A Fast Mach Network IPC Implementation

Joseph S. Barrera III

jsb@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue Pittsburgh, PA 15213

Abstract

This paper describes an implementation of network Mach IPC optimized for clusters of processors connected by a fast network, such as workstations connected by an Ethernet or processors in a non-shared memory multiprocessor. This work contrasts with earlier work, such as the netmsg server, which has emphasized connectivity (by using robust and widely available protocols such as TCP/IP) and configurability (with an entirely user-state implementation) at the expense of performance.

The issues addressed by this work are support for low latency delivery of small and large messages, support for port capabilities and reference counting, and integration with the existing local Mach IPC implementation. Low latency for small messages requires careful buffer and control flow management; this work is compared with other fast RPC work described in the literature. Low latency for large messages, particularly for faster networks, requires an avoidance of copying, which can be achieved through virtual memory support; the modifications that were necessary to make Mach's virtual memory support inexpensive enough to be useful for this purpose are described. The distributed implementation of port capabilities, port reference counts, and port migration is discussed, and compared with that in the netmsg server. Finally, performance data is presented to quantify the speedup achieved with the described implementation.

1 Introduction

Mach IPC has traditionally been extended over the network by use of the netmsg server [Sansom 88, Julin & Sansom 89], a user-level server which uses general purpose protocols such as TCP/IP. While this approach has connectivity and configurability advantages, it has a serious performance disadvantage. In particular, network Mach RPCs are three to five times slower than network RPCs in other systems on comparable hardware.

An effort currently underway to implement Mach abstractions on a non-shared memory multiprocessor yielded a requirement for faster network Mach IPC. Non-shared memory multiprocessors (such as the Intel iPSC/860 and its successors) have interconnects with high throughput and very low latency; it is inappropriate to burden such fast interconnects with a slow IPC implementation. Fast network IPC is particularly important for such machines since much more IPC on such systems will be remote.

This research was supported by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDS972-90-C-0035.

2 Issues and Implementation

The following sections describe various important issues in constructing a network IPC implementation, and how the described implementation addressed them. These issues include support for low latency delivery of small and large messages, support for port capabilities and reference counting, and integration with the existing local Mach IPC implementation. [Draves 90]

2.1 Message sizes and latency

There are two important cases to optimize in Mach IPC: small messages (less than 128 bytes), and large messages (carrying one or two pages of out-of-line data). Small messages are used for most requests and many replies, whereas large messages are used for transferring data, including file and device access and paging traffic. Buffering in operating system servers and emulators is responsible for the lack of intermediate sized messages. For example, the BSD Unix server translates a read system call into a read from a mapped file area; the read thus either generates no message, if the page containing the read data is resident, or a small request followed by a large reply if the page needs to be faulted in.

For small messages, software latency is the dominant cost. Since there is little data, network throughput is irrelevant. Network latency (including the software cost of setting up the send and the receive) can range from tens to hundreds of microseconds, but it requires a well-optimized IPC system for such costs to be noticeable.

For large messages, network throughput and data-dependent software costs become important. The most common data-dependent software cost is the cost of copying the data between buffers. Limiting the number of copies to one on send and one on receive is moderately straightforward; eliminating all copies is significantly more difficult. One difficulty is that some network interfaces can only send from or receive into special device memory; also, some interfaces do not scatter/gather, and thus may require a copy to add or remove headers.

Although the cost of copying data may not affect asymptotic throughput, it does increase the latency of one or two page messages; since such messages are common, copying is worth avoiding. Copying can fail to affect asymptotic throughput when the copying of one packet can be done in parallel with the network transmission of another; this is feasible with Ethernet since memory-to-memory copying is typically several times faster than Ethernet. When transferring a small number of packets, however, there is less chance for such overlap. Faster networks (with speeds much closer to memory access speeds) also increase the significance of copying.

2.2 Small messages

Small message transfers were optimized by borrowing many of the techniques that have been explored in systems such as Firefly [Schroeder & Burrows 89], Amoeba [van Renesse et al. 88], Sprite [Ousterhout et al. 88], and V [Cheriton & Zwaenepoel 83]. For example, context switches are avoided by having the interrupt thread do as much work as it can and having the thread receiving the message do the rest. A context switch on the sending side of an RPC is avoided by not switching to the idle thread when there are no runnable processes; instead, the sending thread spins, waiting for the reply message (or for another thread to become runnable).

