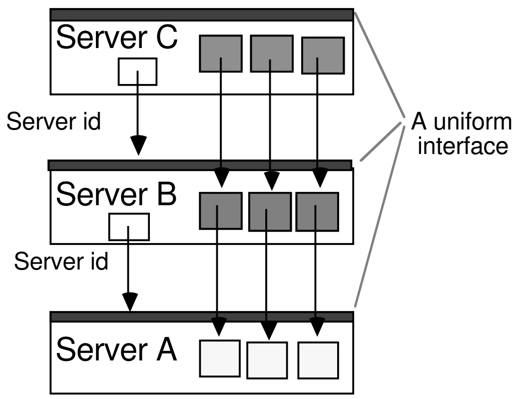

Figure 4-a: Server-stacking with objects.

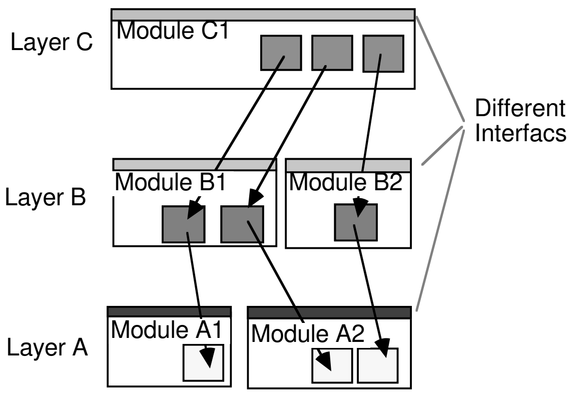

Figure 4-b: Layered programming with objects.

2.4 Layered programming and object- or server-stacking

The concept of server-stacking is similar to that of layered programming. A server corresponds to a module of a layer. A fundamental difference is that in layered programming, interfaces between layers are different, and the configuration and order of the layers are fixed. In server-stacking, the interfaces of servers are uniform, and the configuration and order of the servers can be chosen flexibly.

In layered programming, the concept of objects may not exist. However, if the concept of objects exists and each layer consists of multiple modules, modules can be chosen flexibly as similar to object-stacking (Figure 4-b). In addition, object- and server-stacking can be used within a layer.

2.5 Abstract data types, subtypes and object-stacking

Object-stacking is related to ADT (Abstract Data Type) and subtypes in programming languages [3,6]. A service, a server and an object in object-stacking correspond to an ADT, a class (an implementation of an ADT) and an instance of an ADT (also an instance of a class) in programming languages with ADT, respectively. For example, the file ADT contains the read and the write procedure, and every instance (object) of the file ADT can accept these procedures. In the example described in Section 2.1, Standard File Class, Compression File Class and Encryption File Class are implementations of the file ADT.

ADT does not include the key concepts (2) and (4), which are described at the beginning of this section. (The key concept (3) is interpreted as subtyping.) In object-stacking, every stackable object holds the ids of its lower objects, and calls those lower objects to implement its functions. The lower objects must be instances (objects) of the ADT to which the upper object belongs (or instances of a subtype of the ADT). For example, a stackable file object has instance variables of the file ADT. To implement its functions, it calls the objects which are pointed to by the instance variables.

2.6 Inheritance and object-stacking

Object-stacking is not related to the concept of inheritance in object-oriented programming languages. For example, consider implementing ZFC (Compression File Class), as similar to ZFS described in Section 2.1. Although ZFC can be defined as a subclass of StdFC (Standard File Class), the implementations of the methods of StdFC cannot be inherited by ZFC. The read and the write method of ZFC must be reimplemented to be new ones which perform uncompression or compression. Although the interface of StdFC can be inherited by ZFC, this means that both ZFC and StdFC are two implementations of the file ADT.

It is possible to implement ZFC in an object-oriented programming language without the concept of inheritance. As similar to ZFS, every instance holds the id of a lower object of the file ADT, and sends messages to the lower object to implement its functions.

2.7 Delegation and object-stacking

Object-stacking is similar to *delegation* [11]. A lower object in object-stacking corresponds to a *prototype object* in delegation. There are four major differences. First, in object-stacking, there exists the concept of a class (a server), and an object is managed by a class. In delegation, there is no distinction between a class and an object. Secondly, in object-stacking, the objects which construct a stack object have a uniform interface. In delegation, the interface of an object is usually different from that of its prototype object. Thirdly, in object-stacking, the methods that are not defined in an object are not forwarded to its lower objects, but they are rejected. In delegation, such methods are automatically forwarded (delegated) to its prototype object. Lastly, in object-stacking, no one knows the object that originally receives a message. In other words, every stackable server is a client of lower objects. In delegation, the variable *self* holds the object that originally receives a message.

