A Revised IPC Interface
by
Richard Draves

This research supported in part by the Fannie and John Hertz Foundation.

1. Introduction

The Mach 3.0 IPC facility efficiently supports many different styles of communication, including server-client remote procedure calls, distributed object-oriented programming, and streams. As a building block, the interface provides a general message primitive. The message primitive operates on capabilities for communication ports. The interface remedies numerous problems that were observed in the older Mach 2.5 interface. The implementation provides good performance, running 30-40% faster than Mach 2.5 with data structures 50% smaller. The Mach 3.0 implementation provides backwards-compatibility support for the older interface.

2. Interface Description

The Mach IPC facility provides message-oriented, capability-based interprocess communication. The interface supports several different styles of interaction, including remote procedure calls, object-oriented distributed programming, and streams. The implementation makes use of the VM system to efficiently transfer large amounts of data using a copy-on-write optimization. The Mach IPC interface allows intermediaries to transparently extend the IPC facility. A user-level server extends Mach IPC across networks.

2.1. Remote Procedure Calls

The IPC interface supports remote procedure calls as an important special case operation. The `mach_msg` system call allows tasks to combine send and receive operations. The interface provides send-once rights for ports, designed for use as reply port rights. A notification mechanism allows clients and servers to detect and recover from aborted RPCs. The MiG stub generator aids the construction of RPC interfaces.

At the message level, an RPC consists of a request message sent from a client task to a server task and then a reply message sent from the server task to the client. The server task holds a receive right for the service port, and the client task hold a send right for the port. Many client tasks may have send rights for a single port. The `mach_msg` system call satisfies the needs of both client and server tasks with a single system call. Client tasks use `mach_msg` to send a request message and receive the reply. Server tasks use `mach_msg` to send a reply message and receive the next request.

The interface provides send-once rights, for use as reply ports. The client provides the server with a send-once right for a reply port, to which the server sends the reply. The request message carries the send-once right. The message header contains a field for transmitting rights to a reply port. The holder of a send-once right can use it to send only one message, because the act of sending to a send-once right deletes it. When the client receives its reply message, it knows that the server retains no rights for the reply port.

Send-once rights never disappear; they always result in a message. When a task deallocates a send-once right, the kernel uses the send-once right to send a notification message. If a server dies while processing an RPC, the server's client can recover because it receives this notification message from the kernel.

The interface also provides dead names and dead-name notifications, which allow servers to detect the death of clients and abort computations. When a client dies, the kernel deallocates its port rights, including its reply port. The send-once right for the reply port, held by a server, becomes invalid and turns into a dead name. When the server tries to send its reply message to the dead name, the `mach_msg` call returns an error that informs the server of its client's death. If a server wishes to abort computations on behalf of dead clients, it can request dead-name notifications for the send-once right reply ports. Then when a client's send-once right becomes a dead name, the kernel will send a dead-name notification message to the server, informing it of the new dead name.

Mach provides an RPC stub generator, MiG, which simplifies the construction of distributed systems by implementing familiar procedure-call functionality. For the client, MiG produces stub procedures which pack a request message, use `mach_msg` to send the request and receive a reply message, and then unpack the reply. For the server, MiG produces stub procedures which unpack the request message, call the server's function, and then pack a reply message. MiG also provides servers with a standard function to receive requests and send replies using `mach_msg`. MiG allows clients to send asynchronous messages when they do not need a reply message.

2.2. Object-Oriented Servers

Mach IPC supports the object-oriented construction of distributed systems. In this model, a port represents an object, and messages sent to the port manipulate the object. Send rights for the object port act as opaque object pointers. Ports sets let a few threads serve requests for many objects. A no-senders mechanism allows object servers to garbage-collect unused objects. The MachObjects package simplifies the construction of object-oriented interfaces. The kernel interface represents a good example of an object-oriented interface.

The interface allows object-oriented clients and servers to use send rights as object pointers. Programs can compare send rights for equality because a task only holds one send right for any given port. If a task holds a send right for a port, and receives another send right for the same port in a message, then the kernel reuses the task's existing name for the send right.

Port sets let an object-oriented server manage hundreds or thousands of objects. A port set holds receive rights, and a receive operation on the port set returns the next message sent to any of the constituent ports. The received message specifies the port to which it was sent, so that the server knows what object to manipulate. A server with many objects must rely on port sets, because assigning a separate thread to every object would consume a prohibitive amount of space for stacks and other data structures.

