The Mach 3.0 Multi-Server System Overview

Daniel P. Julin

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

dpj@cs.cmu.edu

DRAFT of July 16, 1991

This is a working document. The information presented herein is subject to change without notice.

1. Introduction

The Mach 3.0 multi-server emulation system is a research effort conducted in collaboration between CMU and the Research Institute of OSF. The main goal is to study the issues and technologies involved in the development of collections of software components or servers that cooperate to emulate the high-level semantics and functionality of various modern operating systems, on top of a Mach 3.0 micro-kernel base. The current effort concentrates on the realization of a prototype for the emulation of UNIX 4.3 BSD.

This paper is divided in two major sections. The first section presents a brief discussion of the strategy and major design considerations for the development of this prototype, followed by an overview of the key elements and ideas underlying this design. The second section provides technical descriptions for most of the major areas of the design and implementation.

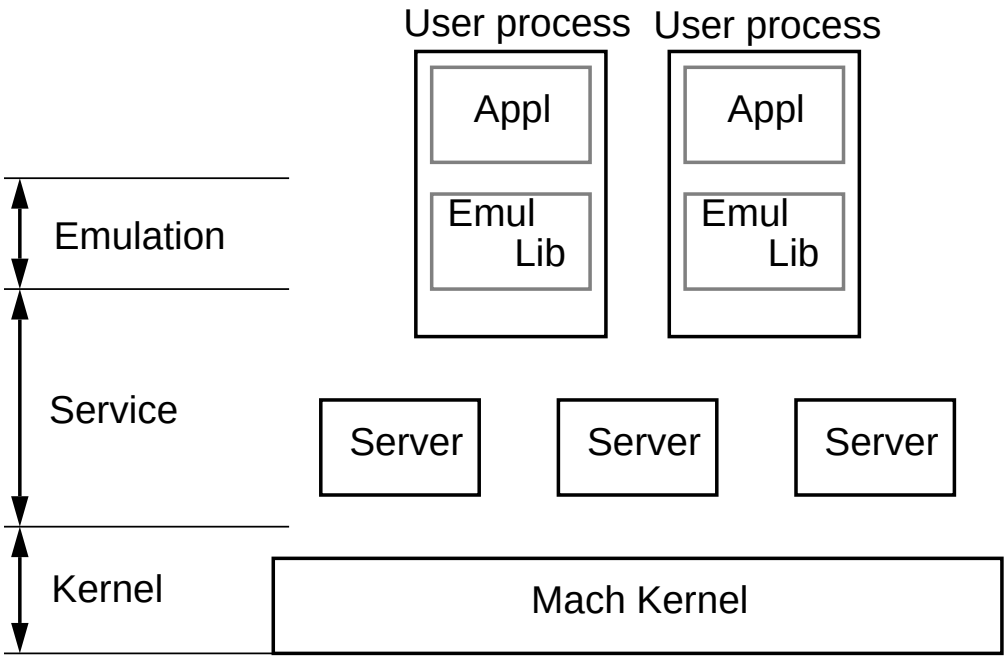

Figure 1: General System Structure

2. Design Strategy

2.1. System Structure

The main goal of the system is to provide an environment for the execution of a number of *emulated processes*. Each such process contains a program originally written for another operating system, and must be made to operate in this emulated environment as if it was running on its native system. The operating system whose features are thus being emulated is often hereafter referred to as the *target operating system*. In the context of an emulation system, the execution environment defined by the Application Programmer Interface (API) of the target operating system is also often referred to as the *operating system environment*, to distinguish it from the API and facilities found in a "native", non-emulated implementation of the target operating system.

Figure 1 shows the general organization of a system performing such emulation. Each emulated process is implemented by a Mach *task*. Beside the program to be emulated, each of those tasks contains an instance of the *emulation library*, which intercepts the system calls invoked by that program. The bulk of the work for the emulation is performed by a combination of individual servers responsible for functions such as file service, network access, process management, etc. These services are normally exported to the client emulation libraries through special libraries or proxies linked with the clients, which facilitate and optimize client-server interactions. Finally, the *pure Mach kernel* provides the basic facilities for the execution of those various components, as well as raw device access. This organization can be viewed as a combination of three software layers that can be designed and maintained independently, corresponding the the kernel,

the servers and their proxies, and the emulation libraries.