2.3 Large messages

Beyond the optimizations required for small messages, optimizing large messages requires that the data size dependent costs be minimized. As discussed above, the primary data dependent software cost is the cost of copying data, and this cost is particularly significant both for latency and for networks that run close to memory speed. The following sections examine two methods for avoiding copying.

2.3.1 Avoiding copies via shared buffers

One method for avoiding copies, used for example by Firefly RPC, is the use of non-pageable buffers shared between user tasks and the network driver. Users construct messages in a shared buffer; the driver then sends directly from the appropriate buffer. Beyond avoiding both mapping and copying operations, this scheme has the advantage of working with network devices that can only view a subset of the physical memory.

The shared buffer area approach has a number of disadvantages, however. First, a single shared buffer area offers no protection against tasks interfering with each other's message operations. Such protection requires a separate buffer area for each sending task; however, this divides the device space into small pieces, which may artificially limit (in a device-dependent way) the maximum message size. Furthermore, separate buffer areas for each task do not protect the driver from malicious users, particularly if the device does not do scatter/gather and thus must create headers in the shared area.

There are also semantic problems with the shared buffer area approach. Most IPC systems offer by-copy semantics for message sending: once the send completes, modifying the data in the sent buffer will not change the data seen by the receiver. In contrast, when data is sent via a shared buffer, the sender must wait for the driver to indicate that it is done with the buffer. Furthermore, the fact that the shared buffer is a limited resource, and that most of the user's data does not live in the shared buffer, makes it probable that the user task will need to copy its data into or out of the shared buffer upon sending or receiving. Shared buffers may therefore only shift the need for copying onto the user task instead of truly eliminating it.

2.3.2 Avoiding copies via virtual memory mapping

The above-mentioned problems with the shared buffer approach, combined with the difficulty of integrating shared buffers with Mach IPC semantics, led to the decision to avoid copying by using virtual memory operations. This approach has precedent in the Mach system, as local Mach IPC uses copy-on-write to avoid copying data until either the sender or receiver attempts to modify the data.

Copying is avoided upon sending by mapping user data into the kernel address space, faulting in any non-resident pages and marking the pages unpageable. The network driver then uses the kernel mapping of the data. When data is received, it is received into kernel buffers which are then mapped into the receiver's address space.

Unfortunately, the existing Mach methods for manipulating out-of-line data were ill-suited for network IPC, and thus the cost of using the virtual memory system to avoid a copy was several times the cost of the avoided copy. The primary problem is that the Mach virtual memory data structures used to represent copied data are optimized for long term copy-on-write sharing of data, typified by address space copies. In particular, out-of-line data in a message is represented by a *copy object*, which is a complicated data structure

which preserves copying and sharing relationships. When out-of-line data was sent across the network using the original virtual memory data structures, several expensive operations occurred. First, the address in the message was converted into a copy object. The copy object was then mapped into the kernel's address space, and the pages were faulted in and made non-pageable. After the data was sent, the data was unwired and unmapped from the kernel address space.

The key observation about the way data is used when sent remotely is that it is used read-only and then quickly deallocated by the driver, and it only needs to be protected by writing from the user until the driver has completed sending it. This short term sharing of the data suggested a *page list* data structure. Every page in the data to be sent is located, faulted as necessary, and placed in list, which is then handed to the network driver.

There are two methods for preventing the user task from modifying the pages being sent. The first is to prevent the user from returning from the kernel as long as the driver needs the pages. This method works well when the user task is performing a send-receive call and thus will wait anyway for a reply before leaving the kernel. This method also is viable if the device requires a copy (for byte-swapping or limited addressing reasons) and thus will be done with the pages as soon as it has made the copies, or if the network is fast enough that the acknowledgement is likely to arrive by the time the sender has unwound from the routines leading to the driver send routine.

