Naming Facilities for Operating System Emulation in Mach 3.0

Daniel P. Julin

Abstract – **The Mach 3.0 micro-kernel provides a foundation for the emulation of various operating systems with a combination of libraries and Mach servers. We present a general-purpose naming facility to help construct such emulation systems by assembling many independent servers. This facility defines a set of standard interfaces for a uniform name space with uniform access mediation and a set of practical mechanisms to integrate servers into this name space, through prefix tables and ancillary servers supporting special mount points. We then discuss the application of this facility to UNIX 4.3 BSD emulation, to a secure system with a centralized trusted name server, and to larger distributed systems.**

I. INTRODUCTION

One of the main focuses of research within the Mach project is the use of a "micro-kernel" architecture to emulate the functionality and programming interfaces of operating systems such as UNIX, VMS, MS-DOS, etc.[1] Several sub-projects are currently underway to investigate different design alternatives within this overall architecture, and to produce working emulation systems for various target operating systems. One such sub-project, often referred-to as the "Mach 3.0 multi-server emulation system", concentrates on identifying general issues and proposing common techniques that can be used to facilitate the design and implementation of emulation systems in general, independent of any specific target. This effort centers on the specification of a flexible common framework for building emulation systems with a combination of independent servers that cooperate to implement the complete emulation. One of the key elements of this framework is a general-purpose naming facility that organizes the interactions between the emulated client processes and the various servers. This paper discusses the design of such a naming facility.

II. BACKGROUND

The proposed general architecture for emulation contains the following elements (many of which are shared to some degree by other Mach emulation designs):

1. the Mach 3.0 micro-kernel, that provides a few basic facilities needed for the execution of the other components. These facilities include *task* and *thread* management, virtual memory management (including user-level control of *memory objects*), an inter-process communication facility (IPC), and raw device I/O.

2. a collection of *system servers* operating in separate user-level Mach tasks, that export all the high-level functionality required of different operating systems. Examples of such servers include one or more file servers, a terminal or "TTY" server, a process manager, an authentication server, a system management server, etc. One of the major ideas of the design is to structure this collection of servers so that it can be easily adapted to support different target operating systems or variation to existing targets. This potential is maximized by separating as much possible the "raw functionality" of the various services from the details of interface or "presentation" specified for each target system. Many servers can simply be reused for different emulations. Others can be used as a skeleton that can be easily modified to suit different requirements.

3. a target-specific *emulation library* transparently loaded in the address space of each process for which a particular programming interface must be emulated (also called *emulated processes*), that intercepts the system calls issued by the user program and handles them by invoking appropriate services in appropriate servers. The emulation library acts as a translator between the more-or-less generic services described above and the interface of the operating system to be emulated, thereby allowing the definition of the interfaces for those services to be guided primarily by considerations of generality, flexibility and efficiency, and not by the idiosyncrasies of any particular target. In addition, the emulation library often maintains local state for each emulated process, and may perform some complex functions locally to reduce the number of interactions with servers.

The components in this system communicate using a combination of the shared memory and IPC facilities provided by the kernel. Only the Mach IPC facility is relevant for the purposes of this paper. It is based on the abstraction of uni-directional message queues named *ports*, with a single *receive right* and multiple *send rights* distributed among various Mach tasks. Port rights are treated as secure capabilities for the ports that they represent and can be transmitted in messages; they can be used at the same time as handles for communication operations and as protected references for individual objects or resources in the

Author's address: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

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.

system. The local IPC facilities exported by the kernel are transparently extended in the network environment by a collection of *network servers*[2].

Every high-level service exported by the servers in the emulation system is defined following an object-oriented paradigm, with operations exported by one or more *items*. The word *item* is used here to avoid confusion with various language-level objects or classes that may be used internally for the implementation of each service. Examples of items (using a UNIX terminology) are files, pipes, sockets, processes, devices, as well as a variety of miscellaneous services such as time, system administration, monitoring, etc. Each item is managed by a particular server; some servers are responsible for only one item (e.g. system administration), while other servers may manage large number of related items (e.g. all the files in one file system). Some items may exist only as long as they are actively being used (e.g. sockets), while others are long-lived and must persist between consecutive periods of use.

The set of operations exported by all the items is subdivided into a number of standard interfaces that can be combined or specialized as needed. Examples of basic interfaces include a standard input/output interface, a network control interface and a uniform naming interface, that constitutes the focus of this paper.