2.8 Meta-objects and object-stacking

Lower objects in object-stacking are not related to *meta-objects* in the Muse operating system [23]. Muse is a distributed operating system based on object-oriented concurrent computing and reflective computing. In Muse, everything is defined as an object. Each object in Muse is supported by one or more meta-objects. A meta-object of objects is an object which provides an execution environment for these objects. A meta-object itself is also supported by other objects (meta-meta-objects).

A server in object-stacking is similar to a meta-object in Muse. The main difference is that in object-stacking the relationship among servers changes in each stack object. In Muse, the relationship between an object and its meta-object is usually fixed. There is no object that is supported by a meta-object and the meta-object is supported by the object reversely. In object-stacking, on the other hand, stackable servers depend on one another. For example, ZFS and CFS in Figure 3 depend on each other.

A lower object in object-stacking is similar to a meta-object in Muse. The main difference is that in object-stacking the behavior of an upper object is not determined by its lower object but by its server. In Muse, the behavior of an object is determined by its meta-objects.

2.9 Generic servers

Constructibility of generic servers is one of the advantages of object-stacking. A *generic server* is a server that can handle the objects of any server. Mask Server and an indirection server are two examples of such servers. Mask server forwards some requests to lower objects and blocks the other requests. For example, the read-only file of a read-write file can be created by masking of write requests. An indirection server forwards most request messages to the lower objects without modifying those request messages. These servers can handle file objects and directory objects.

In object-stacking, a generic cache file server can be built. *A generic file server* means a server that can deal with not only the file objects of the kernel file server but also the file objects of any other file server, such as a filter file server.

3 The ReSC distributed operating system and environments of object-stacking

Object-stacking is originated in the development of the ReSC distributed operating system [18,19,20,24]. ReSC is an object-based distributed operating system, and its kernel provides a good environment of object-stacking. In this section, we show the overview of ReSC as an environment of object-stacking, describe requirements to use object-stacking, and discuss desired features of the kernel for developing distributed operating systems by object-stacking.

3.1 The ReSC distributed operating system as an environment of object-stacking

ReSC is a multiprogramming system for parallel/distributed and sequential applications [19]. Its hardware environment is a set of machines which are connected with a high-speed local area network, as shown in Figure 5. Each machine is called a site. Each site is a shared-memory multiprocessor or a uniprocessor.

In ReSC, applications are classified into two types: parallel/distributed applications and sequential applications. According to the type of each application, the system can be viewed as a centralized system or a distributed system [18].

The ReSC system consists of a *distributed kernel,* external servers, and libraries. The kernel provides only the following facilities. Other system facilities are provided by the external servers and libraries.

(1) Management and fair partitioning of shared resources.
(2) Inter-application communication, authentication, and protection of processes.
(3) Simple and efficient facilities of the underlying hardware.
(4) Standard interfaces.

The first and second are fundamental services for all types of applications. The third is an efficient service for parallel/distributed applications. To accomplish the third facility, the kernel includes complete servers rather than device servers. These servers are called *kernel servers*. The facilities (3) and (4) give the basis of object-stacking. We will discuss it in Section 3.2 (4).

A server outside the kernel is called an *external server*. Each external server is a process or a set of processes which are distributed on different sites in the network. External servers are integrated by using object-stacking, and provide high-level functions. Therefore, sequential applications use the external servers mainly for the high-level functions.

3.2 Requirements for implementing object-stacking

Requirements for implementing object-stacking are summarized as follows:

(1) Objects. The concept of objects is essential. An object is an encapsulation of data and exported procedures. ReSC is an object-based system, and system facilities and resources are provided through objects.

(2) Object Ids (Object Identifiers). Every object must be referenced by an object id in a uniform way. Object ids are used to hold lower objects. This uniformity is required to use any type of objects as lower objects.

In ReSC, every object is identified by a uniform object id. Each object id is a triple

< *site-id*, *server-id*, *object-number* >,

where the *site-id* indicates the location of the server, the *server-id* identifies the communication port of the server, and the *object-number* is an internal number managed by the server.