The interface provides a no-senders detection mechanism. This mechanism allows the holder of a receive right to detect the absence of send rights for the receive right, and garbage-collect the receive right and the object it represents. The task holding a receive right can request a no-senders notification for the right, supplying a send-once right to which the kernel will direct the notification. Then when the last send right for the port disappears, the kernel uses the send-once right to send the task a no-senders notification.

The no-senders detection applies only to send rights, not send-once rights. The receive right holder does not need a notification to detect an absence of send-once rights for a port. Because only the receive right holder can create new send-once rights, and send-once rights do not disappear but always result in a message, the holder of a receive right can keep track of the number of send-once rights without further help from the kernel.

Because send-once rights do not affect no-senders detection, the no-senders notification for a port can be sent to the port itself. This significantly reduces the number of ports used by object-oriented servers, because they do not need to create a notify port for every object port. Clients holding send rights for the object port can not forge no-senders notifications, because the server can distinguish messages sent to send-once rights from messages sent to send rights.

The MachObjects package aids the construction of complex object-oriented interfaces. MachObjects provides functionality similar to Objective C. It allows the definition of classes, the inheritance of methods for handling request messages, and the delegation of operations to another object. Like MiG, it packs and unpacks messages for the application.

The kernel interface demonstrates object-oriented principles. Ports represent kernel objects like tasks and threads, and user programs can manipulate any task or thread for which they have send rights with RPC operations on the ports. This approach lets applications like the Mach 3.0 Unix emulation and debuggers manipulate other tasks, even tasks on other machines, through the standard kernel interface.

2.3. Streams

Because the IPC interface provides message-oriented communication, it directly supports stream protocols. The interface guarantees reliable message delivery. The interface also supplies flow control mechanisms for both readers and writers.

The IPC interface requires reliable, sequenced message delivery. The implementation must deliver every message sent to a port, unless the port is destroyed. Furthermore, the implementation must preserve the order of messages from a single source. The interface does not define the ordering of messages sent by different threads. Network message servers extend these semantic guarantees across networks.

A stream reader adjusts the message queue's length to perform flow control. The reader holds a receive right for the port implementing the stream. Queue of messages at the port may only grow to a limited size, controlled with the `mach_port_set_qlimit` call. By modifying the queue limit the reader controls the buffering in the stream.

The interface also offers stream writers a flow control mechanism. Normally when the writer exceeds the port's queue limit, the writer's send operation blocks until the reader reduces the size of the queue. Alternatively, the writer may request a msg-accepted notification. In this case, the kernel sends the writer a notification message to inform the writer that it can resume queuing messages. This option allows the writer to perform other work while waiting for the reader to catch up.

2.4. Copy-On-Write

The IPC implementation makes use of the VM system to efficiently transfer large amounts of data. A VM optimization, copy-on-write sharing, lazy-evaluates the copy operation. The implementation also allows a message sender to move data into a message, further reducing the transmission cost.

A message body can contain the address of a region in the sender's address space which should be transferred as part of the message, as an out-of-line memory region. When a task receives a message containing an out-of-line region, the region appears in an unused portion of the receiver's address space. Out-of-line data retains the normal copy semantics of message transmission. The sender and receiver can freely modify the out-of-line data. The implementation optimizes the transmission of out-of-line data so that sender and receiver share the physical pages of data copy-on-write, and no actual data copy occurs unless the sender or receiver modifies the out-of-line data. Regions of memory up to the size of a full address space may be sent in this manner.

A message sender can enable a further optimization when it does not wish to retain a copy of the memory region. The deallocate bit in the out-of-line data's type descriptor indicates that the memory region should be deallocated from the sending task. The VM system takes advantage of this information to disable copy-on-write, avoiding the cost of write-protecting the memory region. Instead of copying, the VM system moves the memory region directly from the sending task into the message.

2.5. Transparent Extensions

The IPC interface allows intermediate servers to transparently extend the IPC facility. User-level servers extend Mach IPC across networks; the Mach kernel only directly handles message transmission within a single machine. Programs can interpose on port rights, to transparently intercept message traffic.