III. FUNCTIONALITY AND REQUIREMENTS FOR THE NAMING SUBSYSTEM

In general, a complete emulation system can be viewed as a loose federation of servers that together export and manage all the items needed for a specific emulation. The naming subsystem provides the common "glue" between those servers, and is the mechanism through which emulation libraries in various systems gain access to all the individual items needed to perform each particular emulation function. It defines tools to create a single uniform name space encompassing the collection of all items managed by all emulation servers, a standard protocol to navigate through this name space, and a common framework for access mediation. The basic functions are the resolving of names into items and item creation.

The basic name resolving operation returns a handle for a single item uniquely identified by a path name. This handle can then be used by a client to invoke other operations on the item. The naming subsystem takes care of access mediation for those operations, and garbage-collection when the handle is released by the client. To simplify the overall system, and to permit a pure object- or item-oriented model that does not force the explicit definition of groups of items belonging to a same server, the same naming protocol and resolving operations are used to locate individual servers or servers exporting a single item, and to navigate inside a "subtree" of the name space containing all the items managed by one particular server.

The basic item creation operation combines creation of the new item and entering it at an appropriate location in the name space. This approach provides a natural mechanism to integrate this operation within the access mediation framework, by applying the access control on a directory to mediate the creation of items to be entered in that directory. In addition, it helps guarantee that all new items will be automatically visible in the name space, and avoids the problems encountered when an item managed by one server must be entered in the name space of another.

Within the context of this model, the design of the naming subsystem addresses the following high-level requirements or desirable properties:

multiple configurations: it should be easy to build many different configurations by assembling different collections of servers, to specialize each emulation system for specific user or environmental requirements.

multiple operating systems: the emulation libraries for various target operating systems must be able to present different "views" of the standard name space for each of the corresponding targets, in terms of name space structure, name syntax and functional interface.

integrating new servers: it should be possible to integrate new servers into the system, even if the internal organization of these servers or the particular type of items that they manage were not explicitly part of the original design.

security: the protocols and mechanisms should provide adequate security to control access to items by a variety of clients in a variety of environments and configurations.

performance: the naming subsystem is to be invoked for almost every emulation function. It is critical that its performance be very good, at the same level as that of common naming functions in most operating systems that are to be emulated.

integration into large distributed systems: beyond the (relatively) simple needs of a single localized operating system emulation, the overall design should provide for clean integration into larger naming structures spanning more than one node, more than one emulation, and possibly more than one administrative domain. Specific concerns in that area include issues of robustness, scale, dynamic reconfiguration, and sharing of servers between several "local" systems.

customizable name spaces: to increase the flexibility available to users, particularly in the presence of possibly large, extensible name spaces and in the presence of different target-specific naming requirements, the system should support the definition of different views of the overall name space to be specified by each individual client or system.

Conversely, this naming subsystem is primarily intended as a tool for the construction of emulation systems and the integration of many different servers, and not as a general-purpose facility for distributed computing. Accordingly, a number of features are omitted for the sake of simplicity, adaptability and performance. Other than the fact that Mach IPC is naturally network-transparent, there is no explicit support for traditional high-level functions of distributed systems such as replication or sophisticated caching strategies; such services may be provided,

if desired, through integration of the local naming system into a larger distributed system using different protocols. Complex queries using various attributes or dealing with more than one item at a time can be implemented in specific servers and exported through conventions in the name space and special items, but there is no explicit definition or support for any such facilities in the common framework beside the simple lookup described above. Finally, complex item operations beyond name resolving or item creation are not directly supported; they can be implemented by a combination of a lookup operation followed by operations invoked on the handle representing the item, independent of the naming framework.

IV. SYSTEM OVERVIEW

Several research projects have investigated naming facilities for distributed systems or systems with multiple servers, and have proposed a number of useful ideas and mechanisms[3, 4, 5, 6, 7, 8, 9, 10, 11]. This section presents an overview of the complete naming facility proposed for emulation systems under Mach 3.0, that tries to take advantage of the experiences from these earlier systems.