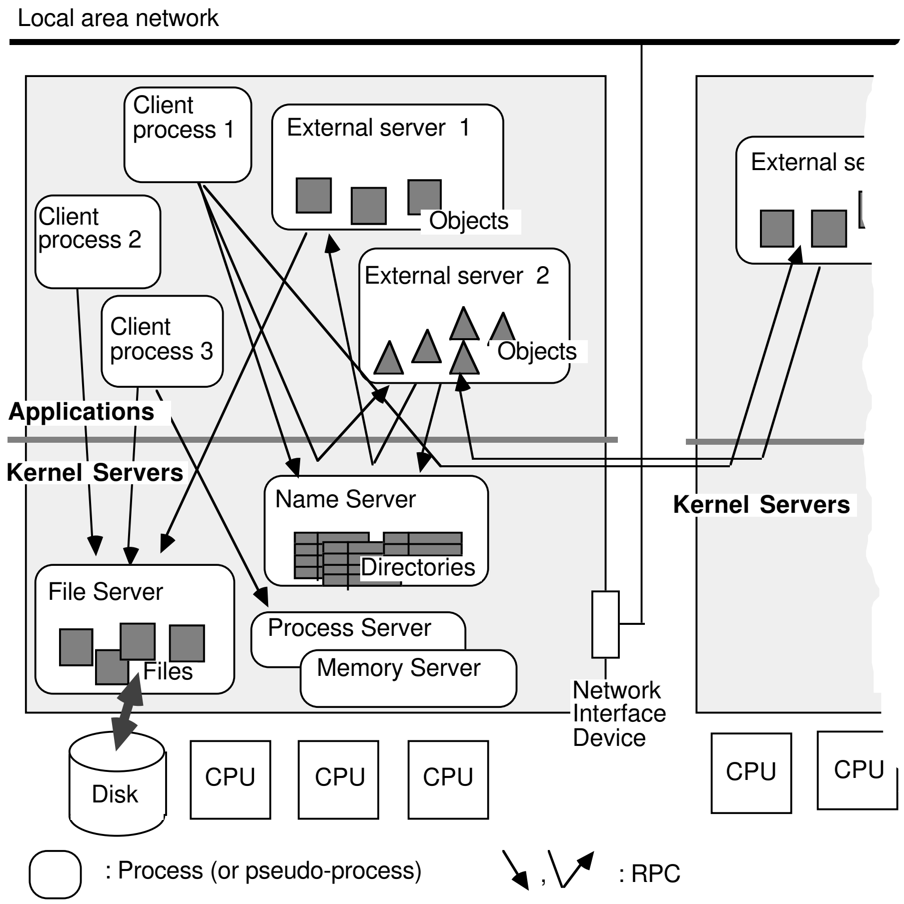

Figure 5: An overview of the ReSC distributed operating system.

The server-id is a unique number which is assigned to each server in the system by the system manager. The distributed processes of a server share a common server-id. (We will discuss site-ids in Section 3.3 (3))

(3) Indirect manipulation of objects. Objects must be manipulated at least by sending request messages indirectly, and the system must provide the facility for sending messages. However, it is not required to map objects into the address spaces of their clients.

In ReSC, each object is managed and protected by its server. The kernel provides an RPC (Remote Procedure Call) facility as an IPC primitive. A client invokes an RPC with object ids as arguments, and manipulates the objects indirectly.

(4) Bottom objects. The system must provide bottom objects because they cannot be implemented by stacking objects. In ReSC, bottom objects are provided by kernel servers.

Bottom objects provide not only basic functions but also standard interfaces. In the V system [4], for example, the kernel disk server provides access to each drive as a raw block device. It is not sufficient to provide only the raw block device interface because this interface is designed for the file server outside the kernel, not for users. In this case, objects which are provided by the file server outside the kernel are suitable as bottom file objects.

(5) Multiple servers for a single service. The system must allow multiple servers for a single service, and the servers must be dealt with in a uniform way. In ReSC, since each object id includes its server id (not its service id), the multiple servers can provide a single service simultaneously. Furthermore, additional servers can provide the same service as existing system services, such as the file service and the name service. The additional servers can be dealt with in the uniform way as the existing system servers.