2.2. Major Design Considerations

The organization above offers the potential to achieve a number of high-level goals, while at the same time creating concerns in other areas. Consideration for these questions guides the whole design and defines the various tradeoffs to be resolved:

Modularity The first major aspect of the structure described above is the separation of the basic services into many independent, modular components. Such modularity is obviously crucial in the area of software development and maintenance, but it also offers the potential for generating systems that are customized for different user-level applications and different operating environments. System engineers can build many system configurations by selecting and assembling components from a collection of general-purpose and specialized servers supporting a wide variety of requirements. Ideally, this approach could also give rise to a new industry for the production of "system components", with the potential to greatly enhance the overall quality and flexibility of future systems.

Extensibility A second major consideration is the ability to extend or alter the design and implementation of an existing system in ways not directly anticipated by the original designers. Examples of such modifications include the addition of new high-level services and functions corresponding to new developments in technology or system programming concepts (specialized file servers, databases, distributed facilities, multi-threading, etc.), and changes to the application programming interface exported by the emulation system to represent different "flavors" of interfaces or to accommodate evolving standards. The keys to realizing this potential are the separation of functions between servers, careful design of server interfaces to facilitate the integration of new services, and careful design of individual components to simplify modifications.

Support for Multiple Emulations Combining the modularity and extensibility aspects mentioned above, it is also possible to reduce the design and implementation efforts necessary to build several emulation systems for different target operating systems. A number of high-level functions to be provided in many operating systems differ only in relatively small details of interface (network access, IPC, file service, etc.). For these cases, it is desirable to design servers than can be easily reused or specialized for each target system. Moreover, in cases where it is impractical to use a whole common server, the design may substantially simplify the task of constructing different servers or other components by identifying common sub-systems and lower-level modules (access control, name resolution, data transfer, etc.) and providing reusable implementations for them.

Beside the immediate question of simplifying the implementation of different emulation systems to be used independently, the ideas presented here also raise two additional questions. First, it may be desirable to support multiple emulations executing simultaneously on the same hardware, either with disjoint sets

of servers or with some sharing of hardware and system facilities. Second, assuming that more than one emulation system is operating on a given node or distributed network, it may be desirable to provide for interoperability between various user programs executing in these different operating system environments, in the form of sharing of software resources visible to application programmers (files, I/O channels, etc.). Although these questions certainly merit further investigation, they are not directly addressed by the current prototype.

Security It is expected that the emulation architecture under consideration can be used to produce systems with very high security levels, potentially higher than those achievable with traditional system implementations. Since each server operates in a separate, protected address space with secure communications channels, the task of verifying the correctness and integrity of each sub-system is greatly simplified compared to the equivalent task in a monolithic system. Moreover, a careful definition of the separation of functions between different servers can in many cases eliminate the need for certification of some servers altogether. However, this architecture also introduces new difficulties. Unlike the situation found in traditional systems, there is no system component that constitutes a natural centralized repository for security-related information and policies. In addition, the very modularity of this organization gives rise to potentially complex interactions between components that can themselves complicate the security analysis. Both of these problems need to be addressed early in the design if this design is to be successful in the area of security.

Practical Considerations In addition to the high-level goals listed above, a practical system should also try to meet a few more mundane requirements:

Performance : The overall performance of the system is very critical to its success and acceptance. Problems in this area stem from the overhead of communications between clients and servers, and from the need of various servers to sometimes communicate with each other to maintain a shared, distributed view of the system state. They must be addressed through careful analysis of all interactions between components, and through the design of appropriate interfaces and communications techniques. On the other hand, since the architecture of the emulation system is quite different from that of more traditional systems, performance in various areas is not necessarily bounded by the performance of existing architectures.

Importing Existing Code: In a large software project such as the one described here, it is extremely desirable to be able to import components or servers built for other projects. For example, it is often expedient to take advantage of code for various file systems or networking implementations, that was originally written for a specific (non-emulated) operating system.

Reasonable Licensing: In view of the desire to import foreign code, the success of a new system is likely to be influenced by the licensing constraints imposed by the various software elements that compose it. Although licensing restrictions per se are not necessarily avoidable or even undesirable (see the suggestion for a "components industry" above), the emulation system must try to avoid relying on components for which those restrictions are too strong, or provide for the easy replacement of such components if desired.

