Generalized Emulation Services for Mach 3.0
Overview, Experiences and Current Status

Daniel P. Julin *Jonathan J. Chew* *J. Mark Stevenson*
Carnegie Mellon University

Paulo Guedes *Paul Neves*[*] *Paul Roy*
Open Software Foundation

November 6, 1991

Abstract

This paper reports on an ongoing project to develop a general understanding of the problems encountered when building emulators for various operating systems at the user-level on top of a Mach 3.0 micro-kernel, and to propose a common framework to construct emulations for a wide range of target systems and environments. It presents an overview and discussion of the major techniques and experiments that characterize the design of the proposed system. Some of the relevant aspects include the combination of several independent servers to create a complete system, generic service interfaces relying on the emulation library as an interface translator, the use of object-oriented technology to define standard interfaces and to simplify the implementation of common facilities, moving portions of the system state and processing from the servers into a smart emulation library, and a few general-purpose facilities that simplify the generation of a complete system. A number of practical observations and experiences are also presented, in the context of the development of a prototype for the emulation of UNIX 4.3 BSD.

1. Introduction

Defining and standardizing a powerful micro-kernel base is only the first step in realizing the potential of the architecture proposed with the overall Mach approach. The next step, and perhaps the one richest in design possibilities, is to learn how to construct a wide range of useful higher-level systems on top of this simple kernel. A particular class of such systems is the so-called *emulation systems*, that implement the application programming interface of an existing complete operating system or *target system* with various combinations of user-level components (servers and/or libraries) operating on top of a Mach kernel. The emulation system provides an *operating system environment* for a number of *emulated processes*, such that programs executing in these processes can operate as if they were running under a native implementation of the target system. The main benefits expected from this overall emulation approach, met at various degrees by

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035 and in part by the Open Software Foundation (OSF). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, or the U.S. government.

[*] Paul Neves is currently with the Swiss Bank Corporation / O'Connor Assoc. L.P., Chicago IL

This paper will appear in the Proceedings of the Second USENIX Mach Symposium, Monterey, November 1991.

different system architectures, include increased modularity, portability, flexibility, security and extensibility, as well as simpler development, debugging and maintenance.

With the increasing maturity of the Mach 3.0 micro-kernel, an ever-growing number of pure and applied research groups are now undertaking to design, build and study such emulation systems for a range of targets that span from relatively simple systems like MS-DOS to considerably larger systems such as VMS[8, 1, 11, 9]. All these systems have a few common characteristics: a Mach kernel, one or more servers, and an *emulation library* associated with each emulated process, that intercepts the system calls issued by that process and redirects them to the appropriate emulation services. But beyond these broad similarities, a number of different approaches are taken. Some systems use a single monolithic server that more-or-less reproduces the internal structure of the native implementation of the OS being emulated. Others attempt to decompose the functionality of the single native kernel into several independent servers. Still others follow a mixed approach, with one main central server and a number of auxiliary servers implementing specific portions of functionality that are more easily or usefully separated. Furthermore, some designs attempt to keep all emulation code strictly out of the kernel, while others define various controlled mechanisms to assist emulation by integrating some specialized modules into the protected kernel space.

In general, most efforts to develop emulation systems so far are being pursued in a mostly "ad-hoc" fashion. The precise organization of the interactions between the components of each emulation system is often largely determined by extrapolating from the architecture of the native implementation of the target operating system. Moreover, each individual component is typically designed specifically for the particular system in which it is to be used, and implemented either entirely from scratch or by adapting code from the native OS implementation. However, it appears that a number of basic design and organizational issues, and even the specification of some high-level functions, are in fact quite similar between various emulation systems for different targets. In response, this paper reports on an ongoing effort to take a broader view of the problem of OS emulation in general, and to propose a common framework to minimize the duplication of effort for the development of multiple emulation systems for the widest possible range of target systems and environments. This range may include several different "traditional" operating systems (UNIX, VMS, DOS, etc.), various specialized systems or configurations related to a single main target, or even completely new programming environments.

2. Approach and Overview

The primary focus of this work is the definition and refinement of a general-purpose architecture for operating system emulation. This design is not directly tied to the emulation of any particular operating system, nor does it attempt to imitate the structure of any particular existing operating system implementation. Instead, this work attempts to take an original look at the problems and opportunities presented by a micro-kernel architecture for emulation, and to develop a global understanding of the pertinent design trade-offs. It concentrates on identifying common issues and proposing general solutions or practical mechanisms to address them. The main challenge is to define such a general architecture so that the resulting systems remain practical and competitive when compared with traditional monolithic kernel-based architectures, particularly in the areas of simplicity of implementation, robustness and, of course, performance.

To provide a concrete environment in which to evaluate the design, we are implementing a prototype of a complete emulation system for UNIX 4.3 BSD, often referred to as the "Mach 3.0 multi-server emulation system". However, this particular prototype is not intended to be the major or single final product of this work. Rather, it is used as a vehicle with which to test general ideas about the system organization and to experiment with particular implementations of these ideas. The overall development of the design proceeds through successive refinements of the prototype, such that each consecutive version constitutes a new working emulation environment with better characteristics than the one preceding it, while allowing all design decisions to be reconsidered at each stage in light of the experiences accumulated during this process. The "final" prototype after the design and evaluation are complete might then be used as a basis for further engineering work to construct a production-quality system for BSD emulation, or for other UNIX emulations.