A. Client-Server Interactions and Access Mediation

Servers give clients access to the items that they manage by handing out *send rights* for Mach ports for which they hold the corresponding *receive rights*. The clients can request operations on each item by sending messages to the appropriate port. The servers associate each such port with a combination of three elements: a designation of a particular item, a set of *user credentials* and a set of *access rights*. The credentials are an abstract representation of the identity of the client or clients that request operations through that particular port. The access rights perform two functions: first, they are a simple specification of which client requests should be accepted or rejected when received on that port; second, the existence of an outstanding send right with particular access rights may be used as an indication of which access modes are currently active for that item (e.g. "read-only", "read-write", etc.). The association of each port with credentials and access rights is immutable, established by the server when creating the port and granting access to each specific client. Since Mach ports are secure capabilities, clients cannot subvert the system by forging or stealing different ports.

The operation by which a client requests access to a given item with specific credentials and access rights is the equivalent of an "open" operation in UNIX or other operating systems. The client may securely specify the desired credentials by sending to the server an authentication token, to be interpreted by a mutually trusted *authentication server*. The exact nature of this authentication service is independent of the naming protocol; the current prototype uses a simple server with Mach ports as secure authentication tokens. To provide a uniform access mediation mechanism and interface, each item is associated with a general-purpose *access control list* to be matched against the user credentials when establishing the valid access rights, and a *type* that defines the association between access rights and each operation exported by the item. Servers may create special "un-authenticated" ports, that can be used to provide clients with references to individual items without allowing them to perform any operations directly, other than to re-authenticate to obtain a new port with different access attributes.

There is no corresponding "close" operation: clients simply deallocate their send rights to release their access to the corresponding items. The Mach kernel sends a *no-more-senders* notification to the server when all outstanding send rights for a port disappear, thereby providing an automatic garbage-collection mechanism that functions even in the presence of client crashes.

Finally, to provide additional flexibility, clients do not normally manipulate the above send rights directly. Instead, each send right is encapsulated in a special body of code loaded in the client's address space, called a *proxy object* or simply *proxy*[12]. 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 directly forwards 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). New proxies are specified by the server and automatically instantiated in a client's address space by the RPC run-time system, as a side-effect of receiving an item reference in a message.

B. Name Service Items

The name space is defined around four basic abstractions, each of which is represented by a special type of item subject to the standard access mediation facilities:

- a *directory* is a container for a collection of entries in the name space, each of which corresponds to an item managed by the same server that manages the directory itself. Each entry is identified by a single-level alphanumeric name. Each directory exports a simple *lookup* operation to translate such names into item references represented by proxy objects.
- 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 that 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 name resolving protocol

allows clients to specify independently whether transparent and normal symlinks should be automatically followed when found at the end of a path. Transparent symlinks provide a simple way to "graft" arbitrary items in arbitrary directories of the name space, with minimal impact on the complexity of all directories.

In the context of the naming subsystem, every other type of item is called a *terminal*, and its exact nature is irrelevant. Terminals do not themselves export any name service operations, and correspond to leaves in the global name space.

Each server that manages more than one item typically implements a graph of directories and subdirectories, with one main directory acting as the "root" for this server, through which it is connected to the rest of the name space. All the other items managed by that server are naturally entered at the appropriate locations in the graph. There is no requirement that this graph be strictly hierarchical: each item must simply appear in at least one directory to be reachable in the common name space.

These abstractions define a common behavior so that similar functions are treated uniformly across all servers participating into the system, but they do not specify a required functionality for all components. Each server may or may not implement symlinks, mount points and even subdirectories as deemed appropriate. More importantly, the directories used in each server need only hold entries for the items and item types managed by that particular server; there is no requirement to implement a "universal directory" that can hold arbitrary items from the global name space.

In addition to their function for name lookup, directories in each server normally also export one or more specialized *create* operations to instantiate new items. The details of those operations, including the type of items that can be created, are irrelevant from the perspective of the naming subsystem. The only condition is that the newly created item be visible as an entry in the directory to which the "create" operation was applied, and that this creation be mediated by the access control mechanisms in effect for that directory.

Finally, each directory may also export a *list* operation to list all the entries available for lookup. To facilitate integration with large foreign name spaces, and in accordance with the general principle outlined above, this operation is only optional for the various specialized versions of directories implemented by each particular server.

C. Name Resolving Protocol

Name resolving is obviously the primary function of the name service protocol. It proceeds by iteration through the following steps:

1. the client acquires access (a proxy) for a root directory from which to start resolving pathnames.

