Client-Server Interactions in Multi-Server Operating Systems: The Mach-US Approach

J. Mark Stevenson Daniel P. Julin

September 1994

CMU-CS-94-191

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes requirements placed upon client-server interaction in a multi-server operating system and how to answer those requirements. Addressed are the problems of binding maintenance in the face of: remote method invocation, forking, binding transfer, authentication, asynchronous interruption, and client crashes.
Design and implementation choices are analyzed. The solutions, used by the CMU Mach-US multi-server UNIX 4.3BSD emulation on the Mach3.0 kernel, are described.
The lessons learned are applicable to multi-server OS design and should be applicable to object based systems that must resolve these binding problems in a "micro" kernel environment.

This research is sponsored by the Advanced Research Projects Agency under contract number DABT63-93-C-0054.

Keywords: Multi-Server Object Oriented Operating System, Mach, Client-Server Interactions, Remote Method Invocation and Interruption, Inter-Process Communication, OS-item, Proxy

1 Introduction

Operating systems are being developed with a kernel-based multi-server structure. Such systems use an operating system kernel to supply system primitives such as tasks, virtual memory, and IPC. Separate servers are used to support derived abstractions (*OS items*) such as files, ttys, and pipes. Additional client-side code may be used to do additional OS computation. Examples of such architectures can be seen in systems such as IBM's "Workplace OS" [Phelan+93], OSF1-AD [Zajcew+93], FSF "GnuHurd", Sun's "Spring" [Khalidi&Nelson93], and Mach-US from Carnegie Mellon University [Julin+91].

The goal of this separation is to achieve several kinds of flexibility in system configuration and system development. These include:

- Rapid development of new or modified subsystems without the potential of crashing or corrupting other services.
- Replaceable services for features like authentication, file systems, or networking.
- Support for multiple "application programmer interfaces".
- Efficient use of multi-processor systems.
- Accessing services on remote nodes, and support for "loosely coupled" OS configurations.

One approach is for these goals to be achieved via object-based software with transparent remote method invocation.

A framework for client-server interactions constitutes a central point for the design of such a multi-server OS. A good framework simplifies the implementation and integration of all the components in the system. Conversely, a bad framework may adversely affect the complexity, performance and expressive power of many components.

The thrust of this paper is to examine the requirements of such a framework of invocation and binding maintenance. A model is developed and the tradeoffs are analyzed. The choices made for the Mach-US multi-server system are described and that system will be examined to determine if the client-server interaction goals were achieved with appropriate performance.

This paper is not about the general architecture and implementation of the Mach-US operation system, which is described in [Julin+91]. However, it is useful at this point to present a brief overview of its design and features. This is necessary to supply needed context for the discussion of how client-server interactions work in a multi-server OS and how they are supported within Mach-US.

2 Background: What is Mach-US

The Mach-US operating system was created as part of the Carnegie-Mellon University MACH project. Mach-US is an OS which uses the CMU Mach3.0 kernel and runs on i386/i486 based hardware supported by Mach3.0 including most IBM clones, and the Sequent Symmetry multi-processor.

Mach-US currently supplies a Mach2.5/4.3BSD API and runs most non-administrative UNIX applications. No re-compilation or re-linking is required.

2.1 Quick System Architecture Overview

Mach-US is a symmetric multi-server OS emulation system. It has a set of separate servers supplying generalized system services (file systems, network server, process-mgr, tty server, etc.) and an emulation library loaded into each user process. This library uses the services to generate the semantics of an industry standard OS being emulated. Some of the system servers are, in part or whole, specific to the OS being emulated while others are meant to be fully generic. Care was taken when writing the OS specific servers to make parts of them reusable for other OSs. There is no central server, either for emulating a specific OS or for general traffic control. All multi-service actions are controlled by the emulation library. Details can be found in [Julin+91].

2.2 Mach-US: Flexibility Through Design

The most important property of Mach-US is its flexibility. It offers a new, highly modifiable OS architecture without significant structural impediments to speed. Some of the features that supply this flexibility are:

- Object-Oriented Generic OS Interfaces: These are C++ object-oriented interfaces (virtual classes/methods for multiple inheritance) that define semantics supplied by the system servers. These interfaces are: access mediation, naming, I/O, net_control(OSI-XTI based), and async notification. The servers support a combination of these interfaces to do their work and an emulation library uses them to emulate the target OS in question.
- Modular Services: Different functionality is separated into these servers: configuration, authentication, root-name, diagnostic, pipenet, ufs, process-mgr, tty, and network (with a University of Arizona xKernel protocol engine [Hutchinson&Peterson91]). This separation makes it simple to develop and debug OS services. One can also add and subtract services as needed for a given invocation of the system.
- Object Library/Code Reuse: There is an extensive object library both for support and implementation of the various generic interfaces as well as general server building blocks. This enables faster prototyping of a new server and eases creation of servers from foreign code. Some highlights of this are support for name-space manipulation, protection, shared-mem, IO, and extensive interrupt/signal support.

3 Client-Server Interaction Model

3.1 Client-Server Interaction Objectives

The main design considerations for the client-server interaction mechanism are:

Integration with the system interfaces: Obviously, the invocation facility must work well with all the system interfaces. This has several important implications. First, all invocations are always initiated by the client and proceed synchronously until some sort of response is generated by the server (optionally, there may be no response). Second, the invocation facility must support dispatching of operations to the appropriate items, without exposing server boundaries. Finally, the invocation facility must also support combinations and subclassing of interfaces and handling of opaque argument types.

Transparency: Although most items are managed by a server distinct from the client and residing on the same node as the client, a few special cases must also be taken into consideration. Some servers may be distributed across a network and shared between several nodes. Conversely, some items, although logically independent from their client, may be best implemented in the same address space as that client. This is typically the case when an item has only one client and has only minimal interactions with the global system state.