2.3. Major Design Directions

Within the guidelines defined by these high-level considerations, the design is based on a small number of key ideas representing the fundamental approach and technologies available to the implementers:

Systematic re-design First of all, the design under consideration constitutes a new approach of the problem of implementing high-level system services, and not an evolution of existing kernel-based implementations. No effort is made to imitate any internal details or data structures from other systems; instead, the design concentrates on identifying and applying the techniques best suited to the proposed micro-kernel, multi-server architecture.

Generic Services A second key idea is the observation that the emulation library is the only system component that interacts directly with the application programs operating in the emulated environment. Therefore, the programming interface of the target operating system to be emulated need only be exported at the boundary between these application programs and their emulation library and not directly by the collection of servers. In the proposed system design, many of the servers are largely "generic", providing a set of services that are useful to a wide class of high-level operating system interfaces and/or configurations. The aspects of interface or functionality that are specific to a particular emulation target are as much as possible concentrated in the emulation library, which then acts as a sort of "translator" or "presentation layer" above the generic service layer. Note that this approach also allows the design of the server interfaces to concentrate on issues of flexibility, security and performance, with questions of simplicity and ease of use by application programmers becoming secondary.

Modular Services In accordance with the objectives established above, and following on the idea of keeping the service layer as generic as possible, the design also attempts to make this layer highly modular. The whole system is treated not as a single rigid construction but as a library of replaceable tools, with a flexible framework for assembling them. The design defines as many independent services in the second layer as possible, each of which to be implemented typically with a separate server. This decomposition is limited by two main considerations. First, when dealing with multiple target emulations, it is important that the set of services so defined not simply be the concatenation of all the specific services for each target environment; commonalities must be properly recognized, analyzed and exploited. Second, interactions between servers are typically 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.

Common Mechanisms and Facilities In a modular system such as the one proposed here, there are many examples of functions and problems that must be addressed in several different servers, or in several specialized versions of the same service: data transfer (I/O), access mediation, management of shared buffers, naming, synchronization, etc. To maximize the potential for extensibility and ease of modification, as well

as to reduce the overall complexity and avoid code duplication, the design concentrates on identifying these common issues and defining "standard" solutions or mechanisms that can be used throughout the system. Examples of this approach range from the creation of a library of code fragments that can be used in many servers, to the definition of general principles about how a whole class of specific issues should be addressed.

Standard, Object-Oriented System Interfaces As an important application of the idea of using common facilities, most high-level functions are defined in terms of an object-oriented approach, with a collection of standard service interfaces. 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, ttys, etc., but in practice, each of those UNIX abstractions may be represented by a more neutral item, corresponding to a generic service that can be specialized for a given emulation 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 grouped in several independent interfaces such as I/O, naming, access control, etc. Several of these interfaces are generic, while others are specific to particular sub-classes of items. They provide a common base for the system specification, that can be extended or specialized as needed to support additions in high-level functionality, or deal with the specific requirements of various target emulations. Note that these basic interfaces do not map directly into the various high-level services; rather, they correspond to low-level services that must be combined to define the complete item functionality. In particular, special attention is given to avoid imposing any particular subdivision of services between servers, or any specific server implementation.

Client-side Processing 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 many client-server interactions by displacing some of the processing required to service various functions from the system servers into the clients of these services themselves. Several system services are defined so that the emulation library itself is responsible for keeping various elements of client state and performing many functions locally: UNIX signal handling, UNIX file descriptor table, pathname resolution, current working directory, `exec()` logic, etc. The various components in the "service" layer are then considered as a second level of service, invoked only when the emulation library code cannot perform all the necessary functions. In addition, many of the interactions between the emulation library and the service layer are themselves optimized, through the use of client-side caching and buffers of memory shared between clients and servers. A special mechanism (*proxy objects*, see below) facilitates this type of optimizations.

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. In accordance with the goal of supporting very secure system implementations, the 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, reasonable compromises for optimization are acceptable with respect to 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.