The overall architecture is shown on figure 1. The major elements in this architecture are the kernel, the emulated processes, and a collection of servers.

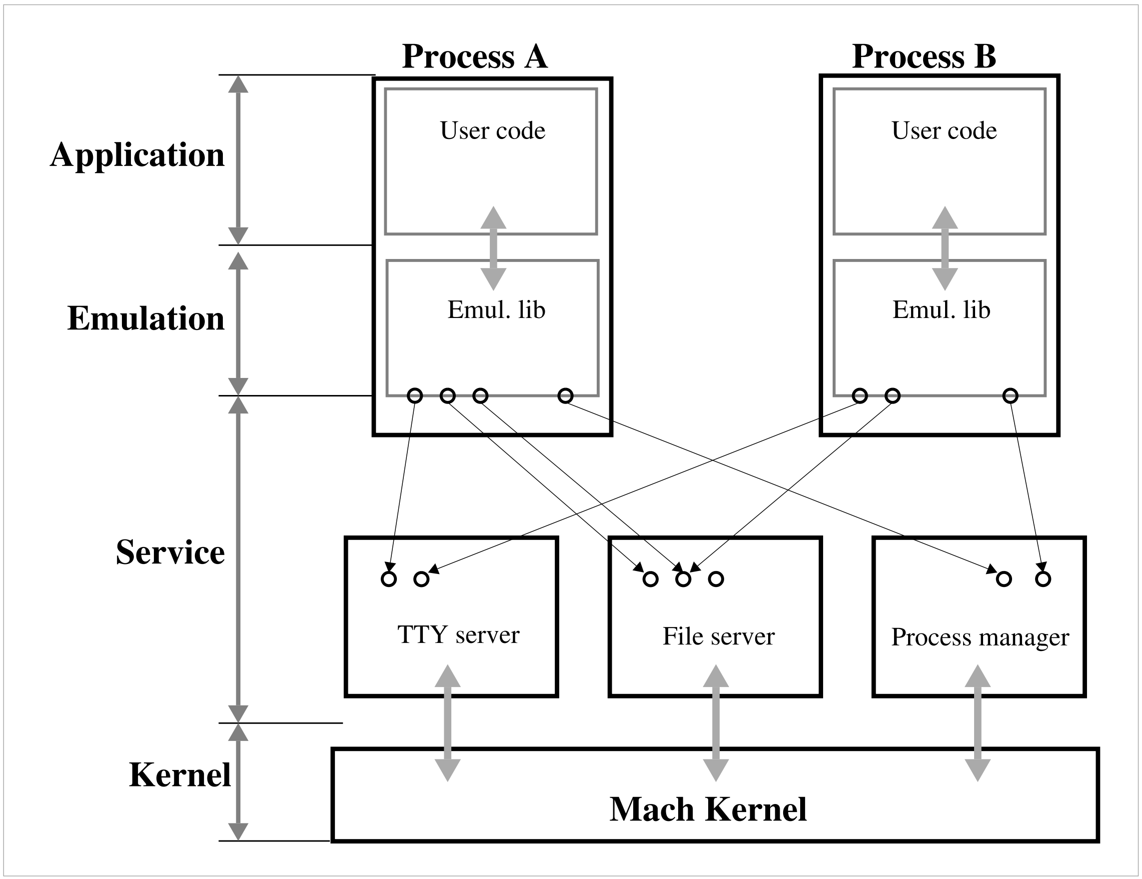

Figure 1: Overall System Architecture

The kernel is the "pure" Mach kernel, exporting only the basic Mach primitives. It is completely general, and contains no code specific to any emulation system. This decision is intended to maximize security and fault isolation.

Each emulated process is implemented in a separate Mach task that contains the unmodified user code from various application programs and an emulation library that intercepts and implements system calls issued by instructions embedded in the user code. Although both of these regions of code reside in the same address space, we often refer to the circumstances when a thread is executing user code as "executing in *user space*", and when a thread is executing code in the emulation library as "executing in *emulation space*". The emulation library is relatively large; it manages a considerable portion of the process state and the emulation functions locally.

The rest of the functionality of the emulation system, that is not handled in the emulation library itself, is distributed among a "federation" of servers each operating in a separate Mach task. These servers are independent of each other and can be replaced or re-configured at will, to provide for maximum flexibility. Examples of servers in the BSD prototype include file servers, a process manager, a TTY server, a pipe/socket server, etc.

The primary mode of communication between components in the system is the Mach IPC facility. Every individual UNIX abstraction (files, sockets, pipes, etc.) in every server is represented by a separate Mach port. However, Mach shared memory may also be used in client-server interactions when appropriate: for example, file data is typically mapped directly into the address space of each client for fast I/O access.

Finally, to maximize the potential for building different configurations or collections of servers, as well as to minimize communication overheads, the emulation libraries communicate directly with each server whenever needed without any central coordinating agent or server. As a consequence, the state of the system is effectively distributed among the various servers and emulation libraries operating in each emulated process.

The following sections elaborate on the main technical aspects of this organization and the basic ideas and experiments that we are evaluating, along with a number of our concerns and experiences with the design and the current prototype.