Accordingly, the invocation mechanism should be flexible enough to allow the use of various mechanisms for communication between clients and servers. Different communication mechanisms may be appropriate for different placements of clients and servers. In particular, when the client and service manager are located in the same address space, the invocation mechanism degenerates into a local procedure call or object invocation facility. In all cases, the actual communication mechanism used should be as transparent as possible to both the clients and servers of the facility. This permits the actual system organization to be changed or extended with minimal impact on the individual components.

Performance: Since every interaction in the emulation framework must go through the invocation facility, the performance overheads introduced by this facility are of critical importance. This is another argument for supporting multiple implementations and modes of communication within the common invocation facility, so as to use the best approach in each individual case.

One particular approach is to use buffers of shared memory between specific clients and servers when possible. This avoids the overheads often introduced by traditional message-based systems. Another approach referred to earlier is to try to minimize the the number of inter-process interactions themselves, through the use of client-side caching and client-side processing.

Inheritance for process creation: In many target systems under consideration, newly created processes must inherit a number of item references from a parent process. To exercise such an item reference, each client needs to establish a client-server interaction. Therefore, the invocation facility must provide an efficient and secure mechanism for new client-server interactions to be established, reconstructed, or inherited during process creation. Such a mechanism should be scalable to a large number of servers, if necessary.

Garbage collection: Since the invocation facility is the focal point through which clients access servers, it is responsible for keeping track of the allocation and deallocation of client-server relationships, and for the appropriate garbage collection of items and communication resources when they are no longer needed. The garbage collection mechanism comes into action when a client explicitly relinquishes or closes access to an item and when the failure of a client process causes all its item associations to be lost. Client-server relationships may change rapidly in an active emulation system, so the garbage-collection mechanism must be efficient. In addition, the clients of the emulation system (i.e., emulated processes) may fail arbitrarily. The garbage-collection mechanism must be robust in the presence of such failures and maintain accurate information at all times. Note that we consider only client failures in this area, and not server failures. The question of handling failures of arbitrary components that implement the emulation system (as opposed to being clients of it) is part of a larger issue of building fault-tolerant systems, and outside the scope of the current project.

Interruptibility: Many client-server interactions may last for arbitrary periods of time, while the server is performing work on behalf of the client or waiting for the availability of various resources. However, most target systems require that it be possible for a client or end-user to abort an ongoing operation and recover control in a prompt and orderly fashion. This implies that the invocation facility must provide adequate support to interrupt pending low-level operations and reflect the interruption to the higher layers of the target emulation library. In the case when the operation to be aborted is not idempotent (e.g., reading or writing on a sequential data stream, accepting a network connection, etc.), care must be taken that no state is lost, and that the server is also informed of the abort in a manner such that it can perform adequate recovery actions.

Access mediation and client identification: Another requirement of a usable system is that each invocation of a specific operation upon a specific item by a specific client be subject to access mediation. Also, the identity of the client performing a given operation must be made available to the item manager for informational or logging purposes (e.g., to keep track of the identity of the creator of a file, or of the client currently holding a global lock, etc.). This identity may consist of various elements of information, such as an authentication identifier (user id) or a process identifier, if there is a need to distinguish between multiple clients operating under the same authentication.

Although most of these functions could be provided at other levels of the system, integrating them with the central invocation facility seems to offer the best potential for efficiency, simplicity, and uniformity. Furthermore, this centralization and uniformity themselves facilitate the verification of the system from a security standpoint.

The addition of these functions to the invocation facility introduces new security considerations in the design. Although the design of secure systems constitutes a field of its own and is outside the scope of this work, the emulation framework must still satisfy common security requirements. In particular, it must not be possible for unauthorized users to intercept invocations performed by other clients or to forge invocations such that they appear to be made on behalf of different clients.

Finally, the definition of access mediation and client identification mechanisms rely on the specification of a model for protection and authentication that is responsible for determining the actual security policy that is to be implemented by the above facilities. The design of such a model is largely orthogonal to the design of the invocation facility itself.

3.2 Semantics of Client-Server Associations

3.2.1 Information exchanged between clients and servers

Any interaction or invocation between a client and a server must explicitly or implicitly be associated with the following information:

- A target item.
- The name or identification of an operation to perform.

- Any arguments explicitly for the requested operation.
- The identity of the client for authorization and information purposes.
- Information allowing the invocation facility to locate the item manager or server for the target item and to route the request to it.

Furthermore, items that are not stateless often need:

- The number of clients currently having access to the item and the identities of these clients.
- The types of operations that these clients intend to perform on the item, or *access mode*.

Obviously, all these categories of information are not necessary in all cases and the management of each element entails some overhead. However, for system uniformity, clarity, and flexibility when modifying or adding or services, it is important that the same model be used for all client-server interactions. The specification of a suitable collection of client information, along with appropriate mechanisms to manipulate it, is one of the key design areas for a common invocation facility.

3.2.2 User-visible abstraction – *bindings*

The need for the servers to keep track of attributes of their clients even when no invocation is in progress indicates that the invocation system must maintain some sort of persistent client-server association. Within this document, we refer to such associations as *bindings*. A binding represents the abstraction through which a client accesses an OS item, and through which any item's manager gains knowledge about any particular client or set of clients. Note: A binding is not an item itself, or even a language level object. It is an abstraction used to describe and discuss a specific logical connection between a client and an item on a server.

When a binding is established, the client indicates that it intends to access the item, along with the type of access that it intends to exercise. That binding is dissolved when the client relinquishes its access to the item. Between these two points in time, it must be possible for the item manager to query for the existence of the binding and to gain access to its relevant attributes. In addition, it must be possible for the item manager to be asynchronously notified when a binding is established or destroyed.

These semantics essentially correspond to the open/close semantics of many traditional systems. However, the use of these words may be misleading. There is no real "open" operation, since the mere fact that a client can designate or reference an item constitutes a form of binding that must be registered within the invocation system. Instead, a client can only gain access to an item and at the same time establish a binding, either by acquiring a reference to an existing item through some sort of lookup operation in a existing name space or by causing a new item to be created in some item manager.