The second method is to protect the pages against writing and let the user return from the send. Instead of using the general and relatively expensive copy-on-write technology, the pages are marked "busy", and protected against writing by calling a pmap routine (which manipulates the page tables directly). When a thread faults on a busy page, it blocks until the page is made not busy; this is a preexisting mechanism in the Mach virtual memory system to indicate that the page is waiting for some event to complete, such as a page to be read from disk. The network driver then marks the page not busy when it is done with it. It does not have to unprotect the page, since the virtual memory system will automatically unprotect the page upon a fault. One reason for not having the network driver unprotect the page is the possibility of the driver accidently granting write permission when it should no longer be allowed.

A separate method for preventing copying is for the user to specify that the kernel should deallocate the sent pages from the user's address space. This option is standard in Mach IPC, and is commonly used by servers which generate but do not cache data, such as pagers or device servers.

These issues are not particular to network IPC; they are applicable whenever the kernel needs to write data to a device, be it a network (used for IPC or standard protocols) disk, or tape. The key feature is the lack of long term sharing, and the fact that the kernel won't write to the data. A separate project has generalized page lists so that they can be used in all such cases. [Black 91]

2.3.3 Avoiding copies on receiving using virtual memory mapping

Copies can be avoided upon receiving as well as sending. When pages of data are received, they are received directly into anonymous pages, unmapped by any user task. These pages are then threaded into page lists, which are then inserted into the Mach message being constructed. Finally, when a user task receives the message, the pages in the page lists are mapped into the task's address space, and the page list pointers in the message are replaced with pointers to the mapped areas. The use of anonymous pages for receiving allows copies

to be avoided without manipulation or examination of the receiver's address space, and without a thread blocked waiting for the message. Other systems, such as V, require that a thread be waiting for a message for the copy to be avoided.

2.4 Integration with local IPC implementation

One of the negative aspects of integrating remote IPC into the in-kernel local IPC system is the possibility of introducing more complexity into an already complex system. Fortunately, the interactions between the local and remote IPC code are limited to two areas: message translation and message queueing.

Local Mach IPC messaging has four stages: copyin, queuing, dequeueing, and copyout. Copyin consists of copying the message buffer from the user's address space into a kernel buffer, and translating ports and out-of-line data into internal kernel representations. The message buffer is then queued on the destination port. When a thread is ready to receive the message, the message is dequeued and copied out, with translation back from kernel to user representations for ports and out-of-line data.

The remote Mach IPC implementation intercepts messages at the queueing stage. When a message is about to be queued on a port, the queueing routine checks whether the port is remote; if it is, it gives the message to the remote IPC system instead of queueing it. This code parallels existing code which checks for messages sent to kernel owned ports such as task, thread, and device ports. Conversely, when the remote IPC implementation receives a message from the network, it inserts it into the local IPC system by calling the queueing routine, as if the message has been sent from a local task.

Similarly, during the translation state, if a message is destined for a remote port, the ports and out-of-line data are translated into over-the-wire representations instead of kernel internal representations. When a message is received from the network, the over-the-wire representations are translated into kernel internal representations before the message is given to the local IPC system.

2.5 Port names, capabilities, and reference counts

The extension of Mach IPC across a network introduces issues such as distributed naming, consistency, and garbage collection. The following sections describe how remote rights are represented, introduce an efficient mechanism for collecting and distributing port usage information, and describe how this mechanism is used to implement port death, port migration, and no senders notifications.

2.5.1 Global port identifiers

When a node sends send rights to a port to another node, it uses a global port identifier to identify the port. This global identifier serves two purposes. First, it allows a node to "merge" port rights, as required by Mach IPC semantics. Second, it provides a method for determining the node to which messages sent to the port should be delivered.

When a task receives send rights for a port that it already holds send rights to, Mach IPC guarantees that the new send rights will have the same name as the old. This merging of port names allows the task to identify two port rights as referring to the same port. The use of global port identifiers allows the kernel to support this merging by allowing it to recognize identical remote send rights.

A fundamental question concerning global identifiers is whether they should encode location information. It is generally cheaper and simpler to locate an object directly from its identifier than to use a separate mechanism to map identifiers to locations; however, some provision must be made for object migration, which invalidates the location information encoded in the object's identifier. Either a new identifier for the object, encoding the new location, must be created and distributed, or the identifier must be entered into a list maintained at every node which lists all identifiers whose location information is incorrect.