Because the sender and the receiver of a message can not detect the presence of intermediaries, the IPC interface is transparent. Port rights carry no location information, so the sender of a message knows nothing about the true receiver. The interface does not give the receiver of a message the identity of the sender. The receiver only knows that the sender possesses a send or send-once right, because port rights are secure.

The user-level extension servers (network message servers, or netmsgservers) take advantage of the transparency and cooperate to forward messages across a network. When a remote task has send rights for a local port, it actually holds a send right for a proxy port on the remote machine. The remote netmsgserver holds the receive right for the proxy port, and the local netmsgserver holds a send right for the local port. When the remote task sends a message to its send right, the remote kernel delivers the message to the remote netmsgserver. The netmsgservers transfer the data over the network, and the local netmsgserver forwards the message to the local port using its send right.

The netmsgserver provides a simple name service which bootstraps this process. User tasks can register send rights with the local netmsgserver. The name service provides a lookup operation that a task can use to acquire send rights that were registered with netmsgservers elsewhere in the network. The lookup can be directed to a particular machine, or broadcast on the local network. The netmsgservers create proxy ports as needed to represent ports elsewhere in the network.

Netmsgservers can use encryption and authentication techniques to protect port capabilities across the network. They can encrypt message data. They make use of the type data in messages to convert between data formats. Implementation choices like level of security and network protocol are made at user-level, and a machine may support multiple extension servers offering different services.

Another application of transparency allows programs to intercept messages. For example, a debugger can monitor all message traffic sent through a send right. The debugger extracts the task's send right and inserts a send right for another port, owned by the debugger, using the `mach_port_extract_right` and `mach_port_insert_right` calls. Unbeknownst to the task, the debugger receives the messages sent by the task and forwards the messages to the extracted send right. Suspect programs may be run in a virtual mode, which intercepts for inspection all message traffic into and out of the task.

3. Interface Advantages

The Mach 3.0 IPC facility remedies several problems that were encountered when using the older Mach 2.5 interface to build complex multi-threaded applications. The old interface does not give complex applications the tools they need to manage their port rights correctly. Clients and servers written to the old interface face problems managing their reply ports. The old interface's design made it difficult for programs to use notifications effectively. Finally, the old interface does not allow programs to recover from some classes of errors. The new IPC interface overcomes these problems without unduly burdening the application programmer.

3.1. Port Right Management

The asynchronous nature of multi-threaded programs greatly complicates port right management. Multi-threaded Mach 2.5 applications commonly contain many subtle bugs because the old interface makes it difficult or impossible for these programs to manage their port rights correctly in the presence of port deallocation. Features introduced in the new interface, reference counts for port rights and dead names, allow multi-threaded programs to overcome these problems.

3.1.1. Reference Counts

Reference counts for send rights allow programs to deallocate send rights safely. For example, consider the operation of a typical client program. The client looks up a service port, acquires a send right for the port, uses the send right to make requests, and deallocates the send right. If several threads in the client use the service concurrently, then the reference count for the send right prevents the threads from incorrectly releasing the send right. When a second thread acquires the send right, the right's reference count is incremented. When a thread deallocates the send right, the reference count is decremented. The client task only loses the send right when the last thread deallocates its reference for the send right.

A library can not solve this problem for Mach 2.5 programs by maintaining user-level reference counts for send rights. Even assuming the mandatory use of a standard library that provides such a coordinating facility, the library may not have complete knowledge of the references for send rights. A task may contain separate binaries, each with its own libraries and data structures attempting to provide reference counts. The Mach 3.0 Unix emulation provides an example of this. The transparent emulation library is a separate binary loaded into the address space of Unix processes. The program and the transparent library can not coordinate their use of send rights because they do not share any data structures or code.

Even without separate binaries in a single task, a potential race condition prevents a standard library from coordinating send right references. A thread in the client task that receives a send right in a message immediately uses the library to register a reference for the send right. However, this leaves a window between when the thread receives the send right and when it registers it. In this window, another thread may release a reference for the send right. If it releases the last reference and the library deallocates the send right, then the first thread is deprived of the send right which it had just received.

The coordinating library avoid this race condition if it ensures that no receive operation is in progress when it deallocates a send right, perhaps by postponing deallocations. This solution eliminates the window, but it severely constrains the program. In many multi-threaded programs, most threads block in receive operations, so the coordinating library will never be able to deallocate send rights. In practice, most Mach 2.5 programs dodge the problem and never deallocate send rights.