Language and Run-time Support Finally, since the present design is essentially new and independent of previous efforts, it is in a position to take full advantage of some modern software facilities not always used to the same extent in more traditional designs. First, many of the common mechanisms and facilities mentioned above, as well as some complete servers, are implemented using object-oriented techniques and languages. The system contains a relatively large library of standard classes that can be used in the implementation of server and emulation libraries. Second, the design itself dictates various extensions or special features that are then incorporated in the implementation language and run-time system. Such extensions include support for controlled run-time binding, garbage collection, interrupt mechanisms, and a sophisticated remote procedure call RPC package integrated with the implementation language that supports complex object data types and limited loading of "server" code into the client's address space.

3. Technical Overview

3.1. Major Components

As indicated above, the system is organized around a collection of independent servers each executing in a separate Mach task and exporting their services through a combination of IPC and shared memory. The various system servers currently implemented or under consideration are:

servers that provide services directly visible to application programs:

- one or more file servers supporting random-access collections of bytes stored in various formats on local disks or remote file servers
- a terminal server ("TTY") for the management of logical and physical terminal devices
- a local IPC server supporting basic communication between application programs (pipes, queues, UNIX-domain sockets,...)
- a process management server or "task master" to keep track of emulated processes and allow them to be operated on by external agents

- one or more network servers providing access to the network and implementing various protocol families
- a device server to control user access to the physical devices controlled by the micro-kernel

servers that support the operation of the other servers:

- one or more *root name servers*, responsible for integrating the various servers together into a single central name space from which they can be located by clients
- an authentication server, acting as a secure repository for information on the identity of users
- a blackboard server to manage any information that must be efficiently shared between several emulated processes and/or servers
- a lock/semaphore server to handle synchronization functions between clients and/or servers
- a configuration/admin/startup server to handle the startup of all the other servers and of the system as a whole, and to keep track of the system configuration (which servers to start, which devices to use, etc.)
- a remote Mach IPC server ("netmsgserver") responsible for forwarding the Mach IPC facility over the network
- a network shared memory server, providing uniform shared memory over a collection of nodes connected by a network
- a diagnostics server responsible for logging all debugging, warning and error messages produced during the operation of the system

The relatively high-level services exported by those servers are in turn defined in terms of a collection of more basic facilities and/or standard interfaces exported by various objects or *items*. In many cases, the system prototype also includes a collection of standard classes or code fragments that can be included in the implementation of various components. The main examples of such facilities and interfaces are:

- a client-server invocation facility, specifying how item operations can be invoked by clients in general
- an access control facility, to determine which operations may or may not be performed on a given item, and to keep track of the identity of the invokers
- a naming interface, to locate and create all items in a single uniform name space
- a simple I/O interface, to transfer data between an item and a client
- a notification facility, allowing client processes to receive asynchronous notifications of various events (signals, AST's, ...)
- a shared state management facility, to handle information shared between multiple components (organized around the blackboard server) and to keep track of information cached inside various emulation libraries

The following sections present more details about the design and current status of some of those basic system facilities.

3.2. Client-Server Communication and Access Mediation

Each item is implemented with a set of software objects as defined by an object-oriented programming facility such as the MachObjects package[**?**] or an extended C++ package[**?**]. This set of objects is itself subdivided into a server-side group residing on the server responsible for the management of the corresponding item, and a collection of objects operating in the address spaces of the various clients.

Servers export access to the items that they manage through a special mechanism called a *proxy object* or simply *proxy*[**?**]. A proxy is a body of code loaded in a client's address space, which acts as a representative with that client for a given item from a given server. Each proxy contains at least a *send* right for a Mach port, for which the *receive* right is held in the corresponding server. All operations on the item are invoked by the client as local operations on the proxy instead of being directed at the server. In the simple cases, the proxy simply forwards or *delegates* all client invocations to the server via Mach IPC, but in other cases, the proxy may perform most or all of the processing locally, thereby reducing the communications overhead and the load on the server. One typical application is to use a proxy to cache information on the client side of a client-server interface; another is to use a proxy to transparently manage a region of memory shared between the client and the server (e.g an open file in a file server), thereby avoiding much communication overhead. A new proxy object is instantiated in a client's address space as part of the protocol through which that client gains access to the corresponding item, with support from the RPC run-time system. Note that the set of all the proxies for the items currently accessed by a given client can be viewed as an additional layer of software within the macroscopic service (or second) layer. Although they reside in the client's address space, those proxies are logically part of the implementations of the servers for the items which they represent. Their nature, as well as their implementation or object class, are specified by the designers of each individual server, and not by the clients.