2. the client ships the entire path name to the initial directory.

3. this directory, or the server managing it, consumes as much of the path name as possible before returning to the client. If the desired item is found, an appropriate proxy is returned. If a symlink or mount point is encountered, the server returns to the client the information contained in that entry, as well as an indication of how much of the path name was traversed before encountering it.

4. if necessary, the client applies the necessary transformations implied by the symlink or mount point, and restarts the resolving loop at step 2 with the remainder of the path name and a new initial directory.

This scheme provides great flexibility for the interpretation of many details of structure and semantics of path names in different operating system emulations. Since whole path names are shipped "as-is" to servers and since the length of the path traversed is always returned, it is often possible to completely ignore the actual structure of path names used by particular servers, including the component separator ("/", "\", "`[...]`", etc.). On the other hand, special syntactic constructs such as the UNIX "." and ".." are handled by scanning path names on the client side, and do not require any knowledge on the server. Symbolic links may easily be interpreted in a client-specific way, for example to allow embedded variables or to support different relationships with the UNIX ".." construct. Finally, there is little dependency between individual servers: a server does not need to know where is it "mounted" in the global name space, and the clients can naturally apply various re-authentication operations whenever the resolving loop moves from one server to another.

This scheme also minimizes the number of client-server interactions by allowing multiple components of a path name to be handled at once, but it may still require each name resolving operation to contact more than one server, or the same server more than once. A special caching mechanism described below alleviates this difficulty.

D. Name Space Organization

The naming subtrees managed by different servers can be assembled into a single global name space through mount points in ancillary servers called *root name servers*. These servers are standard components of all emulation systems. They support 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. Using one or more of these servers, it is possible to construct a variety of global name spaces to fit different needs. Note however that this mechanism does not in itself provide any guarantees of consistency for the resulting system. In particular, it is possible to create cycles in the name space, and to structure the name space in such a way that symbolic links across servers are meaningless. Such concerns should be addressed manually by system administrators, or through automatic configuration mechanisms that are outside of the scope of the naming subsystem.

In addition to this global name space, each client of the system may maintain 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 with 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. In addition, these prefixes can be set-up to adapt the client's view of the common name space to match the expectations of a particular target operating system emulation.

- prefixes can be used as a cache for name resolving, to avoid the processing of mount points and symbolic links on each lookup, and to direct the client immediately to the server containing the requested object.

- prefixes that can be redefined freely by the client itself can be used to dynamically modify that client's name space. A simple application of this feature is for the emulation of the UNIX "current working directory" or VMS "logical names". A more complicated application is to let clients customize their name space by adding subtrees from personal servers, or even by masking existing path names to redirect them to other subtrees.

V. DISCUSSION AND APPLICATIONS

The description above has been centered on a set of tools and features, independently of how these tools can be put to use in practice. The following sections describe the design of a few practical applications that may use the common naming facility, and discusses how the various aspects of this facility contribute to the complete systems under consideration.

A. *UNIX 4.3 BSD Emulation*

The main application of the naming facility is a prototype UNIX 4.3 BSD emulation system currently under development at CMU, that is used as a tool for the evaluation of all the ideas proposed in the design of the common emulation framework. The implementation of this BSD prototype is not yet complete, but large portions are now operational (starting a shell, executing simple commands and editors, compiling programs, etc.). It appears that the common naming scheme is working well in this context.

The core of the name space seen by UNIX emulated processes in this system is provided by one or more *UNIX file system servers* (UFS), each of which exports all the files and directories of one disk partition as a subtree of items following the standard naming interface. Typically, one such server is attached to the prefix "/" in a UNIX-specific prefix table that is part of each client's emulation library. This prefix table performs all necessary translations between the standard naming interface and the corresponding UNIX interface. Other file systems may be directly "mounted" with other prefixes, eliminating the need for interaction with a mount table in a primary file server or in a specialized server for each UNIX `open()` operation.