In Internet [15], only one server can run for each service on a single host. This one-to-one mapping of service to server does not allow to implement object-stacking. For example, the port number of the Internet mail server is fixed. Therefore, it is impossible to replace the original mail server by a new one which stacks on the original mail server.

(6) Free choice of lower objects. Clients must have a control facility for choosing lower objects. In ReSC, each server has an object creation procedure which takes an array of ids of lower objects (cf. Section 3.4).

In NFS, for example, it is impossible to choose lower object in object creation procedures (create, symlink and mkdir). Therefore, object-stacking cannot be used in NFS.

Most distributed operating systems satisfy the requirements (1) to (4) easily. The requirement (5) is essential, and some systems, such as Internet, do not satisfy this requirement. In those systems, object-stacking cannot be implemented. In terms of the requirement (6), it is needed to rewrite system call (server call) interfaces.

3.3 Desired features for constructing distributed operating systems

This subsection discusses desired features of the kernel for constructing distributed operating systems by object-stacking. The ReSC kernel satisfies all the following features:

(1) To assign stable ids to stable objects. It is desired to assign stable ids to stable objects. A *stable object* and a *stable id* mean an object and an id which do not change after a crash and restart of the server. For example, files and directories are thought to be stable objects. It is useful for clients to use the same object id for a stable object because it is not needed to get a new object id after a crash and restart of the server. In object-stacking, as there are many stable objects which hold the ids of stable lower objects, this feature is desired.

In ReSC, each object id is a triple of a site-id, a server-id, and an object-number, as described in Section 3.2 (2). Site-ids and server-ids are stable. Therefore, stability of object ids depends on that of the object-number which is managed by their servers.

(2) Connection-less IPC. It is desired to provide connection-less IPC for constructing a robust system effectively. Unless connection-less IPC is provided, it is needed to reestablish connections between clients and servers after a crash and restart of servers. In object-stacking, as a stackable server is also a client of the servers of its lower objects, the overhead and complexity of the reestablishment become large. In ReSC, we choose RPC as connection-less IPC.

(3) To Support both location-dependent and -independent objects. It is desired to support both location-dependent and -independent objects. This is because in a distributed system there are location-dependent objects inherently as well as location-independent objects. For example, physical disk drives, displays, keyboards and floppy disk drives are location-dependent objects.

In ReSC, this feature is supported by embedding location information in the site-ids of object ids. Site-ids are classified into three types: a unique number which is assigned to a site, the number which indicates the local site, and a number which means a broadcast or multicast address. By using site-unique site-ids, location-dependent objects are supported. By using local, broadcast or multicast site-ids, location-independent objects are supported.

(4) Lightweight processes. It is desired to provide a lightweight process facility. In multiprogramming systems, a lightweight process is the unit of parallel execution within a process, which is the unit of resource allocation and protection. This facility is essential for parallel applications which run on shared-memory multiprocessors. In addition, it is known as a useful facility for constructing a server which deals with multiple clients simultaneously. In object-stacking, as it is needed to create many servers to implement high-level functions, this facility is widely used.

ReSC provides a lightweight process facility with the concepts of *microprocesses* and *virtual processors* [18,19]. A microprocess is a user-level lightweight process and can be realized efficiently without any kernel calls. A virtual processor is an entry of a real processor which is assigned to the application process by the kernel. By using this facility, each parallel application can develop its own application-specific schedulers in its user address space [19].

(5) Forwarding request messages. It is desired to provide the facility of forwarding request messages [4]. By using this facility, reply messages are sent directly from forwarded servers to clients, and the overhead of message passing is reduced. In object-stacking, as there are many *indirection servers* which forward most requests to the lower servers, this facility contributes to improving performance.

(6) To separate name servers from object servers. It is desired to separate name servers (directory servers) from object servers. In object-stacking, a large number of object servers are used simultaneously. If each object server includes its own name server, naming of objects is influenced by the server type. In ReSC, name servers are separated from object servers, as similar to Amoeba [12].

3.4 Creating stack objects

In ReSC, users can deal with a stack object not only as a stack of objects but also as a single object. This facility is realized by the following server procedures:

(1) **create_with_lower:** This is used to create a stack object. This procedure takes an array of object ids and options. The server creates a new stack object with the given object ids, and returns the id of the new stack object.
(2) **copy:** The server creates a new object by *deep-copy*, that is, the lower objects are created and copied recursively.
(3) **create_like**: As similar to the procedure (2), the server creates a new object which has the same structure as the given object. The difference from the procedure (2) is that the contents of the given object are not copied, but an empty object is created. This procedure is also invoked recursively.