On the server side, the group of objects implementing a single item is constituted by a set of *agent* objects, all pointing to a single *agency* object. The agency object is specific to the type of item concerned. It is responsible for maintaining the state associated with the item, and it implements all its exported operations, without concern for access control. The agents are normally independent of the type of the item. They are responsible for all access mediation for operations performed on the item, which they perform by acting as a filter between the clients and the agency. Each agent contains a record of user credentials and a *receive* right for a Mach port, both of which are established when the agent is created and can never change afterwards. The various proxies that hold the corresponding *send* rights use this port to send their requests for service to the agent. They are indistiguishable from each other, and are treated as a set of clients associated with the same user credentials. Since Mach port rights are secure capabilities, each proxy can only contact the server by sending requests to its associated agent and never directly to the agency or another agent. The agent can thus use the stored credentials to perform all the appropriate access checks before forwarding each individual request to the agency, and can provide the agency with a reference to the client's credentials if needed. In this scheme, the agents can be considered as trusted representatives of the clients with the agency; since

they are stored in the server's address space, they cannot be tampered with by the clients. Note that with this system organization, the service interface for a given item is really the set of operations exported by the proxy objects, but the interface that needs to be examined to verify the security of the system is the set of interactions taking place between proxy objects and servers.

Finally, since communication between proxy objects and agents uses Mach IPC, the Mach *no-more-senders* notification mechanism is used to detect the disappearance of all clients, and provides a natural basis for a garbage-collection facility, allowing agencies to take special action when all their agents (and clients) have been deallocated.

3.3. Authentication and Access Control

3.3.1. Protection Model

In accordance with the principle of using generic interfaces, the access control interface is based on a single protection model, of which the models of the various target environments should be either subsets or reasonable variations. This model is organized around three basic concepts:

- each operation exported by an agency is associated with a set of *access rights*. Those rights are normally established and stored in the agent when this agent is created. The appropriate rights must be present in the agent if the corresponding operation is to be allowed to the client.
- each client is associated with a list of *credentials* representing its identity, associated with the corresponding agent. These credentials are a simple ordered list of 32-bit *authentication identifiers*, which may be obtained from the authentication server upon presentation of an unforgeable *authentication token*. The token itself is created by the authentication server, from a user's password.
- each agency logically contains an *access control list* or *ACL*. This list is an ordered list of entries, each containing an authentication identifier and a set of access rights. The agency exports an operation to match a list of credentials against the ACL, and determine the set of access rights thus authorized for that client. In addition, each agency specifies a *privileged identifier*. Presence of this identifier in a list of credentials causes all access checks to be bypassed, and all access rights to be authorized.

3.3.2. Access Control Interface

Given this model, clients acquire access to an item by obtaining access to an agent via a proxy object. The agent contains the credentials of the client, and a set of enabled access rights. Those access rights are specified by the client (under the restrictions imposed by the access control list), which indicates in this way what operations it intends to perform on the item (e.g. read-only file, read-write, etc.). The main primitive in the access control interface, to be invoked against one proxy object, returns a new proxy object representing the same item, but with different credentials and/or access rights, subject to mediation through the ACL and

authentication mechanisms. The system has special bootstrap mechanisms to create an agent/proxy object with "anonymous" credentials, and simple rules for inheritance of credentials when an operation on one item must return access to another item.

The other primitives in the access control interface can be used to query the status of agent (credentials and enabled access rights), find out the privileged identifier for a given item, or manipulate access control lists.

3.4. Naming

3.4.1. Basic Concepts

All items in the system are listed in a single, uniform name space controlled by the standard name service interface. The structure of this name space is hierarchical; it includes leaves for servers implementing a single item such as a device, and whole subtrees representing all the items managed by one server, such as all the files in a file server. Each server managing such a subtree must conform to the name service interface; however the internal implementation is left to the discretion of the designer.