Other UNIX objects such as *tty's*, devices, pipes and many flavors of sockets are all implemented as items in separate servers that export them through their own internal directories or naming subtrees. Transparent symlinks in the file server are used whenever such an item must be made visible somewhere in the UNIX file system space. To provide a context for the interpretation of these transparent symlinks, as well as to provide a way for emulation libraries to manipulate the target items directly and to create new items, all the relevant servers are attached to mount points in a single root name server, that is itself attached to a "/servers" prefix. For example, to create a new UDP socket, the UNIX emulation library might simply resolve the name `/servers/net/udp`, and invoke a socket creation operation on the directory returned by this lookup. This scheme has the major advantage of not requiring each file server to support complex "foreign" entries in its name space beyond simple transparent symlinks that can easily be implemented with the help of ordinary symbolic links, even though a native UNIX implementation would.

Finally, the prefix table also maintains several elements of UNIX user state without requiring any support from system servers. The main examples of such state are the UNIX "current directory", the "current root" as established by the UNIX `chroot()` system call, and the "controlling terminal" (*/dev/tty*).

B. *Segregated Name Servers*

In the examples discussed so far, the same server is managing a number of "terminal" items and the portion of the name space in which these items are located. Such an organization can be referred-to as an *integrated name server*. However, there are instances where it is advantageous to separate these two functions, with an organization called a *segregated name server*. One such case is presented by a secure UNIX implementation currently being implemented in the Mach emulation framework, targeted at the B3 level of security[13]. In this system, all operations by which clients gain access to items are concentrated in a single central name server that is subject to very careful verification, and that operates at a high level of trust. The items themselves are managed by a number a separate *item managers* that are typically not as trusted as the name server. In some cases, the low-level security mechanisms of the system may even prevent those item managers from transferring port rights to and from their clients, thereby making it impossible for them to grant access to services directly. Under these conditions, the operations exported by each item, that deal with the protection of that item, must be handled by the name server and not the item manager: get/set access control list, authenticate user credentials, audit the granting and releasing of access to the item, etc.

The common naming facilities support an implementation of this architecture, based on a few special mechanisms. First, the name resolving and item creation operations are handled by directories in the name server, which must interact with the item manager to determine exactly what to return to the client and themselves construct the proxy and send rights to be used for client access. This interaction follows an internal protocol independent of the common naming interface. While such an

interaction is taking place to decide on the creation of a new item, the name server must protect the tentative name of the new item in its containing directory. This can be achieved by delaying the execution of other conflicting operations until the creation is complete, or by exporting a special "reserved" item as a place-holder when appropriate.

Second, since some item operations must be handled by the name server, each proxy for terminal items must contain two send rights. One of these rights is used to send requests to the name server, and the other is used to send requests to the item manager. The proxy code itself may perform the appropriate dispatching, thanks to the large degree of separation between the primitives in the name service interface.

C. *Extension to Large Distributed Systems*

A third area of interest, which is still mostly of a speculative nature in the context of the present system, is the extension of the name space available in a single emulation system to encompass much larger domains. The common naming facilities offer two main approaches to address this issue:

First, the same flexibility and simplicity of the service interfaces that make them suitable for use in a variety of emulation systems also facilitate the integration in such a system of special servers that act as gateways into other names spaces or distributed systems. One very simple example of this approach is a *NFS server* used in the BSD emulation prototype, that translates local naming requests and forwards them over a network using the NFS protocol[14]; other systems that may be considered for integration along the same principle range from straightforward distributed file systems to less conventional distributed naming services. When necessary, the gateways may extend to functionality of the basic naming system, either by adding more structure to the largely structure-free pathnames, or by building upon the object-oriented nature of the interfaces to add new primitive operations.

Second, since Mach IPC is the primary mode of communication between components, it is possible for clients to access servers on other nodes or other systems directly. Since there is no requirement that the global name space formed by all accessible servers be strictly hierarchical, and since servers need not be aware of their own location in that name space, any combination of root name servers, mount points and prefixes is allowed. Individual servers or groups of servers may be made available at arbitrary locations in the name space of remote clients, without affecting in any way the operation of the local system to which these servers participate. The prefix table, by generalizing the notion of "root", allows such extensions of the name space to be gracefully integrated in an existing system, and eliminates most of the performance issues associated with scale on more centralized systems.

The *netname service* provided by the basic Mach network IPC facility constitutes a convenient mechanism for bootstrapping of such inter-node relationships. This service implements a simple unprotected remote lookup for selected send rights identified by the combination of an IP address and a single-level name. The initial access to a foreign name space is established through the lookup of an "un-authenticated" send right for an appropriate base directory in a remote server, with which a client can construct a simple proxy and re-authenticate to obtain a more complete proxy under the standard access mediation facilities.