By using the procedure (1), a client can make a stack object flexibly. By using the procedures (2) or (3), a client can make a stack object without knowing its structure. The procedure (2) is used in a copy command. The procedure (3) is used for programs to create a new empty object which has the same structure as an old one.

Template object bases: In ReSC, frequently used objects are stored in a directory. This directory is called a *template object base*. For example, the following objects are stored in the directory:

- std-file
- compressed-std-file
- encrypted-compressed-std-file

The file std-file is an empty file object of StdFS, which is described in Section 2.1. (An empty file of StdFS is registered with the symbolic name "std-file" to the directory.) The file compressed-std-file is an empty file object of ZFS, which holds an object of StdFS as its lower object. The file encrypted-compressed-std-file is an empty file object of ZFS, which holds an object of CFS as its lower object, where the object of CFS holds an object of StdFS as its lower object.

Objects in this template object base are created by a system manager with the procedure create_with_lower. They are used by a user who copies them into his/her working directory.

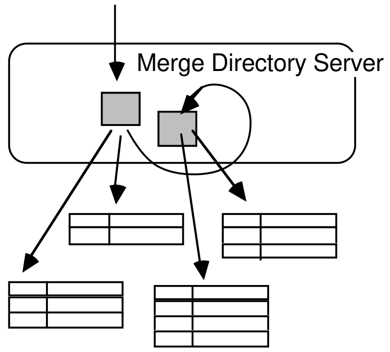

Figure 6: Merging four directories.

4 An example of a stackable server

In this section, we show the implementation of Merge Directory Server as an example, and describe the structure of a typical stackable server. In Merge Directory Server, each object takes multiple directory objects as lower objects, and provides a directory object which holds all the entries of the lower directory objects. In Figure 6, four directories are merged by this server. In Plan 9 [14], this facility is provided by the kernel. By using object-stacking, it can be implemented outside the kernel.

In the implementation of Merge Directory Server, the following techniques are used:

(1) It is implemented as a cache server. This is one of the unique points of object-stacking. We will describe it in Section 4.2.
(2) A lightweight process is created for each RPC acceptance. Lightweight processes are used to realize parallel processing of requests, mutual exclusion, mutual calls, and recursive calls.
(3) Directories are thought to be stable objects, and they should be unchanged after a crash and restart of their server. To implement stable directories, we use structures in files and objects in main memory. This is a traditional method for implementing stable objects.

This server is a group server. A group server means that each object takes multiple lower objects to realize grouping of objects [17].

4.1 Components

Merge Directory Server consists of the following components:

The RPC acceptance lightweight process. A lightweight process is waiting for requests from clients by invoking an RPC acceptance primitive. When a request arrives from a client, this lightweight process checks the procedure number in the request, and creates a new lightweight process to call the corresponding *RPC entry procedure*. If there is no such RPC entry procedure, the lightweight process sends an error to the client.

RPC entry procedures. These are called as top routines of lightweight processes by the RPC acceptance lightweight process. The flow of each entry procedure is as follows:

(1) Each entry procedure calls *the local object manager* explicitly, and translates the object id in the argument of RPC into the pointer to *the local object* which is allocated in heap memory. If there is no such object, the entry procedure sends an error to clients and terminates the current lightweight process.
(2) It calls the corresponding *local procedure* with the pointer to the local object as an argument. If the object does not accept the procedure, a dummy procedure is called and an error is sent to the client by the dummy procedure.
(3) It sends the result of the local procedure to the client.
(4) It releases the local object. (The local object stays in a cache for future use.)
(5) It terminates the current lightweight process.

Local procedures for objects in memory. These procedures are similar to methods in object-oriented languages and exported procedures in ADT. The distinctive features of local procedures are as follows:

- Checking of access rights is needed.
- Locking is needed.

Locking is implemented by using a specific lock primitive of the lightweight processes.

The local object manager. This module provides the following facilities.

(1) Allocation of a new local object and its object id.
(2) Translation of an object id into the corresponding local object in heap memory.
(3) Management of the caches of local objects.