The name space is defined around five fundamental entities, all represented by items, and exporting the normal access control interface:

- a *terminal* is any item which is at a leaf in the name space, that is, which does not itself export any name service operations such as lookup.
- a *directory* is a container for a set of entries in the name space, identified with a single-level name. It exports a *lookup* operation, which returns a proxy object for the item designated by the given entry name.
- a *symbolic link* or *symlink* is essentially identical to the symbolic links found in traditional operating systems. It contains a path name used to redirect the name resolution when it is encountered during a path name traversal. The path in the symlink can be absolute, or relative to the directory containing the symlink.
- a *mount point* operates like a symlink to redirect a name resolution, but instead of containing a path name, it contains a direct reference (proxy and associated Mach port) to another point (directory) in the name space, possibly managed by a different server. This reference is returned to the clients in the form of a simple proxy object representing the target of the mount point.
- a *transparent symlink* is exactly identical to a normal symlink, with the exception that the system allows clients to specify independently whether transparent and normal symlinks should be automatically followed when found at the end of a path by the pathname resolving logic. Transparent symlinks provide a simple way to “graft” arbitrary items in arbitrary directories of the name space, without increasing the complexity of all directory servers beyond support for symbolic links.

3.4.2. Name Resolving

With this organization, name resolving proceeds in the following manner:

1. the client ships the entire path name to a base (root) directory.
2. this directory, or the server managing it, consumes as much of the path name as possible before returning to the client. If the desired object is found, an appropriate proxy object is returned. If a symlink or mount point is encountered, the server returns to the client the information contained in that object, as well as an indication of how much of the path name was traversed before encountering it.
3. if necessary, the client applies the necessary transformations implied by the symlink or mount point, and restarts the resolving loop at step 1, possibly with a different starting directory.

Note that this scheme leaves to the client all liberty to interpret symlinks in a way specific to any given environment. Similarly, concepts such as the UNIX "." and ".." are interpreted by the client, not by the server. Finally, it is not even necessary for the client to know the path name component separator character ("/", "\", ...) in use in a given server, since each server will consume those characters naturally when it receives the whole path name.

3.4.3. Object Creation

The name space is also used as the framework to control the creation of items. Each item in the system, with the exception of the root directory on each server, can only be created in the context of a directory, by an operation exported by that containing directory. In this way, all items are automatically named and reachable, and the access control list on the containing directory can be used to mediate item creation. Moreover, this scheme allows items to be created, while avoiding the need to export any abstraction of a server at the user level. Any item can be subsequently detached from its parent directory, but in this case, that item can only be volatile, and disappears as soon as its last current client releases it.

However, this scheme leads to one complication. Although all directories fulfill the same basic functions with respect to the name service interface, the structure of the name space is such that many directories are embedded inside various servers. Those servers do not necessarily support all kinds of items. To resolve this problem, the name service interface defines a specific *type* for each item, and each directory possesses a special attribute that is the list of all item types that it may contain. In this way, directories in various servers are "specialized" with respect to item creation.

3.4.4. Other Functions

Next to the basic functions of lookup and creation, the name service interface specifies a number of other functions for removing or renaming entries, listing the contents of directories, etc. In addition, every item in the name space exports a set of operation to query various *attributes* or groups of attributes, giving generic information about that item.

All the primitives in the name service interface are normally intended to be exported by servers that implement at the same time a set of terminal items, and the portion of the name space that contains those items. However, when security requirements are very severe, it is desirable to split these aspects between two servers: an item manager, which implements the terminal items, and a *segregated name server*, which is responsible for the naming and access control functions. In this way, the segregated name server can operate in a separate address space from the item manager; it can be easily standardized, and formally verified. To distinguish it from this segregated mode of operation, the normal organization is sometimes called *integrated name server*.

3.4.5. Name Space Organization

All the subtrees in the various servers are assembled in a single name space through mount points in one or more system or *root* name servers. These servers are standard components supporting directories, mount points and symbolic links, but no terminal entries. They do not maintain any persistent storage; their naming hierarchy must be created dynamically during the system startup and configuration process.

In addition to this global name space, each client of the system maintains a separate *prefix table*. This table contains a set of path name prefixes, and corresponding proxy objects representing items in the name space that are associated with those prefixes. Every name resolution performed by a client starts by a search in the prefix table to find an initial directory to handle the path name. This mechanism is used for several functions:

- some prefixes established at bootstrap time represent a few key directories in the name space. They provide good starting points for name resolution without incurring the overhead of contacting a single server implementing the "root" directory.
- prefixes provide a level of indirection between clients and the global name space. In this view, they can be used as a form of client-specific mount points, to customize the name space by specifying different names for various subtrees, and even by masking existing path names to redirect them to other subtrees. Among other applications, this feature allows clients operating under various environments to observe a name space that fits the expectations of those environments.
- prefixes can act as a cache for name resolving, to avoid the processing of mount points and symbolic links on each lookup, and direct the client immediately to the server containing the requested object.