3.1.2. Dead Names

Dead names give programs time to recognize the asynchronous deletion of a port right. When a port is destroyed, because its receive right is deallocated, the port's send rights become invalid. The new interface changes the invalid send rights into dead names. When a task tries to send a message to a dead name, the resulting error lets the task know that the port died and its send right is no longer valid. The task can then deallocate the dead name.

Like send rights, dead names have reference counts so that multi-threaded programs can safely manipulate them. When a send right with multiple references becomes invalid, the resulting dead name assumes the send right's references. As different components of the application notice the dead name and deallocate it, the dead name loses references. When the dead name loses its last reference the name becomes available for reuse by another port right.

A multi-threaded Mach 2.5 program is vulnerable to port name reuse when a port right becomes invalid. For example, after a thread in a client task looks up a service port and receives a send right for the port, the port may die. The Mach 2.5 implementation deletes port's invalid send rights immediately. Another thread in the client task can create or receive a port right which happens to reuse the name of the invalid send right. Then the first thread will inadvertently use the wrong port right when it tries to make a request. In practice, this problem does not afflict old programs because the Mach 2.5 implementation allocates port names sequentially, so that names are very rarely reused. An implementation that picked the smallest name not currently used would break many Mach 2.5 programs.

3.2. Reply Port Management

Mach 2.5 clients and servers face several reply port management problems stemming from the old interface's use of send rights for reply ports. RPCs do not leave a client failure-isolated from the server. Servers must receive and discard many unwanted port-deleted notifications. Finally, clients can not recover when a server dies during an RPC. The new interface provides send-once right reply ports, which solve these problems.

The Mach 2.5 RPC model does not leave clients isolated from bugs or faults in the server after an RPC finishes. A faulty server can precipitate mysterious failures in any client which has used it. This negates some of the failure-isolation advantages gained by putting the client and server in separate address spaces. Because the client gives the server a send right for the reply port, the server can potentially send many messages to the reply port. Normally, the server sends only the reply message and then retains the reply port as an unused send right. However, clients can not count on this behavior. A buggy server may send additional messages to a reply port, and these messages will interfere with later RPCs made using the reply port. The only recourse for the client is to use a separate pool of reply ports, with one reply port per thread, for every service that the client uses. This alternative is prohibitively expensive, because it greatly increases the number of ports consumed by the client and because selecting the correct reply port for an RPC increases the RPC cost.

From a Mach 2.5 server's viewpoint, the reply port send rights it retains clutter its port name space. After a server sends its reply, it can not deallocate the send right reply port, because this exposes it to the previously discussed port deallocation problems. Even if a server could safely deallocate the reply port right, the deallocation operation would make RPCs significantly more expensive. When a client program exits, its reply ports are deallocated. Every server which a client thread has used receives a port-deleted notification, which the servers must receive. The death of a client precipitates a flurry of context-switches as servers wake up to receive and throw away the port-deleted notifications.

When a Mach 2.5 server dies while handling an RPC, the client remains blocked waiting for a reply message that will never come. The client can not recover from the server's death. A time-out is not an appropriate solution to this problem, because very often the client can not pick an appropriate time-out value.

The new IPC interface provides a reply port field in messages, which carries send-once rights, to support RPCs. A client program wishing to make RPCs allocates a reply port, for which it holds the receive right. To perform an RPC, the client sends a request message to the server, using a send right for a service port held by the server. The request carries a send-once right for the reply port. The server sends its reply message to the send-once right, removing the send-once right from the server's port name space, and the client program receives the reply using its receive right for the reply port. If a client dies during an RPC, the server's send-once right turns into a dead name. When the server tries to send the reply, an error return notifies it that its client has died. If a server wishes to receive a more timely notification of the death of a client, the server can request a dead-name notification for the client's send-once right reply port. If a server dies during an RPC, the client receives a send-once notification for the reply port, so it unblocks and can recover from the failure.

3.3. Notifications

The new IPC interface's strategy of only generating notifications after an explicit request from a user task solves several problems with the Mach 2.5 notification model. The old model makes it impossible for different program components to share notifications. Most notifications generated by the Mach 2.5 kernel represent wasted effort because tasks do not want the notifications. Finally, the asynchronous generation of notifications makes it difficult for tasks to handle the notifications that it does want.

