Object-Oriented Interfaces in the Mach 3.0 Multi-Server System

Paulo Guedes
OSF Research Institute
11 Cambridge Center
Cambridge, MA 02142
pjg@osf.org

Daniel P. Julin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dpj@cs.cmu.edu

Abstract

The Mach 3.0 multi-server system decomposes the functionality of the operating system between a micro-kernel, a set of system servers running in user-mode and an emulation library executing in the address space of applications. The interfaces provided by the system servers are object-oriented and both the servers and the emulation library are written in an object-oriented language.

In this paper we present how the interfaces between the components are specified and implemented to guarantee consistency and early detection of errors, yet maintaining the flexibility to extend and configure the system by adding new or modified servers without affecting existing pieces.

1 Introduction

A current trend in operating system design is to attempt to distribute the complete system functionality between a micro-kernel and a set of system servers executing in user mode and communicating by message exchange [6, 1]. This approach provides a clean separation between the different parts of the system, thus easing their development, maintenance and extension. Most of the operating system functionality is provided by the servers and can be developed and tested like any user application.

Object-oriented programming provides another dimension of software development tools to help meet the same objectives of easier development, maintenance and extensibility. One way to apply these tools to operating system design is to define an object-oriented model for all interactions between clients of the operating system and that system itself, and to use object-oriented techniques to structure those interactions.

The Mach 3.0 multi-server system uses both these approaches to operating system construction. The functionality of the operating system is provided by a three-layer architecture composed of a micro-kernel, a set of system servers and emulation libraries executing in the address space of user programs. The interactions between system components are defined using an object-oriented model, and the system itself is implemented using an object-oriented language.

The next section presents a brief overview of the Mach 3.0 multi-server. The rest of the paper describes how the interfaces exported by the servers are specified and implemented in C++. Static-type checking is used to guarantee consistency and early detection of errors, yet the system is flexible enough to allow independent development of its components and can be extended and configured by adding new or modified servers without affecting other servers or the emulation library.

2 Architecture

The "multi-server emulation system" for Mach 3.0 [8] is a research system that combines the use of several independent servers and an object-oriented approach. The primary goal of the project is to provide a generic architecture for the emulation of different operating systems. The current prototype provides binary compatibility with UNIX[1] BSD 4.3. There are three major layers (see Figure 1.). At the bottom is a standard Mach 3.0 micro-kernel, providing the basic Mach abstractions: virtual memory management, inter-process communication, task management and device handling [4].

Next comes a collection of mostly generic servers executing as tasks in user mode, that implement all the high-level functionality required of a complete operating system, independent of any given system's interface: file management, process management, networking, etc. Many of those servers are written specifically for use within this system, using an object-oriented structure and a collection of re-usable classes to simplify their development.

Finally, the third and higher layer consists of a collection of emulation libraries executing in the address spaces of us-

[1] UNIX is a registered trademark of UNIX System Laboratories in the United States and other countries.

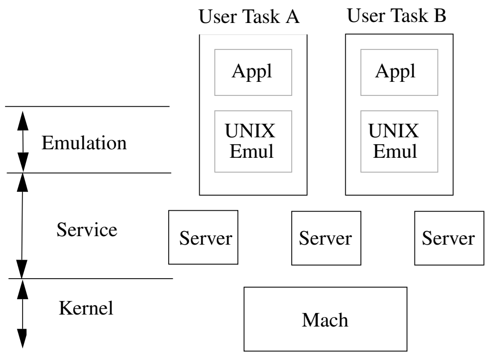

Figure 1: Architectural layering of the Mach 3.0 multi-server emulation system.

er tasks, that provide access to the generic services exported by the servers, and themselves export the specific programming interface of a given operating system (e.g BSD 4.3). When an application program executes a system call (e.g. a UNIX system call), it traps into the Mach kernel, which redirects execution into the emulation library (this is called the trampoline effect). The system call is then executed by sending one or more requests to the system servers.

Communication between the emulation library and the servers, and between the servers, primarily uses Mach IPC, but optimized mechanisms such as shared memory and file mapping are also available. Those low-level communication facilities are accessed through special proxy objects [7] loaded within the address spaces of clients, that act as representatives for each server-side object with each particular client. As a result, the actual communication code is well separated from the code of the clients themselves.