Three factors led to the decision to encode port information in the global identifier, and to change identifiers upon port migration. First, port migration is very rare, whereas sending to a port is very common. It is therefore acceptable to complicate port migration in order to reduce the space and time cost of determining port location. Second, the mechanisms required for distributing new identifiers and invalidating old identifiers are already required by other aspects of Mach IPC, such as no-senders notifications. Finally, identifier replacement can be performed with no user impact, since tasks use per-task port names and never see global port identifiers; this decision would have been more difficult if all tasks shared the same port name space.

2.5.2 Proxy ports

A proxy port is the local representative of a port whose receive rights lives on another node; it is created the first time a node receives send rights to the port. A proxy port is treated like a normal port by the local IPC code, and thus automatically maintains per-node usage information about the port. In particular, it maintains local send right counts, which is critical for implementing distributed no senders notification. The global identifier for a port is contained in every proxy for that port. This allows nodes to merge send rights that they have seen before; it also allows the remote IPC code to know where to send a message when the local IPC code attempts to send the message to a proxy.

2.5.3 The periodic token

The periodic token solves several information distribution problems in an inexpensive manner. It allows a single node to broadcast information to all other nodes. It also allows a node to collect information from a large set of nodes. Finally, it allows nodes to know when such information has been seen by all other nodes.

The periodic token is a writable message that is periodically sent on a fixed path through every node in the system. The token passes by each node three times each period. During the first pass, each node writes information into the token. During the second pass, each node reads information from the token. The third pass simply informs each node that all information carried by the token has been seen by every node. To reduce the overhead due to processing incoming token messages, the token pauses for a few seconds in between periods.

The primary advantage of the periodic token is that it batches many small and often self-cancelling messages into one larger message, greatly reducing message handling overhead. It is particularly appropriate for port usage information, since such information is intrinsically self-cancelling (since intermediate reference count values are uninteresting), and does not need to be acted upon any more than once every few seconds.

2.5.4 Detecting no senders

A task can request a no-senders notification message when it attempts to receive from a port for which no task has send rights. This notification allows servers to garbage collect objects which are no longer in use. Detecting no senders is easy in the centralized case; it is significantly harder in the distributed case.

In the centralized case, one data structure describes all current usage of the port, including the number of send rights held by tasks and queued messages in the system. This count is updated by every operation that changes the send right count; it is therefore a simple matter to send a notification to the receiver when this count drops to zero.

In the distributed case, the send right count is distributed across all nodes, and in general no one node has an accurate global count; all a node knows is the local send right count. This lack of knowledge has two causes. First, any holder of either send or receive rights to a port can generate new send rights. If receive rights were required to generate send rights (as is the case for send-once rights), then the holder of receive rights could maintain an accurate count. Second, when a node sends send rights to another node, the sending node cannot assume that the global send count has increased, since the send right might be merged by the receiving task on the receiving node.

A naive approach to implementing no senders detection, in which a pessimistic yet up-to-date send count is maintained at the node holding receive rights, would generate a huge amount of message traffic. In such an approach, a message would be sent to the holder of receive rights whenever a send right was sent from one node to another; another message would be sent by the receiver of the send right if the send right count was not to be incremented. Such a count is pessimistic because it overestimates the global send count. When an accurate count is not possible, a pessimistic count is necessary to prevent incorrect no-senders notifications.

The periodic token can be used for a much more efficient implementation of no senders detection, since the periodic token introduces a fixed, small number of messages into the system. In such an implementation, each node associates a transit count with each proxy; this transit count is incremented when a node sends the send right associated with the proxy, and decremented when it receives such a send right. The purpose of the transit count is to count all send rights held by messages that have been sent and not received. Each period, as the token circulates, each node writes the send count and transit count of its remote send rights into the token.

If the token could visit every node instantaneously, then no senders could be detected by looking for uniformly zero send and transit counts. Unfortunately, as the token passes from one node to the next, the second node can send a send right to the first, receive it back from the first, and then deallocate it; this leaves the already scanned first node with a send right and the second node with a zero transit and send count.