The new interface only sends notifications upon request. The notification request specifies a send-once right for the notify port. The kernel sends the notification to the supplied send-once right.

Because the old interface obliges the kernel to send port-deleted notifications for a dead port's every extant send right, the kernel often generates unwanted notifications. Tasks must receive these notifications, so the death of a port triggers unnecessary context-switches and receive operations. The new interface avoids this problem, because tasks only receive notifications that they have requested.

The old interface directs all notifications to a single task-wide notify port, so separate program components or libraries that try to receive notifications suffer from contention. A standard user-level library to distribute notifications can not solve this problem, because a task may contain multiple binaries which wish to receive notifications. The new interface avoids this problem, because program components can specify send-once rights for different notify ports when they request notifications.

The old interface generates notifications spontaneously. If a task receives a send right for a port, and the port dies, the kernel immediately generates a notification. The task may receive and process the notification before it processes the received message that contains the dead send right. Because the task has not examined the send right, the port-deleted notification does not make sense to the task. The new interface avoids this problem, because the kernel will not generate a notification until the task examines the received message and explicitly requests a notification for the send right.

3.4. Error Recovery

The old IPC interface suffers from poorly defined semantics for many exceptional conditions. Mach 2.5 programs can not recover from some interrupted send operations, which may destroy critical port rights and out-of-line memory. Resources shortages during an old `msg_receive` system call may result in the destruction of ports and data, with no error indication given to the receiving task.

The old interface pretended that the send and receive message operations are atomic. Unfortunately, the Mach 2.5 implementation does not make send and receive atomic operations. Mach messages are complex objects, holding an unbounded number of port rights and out-of-line memory regions. It is very difficult, if not impossible, to make send and receive atomic. The new interface recognizes this and copes with the possibility of partially completed operations.

3.4.1. Interrupted Send

Send operations can not always queue a message containing resources like port rights or out-of-line memory. When this happens, the new interface returns the resources in the message to the sending task. If the task wishes, it can retry the send operation. The old interface destroyed the resources in the message.

The send operation has two phases. First, the kernel copies the message into an internal buffer, translating port rights and making out-of-line memory regions copy-on-write. Second, the kernel tries to queue the message. If the message can not be eventually queued, because the send times out or is interrupted, then the kernel must dispose of the message. However, the message may contain precious resources like a receive right or the sole copy of an out-of-line memory region.

The new interface specifies that an incomplete send operation returns the unsent message to the sender, giving the sender the port rights and out-of-line memory in the message. The sender can then retry the transmission. This return operation resembles a normal receive operation, and is known as a pseudo-receive.

The Mach 2.5 interface makes no provision for returning the contents of the message to the sender in these situations. Because it destroys the message, resources which are only present in the message may be irretrievably lost. For example, receive rights carried in the message are deallocated, destroying the ports. If out-of-line memory is sent with the deallocate option, then the sender does not retain a copy of the memory. This possibility means that if the send operation might block, the old `msg_send` system call is inherently unreliable when used to send receive rights or out-of-line memory with the deallocate option.

3.4.2. Partial Receive

Because a receive operation consumes resources, sometimes a resource shortage might prevent the reception of some resources carried in a message. The old interface does not allow programs to recover from this possibility, because the old `msg_receive` system call returns no error indication. The new interface specifies a simple semantics for partial receive operations that allows programs to perform error recovery.

When a task receives a message, the kernel transfers to the task the resources, port rights and memory regions, held in the messages. This process consumes resources. A port right may require a new name drawn from the task's port name space, and out-of-line memory requires an unused region in the task's address space. In addition, the kernel may need to allocate internal data structures. Because the receive operation is not atomic, an allocation may fail after some resources have been transferred.

The Mach 2.5 interface does not specify the outcome of the receive operation when a resource allocation fails. In practice, the receive operation silently deallocates any troublesome port rights or out-of-line memory. The receiver gets no indication of a problem with the receive operation.

When some port rights or memory regions can not be received due to resource shortages, the new interface specifies the semantics of a partial receive. The partial receive attempts to give the receiver all the resources in the message. The partial receive destroys those resources that can not be transferred. The transfer of out-of-line memory containing port rights guarantees that the receiver never gets the port rights but not the memory. A task never receives a resource that it is not told about in the received message. The partial receive returns an error code that indicates what type of resources were lost.