- some prefixes that can be redefined freely by the client itself are used to generalize the notion of a "current working directory", without requiring that any state be kept in any servers.

3.5. Input / Output

3.5.1. Generalities

The input/output interface is responsible for all transfers of data between an emulation client and an item, representing abstractions such as files, pipes, sockets, etc. It specifies a collection of basic operations to manipulate simple unstructured data, and provides a basis for the definition of “derived” operations to handle more complex data elements (i.e. data with attributes, etc.). In addition, the I/O interface supports the use of shared memory buffers as an alternative to IPC for data transfers, and manages independent concurrent access to the same data by multiple users.

3.5.2. Types of Client-server Interactions

The first question for the specification of the I/O interface is to determine the nature of the I/O interactions, and how they are initiated: existing systems include concepts of data sinks, sources, pumps, asynchronous and synchronous I/O, polling, etc. To keep the standard interface as simple as possible, the standard interface specifies that all I/O operations should be synchronous, and initiated by the client: the client requests each individual data transfer explicitly by invoking an operation (read or write) on the data-managing item, and blocks waiting for that transfer to complete. Under no circumstances can an item effect more than one data transfer in response to a single client request, or interact with a client that is not blocked waiting for that interaction. All I/O primitives also specify two mode flags that modify the type of client-server interaction performed:

- the *wait* flag specifies that the I/O operation should block until it can be completed. Otherwise, the I/O operation would return immediately with a failure code instead of blocking.
- the *probe* flag specifies that the item should report exactly how the specified I/O operation would be performed (number of bytes transferred, wait, etc.), but not actually transfer any data. The state of the item cannot be modified by the invocation of an I/O operation with the probe attribute.

These flags, in conjunction with the use of multiple synchronous threads in the client emulation libraries as needed, provide enough functionality for the various other useful types of I/O interactions.

3.5.3. Specification of Data Elements

The second question is to determine how a client can specify, in an I/O request, the exact portion of all the data represented by an item that is to be the object of a transfer. Although the I/O interface should in principle ignore the structure and nature of the data that it is manipulating, this problem requires at least the definition of data elements that can be individually designated within the set of all data. For this purpose, each data-managing item is defined to be either byte- or record-oriented, depending on the intrinsic nature of the data set that it represents.

A byte-level I/O item represents a collection of bytes such that each byte can be manipulated (read or written) by clients independently of all other bytes in the collection. Examples of such data sets include UNIX files, UNIX pipes, etc. The I/O interface designates every byte in a byte-level data set with a unique 64-bit *offset* and provides operations for the transfer of individual bytes or of ranges of bytes with consecutive offsets, designated by a starting offset and a length.

A record-level I/O item represents a collection of bytes grouped in *records* such that each record must always be manipulated as a whole. Records may be all of a fixed size or have arbitrary variable sizes. Examples of such data sets include record-oriented files (not UNIX), packets on a network connection, etc. The I/O interface designates every record with a 64-bit *record number*. To avoid the problem of marking record boundaries in a collection of variable-length records stored in a single buffer, the simple I/O primitives define operations for the transfer of only one record at a time. An extended version of the I/O interface (see below) may support the manipulation of ranges of records with consecutive record numbers.

The I/O interface also defines a special mode flag named *truncate* to control the behavior of the I/O items when manipulating ranges of bytes or records. If set, this flag specifies that, if the item can only currently transfer (read or write) fewer data elements than have been specified in the client request, the I/O operation should transfer as many elements as possible at once and ignore the rest of the range. Otherwise, if this flags is not set, the I/O operation should fail or block until the complete range of data elements can be transferred in one operation.

3.5.4. Sequential vs. Random Access

For each fundamental operation (read/write, byte/records, ...), the I/O interface defines two variations for random and sequential access. For the random access primitives, the client specifies in the request the offset or the record number of the data elements to transfer. For the sequential access primitives, the offset or record number is returned as a result of the data transfer operation (read or write), to be used in future references to the data elements in question.