Conversely, there is no need for an explicit "close" operation, since the client does not have to provide any explicit information or arguments at that time. In any case, a garbage-collection mechanism is needed to detect when a client has crashed and implicitly destroy its existing bindings. An explicit close operation should be used only if such an operation can be implemented more efficiently than the default garbage-collection mechanism, but it does not itself provide any additional functionality.

In practice, it is natural to represent a binding on the client-side with a *handle* that can be manipulated by the client and used to request invocations on the item. It is also natural to associate this handle with the location information needed to route invocations to the correct item in the correct server or item manager.

In addition to all the elements discussed above, the last category of information that might belong with a binding is the identity of the client itself. This identity may sometimes be required by the item manager as a static attribute of the binding, and is more often required dynamically when processing an invocation. At least part of this client identity must consist of secure authentication information. Consequently, establishing this information and transferring it between client and server may be expensive. Therefore, it is advantageous to attach this information directly to the binding and not to individual invocations. The item manager may store the information when the binding is established, thus amortizing the cost of manipulating across further invocations, with essentially no impact on the semantics and usefulness of the invocation facility.

In summary, a binding consists of: an access mode, location information for the communication subsystem, a handle, and a client identity.

3.2.3 Transfer of bindings and binding inheritance

Many bindings come into existence not as a result of an explicit operation performed by the client, but instead are inherited as a side-effect of a process creation (e.g., UNIX *fork()*, etc.).

However, setting up a new binding in a client is a complex operation. The client identity must be established and verified, the item manager must be notified of the new binding, a client-side handle must be created, location information must be maintained or updated, etc. Creating new bindings at fork time would mean that a new child process would explicitly register itself with each item manager for each item for which it needs a binding. Furthermore, since there is by definition no way for a process to reference an item without already possessing a valid binding (both the location information and the security aspects of the access to the item are themselves part of the binding), the parent process and each manager would need to explicitly collaborate to determine the list of items and bindings that should be transferred to the child process.

Obviously, this scenario is quite expensive. The cost to create a new binding is essentially inherent to the model chosen for the system and the requirements that justify it and cannot easily be reduced in the general case. However, the model can be modified to use memory inheritance to logically copy the binding when the child task is created. This scheme is possible at the expense of a small loss of generality in the semantics associated with bindings.

Representation of handles

First, the handle and location information for the binding, which are stored in the client's address space, must be of a form such that they can be inherited through the Mach *task_create()* operation. Although this requirement is not immediately met for the most obvious implementation, it does not constitute a serious difficulty. This matter will be discussed further in section 4.2.

Undifferentiated processes

Second, there is no general way for an item manager to securely differentiate between the parent and the child process, except through some form of secure registration with the item manager, which is precisely what we are trying to eliminate. Under a strict interpretation of the requirements, this problem invalidates the inheritance scheme, since a binding is supposed to provide a complete and secure identification of the client. However, if we subdivide this client identity between a *user identity* and a *process identity*, a compromise is possible. The user identity corresponds to a logical user of the system, typically identified through some sort of authentication mechanism at login time. In turn, each user may control several processes, each of which has a different process identity. The compromise consists in relaxing the security requirement for the process identity alone.

The user identity is usually the cornerstone of the protection system, and must be handled securely. But for the large majority of process creations, the user identity of the child is the same as that of the parent, so that simple inheritance presents no problem. In fact, such inheritance is desirable to avoid the need for a new interactions with the authentication facility at each process creation. In the relatively uncommon cases where a user identity must change, some re-registration with the authentication service and the various item managers seems to be unavoidable

On the other hand, for many target systems (such as most variants of UNIX), the process identity is not part of the protection model. In such a case it is acceptable to dissociate this information from the binding. This simplification may make it possible for one process to masquerade as another, but not to usurp different user privileges, so this solution does not compromise system security.

Immutable bindings

Third, if an item manager cannot distinguish between several clients belonging to a single client family, it is not acceptable for one such client to unilaterally change some attributes of the binding (access mode, identity, etc.), thereby affecting the operation of the other clients without their control or knowledge. The simplest solution to this problem is to make all sensitive attributes of the binding immutable, that is, they are established atomically when the binding is created and can never be modified afterwards. This approach may necessitate the creation of additional bindings with different sets of attributes in some operational situations, but in practice it appears that the need for such dynamic changes of attributes is sufficiently infrequent to justify this restriction.

No count of clients

Finally, we must give up on keeping an accurate count of the number of outstanding clients for a given item, since keeping such a count would be equivalent to keeping track of all process creation and other transfer operations affecting each item. It appears that in all practical cases, the system only requires that item managers be aware of the *existence* of clients with a particular identity and access mode, not of the actual number of these clients. Furthermore, Mach provides an efficient garbage-collection facility (see 4.3) that can detect the presence or absence of clients, but not their number. In view of these considerations, such a restriction in semantics also seems acceptable.

4 A Design for the Model and Objectives

4.1 Programming Aspects

4.1.1 Representing and producing invocations

The first basic programming issue with respect to programming in the context of the client-server interaction facility is how the invocations themselves should be presented. The chosen mechanism is to represent item invocations as standard C++ language method invocations. It is the responsibility of the remote invocation system to make such method invocations appear as if they were local to the current executable when they are not.

The Mach RPC stub generator named Mig is inappropriate for this function. Instead, Mach-US has a remote invocation package that is better suited to the object-oriented view taken in the system, and that provides additional flexibility for various extensions in functionality. This package can be subdivided in three parts: a run-time system that transports invocations and item references between client and server, a set of client-side objects that interface between the clients and the run-time system, and a set of server-side objects that interface between the run-time system and the code implementing each item in the item manager. The run-time system itself is not directly visible to the programmers of any emulation system; its most important features will be discussed in section 4.2. The other two aspects bear directly upon the view of the emulation framework presented to the programmers of any emulation system, and are discussed below. More details about this package can also be found in [Guedes&Julin91].