The netname identifications of such remote initial ports may simply be transmitted outside the realm of the distributed system altogether, or they may be stored as strings in other established name services such as the Internet BIND facility[15]. With the latter scheme, it is conceivable to provide each local system with a special gateway server that gives it access to a "super-root" grouping all the Mach 3.0 systems on a network.

VI. SUMMARY AND OPEN ISSUES

A flexible naming facility is a powerful tool to construct systems by assembling a collection of independent servers with minimal per-system effort. The facility described here fulfills this role by providing a uniform mechanism to connect clients and servers, to handle access mediation, to locate objects in complex name spaces, and to integrate arbitrary servers in the overall system. It supports the general requirements of a variety of emulation systems and is a good starting point for use in larger distributed systems.

We continue to experiment with our BSD prototype, and to refine the design of the common emulation framework to match our experiences and meet new requirements as we encounter them. A few of this issues under consideration in the context of the naming subsystem are:

- The name service protocol may have to be extended to support an integrated facility for mandatory locking of access to items. Currently, such functions are implemented by specialized item operations outside the naming facility.
- The naming facility exports only simple items and generally avoids manipulating servers as a whole. There is a need for an abstraction representing a server itself for system management.
- Symbolic links are insufficiently specified. Their interpretation depends on each client's prefix table, and can only be kept consistent through strong conventions regarding the contents of such prefix tables. In addition, we may also wish to define symbolic links that are relative to special points in the name space, beside the directory that contains the link itself.
- The prefix table considerably complicates the transmission of a client's context for remote execution. Mechanisms must be designed to simplify this operation. In addition, more facilities are needed to manage prefix table entries that are shared between multiple clients.

ACKNOWLEDGMENTS

The work discussed here incorporates contributions from a great number of people at Carnegie Mellon, at the Research Institute of the Open Software Foundation and at Trusted Information Systems. The list of names has now grown too large to be printed, but the efforts of everyone are greatly appreciated.

REFERENCES

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

[2] R. D. Sansom, D. P. Julin, and R. F. Rashid, "Extending a capability based system into a network environment," in *Proceedings of the ACM SIGCOMM 86 Symposium on Communication Architectures and Protocols*, pp. 265–274, ACM, August 1986. Also available as Technical Report CMU-CS-86-115.

[3] L.-F. Cabrera and J. Wyllie, "QuickSilver distributed files services: An architecture for horizontal growth," Tech. Rep. RJ 5578, IBM Almaden Research Center, Apr. 1987.

[4] D. Cheriton and T. P. Mann, "Decentralizing a global naming service for improved performance and fault tolerance," *ACM Transactions on Computer Systems*, vol. 7, pp. 147–183, May 1989.

[5] D. Comer and T. P. Murtaugh, "The Tilde file naming scheme," in *Proceedings of the 6th International Conference on Distributed Computing Systems*, pp. 509–514, IEEE, May 1986.

[6] M. Guillemont and J. L. Martins, "UNIX into the distribution age: the CHORUS operating system." Internal document, INRIA, number EXP 4 540.6, 1987.

[7] B. W. Lampson, "Designing a global name service," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pp. 1–10, ACM, Aug. 1986.

[8] K. Lantz, J. Edighoffer, and B. Hitson, "Towards a universal directory service," in *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pp. 250–260, ACM, Aug. 1985.

[9] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba – a distributed operating system for the 1990s," *IEEE Computer Magazine*, vol. 23, May 1990.

[10] D. B. Terry, "Structure-free name management for evolving distributed environments," in *Proceedings of the 6th International Conference on Distributed Computing Systems*, pp. 502–508, IEEE, May 1986.

[11] 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*, pp. 184–189, IEEE, May 1986.

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

[13] J. Graham and W. Morrison, "Trusted Mach system architecture." Internal document, Trusted Information Systems, Glenwood, MD, April 1990.

[14] Sun Microsystems, Inc., Mountain View, CA, *Networking on the Sun Workstation*, 1986. Part No 800-1324-03.

[15] J. M. Bloom and K. J. Dunlap, "Experiences implementing BIND, a distributed name server for the DARPA internet," in *Proceedings of the 1986 summer Usenix*, pp. 172–181, Usenix, 1986.