4. Implementation

The Mach 3.0 implementation takes advantage of several properties of the IPC interface to provide good performance. The implementation uses highly tuned data structures that represent a port in 64 bytes and a port right in 16 bytes. At the same time, the data structures expedite important operations like looking up port rights given a task's name, creating port rights, and deleting port rights. When possible, the IPC code makes use of a scheduling optimization, hand-off scheduling, to reduce the context-switch overhead. In combination these optimizations make the implementation faster and smaller than the Mach 2.5 IPC implementation.

4.1. Data structures

The Mach 3.0 implementation represents port rights with a hybrid data structure. Most port rights fit into a per-task table, at 16 bytes per port right. The name of the port right is its index into the table. Because a task can rename its port rights to create a sparse port name space, the implementation must accommodate port rights that do not fall into the task's table. A per-task splay tree, a self-adjusting binary tree, represents these overflow port rights in 32 bytes. The kernel dynamically adjusts the division between the table and splay tree, growing the table as needed, to minimize memory consumption.

The new implementation solves the reverse translation problem, finding a name given a task and a port, in several different ways. Every port contains a pointer to the task holding the receive right and that task's name for the receive right. If that check fails, the kernel looks for a send right in the task's table of port rights. A closed hash table that translates send rights to port names is folded into the table of port rights. Finally, the kernel checks a global open hash table to find send rights represented in a splay tree.

This data structure takes advantage of two properties of dead names. First, dead names allow tasks to cope with an implementation that reuses names quickly. The per-task tables contain a free list of unused entries. The entry for a deleted port right is pushed onto the free list, and it will be reused for the next port right to be allocated. Second, dead names allow the implementation to lazy-evaluate port destruction. The data structure does not allow the kernel to traverse a dead port's send rights and convert them to dead names. Instead, the kernel converts the send rights to dead names as it comes across them. The kernel uses a separate list of send-once rights attached to the port to generate dead-name notifications.

The hybrid data structures makes the important operations efficient. A name lookup, to convert from a task's name for a port to an internal port pointer, performs a bounds check and indexes into the task's table. The allocation of a send-once right pops an unused entry off of the free list, and the deallocation of the send-once right pushes the entry back onto the free list. Because every send-once right gets its own port name, send-once rights are not entered into the reverse hash tables. A simple RPC with a send-once right reply port does not use the reverse translation algorithm.

The Mach 2.5 implementation uses a straightforward data structure. It represents every port right with a 52-byte structure, and every port with an 88-byte structure. Each port right is on four doubly-linked lists. Each port has a list of its rights, each task holds a list of port rights, and each port right belongs to two global open hash tables. The hash tables perform the translations from task and name to port and the reverse translation from task and port to name. This representation of port rights makes all operations possible, but no operations are particularly efficient.

4.2. Hand-off scheduling

The hand-off scheduling optimization significantly reduces the cost of an RPC. Hand-off scheduling directly transfers control of the processor from one thread to another. The new IPC interface allows the Mach 3.0 implementation to take full advantage of hand-off scheduling.

The `mach_msg` system call allows the implementation to do hand-off scheduling in both directions of an RPC. A fast RPC implementation makes two hand-offs. First the client threads hands-off to a server thread blocked doing a receive, leaving itself blocked waiting for the reply. Then the server thread hands-off to the client, leaving itself blocked waiting for the next request. If both client and server use the `mach_msg` call to combine the send and receive operations, the implementation will avoid queuing and dequeuing of the messages and instead hand messages and control of the processor directly from sender to receiver.

The Mach 2.5 implementation only achieves a hand-off in one direction, from client thread to server thread. The old interface's `msg_rpc` system call, used by the client, does not provide enough flexibility. `msg_rpc` first sends to the destination port a message that carries a reply port, and then it receives from the reply port. Servers can not use `msg_rpc` to send a reply message and then receive a request, because reply messages should not carry a reply port and `msg_rpc` can not receive from a port set.

4.3. Performance

Space and time are both important performance measures. The Mach 3.0 Unix emulation relies heavily on the Mach IPC facility. A current single-server Unix system uses approximately 2000 ports. The emulation of many Unix system calls requires one or more Mach RPCs. The multi-server system under development makes even greater use of the IPC system.