This module is similar to the inode management module in UNIX. An object id, a local object, and a structure stored in a file correspond to *an inode number*, *inode*, and *disk inode* in UNIX, respectively. Differences from UNIX are as follows:

- Unlike UNIX, two files are used to store stable structures: One is the files for the fixed length parts. The other is the file for the variable length parts. In UNIX, disk inodes and data, which correspond the fixed and the variable length parts, are stored in disk blocks.
- Like capabilities in Amoeba [12], object ids in ReSC are handled by users. An object number in an object id consists of a serial number, a node number, and a check field. The serial number guarantees uniqueness of the object id. The node number denotes the offset of the fixed length part in the file, as similar to an inode number in UNIX. The check field is used to prevent users from forging new object ids or tampering with existing ones, as similar to Amoeba.

4.2 Implementation of Merge Directory Server by Caching

In this subsection, we describe the implementation of major procedures in Merge Directory Server as a directory server.

create_with_lower: This procedure takes the object ids of the directories to be merged as lower objects. First, the server calls the dir_nop (directory no operation) procedure of each lower object, and inquires of it whether it is a directory. Next, the server creates a new object, and stores the ids of the lower objects into the new object. Last, the server returns the id of the new object.

dir_read (read directory): This procedure returns all the entries of the lower directory objects of a given object. First, the server checks the cache of the given object. If the cache exists, the server calls the the getattr (get attributes) procedure of each lower directory object to get the last modify time, and checks the consistency of the cache. If no cache exists or the cache is inconsistent, the server calls the dir_read procedure of each lower object, merges all the contents of lower directories, and makes a consistent cache. The server returns the contents in the cache to the clients. The cache remains in memory for future use.

dir_lookup: This procedure takes a symbolic name of an object and returns the corresponding object id. Like the procedure dir_read, the server makes a consistent cache of the given object. The server searches the cache for the given symbolic name, and returns the object id which is associated with the symbolic name.

4.3 Caching by using objects of other servers

Since Merge Directory Server handles only directories, the caches of active objects can be placed in main memory. If large caches are needed, objects of other servers can be used.

For example, ZFS (cf. Section 2.1) uses temporary files of other file servers to store uncompressed data. When ZFS receives a read or write request to an object, the server uncompresses the whole data of its lower file object, and writes the uncompressed data to a temporary file. Subsequent requests to the file are directed to the temporary file. If the server does not receive messages to the file for a while and the temporary file is modified, the server reads the temporary file, compresses the data in the temporary file, and writes to the lower file object.

5 Multiple views and caching

Multiple views of an object mean that the object can be used through multiple interfaces. Multiple views can be realized by a conceptual object which consists of a lower object and multiple upper objects. In Figure 7, a lower file object is referenced by three upper objects: an object of an indirection server which forwards all the requests to the lower object, an object of Mask Server which blocks write requests (cf. Section 2.9), and an object of a filter file server which translates upper characters into lower ones. As a result, the three views of the file are created.

View 1: The read-write view.
View 2: The read-only view.
View 3: The lower case view.

5.1 Multiple views by accessing multiple layers

Every stack object provides its multiple views by allowing access to lower level objects [20]. Figure 8 shows a stack object with three views.

Allowing access to lower level objects may cause some troubles. For example, consider a read-only file which is created by stacking an object of Mask Server on a read-write file object. Although the upper object masks the write request, the lower read-write object can be directly accessed without passing the upper object. By keeping the ids of lower objects secret, this direct access is prohibited. To keep ids secret can be realized by using a cryptographic technology, as similar to *capabilities* in Amoeba [12].

5.2 Maintaining cache coherency

In providing multiple views of an object, maintaining cache consistency is one of the problems to be solved [24]. In Figure 7, if View 1 (the writable view) receives a write request, the caches of the other views may be inconsistent. A solution of this problem is to introduce a coherent cache file server. As shown in Figure 9, the cache of every view is related to one another. When this server receives a write request for a cache, the server invalidates the other related caches.

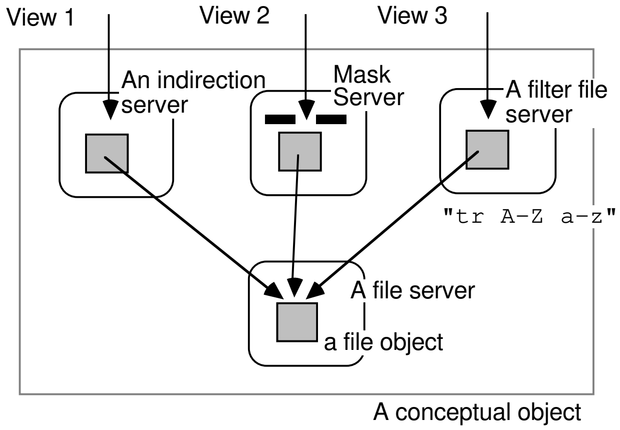

Figure 7: An object with three views by three upper objects.