3. Major Design Considerations and Experiences

3.1. Generic Service Layer

A first key idea underlying the design is the definition of two separate functional layers in the emulation system architecture, corresponding to generic and target-specific functions. Such a separation is suggested by the observation that the emulation library is the *only* system component that interacts directly with the application programs operating in the emulated environment; it provides the necessary indirection allowing the decoupling of the interface provided by the collection of servers and the application programming interface. Accordingly, the first layer, or *service layer*, contains a collection of components or servers that provide basic services, defined in such a way as to be as generic as possible, so that they can be useful for the emulation of several different high-level operating systems and/or for various configurations. The second layer or *specialization layer* implements the aspects of interface or functionality that are specific to a particular emulation target, and operates as a translator between the emulated application programming interface and the generic services exported by the service layer. This specialization layer is concentrated as much as possible in the emulation library.

This approach leads to the definition of a new "system programming interface" between the emulation library and the service layer that is not directly visible to application programs. Its design can thus concentrate on issues of security, performance and most of all flexibility, thereby increasing the potential for making the service layer more generic. Issues of simplicity and ease of use of this interface by programmers, while still important, are nonetheless secondary when compared to this main consideration. Similarly, the modes of interactions between the components inside the service layer can be defined freely to maximize the flexibility, configurability and reusability of this layer, with little concern for the requirements of application programs in each specific target system.

In practice, the service layer can, of course, only be partially generic. Current observations from the design of our prototype indicate that high-level services such as file management, network access and local "pipe-style" interprocess communication, can be expected to be directly useful for a variety of operating system environments. In addition, many ancillary services that are not directly exported at the application program level, such as authentication, location of servers (coarse-grain naming), configuration management, etc., can also be reused in many different system configurations. On the other hand, process management remains largely tied to the semantics of each particular target operating system, mainly because of the complexity of the definitions of groups of emulated processes and the associated rules for access mediation. However, we have started and continue to develop a set of low-level core components that can be assembled and customized to construct different variations of process managers. Finally, terminal management ("TTY") has not been considered for generalization so far; the main difficulties in this area stem from the semantic richness of the line management strategies, together with the multiple user-controllable options (*ioctl()*).

3.2. Standard Object-Oriented System Interfaces

To capitalize on the idea of using generic services, most high-level functions are defined in terms of an object-oriented framework. Each operating system service is represented by one or more abstract *OS objects* or *items*, exporting a well-defined set of operations. Examples from the UNIX domain are files, pipes, sockets, tty's, etc., but in practice, each of those UNIX abstractions may, of course, be represented by a more neutral item, corresponding to a generic service that can be specialized by the emulation library for a given emulated environment. Each server normally implements a large number of similar, but independent items. Note that the various servers and items may themselves be implemented using object-oriented techniques; the word *item* is used to avoid confusion with the actual objects used at the implementation level.

The operations exported by each item are categorized into several independent functional groups which are standard throughout the system. Each such group of functions is represented by a specific "standard" interface. These interfaces are important both to allow a degree of target-system independence at the service level, and to allow for the easy combination and integration of many independent servers or services. The major categories of functions currently defined for our BSD prototype, that appear to be well-suited in both of these respects, are:

access mediation: access to all entities (*items*) in the system is mediated through a standard facility using general-purpose access control lists and a uniform representation for user credentials. The requirements of particular target OS'es can typically be handled with special initialization and interpretation of the standard abstractions. There is a "secure" variation of this system, targeted at the B3 level of security[11].

naming: all entities in the system can be named and accessed through a uniform name space, that also handles garbage-collection and access mediation for item creation. There is a common name resolution protocol to navigate the global name space, locate servers and locate individual items inside a server (e.g., files in a file server, pipes in a pipe server, etc.). This name space is largely independent of syntactic issues for path names, such as the interpretation of symbolic links, UNIX "..", etc. It supports user-centered naming through the use of *prefix tables*[12].

I/O: the I/O interface provides a neutral model for the identification and transfer of data. It supports both byte-level and record-level data access, as well as sequential and random access operation. Most issues of synchronization are left to the clients (emulation libraries) for maximum flexibility: when necessary, asynchronous operations are simply implemented with multiple client threads. In addition, various caching and buffer management policies can be implemented within the same I/O framework.

network control: the network control operations deal with the creation and management of transport endpoints, to the exclusion of actual I/O on those endpoints. This interface is largely inspired by XTI[6], with some changes to facilitate sharing of endpoints between multiple clients, and for integration in the uniform name space.

asynchronous notifications: this subsystem can be used by servers to deliver all kinds of notifications to clients, such as UNIX signals, VMS events, etc.

Each of these basic interfaces does not directly define the complete functionality exported by any individual item; rather, they correspond to lower-level services that must be combined to define the complete item. For example, network endpoints typically export operations from both the I/O and network control interface categories.

Many primitives of various target operating environments can simply be implemented with an equivalent primitive from a standard system interface, or with a combination of such primitives. However, it does not seem either possible or practical to define all standard interfaces to incorporate every feature of every conceivable target system. Therefore, there must be exceptions that must be handled with more specialized system primitives. The object-oriented framework provides a sound basis for the definition of such specialized functions, either through the addition of new specialized interfaces alongside the standard ones for various specialized items, or through the "derivation" of specialized interfaces from the standard ones by adding more complex or modified operations. Since the definitions of all the items are cleanly separated from each other, such extensions can be made with minimal impact on the system as whole.