In general, most types of data access can be implemented as applications of the pure random-access model with a single scalar data identifier, augmented with external support such as table of contents, indices, counters, etc. In particular, the traditional UNIX sequential file reading semantics (“seek”) are implemented with a “current offset” maintained by the client emulation libraries and not by the I/O items. This approach

relieves the I/O items of the responsibility to keep track of multiple seek offsets for several unrelated groups of clients, and facilitates the use of memory-mapped files and the handling of multi-threaded clients. The sequential-access primitives are intended to be used for the special cases of I/O interactions that have an intrinsically sequential character not imposed by the clients, such as pipe I/O, TTY I/O or multiple clients appending to a file.

3.5.5. Data Manipulation

Most naive applications and application programming interfaces manipulate I/O data by copying to and from buffers allocated and specified by the clients. However, to maximize the performance of data transfers through shared memory buffers as well as between modules inside individual servers, it is desirable to manipulate data through pointers to long-lived, system-controlled buffers that can be simply passed without copying along a chain of data processing modules.

In the interest of simplicity, the standard I/O interface is divided between a basic and an extended interfaces corresponding to those two modes of operations. The basic interface only supports copies to/from client buffers, and is exported to all clients (emulation libraries). The extended interface defines additional data structures (blocks and records) to represent data buffers, and a set of primitives that parallel the "basic" primitives but operate on those data structures. This extended interface is not currently exported to external clients, but it is used for the internal implementation of various servers and proxies in the system prototype. It may be exported for sophisticated clients in the future. Note that thanks to its more sophisticated mechanisms for data representation, this extended interface can also support the manipulation of ranges of records and provide scatter-gather capabilities for blocks of data.

3.6. Asynchronous Notifications

The notification facility allows various service providers to initiate interactions with emulation clients and inform them of special events happening in the system. Its main use is for the emulation of UNIX-like signals.

The fundamental abstraction in this facility is an *exception item* instantiated inside the emulation library of each client process. This object exports a simple operation to deliver a notification to the target process. Such a notification contains a simple numeric code, from a unified space of error and exception codes. It can be handled by the emulation library in any way specified by the implementer of that library; the most common approach is to raise an exception at the programming language level in the client process.

To allow external servers to locate and access those exception items, each emulated process is represented by a *task item* managed by the process management server or *task master*. This server makes those items available in the global name space, and takes care of access mediation in the normal fashion. Among a number of primitives concerned with control of the target task, the task item exports an operation to forward a notification to the target emulated process.

3.7. Support for Shared State

In a typical emulation system, there are many instances of information that must be shared between several components:

- UNIX file descriptor state, shared between multiple UNIX processes
- file size and control information, shared between a file server and potentially many clients
- information maintained by various servers and cached in the emulation library
- etc.

The most direct approach to deal with this issue is to integrate support for each specific data sharing requirement rigidly in the design of each concerned server or client. However, in keeping with the design guideline of attempting to minimize the number of specialized mechanisms in the system, a general purpose facility is currently under consideration, that can be used as a basis for many different applications.

The model for the shared state management facility is based upon the abstractions of *shared records* and *blackboards*, managed by a central *blackboard server*. Each element of data in the system that must potentially be shared by several components is represented as a separate abstract record allocated by the blackboard server and uniquely identified throughout the system. Each client of the sharing facility is associated with a private blackboard item in the blackboard server, that holds the collection of shared records accessible to this client. The data stored in those shared records can be manipulated by each client through simple operations exported by the proxy for the blackboard. Note that in general, by the very nature of a shared state facility, a single shared record may be accessible through many client blackboards. Clients use an explicit *attach* primitive to gain access to specific records in the context of their own blackboard.

The preferred implementation of this facility is for each blackboard to correspond to a region of memory shared between one client and the blackboard server, storing a copy of all the relevant shared records. The blackboard server is then responsible for keeping track of client accesses and for copying the most up-to-date information into each blackboard as appropriate. When this full shared-memory implementation is not possible, the blackboard facility can also be implemented with other caching schemes at the client level, or even with no caching at all, by forcing each client to communicate with the blackboard server via RPC for each data access.