4.1.2 Client view — *proxies*

Each binding on the client side is represented with a special mechanism called a *proxy object* or simply *proxy*[Shapiro86]. 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. 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 application 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 run-time system. 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 service layer typically implemented by the collection of servers. 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.

In this context, the handles referred to above, that are used to designate individual bindings on the client side, correspond simply to the address of specific proxies. The memory allocated for such proxies can naturally be inherited across a *task_create()* operation and retain the same address in the child address space, so that the inheritance of bindings can be handled without any explicit intervention or modification in the client code. However, the proxies themselves may contain state (such as port-rights) that is not automatically inherited across *task_create()*. We describe a mechanism to deal with this question later in this section.

Finally, the set of methods exported by a proxy need not be the same as the set of method invocations or messages exchanged between a proxy and its corresponding item manager. Only the first set of methods are visible to clients of the system and constitute the actual exported item interface. This provides another useful level of indirection between clients and servers that can be put to use to implement any number of optimizations. Furthermore, the specification of the second set of methods is purely a private matter for the designer of each individual server, since each server must also supply its own proxies. Therefore, this interface need not be standardized in any way.

4.1.3 Server view — *agents* and *agencies*

For the simple purpose of transferring invocations, no special user-visible mechanism is needed on the server side. The run-time system can take the full responsibility for calling the target function of method directly, without need for any proxy-like stub. The only requirement is that the address of all remotely accessible functions be made available to the remote invocation system. This is accomplished with a set of special compile- and initialization-time declarations (see [Guedes&Julin91]).

Making binding information available to item managers

First, we must decide how the information associated with a binding is made available to the code implementing the target function or method. On the client-side, this information is implicitly represented by the address that the client uses as a target for invocation, corresponding to the address of the proxy object. But on the server side, the situation is reversed: the invocation is received on a binding handled internally by the remote invocation system, and forwarded to an object representing an item that is the same for all bindings associated with the same item.

This issue can be resolved in two major ways: add an explicit argument to each remote call to specify the binding information, or make this information available implicitly through some other language-level mechanism such as thread-specific data or a global variable.

The use of an explicit credentials argument is certainly simple, and has been adopted in several existing systems, such as the *vnode interface*[Kleiman86]. However, such a solution presents many awkward properties. Since it is never clear where the access mediation will be needed, the extra argument tends to percolate to many lower-level functions in addition to the main exported remote functions. This may lead to increased overhead simply for the propagation of the access mediation information. Conversely, the use of this extra argument is not uniform: many internal functions do not require it at all, and it is not clear if they should have a dummy argument. In particular, this information cannot be specified on the client side, since by its very definition it must be supplied by the (secure) invocation system. But removing the extra argument on the client-side destroys the symmetry between client- and server-side interfaces for the same operations. Similarly, not specifying the extra argument on purely local or internal functions (where there may not be a meaningful value for it at all) destroys the transparency between local and remote functions.

In comparison, the implicit specification of the mediation information seems much more attractive, and has therefore been adopted for the emulation framework. Since servers are in general multi-threaded, a specific key for per-thread data[PThreads93] is allocated to hold a pointer to this information for all threads in a server operating on behalf of a client. This "mediation pointer" is established transparently by the run-time system, and can be freely accessed by any code module that requests it. Given current implementations of such per-thread data pointers, it appears that the small possible increase of overhead for this type of access compared to the use of explicit arguments is not significant, particularly in view of the costs for copying this argument on potentially many stack frames.

Representation of binding information

Second, the binding information is represented by a mediation object or *agent*. This object is used as a filter between the remote invocation system and the actual code implementing each item. The client-server interaction mechanism is organized in such a way that invocations on each binding are dispatched to an agent object corresponding to the binding instead of being dispatched directly to the item. The agent verifies that a given invocation is allowed under the current access mediation policies, updates the per-thread mediation pointer and forwards the invocation to the actual item.

An advantage of the use of agent objects in this way is that the same implementation of an agent can often be used for many different types of items, using a table-driven mechanism to apply the mediation for specific invocations. In this way, the access mediation code can be cleanly separated from the implementation of the item themselves, thereby providing additional confidence in the correctness of the system. On the other hand, this approach creates some other difficulties with respect to the implementation of items. First, the concept of a pointer to an item within a server becomes somewhat muddled: is it a pointer to the item itself, or to one of its agents? Second, when code internal to an item must invoke another primitive of the same item, should it again go through the current agent, or should it operate on the item directly? In practice, this complexity is a cause for concern, but it seems to represent complexity that would have occurred independently of the agent model.

Keeping track of the active bindings

Finally, to complete the functional model defined above, it must be possible for an agency to keep track of the set of active agents or bindings that currently have access to it. In the default implementation, this is realized by having each agent explicitly register and un-register itself with its agency when that agent is created or destroyed. All that such a registration or un-registration call needs to represent the information associated with the agent is the per-thread mediation pointer. However, the current specification of these calls contains an explicit representation of the access rights associated with the agent. This simplifies the implementations of agencies that must keep track of the current maximum active access but do not care about the set of active credentials; this is by far the most common case.

4.2 Underlying Mechanisms for Communication

As discussed above, proxies provide a convenient indirection between clients and the remote invocation systems and constitute the only abstraction that is visible to those clients. Underneath this level, any number of communication mechanisms may be used. However, unless the proxy and the item manager reside in the same address space, any acceptable communication mechanism must provide some minimal guarantees with respect to preserving the access mediation semantics defined for bindings. In this view, it is useful for the complete system specification to define at least one standard general-purpose communication mechanism that meets all the requirements outlined above.