It is clear that a trade-off exists between defining standard interfaces that represent the union of many different target systems, and resorting to specialization of interfaces to handle some less-common or less-generalizable features. In general, a complete system contains a mix of generic and specialized interfaces, as well as a mix of generic and specialized servers. One significant example of this trade-off is the definition of a collection of attributes for various items, that can be used to implement the UNIX `stat()` operation: the set of all possible attributes and combinations is simply too large. Accordingly, we have defined a "reasonable" set of standard attributes, and rely on additional specialized operations to manipulate all other attributes. Some other examples of OS features that seem best handled through specialization are the combination of multiple I/O channels into one (message queues in UNIX System V, reading out-of-band data inline on BSD sockets), complex protections that do not fit the simple access control list scheme (AFS, UNIX "sticky bit" on directories), special blocking options for `open()` on UNIX TTY's, etc.

Regardless of the resolution of the trade-off between generalization and specialization, it remains the case that the generic interfaces are typically more complex than the corresponding ones in any particular target OS, since they must often support a richer set of features, and tend to have many service options. As mentioned earlier, this complexity is not visible to application programs and is thus not a primary problem, but it does significantly complicate the task of providing correct and complete server implementations of each interface. We have found that a good "standard library" of reusable components or building blocks is crucial to solving this problem.

In addition, in order to maximize flexibility, complex operations are sometimes decomposed into several simpler primitives in the generic interfaces. For example, most UNIX file system operations are implemented with a combination of one or more steps to navigate through the system name space to locate a particular entity in the file system, followed by one or more additional steps directed at the item returned in the previous phase. Similarly, several steps are typically needed to completely set-up a socket (creation, establishment of service options, installation in the user-visible name space). This approach gives rise to two new considerations:

First, whenever the state of the system or item is visible between individual steps of such a sequence of operations, there is a question as to the management and protection of that state. The design of the item interfaces is such that, in most cases, this does not cause significant difficulties. The relevant state information is either naturally made visible and accessible to all (e.g., item references or locks) or it is abstracted away through the use of options and information maintained by the client and provided explicitly with each primitive (e.g., I/O modes and current offset). In one case, however, we have had to introduce a special mechanism: it is possible to create a "reserved entry" in a directory to lock a name until a complete entry can be established. This entry prevents another entry from being created with the same name, but it cannot be looked-up in the traditional fashion. In general, we have considered introducing a general-purpose locking facility to protect groups of operations, but have not found such a facility to be required so far.

Second, and probably more importantly, the use of sequences of operations to perform common tasks raises legitimate performance concerns. This question introduces another design trade-off, to define a set of special "composite operations" to optimize selected functions without unduly compromising the flexibility and reusability of the various servers. The number of these operations must be kept small, because they may in principle have to be implemented by every server participating in every emulation system. The prime example of this question is again provided by the UNIX *stat()* operation: the design defines a composite operation that combines a simple name resolving step to locate a file and a simple operation to retrieve a set of common attributes. This composite operation only covers the most common usage pattern in the system; if the resolving process has to be more complex (e.g. follow mount points or symbolic links), or if special attributes must be retrieved, the clients must fall back on the more primitive operations.

To try to provide another approach to this problem, we are also planning to investigate extensions in the

RPC system to allow "batching" or grouping of several primitive operations in a single high-level invocation without requiring changes to the simple interfaces.

3.3. Modular Services

In order to maximize the overall flexibility, and to take advantage of the potential offered by the idea of reusable services, the design aims to define as many independent servers as possible. Those servers can then be used as a collection of standardized building blocks that can be assembled in various ways to create different systems. In addition to its great flexibility, such an architecture also increases the security and robustness of the system by isolating faulty or potentially malicious components into separate, protected address spaces. Moreover, it sometimes simplifies the implementation of some servers, by allowing the use of multiple instances of one server instead of a single more complicated server (for example, one simple file server for each disk partition). However, such a desire for maximum modularity must be balanced by a number of practical considerations:

- Interactions between servers are more expensive than interactions between modules inside a particular server. The separation of services and the corresponding interfaces must be carefully defined to minimize those interactions. We have tried to limit each basic user-level operation to require the intervention of only one server. There are two main examples where this principle creates difficulties in our prototype: the interaction between the process manager and the TTY server to handle job control, and the general problem of UNIX `fork()`, where the parent emulated process, which is normally in contact with many servers, must arrange for the child process to inherit access to all those servers, along with a consistent client state. We have no solution to the job control problem. We handle the `fork()` problem by defining the system interfaces such that a single association between a client and a server (i.e. a Mach port) can be transparently shared between all the processes in a UNIX process tree, without requiring the explicit intervention of each server. Note that with this scheme, from the perspective of a server, all the client processes that share the same authentication credentials are normally indistinguishable.

  In general, the separation between servers, and between servers and clients, increases the importance of distributed shared state in the system. Much of this difficulty is solved by appropriate selection of the location of each item of information throughout the system (see "Smart Emulation Libraries" below). In addition, we are investigating the possibility of using a general-purpose facility (blackboard service) that allows more-or-less arbitrary data elements to be shared efficiently between any number of servers and clients, according to a schema that can be adapted to the needs of various specific emulations.