The following table presents performance numbers for a DECstation 3100 running a Mach 3.0 kernel, version XMK23, with single-server Unix emulation, version XUX16.

		Mach 3.0	Mach 2.5
kernel ports	976		
user ports	1320		
total ports	2296	146944 bytes	202048 bytes
kernel rights	976	0 bytes	50752 bytes
rights in tables	2447		127244 bytes
total size of tables	3032	48512 bytes	
rights in trees	37	1184 bytes	1924 bytes
total size		196640 bytes	381968 bytes
null RPC		125us	210us

Table 4-1: Performance Comparison

4.4. Implementation History

The implementation of the new interface went through several phases, culminating in the current Mach 3.0 implementation. The goal was to test the new interface with a prototype before putting substantial time into the radically different implementation that was envisioned.

A modified version of Mach 2.5 first implemented the new interface. The augmented Mach 2.5 data structures did not perform well, principally because the creation and deletion of send-once rights was relatively slow. However, the prototype implementation did allow test programs to exercise the new interface. The test programs demonstrated the usefulness of the new features. This evaluation was used to refine the new interface.

The core of the new implementation, about 80% of the total including all the data structures, was developed and tested as a user-level library. First, interactive programs tested the low-level data structures. After the higher-level message functions and kernel calls were written, they were tested with communicating user-level coroutine threads. Finally, the code was moved into kernel space and mated with the Mach kernel. This development process produced an exceptionally trouble-free implementation.

The Mach 2.5 netmsgserver runs under the new interface, using the compatibility code. Although new clients and servers can use the netmsgserver to communicate over the network, the performance is oppressively poor. Because the netmsgserver sees the send-once right reply ports as send rights which are destroyed, after every RPC it goes through a broadcast protocol to destroy the reply port. With knowledge of send-once rights, a revised netmsgserver can avoid this performance pessimization.

5. Compatibility

The Mach 3.0 IPC facility includes complete backwards-compatible support for the older Mach 2.5 interface. This design requirement motivated some aspects of the Mach 3.0 interface. Programs written to the older interface make use of a compatibility mode that supports the older system calls and kernel calls. The compatibility mode allows programs to use both interfaces simultaneously. Programs written to the new and old interfaces can even interoperate, exchanging messages and forming client-server relationships. The compatibility mode does not affect the performance of programs written to the new interface, and if desired the Mach 3.0 kernel can be compiled without the compatibility code.

5.1. Compatibility Requirements

The compatibility requirement motivated some important design decisions for the new interface. If it were not for compatibility considerations, the IPC interface could be potentially have simplified message formats and port right management.

The interoperability requirement prevented radical changes to the message format. The Mach 3.0 message format, a message header followed by typed data, differs from the Mach 2.5 message format in only minor details. Because the formats are so similar, programs using the compatibility mode can interoperate with programs written to the new interface. The implementation converts between the two formats when a programs uses the old system calls to send and receive messages.

The new interface allows reply ports to be send rights or send-once rights. It would be simpler if only send-once right reply ports were allowed, but this would make it difficult to convert some programs to the new interface. These programs use the reply port field as a general mechanism for conveying a send right, instead of restricting its use to RPCs.

Compatibility considerations motivated the design of reference counts for send rights. An alternative design, without reference counts, would make send rights behave like send-once rights. When a task would receive a port right, the port right would always appear in the task’s port name space with a new name. A task could potentially hold many send rights for a single port. This alternative would simplify the interface and the implementation, but it would prohibit backwards-compatibility. Some programs rely on the current behavior to perform authentication and naming functions by comparing send rights.

5.2. Compatibility Support

The compatibility code supports the complete Mach 2.5 interface, with no exceptions. This does not mean that it is impossible to write a program which will determine under which implementation it is running. However, it does guarantee that correct programs will continue to operate. The compatibility code hides new features like send-once rights from old programs.

The compatibility code supports programs which use both the old and new IPC interfaces. This allows a program to be gradually converted from the old interface to the new interface. For example, libraries May be written to different interfaces. This occurs frequently in practice, because the Mach 3.0 transparent emulation library uses the new IPC interface. Any process using the old interface has the emulation library in its address space, using the new interface to implement the Unix emulation.

Although the new and old interfaces are similar in many respects, they share no names. The new interface prefixes every name with "mach_" or "MACH_". This prevents name conflicts between the interfaces. It allows a common set of include files to serve both interfaces, and allows a binary to call freely functions from both interfaces.