The service and the emulation layers were initially written in an object-oriented environment called MachObjects [3], consisting of a package of C macros and library routines, and providing dynamic typed objects, delegation, default methods and a generic, transparent RPC mechanism. A partial prototype written in C++ is under development. Some of the issues addressed in this paper relate only to the object model, being largely independent of the implementation language. Where this is not the case, we will refer to the C++ implementation.

3 Object-Oriented Interfaces

All the services provided by the various servers are defined in terms of operating system objects such as files, directories, devices, transport endpoints, etc. To avoid confusion with the objects used as part of the internal implementation of some servers, those exported operating system objects are often also referred to as *items*. Different items export different subsets of all the operations specified in the complete system interface, grouped in specialized interfaces such as naming, I/O, network control, device control, etc. The abstraction of a server is not explicitly exported to clients; instead, those clients only have access to a number of items, and cannot directly determine which items are managed by which servers. This gives the server writer maximum flexibility in the implementation of the servers, which may be split, combined, or in general optimized, without affecting the clients.

The interfaces provided by the service layer are defined through a set of types, each defining the operations that may be invoked on the items of that type. Parameters may be of basic types (integers, strings, etc.) or of other complex types; item references may be sent and received as parameters in operations.

An interface type may be defined by derivation from one or more base types, using single or multiple-inheritance. Single inheritance is used to reflect a common relationship between items in an operating system where all of them share a basic, common interface that is extended in different ways by different items. For example, all the items managed by the Unix filesystem share a basic interface with operations to read and set attributes, change protection, retrieve the time of last access and modifications, etc. This is modeled by defining a basic interface type with the operations available to all items, and deriving files, directories, symbolic links, mount points, etc. from this basic type. Some cases are more complicated: network endpoints, in some combinations of domain and protocol, require inheriting interfaces from more than one base type. In these cases multiple-inheritance is used to inherit interfaces from all the relevant base types.

In the C++ prototype, interfaces are specified by C++ abstract classes. Static type-checking ensures consistency between clients and servers and provides early detection of programming errors.

An important goal of the object model is to facilitate the introduction of new interfaces, or the extension of existing ones, without modifying existing emulation libraries and servers. Two basic object-oriented programming techniques are fundamental to achieve this: inheritance and polymorphism.

Extending an existing interface is achieved by deriving a new interface class from the existing one. At the server side, the new class may completely re-implement the old methods or simply rely on the implementation provided by the base class. At the client side, the old interface class may still be used. Clients are not affected and therefore need not be changed, although they cannot use the extended functionality of the new interface.

One of the limitations of this scheme is that new versions of the servers must maintain the communication interface to old clients. A new version of a server that improves the communication with the client, or simply fixes some bugs but requires a new proxy, cannot be installed without changing the existing clients. This can be avoided by dynamically linking proxies with the client's code, thus allowing a new version of a proxy to be installed without modifying the client. This is made possible by the fact that the new proxy still conforms to the type expected by the client.

Clients using an extended interface must sometimes be able to communicate with old servers that export only the old, base interface. In such a case the client must be able to check at run-time the dynamic type of the server it is communicating with.

There are other situations where run-time type checking is also needed. The Unix `open()` system call, for example, must return an object of a base type because when the name of the object is looked-up its type is not known. However, some operations are performed differently according to the actual type of object returned. The system must have a mechanism for detecting the run-time type of objects, which is used by clients when they need to differentiate between disjoint derived interfaces.

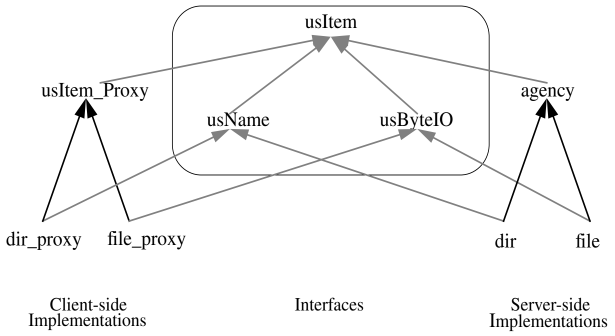

Figure 2: Class hierarchy. Classes `usItem`, `usName` and `usByteIO` define the interfaces. Classes `agency`, `dir` and `file` provide server-side implementations. Classes `usItem_proxy`, `file_proxy` and `dir_proxy` provide client-side implementations.