- The authentication logic in the system can become complicated, since each server is normally responsible for verifying the identity of its clients independently. This policy also introduces additional difficulties if different authentication schemes are to be used with different servers and if the system must handle mutual suspicion between clients and arbitrary servers. The standard access control and naming interfaces offer basic support for these requirements by providing a good separation between the functions related to normal client access and the authentication function itself: clients may be arbitrarily requested to re-authenticate themselves whenever they start referring to any specific new item, and the particular authentication mechanism to be used may be different in each case.

- Although each server typically performs a different high-level function, there are many low-level functions that must be performed similarly by all servers, such as access mediation, name space management, buffer management, etc. Modules implementing most of these functions are stored in a common library used by all servers, but the overall run-time memory usage of the system is considerably greater than that of an equivalent monolithic system. This problem could be partially solved by the use of a shared library to eliminate duplication of code segments, but it remains the case that much data space is essentially wasted.

In general, the design of the system interfaces is such that clients (emulation libraries) are unaware of which server is handling which item, so that servers can be combined or separated freely without requiring changes in the clients. The few elements of state that are logically associated with each client-server pair (mostly authentication information) appear to the clients as if they were attached to each individual item, and are transparently replicated and inherited among multiple items managed by the same server as appropriate. However, there is also a need for a global knowledge of the system, to handle operations such as orderly shutdown, startup, administration and maintenance, etc., as well as to keep track of information needed by all components, such as the current time zone, node identity, etc. We are currently building a *system configuration server* to keep track of all the servers in the system and to perform those duties. Finally, the question of implementing global system usage accounting and of enforcing global resource limits has received little attention so far; the main difficulties in this area are the need for servers to identify individual client processes (and not just groups of processes with the same identity), and of efficiently collecting them among a collection of independent servers.

In practice, our BSD prototype contains or will contain the following servers:

Servers that provide services directly visible to application programs (through emulation libraries):

- one or more file servers, handling different file systems (UFS, NFS, AFS, etc.) and/or physical devices.
- a terminal server managing all serial lines and implementing the equivalent of all UNIX tty's and pty's.
- a local IPC server or "pipenet server", responsible for all intra-node emulation-level (non-Mach) IPC: pipes, UNIX-domain sockets, and possibly System V queues and VMS mailboxes.
- a process management server or "task master", to keep track of all emulated process and process groups, and handle signal dispatching.
- one or more network servers implementing socket-like entities for a variety of network protocol families. This server is currently derived from the x-kernel from the University of Arizona[7].
- a device server handling raw access to the hardware devices, grouped in a "/dev" directory.

Servers that support the operation of the other servers:

- one or more *root name servers* tying all the other servers into a single hierarchical name space through a collection of *mount points*.
- a simple authentication server providing trusted translations between client *tokens* and explicit *credentials*.
- a blackboard server managing distributed state stored in pages of shared memory mapped in various servers and client emulation libraries.
- a configuration/admin/startup server that starts all the other servers and provides centralized access to global information.
- various ancillary servers not directly related to the emulation system, for diagnostics, network shared memory, network IPC, etc.

3.4. Client-side Processing and Smart Emulation Libraries

Independent of providing a convenient mechanism for introducing a separation between a generic and a target-specific layer in the system, the emulation library also provides an opportunity to optimize or simplify many client-server interactions by displacing some of the processing required to implement various functions from the system servers into the clients of these services themselves, and to concentrate system state in these clients. This approach is illustrated by three main strategies:

- The design of many service interfaces is such that the client itself is primarily responsible for managing important pieces of information (UNIX file descriptor table, signal mask and handlers, current working directory, etc.) and performing complex functions (pathname resolution, `exec()` logic, etc.).
- Whenever a server grants access for a given item to a given client, a special code fragment or *proxy object*[10] is installed by the run-time system in the address space of that client to act as a local representative for the associated item. This proxy, which is defined and supplied by the server, provides a convenient level of indirection between clients and servers. Simple proxies just forward all client requests directly to the server via Mach IPC, but more complex ones can implement optimizations such as caching item information, or managing a window of shared memory containing file data or other I/O or control information. Proxies are currently statically linked with each emulation library, but we expect that they will eventually be dynamically loaded.
- The emulation library can itself operate as an active element, to simplify and relieve some of the burden on servers. For example, the UNIX emulation library currently contains an active thread to handle incoming asynchronous notifications and transform them into the appropriate UNIX signals. We also plan to use active emulation library threads to implement most forms of asynchronous I/O.

It is clear that there are limitations to the use of client-side processing. There are many system functions that require synchronization and sharing of information or resources between several clients. Although there are mechanisms to handle a number of these problems with minimal server overhead, the need for external agents or servers cannot be completely eliminated. More importantly, the decision to place more responsibility for various system functions in an emulation library that is not protected from incorrect or malicious user programs has obvious implications in the areas of robustness and security. Clearly, the more code and information that is stored in the emulation library, the greater the risk is of accidental or intentional modification, which can lead to very complex failure modes. One interesting example of this problem is presented by the `copyin` and `copyout` routines, that are used in traditional UNIX systems to transfer system call arguments between user space and the protected system space. In our emulated system, these routines may verify the validity of the user buffers when they are invoked and even copy the associated data to separate emulation-space buffers, but there is no way to effectively protect this data from modifications at any time while it is being used inside the emulation library.