The natural approach to implement the low-level communication aspect of client-server interactions is to rely on the Mach IPC system: clients hold send-rights to ports and item managers hold receive rights. By virtue of its specification based on secure port capabilities, the Mach IPC facility is adequate both as a mechanism for the transport of raw data between address spaces and for the secure representation of object references [Draves90]. Clients cannot forge send-rights, and therefore cannot acquire access capabilities that have not been explicitly given to them by an item manager. Conversely, by using different ports for communication with different clients, an item manager can securely distinguish between invocations requests from these different clients. Furthermore, when forking, these send-rights can be easily given to a child by its parent via the Mach *port_insert_right()* operation. They can also be transferred via Mach IPC to any process, child or otherwise. NetIPC software from the xKernel project at the University of Arizona [Orman+93] makes the port semantics available among loosely coupled hosts.

4.2.1 Ports Identify: Servers, Clients, and Items

Obviously, a send-right for some server port must be used by a proxy for shipping invocations to its server. However, there are two other problems to be addressed at this point: how to securely identify a given binding to a server and how to identify an individual item being invoked against.

This general scheme admits three variations:

- The "*port-per-server*" option would use one server port to receive all of a given servers invocation requests from all of its clients. Then, in each request, a *"credentials port"* would be sent to identify the client/user and ensure security. Also, a "binding handle" would be sent to identify individual bindings within the server. This handle can be a simple unsecure number and could be verified by the server in the context of the credentials port.

- The "*port-per-pair*" option would use a port for each client-server pair. Again, a binding handle would be sent to identify individual bindings within that server. This handle is secure since it can only be received via the client-server port.

- Finally, the "*port-per-binding*" option would represent each binding with a different port.

Each of these variations can provide the required functionality and has been used in various message-based systems. The choice among them is governed by considerations of transfer of bindings, overall system complexity, and speed.

Port-per-server

While this approach is arguably the natural way programmers like to think of client-server interactions, it has some problems. To transfer a binding to another client, it is necessary either to transfer the clients credentials port or to cause the target client to re-acquire the binding with its own credentials port. This may be no problem for inheritance where the child could have the same credentials port as the parent. But transferring a credentials port to a third-party task would make it easy for any binding guarded by the credentials port to be spoofed by the other task. Restricting the credentials port to guard fewer bindings quickly deteriorates into something with functionality approaching port-per-pair or port-per-binding. That functionality would have much more complex semantics invalidating the simple method call model for invocation transparency. It also raises questions of how to coordinate, among servers and clients, the creation of such a restricted credentials port. Alternately, re-acquisition of the binding with the target tasks credentials may be impossible, thus restricting the way items may be shared. Third-party sharing of bindings without re-acquisition is almost required for UNIX *setuid exec* and is a generally useful mechanism.

A serious problem with using a credentials port to supply identity guarantees with each message is speed. In any optimized system or kernel, transfer of regular data will be faster than transfer of secure data. Hence, shipping such a port in every message would be costly. It is better for a server to get its client identity assurances based upon the port from which it receives a message, not on what is in that message. In that way, the security costs are incurred only when a binding is created using new credentials.

Port-per-pair

This approach avoids the credentials port transfer costs. Yet it suffers from some problems when transferring bindings between processes. As in the port-per-server case, a client cannot safely ship a proxy's send-right for the server to a third-party process. That process would be able to access any of the first client's bindings to the server in question. Port-per-pair also cannot handle third-party sharing.

There is also a problem with proxy inheritance. If a parent does not share its server-ports with its child, then that child would need to re-acquire all of its bindings with each of its servers. Yet sharing them leaves all of the parent's bindings available to the child, not just the one the parent wants to share, even ones that do not exist when the child is created.

Port-per-binding

This approach clearly has the best granularity. A binding can be shared with a child or third-party without any possibility of spoofing another binding. The port which represents the binding represents exactly one item, making request multiplexing simple. It also has some garbage-collection advantages (see section 4.3.) While it is true that there are many more ports needed for this method, ports themselves are fairly inexpensive in Mach.

Because of the above considerations, Mach-US uses a port-per-binding implementation.

The structure of the remote invocation system is shown in figure 1.

4.3 Garbage-Collection

4.3.1 Background

Inside a single address space, the most practical mechanism available for garbage-collection with the current state of language facilities under Mach is reference counts[Draves90]. Such a mechanism has amply proved its adequacy with other Mach applications and even within the Mach kernel itself. Given that the primary goal of the present project is to construct a framework for emulation in a multi-task environment, and not to advance the state of language research, we have decided to use this same technique within the emulation framework.

The invocation facility is responsible for extending the concept of reference counts across client-server boundaries. In other words, if a proxy in a client's address space has access to an item through an agent, the invocation facility must implement a *logical reference* from that proxy to that agent. In the context of the local reference counts used for intra-server implementation, this logical reference corresponds to a local reference held by the remote invocation system for the agent, to be released when that agent is no longer accessible from the outside through the remote invocation system. In turn, the agent normally holds a local reference to the agency, thereby ensuring that an item remains accessible inside the server as long as it has active bindings. Since we are not distinguishing between multiple