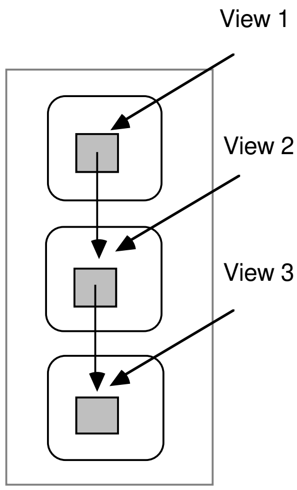

Figure 8: A stack object with three views.

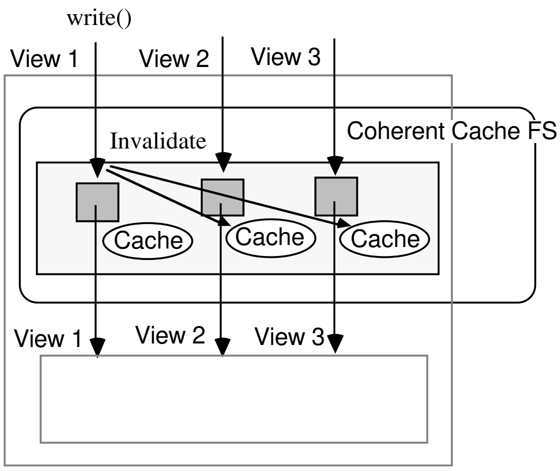

Figure 9: Coherent Cache File Server.

6 Current status and future work

We have implemented two environments of object-stacking. One is the ReSC kernel on Omron Luna88k, a shared-memory multiprocessor workstation. In this environment, the local version of the ReSC RPC and kernel servers are operational. We are now implementing the network communication facility.

The other is the environment that is implemented by using Sun RPC and NFS [13]. The ReSC RPC is emulated by using Sun RPC. A bottom file server is implemented by calling the NFS server directly.

Object ids in the latter environment are realized as follows:

- Site-ids: We use Internet host addresses as site-ids.
- Server-id: For each server, we allocate a unique small (4-byte) integer as its server-id. This is used to calculate the port number of UDP/IP for Sun RPC.
- Object-numbers: An object number is a 32-byte opaque. This is as long as an *NFS file handle*, that is, an object id in NFS. This makes it easy to implement the bottom file server which calls the local NFS server directly. In the bottom file server, each object takes an NFS file of NFS, and its object-id includes the file handle of the NFS file.

We are developing stackable servers in the C language. It is a confusing task for us to write a stackable server because a stackable server is not only a server but also a client. Both the client stub and the server stub for each RPC entry are used in a stackable server's program. Currently, we give them different names. The same name should be given to them because they provide a similar function. This can be realized by using a programming language which provides overloading, like C++[8].

We are developing a filter server which translates NFS protocols to the server calls of ReSC. Since the upper interface differs from the lower one, this is layered programming, neither server- nor object-stacking. This server enables UNIX applications to use stack objects in ReSC. However, stack objects cannot be created by UNIX applications, as described in Section 3.2 (6) and Section 3.4. Therefore, stack objects must be created by using special commands which invoke ReSC server calls.

7 Conclusion

Object-stacking is a model for structuring object-based systems, and a mechanism for integrating multiple servers. This model is very simple and powerful, and used in the wide range of object-based systems and applications. In this paper, we have described the object-stacking model and the structure of the ReSC distributed operating system based on this model. ReSC is an object-based distributed operating system for parallel/distributed and sequential applications. The kernel of ReSC provides a good environment of object-stacking.

In ReSC, each stackable server outside the kernel provides a high-level function, such as, filtering, caching, or merging directories. These servers are used together by stacking their objects.

Several stackable servers have been implemented on the ReSC kernel and the environment that is implemented by using Sun RPC and NFS. In our future work, we would like to develop a toolkit for object-stacking and formalize object-stacking as a programming model.