In accordance with the goal of supporting very secure system implementations, our design policy is to avoid any optimization that can allow a malicious emulated client to gain unauthorized access to protected resources, or to affect the integrity of another emulated client in ways that would not be possible simply through the use of the "published" application programming interface being emulated. On the other hand, we believe that reasonable compromises are acceptable in trading-off performance against the robustness of individual clients. An incorrect user program may be allowed to corrupt the data structures of its own emulation library. As a result, it may even modify the external behavior of the whole emulated process, but only in ways that could also be achieved with a different (correct) user program. The key element that permits the implementation of such a policy resides in the use of port capabilities for all individual items manipulated by the client: these can be destroyed or even swapped, but never actually forged. However, this approach requires very careful design, and we know of no formal method to guarantee its correct application.

We have also considered a scheme in which a substantial portion of the emulation library is kept separate from the emulated process itself in a different protected Mach task, so that each emulated process is represented by two Mach tasks. However, the communication bandwidth between an emulated process and such a separate library is per necessity lower than that of the fully-shared case, since at least some data must cross address space boundaries, and since control must be transferred between threads. In addition, the cost of creating a new emulated process is typically significantly higher, since two Mach tasks must be created instead of one. Consequently, the decision to use a protected or unprotected emulation library must be made by balancing the added performance of one against the added robustness of the other. We feel that in a system such as UNIX, where `fork()` operations are frequent, the unprotected approach is probably preferable. On the other hand, in a system such as VMS, where process creation is rare, the trade-off may lean the other way. Note that this particular trade-off does not detract from the potential for reuse of server components and generic services for the emulation of different target systems, since emulation libraries are necessarily specific to each target operating system.

Two other schemes have also been proposed with respect to the location and protection of the emulation library: placing it in kernel space within each emulated task, and displacing all of its function into a single centralized server independent of the emulated processes themselves. Both of these schemes are obviously feasible, but they contradict our basic design decisions to avoid including specialized emulation code in kernel space and to avoid using a centralized coordinating agent, respectively. Consequently, they have not been considered further within the scope of this particular investigation

Independent of these high-level considerations, the increased complexity of smart emulation libraries also creates very significant difficulties at the technical level: increased size, expensive management of client state and complexity due to multi-threading.

The size of a smart emulation library tends to be considerably larger than that of a simple library that blindly forwards all system calls to one or several servers. This results in an increased load on the virtual memory system, both in the form of normal paging activity and in the form of copy faults when creating new emulated processes (UNIX `fork()`).

Smart emulation libraries also tend to contain significant amounts of state that must be inherited or re-created during process creation. Simple memory can be naturally inherited through the services of the Mach kernel, but other elements of state cannot, such as active threads and port capabilities. This introduces potentially significant overheads. In particular, it is clear that a trade-off exists with respect to inheritance of information simply cached in the emulation library, versus simply re-creating this cached information when and if needed. This trade-off has not yet been satisfactorily explored.

Finally, smart emulation libraries naturally need to be multi-threaded, both to support multiple user threads issuing system calls, and to support active threads internal to the emulation library. We have found that the existing implementations of the Mach cthreads library are poorly adapted to operate inside an emulation library. We have had to introduce modifications to handle interactions with threads in the emulated process itself, for correct transfer of thread state at `fork()` time (clean-up stacks and cthread data structures), and to implement an interrupt handling facility for signal delivery. Even with these extensions, multi-threaded emulation libraries are still difficult to write. Furthermore, it does not appear that newer implementations of user-level threads libraries will reduce the magnitude of this problem. In particular, the current trend towards mixing coroutines and kernel-level threads clearly complicates matters, since the emulation system must often still deal with "raw" kernel threads for Mach exceptions and for interrupts.

3.5. Object-oriented Service Library

In order to avoid unnecessary duplication of code, the system design includes a common library implementing many functions that must be provided in several different servers, or in several specialized versions of the same service. Such a library, through its modularity, greatly contributes to the extensibility and ease of modification of the system and also reduces the overall complexity of the entire system implementation. Our prototype contains a growing collection of reusable code fragments or objects; the main areas currently covered are:

- access mediation for arbitrary items (access control lists, user credentials, authentication, etc.).
- simple building blocks to construct a name space collecting all the items managed by one server, and to handle item creation and deallocation when appropriate.
- mapped-file access, including pagers and appropriate proxy objects.
- sequential streams of data following the standard I/O interface (connections, buffering, data transfer across address spaces, etc.).

- simple pathname resolution from the client side, including a user-controlled prefix table and caching of mount points and symbolic links.
- a layer of code to export the standard I/O and naming interfaces from a server based internally on *vnodes*[5].

Considerable effort has been put in the design and implementation of this common library, and we have been served very well by this approach in practice. Some servers (root name server, pipenet server) are constructed almost exclusively from modules in this library. The only non-common code in these servers is a small *main* procedure for startup and a few lines of code to specialize the operation of various modules in the library and supply some operating parameters. This specialization is normally realized through derivation in the class hierarchy defined by the library. Other servers (UFS, NFS, TTY, network) contain a large body of "internal" specialized code, often inherited from other sources. They have been integrated in the system in a matter of hours simply by linking them with an "upper-layer" of code from the common library.