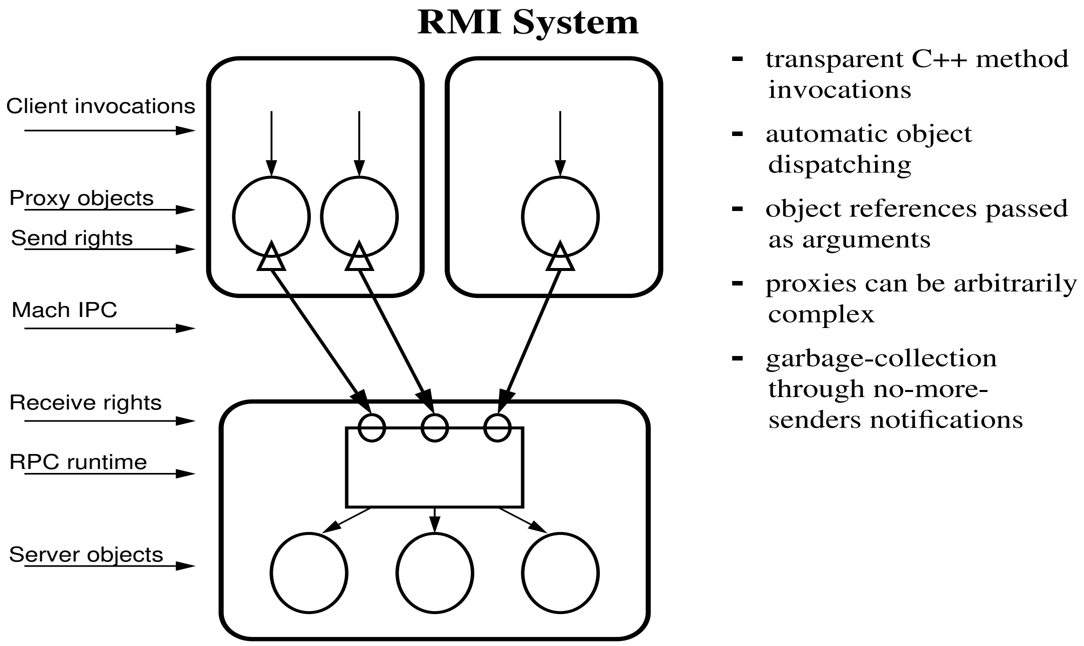

Figure 1: Remote Method Invocation System

clients of the same agent, the remote invocation system holds only one reference to the agent representing simply the existence of clients. It does not maintain an actual count of those clients. Also, in this context, clients are not necessarily limited to emulated processes: it is possible for one server to be a client of another server.

4.3.2 Handling the disappearance of bindings

The garbage-collection facility must deal with two different types of events: a normal explicit release of a binding by a client, and a crash of a client process in which all active bindings are implicitly released. Client crashes present the greatest challenge, and a solution for this question is necessarily general enough to also handle explicit release operations. Additional mechanisms to handle explicit releases need to be introduced only if the general solution selected is too expensive for this other common case.

Any facility to handle garbage-collection of bindings when a client process crashes may be classified according to how it fulfills two independent functions: how it detects a crash and, once a crash has been detected, how it establishes the list of bindings that are being destroyed.

Detecting Crashes

There are three main approaches in a Mach environment for user-level server to monitor one of its clients (implemented by a separate Mach task) and detect a crash of this client:

polling: The server periodically exchanges messages with its clients to determine the continued validity of their bindings. Though logically simple, this method generates message traffic and scheduling load even when the system is not performing any useful work. Furthermore, to service these messages each client must contain a special thread/event-handler.

port-death notification: The server holds a *send-right* for a port for which the client holds the *receive-right* and requests that the Mach kernel generate a port-death notification when that *receive-right* is destroyed. Such a destruction of the *receive-right* indicates that the client has crashed or has explicitly deallocated the port. This method avoids the problems associated with polling by using Mach IPC features to learn of client death, yet it has some drawbacks of its own. In addition to the added port manipulation overhead, a fatal problem is that receive-rights in Mach IPC can only be held by one process. Hence this method of binding death detection would force re-registration of bindings after forking, an option discarded as too costly in section 3.2.3

no-more-senders notification: The server holds a *receive-right* for a port for which one or more clients hold *send-rights* and requests that the Mach kernel generate a no-more-senders notification when all extant *send-rights* are destroyed. Such a notification can be interpreted to signify that all clients having access to that port have either crashed or have deallocated their *send-right*. It is not possible to detect changes in one particular client if several clients hold rights for the same port.

Chosen Mechanism

Mach-US uses no-more-senders notification: it does not require any active participation from the clients. It only comes into action when *all* outstanding send-rights have disappeared instead of detecting each the destruction of each individual send-right, and the kernel-generated notification is typically faster than a complete user-level message exchange. Unlike the situation described for the use of port-death notifications, every client and server pair always possesses a port with the required distribution of rights: the port by which the client sends requests to the server.

This strategy also naturally handles the explicit release of individual bindings, and can be expected to be more economical than most specialized user-level message protocols that fulfill the same purpose. This is done by taking advantage of fact that there is exactly one send-right per binding.

There is one drawback: it is possible for a large number of no-more-senders notifications to be generated at once when a client crashes and releases all of its bindings. This problem is mitigated by the fact that each notification is only generated when no other client holds a binding associated with the same port, but it is not always negligible.

4.4 Interrupting Pending Invocations

4.4.1 Background

Like the garbage-collection mechanism used for the implementation of the complete emulation framework, the mechanism used to interrupt pending operations is not confined within the remote invocation facility. It may be necessary to interrupt not only remote invocations, but also many other types of local computations or blocking points. Moreover, once an operation is interrupted, various language-level mechanisms are required to specify and control which actions the interrupted programs must take, how to respond to the interrupt, how to continue after it, etc. All these issues are part of the overall programming model for the implementation of clients and servers.

In this context, the remote invocation facility is responsible for providing a mechanism allowing the overall language-level interrupt facility to interrupt pending remote invocations.

4.4.2 Basic functions for interrupting remote invocations

The interrupt mechanism provided by the remote invocation facility must fulfill three basic functions:

- Arrange for the client code to resume control when a pending invocation is interrupted. This is the primary function of the interrupt facility.
- Notify the server that the invocation has been interrupted, to give it an opportunity to abort long-running computations or perform other appropriate cleanup functions. Different servers may or may not choose to act on such a notification, but such a decision should be independent of the client.
- Ensure that any reply generated by the server does not get lost by being sent to a client that is no longer prepared to handle it. There is an inherent race condition in this requirement, in that the reply may be sent back to the client at the very same instant that the client is sending an interrupt request to the server. This race is important because many operations in the emulation framework cannot easily be made idempotent. For example, it is in general not acceptable to lose incoming data on a sequential data stream when a “read” operation must be interrupted. Similarly, it is certainly not acceptable to lose incoming connection requests on a transport endpoint set up to accept such requests.