References

[1] M.J.Acceta, R.V.Baron, W.Bolosky, D.B.Golub, R.F.Rashid, A.Tevanian, and M.W.Young: "Mach: A New Kernel Foundation for UNIX Development", USENIX 1886 Summer Conf., 1986.

[2] A.P.Black: "Supporting Distributed Applications: Experience with Eden", Proc. 10th ACM Symp. on Operating System Principles, pp.181-193, 1989.

[3] L.Cardelli and P.Wegner: "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys, Vol.17, No.4, pp.471-522, 1985.

[4] D.R.Cheriton: "The V Distributed System", CACM, Vol.31, No.3, pp.314-333, 1988

[5] R.Chin and S.Chanson, "Distributed Object-Based Programming Systems", ACM Computing Surveys, Vol.23, No.1, pp.91-124, 1991.

[6] S.Danforth and C.Tomlinson: "Type Theories and Object-Oriented Programming", ACM Computing Surveys, Vol.20, No.1, pp.29-72, 1988.

[7] P.Dasgupta, R.J.LeBlanc,Jr. and W.F.Appelbe: "The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work", Proc. IEEE 8th Intl. Conf. on Distributed Computing Systems, pp.2-9, 1991.

[8] M.A.Ellis and B.Stroustrup: "The Annotated C++ Reference Manual", Addison-Wesley, 1990.

[9] R.G.Guy, J.S.Heidemann, Wai Mak, T.W.Page,Jr., G.J.Popek and D.Rothmeier, "Implementation of the Ficus replicated file system", USENIX 1990 Summer Conf., pp.63-71, 1990.

[10] E.Levy and A.Silberschatz, "Distributed File Systems: Concepts and Examples", ACM Computing Surveys, Vol.22, No.4, pp.321-374, 1990.

[11] H.Lieberman: "Using Prototypical Objects to Implement Shared Behavior in Object Oriented System", OOPSLA86, pp.214-223, 1986.

[12] S.J.Mullender, G.Rossum, A.S.Tanenbaum, R.Renesse and H.Staveren: "Amoeba: A Distributed Operating System for the 1990s", IEEE Computer, Vol.23, No.5, pp.44-53, 1990.

[13] "Network Programming", Sun Microsystems,Inc 1990.

[14] R.Pike, D.Presotto, K.Thompson and H.Trickey: "Plan 9 from Bell Labs", proceedings of UKUUG, 1990,

[15] J.Postel ed: "Transmission Control Protocol", RFC 793, 1981.

[16] D.S.H.Rosenthal: "Evolving the Vnode Interface", USENIX 1990 Summer Conf., 1990.

[17] K.Shimizu, M.Maekawa and J.Hamano: "Hierarchical Object Groups in Distributed Operating Systems", Proc. IEEE 8th Intl. Conf. on Distributed Computing Systems, pp.18-24, 1991.

[18] Y.Shinjo and Y.Kiyoki, "Harmonizing a Distributed Operating System with Parallel and Distributed Applications", First Intl. Symp. on High-Performance Distributed Computing, 1992. (To appear)

[19] Y.Shinjo and Y.Kiyoki: "ReSC: A Distributed Operating System for Parallel and Distributed Applications", Proc. First Intl. Conf. on Parallel and Distributed Information Systems, 1991.

[20] Y.Shinjo and Y.Kiyoki: "Multiple Views of the ReSC Distributed Operating System", Workshop on Operating Systems and Object Orientation at OOPSLA/ECOOP'90, 1990.

[21] V.Srinivasan and J.C.Mogul: "Spritely NFS: Experiments with Cache-Consistency Protocols", Proc. 12th ACM Symp. on Operating System Principles, ACM Operating System Review, Vol.23, No.5, pp.45-57, 1989.

[22] A.S.Tanenbaum and R.Renesse, "Distributed Operating Systems", ACM Computing Surveys, Vol.17, No.4, pp.419-470, 1985.

[23] Y.Yokote, F.Teraoka, A.Mitsuzawa, N.Fujinami and M.Tokoro: "The Muse Object Architecture: A New Operating System Structuring Concept", ACM Operating System Review, Vol.25, No.2, pp.22-46, 1991.

[24] "Workshop on Object-Orientation in Operating Systems", OOPSLA/ECOOP'90 Addendum to the Proceedings, edited by V.Russo and M.Shapiro, pp.81-91, 1990.