Several of these modules were first developed in earlier versions of our prototype using a straight "C" coding style, but it became quickly evident that this style was poorly adapted to writing the kind of polymorphic, reusable code desired for such a library. The library has since been converted to use object-oriented technology (first using MachObjects[4], then C++), augmented by a powerful remote procedure call package closely integrated with the object-oriented language itself (see below). We have found that this switch in implementation style has considerably simplified the task of writing new modules and integrating them with the rest of the system. Several observations can be made with respect to the use of object-oriented technology in this context[3]

First, multiple-inheritance significantly simplifies the design of the library. We use it in several instances for the combination of independent interfaces for the definition of items, as suggested above. We also use it routinely for the combination of independent pieces of functionality for the implementation of items, for example to add a protection module or a buffer management module to an item representing a file or socket. In addition, in the C++ implementation of the library, we use multiple inheritance to allow classes implementing various functions in the system to inherit both from base classes in a normal "implementation" hierarchy, and from classes in an abstract hierarchy representing the standard system interfaces (used for defining the same interfaces on the client- and server-side).

Second, we need some form of runtime type checking in the language. Although static type checking clearly offers important benefits in terms of program reliability and maintenance, it cannot be used in all situations, specifically for the implementation of method invocations across a client-server boundary. When first resolving a name to obtain a reference to an item in our standard naming subsystem, as with the UNIX `open()` primitive, only the name of that item is known. The actual type and the exact collection of operations exported by that item can only be determined at run-time. This information must then be explicitly obtained by the client upon acquiring or using a proxy for that item. The total number of defined interfaces makes it very impractical to define a single exported type that exports the union of all possible operations. Further, if we allow for maximum flexibility and the definition and integration of new servers and interfaces over the life the system, even the complete set of possible operations may not be known in advance. With the MachObjects system, this problem is moot since all method invocations are checked at run-time. In the C++ system, we use static type checking wherever possible, coupled with a package similar to the NIH class library[2] to provide the required extended functionality for run-time type checking.

Lastly, it is very useful to be able to manipulate classes and methods as first-class abstractions in the language. The proxy mechanism, which is based on the run-time specification of a class to instantiate in a client, obviously creates a need to refer to classes explicitly and directly. In addition, the RPC package integrated with the object-oriented programming environment manipulates method references directly at run-time to transparently forward some invocations without the need to generate explicit stubs, thereby greatly enhancing the flexibility of this system and speeding-up the design and prototyping process. Method references are also manipulated by the access mediation subsystem, which uses a simple table lookup to determine if a given method invocation is to be allowed for a given caller and object combination. As is the case for run-time type checking, these features are handled trivially in MachObjects, and with the help of an extension package in C++.

3.6. Hiding Complexity in the Tools

In addition to the common library mentioned above, the design also attempts to reduce the complexity directly visible to implementors by aggressively hiding functionality in a collection of powerful run-time and library tools. These tools can be used largely as opaque “black boxes”; they create another degree of fine-grain layering in the overall system design. The design of many such complicated tools could easily constitute a research project in itself, considerably beyond the scope of our current goals. Accordingly, our focus is not on developing ideal general-purpose solutions in this area, but on solving the specific problems and issues raised in our design, and only as we encounter them. Our success with this approach so far encourages us to continue in this direction.

A first example of this approach is the use of the extended cthreads library described above, that takes care of many of the difficulties inherent in the implementation of emulation libraries. The main complexity introduced at this level is a facility used to interrupt pending operations in the emulation library. This facility must be invoked whenever the reception of a UNIX signal must cause the execution of a system call to be prematurely aborted. It includes a relatively general exception handling mechanism, and it interacts with the RPC system to propagate such interruptions to various servers as appropriate. Another example is the set of extensions to the object-oriented run-time system described in the previous section.

More importantly, the last and probably most far-reaching example is a sophisticated remote procedure call package integrated with the object-oriented language used for the implementation of the prototype. This RPC package is the key to many of the features mentioned in the rest of this overview. It transparently converts local method invocations into RPC’s as appropriate, and makes it very easy to define new RPC’s. It handles interruptions of arbitrary RPC’s through a special message protocol that is implemented in all servers. It automatically takes care of transferring item references across address spaces and of instantiating the correct proxy objects. Finally, it completely hides all details of the management of ports and Mach IPC semantics from its users, including taking care of automatic garbage-collection of item references using the “no-more-senders” facility of the Mach IPC system.

Not surprisingly, this RPC package turns out to be very complex, and to be a significant factor for the performance of the prototype. However, we feel that most of this complexity is essentially unavoidable, and would simply have to be distributed through other parts of the system if it was not concentrated in the RPC module. Therefore, we plan to continue to extend the RPC package in directions that we think will improve the performance and flexibility of the overall system. Some proposed experiments include automatic initialization of complex proxies, the use of polymorphic abstract data types as arguments for certain calls, and batching of operations to reduce the total number of messages used in the system.