Each of these functions contributes a few considerations for the design of the remote invocation interrupt mechanism.

4.4.3 Waking the client

From the client's perspective, a thread blocked in a pending invocation is actually blocked inside the kernel `mach_msg()` primitive, waiting for a reply message to be received. It is unblocked either by the reception of a message or by an explicit Mach `thread_abort()` operation. In terms of waking the client, there are two main questions: when and how.

There is a desire to wake the client as soon as is possible, yet for many invocations, it is important to know if the invocation completed or was interrupted. To get that information, it is necessary to notify the server that it is being interrupted and wait for an acknowledgement of that interrupt or a reply to the invocation.

On the other hand, such waiting assumes that the server reacts promptly to the notification. This places each client at the mercy of any server with which it may be in contact, having no obvious way to interrupt a client waiting on an unresponsive server. Unless the `mach_msg()` operation on the client was initiated with an explicit time-out (which cannot normally be the case), the client will simply remain blocked forever. Such a limitation is undesirable in a system where one of the goals is to make it easy for individual users to add arbitrary servers. Furthermore, this scheme forces every server to handle interrupt notifications, even if it would not make any use of such a notification in the context of its own internal operation.

Therefore, the chosen solution is to send an interrupt notification and timeout if a reply/acknowledgment is not received promptly. A reply/acknowledgement would cause the pending `mach_msg()` to complete naturally. If the timeout occurs first, the only remaining option is to `thread_abort()` the `mach_msg()` and react as if the initial invocation failed.

4.4.4 Notifying the server

A server is typically handling many concurrent invocations from many different clients. The main issue, when notifying a server that a pending invocation is being interrupted, is how to identify the particular invocation (and the corresponding client) to the server. There are two obvious choices: to deallocate the reply port (using port-death notification to tell the server of the interruption) or to send an explicit interrupt message on the binding in question.

The problem with the logically simple act of deallocating the reply port is that it makes it impossible to avoid losing a normal reply from the server should one have been calculated at the wrong time. This problem is most severe when the client and server are on different nodes. Even if some way were found to avoid losing a reply, it would still be required to find some way to undo the actions that led to its creation. This would be necessary for every possible reply.

The use of an explicit interrupt message is considerably more complicated and expensive, but it avoids the difficulties seen with port deallocation. How it avoids these problems is described in section 4.4.5.

Identifying an invocation in an interrupt message

An interrupt message is not sufficient to uniquely identify one particular invocation, since several invocations and several clients may be active on the same binding at the same time. The number of such active invocations is usually small, and certainly much smaller than the total number of invocations processed by the whole server on all bindings for its items, but it is not always equal to one. To solve this problem, one can either tag each invocation for possible later interruption, or find some acceptable way to relax the interrupt semantics.

Choosing an invocation tag is not easy. The most obvious choice would be to ship another right to the reply port in the interrupt message. For speed reasons, these reply port rights must be send-once-rights. Yet, in Mach, two send-once-rights for the same port will not have the same values and there is no way to compare them. Sending an additional send-right as a tag with each invocation would also significantly slow down communication (see section 4.2.1). Just using a numeric tag would not securely prevent other active invocations on a binding from being interrupted. This is probably acceptable given the restrictions on how a given binding is shared. But the problem of assigning a unique identifier across all holders of a binding is not trivial (especially if clients reside on separate nodes).

An alternative to tags is to relax the interrupt semantics to specify that when an interrupt message is received by the server, *all* pending invocations on the associated binding should be interrupted. This solution obviously provides the minimal overhead for the normal (non-interrupt) case, since there is no tag whatsoever. It also does not affect the security of the mechanism any more than the use of numerical tags, since interrupts are still limited to a single binding family. However, it does require that each client be prepared for any invocation to be interrupted unnecessarily, and the the client must restart such interrupted invocations, as appropriate.

This model of interaction is similar to the one defined for the semantics of condition variables in many threads packages [PThreads93], where the implementation allows threads waiting on the condition to wake-up even if the condition has not been signalled. In practice, such a complication does not create significant difficulties. The nature of most invocations and syscalls is such that the system must be prepared to restart them after an interrupt occurs anyway, whether such an interrupt was legitimate or not. Moreover, interrupts are rare in normal operation, and the overhead and complexity introduced by the restart requirement can be almost completely restricted to this special case and avoided in the common, non-interrupt case.

According to these various considerations, the no-tag solution appears to be the most appropriate and is the one in use.

4.4.5 Interrupt processing and reply

Atomic success or failure

The obvious solution to the race condition presented by the generation of an invocation reply is to allow interrupts to fail or succeed atomically. If a reply is generated, the invocation should simply complete normally and return whatever information was provided with that reply. If an invocation reply is not generated by the time the interrupt is processed, the invocation system should return a special interrupt completion code with a guarantee that no further reply will be generated or received for this invocation. These semantics are not ideal in that they may force every client of the invocation system to handle one, possibly undesired, normal invocation reply, but they are sufficient to avoid any lost information or uncertainty regarding the status of an invocation.

How is this done

Upon receiving an interrupt message, the server checks if the invocation is still in progress. If it is not, then the reply message is already on its way, and the interrupt should simply be ignored. Otherwise, the server should arrange for the invocation to be aborted as soon as is possible. If and when it is aborted, a special reply message is sent to the invocations reply port. It acknowledges the interrupt in lieu of the normal reply. If the invocation completes before it can be interrupted, then a normal reply occurs. Note: Mach-US supplies an extensive, easy to use language level mechanism for servers (or clients) to describe when and how they can be interrupted. This mechanism has been simply applied to pre-existing code. Its details are not important to this discussion, and are outside the scope of this paper.

As a further optimization, the invocation system knows that some methods represent simple calculations that will never need to wait. For these methods, the interrupt systems overhead is not incurred and the method is always permitted to run to completion.