4 Implementation

A key component of the system is a C++ class library containing a collection of basic classes used to construct clients and servers. Figure 2. shows a small portion of the class hierarchy for this library, focusing on how the interfaces are specified and implemented. The complete system contains several additional interface definitions, but they all follow the same structure and are not discussed here to simplify the presentation. The library also contains many other classes used for naming, authentication, buffer management, etc. and covers several other aspects, such as a C++ remote invocation package on top of Mach IPC, garbage-collection of objects between clients and servers and a mechanism for dynamic type conversion, which are outside the scope of this paper [2].

The basic structure of the library is as follows: a hierarchy of abstract C++ classes defines the interfaces to the various items; this hierarchy is shadowed in the client and in the server side by similar hierarchies with the classes that implement those interfaces. For a given interface class, there is at least a pair of derived implementation classes, one with the client-side implementation and the other with the server-side implementation. There may be, and in general are, several implementation classes for a given interface class at different servers (e.g. files and pipes are different implementations of the byte-oriented I/O interface). Clients use only the interface classes. However, the objects created by the remote invocation mechanism are instances of the implementation classes at the client side, called the proxy classes. These objects may perform some local processing and/or communicate with their peers at the server side.

The consistency between the interfaces as seen by the clients and as implemented by the servers is guaranteed by the fact that both these classes inherit from the same interface class. The static type checking of C++ ensures that inconsistencies are detected at compile time. On the other hand, the separation of the implementations ensures that servers may be modified without requiring changes in the clients, as long as the interfaces between them remain unchanged. This allows different people, possibly in different organizations, to independently develop servers that can be plugged-in and work together with the rest of the system.

Implementation classes use multiple-inheritance to inherit both from the interface hierarchy and from the implementation hierarchy. Each class inherits the interface of its level and the implementation of the level above. For example, class `dir_proxy` in Figure 2. inherits from `usName`, which is the interface of its level, and from `usItem_proxy` which is the implementation of the level above, `usItem`([5] presents a similar model for separating interface and implementation in C++).

The C++ library contains most of the generic classes used to construct servers, including naming, authentication, access control, etc. Each server uses and extends the basic functionality, adapting it to its particular needs. A high degree of code sharing between the different servers is thus achieved, thereby facilitating their development and maintenance.

5 Current Status and Conclusions

The current C++ prototype contains a name server, a pipe and socket server and a BSD 4.3 emulation library written in C++. Some existing servers written in MachObjects, like the file server, the process management server and the terminal server were reused with minimal or no changes. The BSD 4.3 emulation, although not yet complete, supports a number of utilities like login, bash, gcc, vi, make, etc.

An important step to make operating systems more extensible and easy to configure is to re-architect the system in separate components that can be executed as servers and interact with other servers, possibly written by different organizations. Another step is to use object-oriented technology to specify and implement the interactions between the several components. The use of both these techniques in the Mach 3.0 multi-server results in a system that is extensible, flexible and made of replaceable parts.

References

[1] F. Herrmann, F. Armand, M.Rozier, M. Gien, V. Abrossimov, I. Boule, M. Guillemont, P. Leonard, S. Langlois and W. Neuhauser, "Chorus, a New Technology for Building UNIX Systems," *Proceedings of EUUG Autumn*, Cascais (Portugal) October 1988.

[2] P. Guedes, D. Julin, "Writing the Mach 3 Multi-Server System in C++," *Design review of the Research Institute*, Open Software Foundation, Feb 1991.

[3] D. Julin and R. Rashid, "MachObjects," *Internal document*, Mach project, Carnegie Mellon University, 1989.

[4] Keith Loepere, Ed. "Mach 3 Kernel Interface," *Open Software Foundation and Carnegie Mellon University*, 1990.

[5] Bruce Martin, "The Separation of Interface and Implementation in C++", *Proceedings of Usenix C++ Conference*, Washington, D.C., April 22-25, 1991.

[6] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi. "Mach: A foundation for open systems," *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109-113. IEEE Computer Society, September 1989.

[7] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," *Proceedings of the 6th International Conference on Distributed Computer Systems*, pages 198-204, Cambridge, MA (USA), May 1986.

[8] D. Julin, J. Chew, P. Guedes, P. Neves, P. Roy and M. Stevenson, "Generalized Emulation Services for Mach 3.0 - Overview, Experiences and Current Status"", *To appear in Proceedings of Usenix Mach Symposium*, Monterey, CA., Nov 20-22, 1991.