4. Status and Evaluation

The main observation to be made at this stage is that the overall architecture appears to be sound and effective. The current system prototype for BSD does work with completely modular services and has already reached a relatively high level of functionality and practical utility. The flexible technology for interface definition is adequate and we expect that it will continue to be successful as we refine that prototype. The framework for integrating servers into the system, centered on the naming and access mediation components, has proven to be extremely useful. Finally, the use of modular servers has been and continues to be invaluable in helping with the incremental construction and debugging of the system: we routinely restart or replace individual servers while the whole system is operating, and debug them “on-the-spot” using the Mach task and thread control facilities.

The BSD prototype currently implements all the essential functionality for `fork()` and `exec()`, general file access services, signal and process management (including interrupted system calls), TTY management, pipes, and UNIX-domain and IP/UDP sockets. In practice, the system can be started or "booted" on top of the *POE* low-level emulator, run an emulated login process, start a shell and run various editors and common UNIX commands, up to and including compilation of programs. Still missing are TCP sockets, raw device access, and a number of more specialized or less frequently used features in the above subsystems, for example asynchronous I/O, set-uid `exec()`, sharing of file descriptors between processes, resource control, etc. We continue to add new features in these areas and to integrate new facilities. The major implementation activity at the present time is a conversion to use C++ as the standard implementation language.

We have not made any significant efforts to improve the robustness and performance of the prototype until its level of functionality was sufficient to make such efforts meaningful. We feel that this point has now been reached, and detailed evaluation and tuning should begin in earnest as soon as the conversion to C++ is complete. A first short-term objective is to make the prototype self-hosting.

The degree to which various elements of this system are reusable is still a matter of discussion, but early observations are encouraging. We feel that the design of many servers and high-level components, coupled with the generic service interfaces, has very good potential for flexibility and reusability under various operating conditions. In addition, the smaller-level reusability of the pieces of the common service library has already been largely demonstrated; we believe that the current prototype could not have been built without it. A desirable further step in this direction should be the construction of a similar library for the implementation of emulation libraries.

On the negative side, the size and complexity of the current prototype are clearly causes for concern. The memory usage of the multiple servers is large, and the load that they impose on the machine is important. Re-compilation of the prototype is slow, due to the volume of code involved. Several of our low-level mechanisms and sub-systems are very large and seem overly complicated, for example the facility to handle interrupted system calls and all the logic around UNIX `fork()`. Hopefully, the magnitude of these problems will decrease as the design and its implementation become more mature, but it is not clear if they will ever be completely resolved.

Finally, we cannot yet draw conclusions on the experiment with the use of "smart" emulation libraries in general or the degree to which this general approach can and should be applied. There are practical examples of other systems where this approach has been useful[1], but none that goes as far as the present design. The difficulties in terms of complexity, robustness and increased *fork()* overhead cannot be ignored. We will reserve final judgment on this experiment until more experience has been gained with implementing and using systems that rely on this aspect of the architecture.

5. Conclusion

We are conducting experiments and proposing a number of techniques with the goals of helping the development of emulation systems, and of minimizing the duplication of effort for the implementation of several independent emulations. There is no question that achieving these goals is highly desirable; the present effort, although still incomplete, shows that there are many opportunities to make significant progress in this direction. Furthermore, several of the proposed general-purpose components or mechanisms constitute significant practical steps toward improving the current state of the technology, and the results from our BSD prototype encourage us to continue this effort. We expect that the experiences from this investigation, combined with the lessons from other efforts in the same area, will eventually lead to the development of a complete practical framework for the implementation of many emulators, and maybe to the emergence of a new software industry for the production of replaceable, modular components for Mach-based systems.

Acknowledgments

Many people at CMU and at the Research Institute of OSF have participated in various stages of the design of the system and the implementation of successive versions of the BSD prototype. Beside the authors, they are: Robert Baron, Alessandro Forin, Jeffrey Heller, Michael Jones, Keith Loepere, Douglas Orr, Richard Rashid, Franklin Reynolds and Richard Sanzi. In addition, the whole Trusted-Mach team at Trusted Information Systems has often contributed many valuable insights.

References

[1] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the 1990 Summer Usenix*. Usenix, June 1990.

[2] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1990.

[3] Paulo Guedes and Daniel P. Julin. Object-oriented interfaces in the Mach 3.0 multi-server system. To appear in the proceedings of the Second International Workshop on Object-Orientation in Operating Systems, Palo Alto, 1991.

[4] Daniel P. Julin and Richard F. Rashid. MachObjects. Internal document, Mach Project, Carnegie Mellon University, 1989.

[5] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 summer Usenix*, pages 238–247. Usenix, 1986.

[6] Open Software Foundation. *OSF/1 Network Programmer's Guide*, 1990.

[7] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. The *x*-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[8] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, and Richard Sanzi. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113. IEEE Computer Society, September 1989.

[9] Richard Rashid, Gerald Malan, David Golub, and Robert Baron. DOS as a Mach 3.0 application. To appear in the proceedings of the Second USENIX Mach Symposium, November 1991.

[10] Marc Shapiro. Structure and encapsulation in distributed computing systems: the Proxy principle. In *The 6th International Conference on Distributed Computing Systems*, Boston (USA), May 1986.

[11] Trusted Information Systems, Inc. Trusted Mach system architecture. Internal Report, April 1990.

[12] B. Welch and J. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 184–189. IEEE, May 1986.