Variations on this scheme would include explicit acknowledgments of every interrupt or automatic destruction of truly unwanted replies on the client side. But these variations simply add complexity for every interrupt, without adding any significant functionality or robustness to the scheme. Moreover, they raise additional questions with respect to the interruption of multiple pending invocations with a single interrupt message. Accordingly, neither has been adopted.

5 Status/Numbers

Note to reviewers: Final paper copy will have similar, more recent numbers.

This section examines some measurements of Mach-US and its UNIX emulation. While these measurements include more than just the client server interaction times, they are supplied to demonstrate that Mach-US, using its complex binding and invocation system, has similar end-to-end performance to other UNIXs.

The timings given compare the Mach-US UNIX emulation with those benchmarks for the Mach3.0 single server (Mach-UX) system [Golub+93]. This data is not intended to be a source of detailed analysis of the performance of Mach-US.

- The *compile-test* is a Mach classic that compiles nine medium-small programs. Running the compile-test (without the load phase because the results existed locally) we see the following UNIX occurrences: 9 compilations, 48 emulated processes (9 * (cc + cpp + ccom + as + ld)) + (2 * date) + csh, 27 files created, system calls: 9290. Yet because of the intelligent proxies and emulation library, there were only 2430 outgoing RMI's. Running the compile-test again (with the load phase) on a HP 386/25c Vectra with 16Meg, takes 27 seconds for Mach-US versus 21 seconds for Mach-UX (MK78 - UX39/US48).

- The *parallel-compile-test* is a related test that runs any number of compile-tests at the same time. On a Sequent Symmetry hardware with 20 i386/16Mhz processors and 32Meg of memory, with 10 compile-tests, we see Mach-UX(UX38/MK78) = 146 seconds and Mach-US(US48/MK78) = 139 seconds. Mach-US is slightly faster.

- *FTP* is a very high level test of the Mach-US network service. FTP is run on machine1, a i486/50Mhz with 24Meg of memory. FTPD is run on machine2, a i486/25Mhz 16Meg of memory (UX39/MK78). The software running on machine2 does not change during the test. With machine1 running Mach-US(US49/MK83) doing either a *get* or a *put* of a 750Kbyte file from/to */dev/null*, the throughput is 210KBytes/sec. Using Mach-UX(UX39/MK83), FTP puts run at 120Kbytes/sec and gets run at 150Kbytes/sec.

- Comparing *remote-invocation* times on the Vectra we find that a raw fast path CMU Mig invocation takes 340usec, a simple Mach-US RMI (without interrupts or access control) takes 653usec, a complex RMI (passing and returning an object reference) takes 1228usec. Adding interruption adds another 390usec to each (US44/MK65).

The compile-test and other straight-line timings have shown that Mach-US generally runs 10-25 percent slower than Mach-UX for high level benchmarks.

The UNIX emulation portion of the system is largely un-tuned. This is because the chronically limited Mach-US resources have been concentrated on improving its functionality and robustness. Despite this, some macro-benchmarks show results in the same range or better than those obtained with Mach-UX. There are a number of known areas where significant performance gains can be achieved through straightforward engineering efforts.

This straight-line performance gap is significant, yet Mach-US achieves the level of performance that it does in a more complex, and much richer environment, than the systems to which it is being compared. These results are sufficient to show that performance should not be the issue when it comes to judging the suitability of this system and its client-server support for practical applications.

6 Conclusions

This paper has described issues of binding maintenance in an object-oriented multi-server operating system and solutions to those problems in a Mach3.0 kernel environment.

Mach-US has implemented a mechanism for client-server interaction that is sufficiently rich to support an object based operating system structured as a collection of operating system services, with OS services partly implemented in each client process. It has also demonstrated an acceptable level of performance for this implementation even with its increased flexibility.

This research included how to achieve:

- Remote method invocations while hiding server boundaries.
- Transparency of server communication mechanism and location.
- Safe transfer of bindings between clients without re-registration.
- Flexible garbage collection of deallocated or crashed bindings.
- A server-friendly interruption mechanism.

The lessons learned are applicable to multi-server OS design and should be useful to object based systems that require similar features in a "micro" kernel environment.

References

[Draves90] Richard P. Draves. A Revised IPC Interface *Usenix Mach Symposium Proceedings*, Oct. 1990.

[Golub+93] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. *USENIX Summer 1990 Conference Proceedings*, June 1990.

[Guedes&Julin91] Paulo Guedes and Daniel Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. *Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems*, October 1991.

[Hutchinson&Peterson91] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An architecture for implementing network protocols. In *IEEE Transactions on Software Engineering* 17(1):64-76,Jan.1991.

[Julin+91] Daniel P. Julin, Jonathan C. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized Emulation Services for Mach3.0: Overview, Experiences and Current Status *Usenix Mach Symposium Proceedings*, November 1991.

[Khalidi&Nelson93] Y.A. Khalidi and M.N. Nelson. An implementation of UNIX on an Object-oriented Operating System. *USENIX Winter 1993 Conference Proceedings*, January 1993.

[Kleiman86] Kleiman, S.R. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *USENIX Summer 1986 Conference Proceedings*, 1986.

[Orman+93] H. Orman, E. Menze III, S. O'Malley, and L. Peterson. A Fast and General Implementation of Mach IPC in a Network. *Usenix Mach Symposium Proceedings*, April 1993.

[PThreads93] IEEE Standard Portable Operating System Interface - Threads Extension. IEEE Std. P1003.4 Draft 8, October 1993.

[Phelan+93] James M. Phelan, James W. Arendt, and Gary R. Ormsby. An OS/2 Personality on Mach. *Usenix Mach Symposium Proceedings*, April 1993.

[Shapiro86] 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.

[Zajcew+93] R. Zajcew, P.Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, Netterwala. An OSF/1 Unix for Massively Parallel Multicomputers. *USENIX Winter 1993 Conference Proceedings*, January 1993.