The compatibility mode operates on a per-port right basis, not per-task. For example, a send right received with the old `msg_receive` system calls is marked internally with a compatibility flag. If the port is destroyed, the send right disappears and the kernel sends a port-deleted notification to the task's notify port, implementing the old semantics for port destruction. The same task may also hold other send rights received with the new `mach_msg` system call. When those ports are destroyed, the send rights turn into dead names.

The compatibility code disguises send-once rights as send rights. When a task receives a send-once right with an old system call, like `msg_receive`, the task gets what looks like a send right. However, when the task sends a message to the faux send right, the right disappears and the task receives a port-deleted notification. This behavior preserves the essential send-once character of the right. It allows servers written to the old interface to handle clients using send-once right reply ports.

5.3. Implementation Impact

The compatibility code does not affect the Mach 3.0 interface's performance. The code does slightly increase the size of the kernel, but a build-time option allows the code to be removed if this is a problem. In many respects, the Mach 3.0 compatibility code is a better implementation of the Mach 2.5 interface than the Mach 2.5 implementation, because the compatibility code does not suffer from the old implementation's bugs. However, the compatibility code does perform more slowly than the Mach 2.5 implementation. The new implementation does not optimize the common paths through the compatibility code.

6. Conclusion

The Mach 3.0 IPC facility revises the Mach IPC interface to improve support for multi-threaded programs, remote procedure calls, and object-oriented programming. The new interface corrects several problems in the older interface that made it difficult to use correctly. The Mach 3.0 implementation meets the needs of demanding applications, like the Mach 3.0 Unix emulation, with substantially improved space and time performance. Finally, the implementation supports the old Mach 2.5 interface with a compatibility mode that lets programs gradually migrate to the new interface.

I. Interface Summary

mach_msg(msg, options, send_size, rcv_size, rcv_name, timeout, notify)
Send a message, receive a message, or do both. When sending, the message header contains a right for the destination port. Some options use the timeout value and the notify port.

mach_host_self()
Returns a send right for a port representing the caller's host.

mach_task_self()
Returns a send right for a port representing the caller's task.

mach_thread_self()
Returns a send right for a port representing the caller's thread.

mach_reply_port()
Allocates a port in the calling task and returns the name of the receive right. This call provides a way of allocating a reply port that does not use a reply port. The implementation may optimize the allocated port for use as a reply port.

Table I-1: System Calls

mach_port_allocate(task, what_right, OUT name)
Allocates a port, port set, or dead name in the task. Returns the name of the new right.

mach_port_allocate_name(task, what_right, name)
Allocates a port, port set, or dead name in the task. The new right has the specified name.

mach_port_get_refs(task, name, what_right, OUT refs)
Returns the number of references held by the task for the specified port right.

mach_port_mod_refs(task, name, what_right, delta)
Modifies the number of references held by the task for the specified port right.

mach_port_deallocate(task, name)
Deallocates a reference for a send right, send-once right, or dead name.

mach_port_destroy(task, name)
Deallocates all references and rights denoted by the name.

mach_port_type(task, name, OUT type)
Returns the type of the specified port right.

mach_port_names(task, OUT names, OUT types)
Returns a list of the names in the task's port name space, with a corresponding list of port right types.

mach_port_rename(task, old_name, new_name)
Renames a port right.

mach_port_get_receive_status(task, name, OUT status)
Returns status information for a receive right.

mach_port_set_qlimit(task, name, qlimit)
Modifies a receive right's queue limit.

mach_port_set_mscount(task, name, mscount)
Modifies a receive right's make-send count.

mach_port_move_member(task, port_name, set_name)
Moves a receive right into or out of a port set.

mach_port_get_set_status(task, set_name, OUT members)
Returns a list of the members of a port set.

mach_port_request_notification(task, name, which, sync, notify, previous)
Uses the supplied notify send-once right to request the specified notification for a port right. Returns the previously registered send-once right, if any.

mach_port_insert_right(task, name, right)
Inserts the supplied send, send-once, or receive right into the target task, giving it the specified name.

mach_port_extract_right(task, name, how_extracted, OUT right)
Returns a send, send-once, or receive right extracted from the target task.

Table I-2: Kernel Calls

Table of Contents

List of Tables