A correct algorithm does exist which requires two passes and a new export flag. This new flag indicates whether any send rights were sent by the node since the previous pass. No senders can now be detected if two passes return zero send and transit counts, and if the export flags indicate that the right was not sent at all between the first and second pass.

2.5.5 Port death

Port death is easily implemented once no senders has been implemented. When a port dies, its death is broadcast using the periodic token. The port and its global identifier can

then be garbage collected as soon as no senders is true. Note that no senders automatically accounts for send rights in transit; simply waiting until the token announcing port death has been seen by every node does not.

2.5.6 Port migration

Like port death, port migration is easily implemented with the help of no senders detection. When a port is migrated from one node to another, a new identifier is allocated with correct location information. This new identifier is broadcast to each node using the periodic token; however, the old identifier is used until every node knows that every other node has seen the new identifier. The danger of using the new identifier too soon is that a node that has not been told about the new identifier will not realize that it refers to the same port as the old identifier, and thus will not correctly merge the port right. When no senders is true for the old identifier, all nodes are free to garbage collect the forwarding information associating the old and new identifiers.

2.6 Reliable delivery and flow control

A network IPC implementation must provide reliable delivery. Unreliable delivery can be caused by packets destroyed or lost by the network, or by packets being dropped by a node because it lacks buffer space. The described implementation was originally designed for reliable networks and thus used a protocol that only handled lack of buffer space through the use of negative acknowledgements. This protocol has since been extended for unreliable networks by adding timeouts while retaining negative acknowledgements.

2.6.1 Protocol for a reliable network

In the original protocol, designed for reliable networks, every packet sent to a node is either positively or negatively acknowledged. A positive acknowledgement allows the sender to send another packet; a negative acknowledgement requires the sender to resend the current packet. A negative acknowledgement is sent only when buffer space was not available when the packet was received, but is available now; it may thus be delayed for however long it takes for the receiver to find more space.

The ability to acknowledge (positively or negatively) every packet requires cooperation from both software and hardware, including limited hardware support for flow control. The software must recognize when it only has one buffer left, and continually reuse that buffer to receive every incoming packet and record nodes that require negative acknowledgements. The hardware must not lose packets that are sent to a node that has not yet rearmed its interconnect with a receive buffer; in practice, since the interconnect does not have infinite buffer space, this means that the interconnect must be willing to delay network sends. The interconnect in the iPSC/860 provides this capability; presumably a token ring could be designed to do so as well. Note that the existence of hardware flow control does not obviate the need for software flow control, as hardware flow control makes the network partially unusable for as long as it needs to hold onto a packet, which can in the worst case cause system wide deadlock.

The primary motivation for using negative acknowledgments is the avoidance of timeouts to detect buffer space overflow. Timeouts are a poor mechanism in this case because of the large variation in time that it takes for a node to find more buffer space, which makes it difficult to select an appropriate timeout value. In some cases, a node just needs to schedule

a thread to perform memory allocation that cannot be performed at interrupt level; in other cases, the node will need to page out to disk before it will have sufficient buffer space.

2.6.2 Adaptation for an unreliable network

To extend this protocol to unreliable networks such as Ethernet, it was necessary to add timeouts and retransmission. Once these mechanisms have been added, negative acknowledgements are no longer necessary; however, we decided to retain negative acknowledgements for two reasons. First, we wanted to use a common protocol for reliable and unreliable networks, and did not want to give up the advantages that negative acknowledgements provide in the reliable case. Second, it is difficult to find a good timeout value when timeouts are used for both packet loss and buffer space depletion.

To combine negative acknowledgements with timeouts, we added two more types of messages: acknowledgement requests and quench requests. When a node sends a packet, it expects an acknowledgement (positive or negative) within two timeout periods. If it does not receive one, it sends an acknowledgement request and waits again. In turn, when a node receives a packet when it has no buffer space, and if it still has no buffer space one timeout period later, it sends a quench message to the sender to prevent a series of acknowledgement request messages. When buffer space becomes available again, the receiver sends a negative acknowledgement, repeated as necessary.

When timeouts are used only for detecting lost packets, it becomes possible to use a much shorter timeout. This is particularly important on inexpensive workstations and on personal computers, which combine a low latency network (Ethernet) with lossy and unreliable network interfaces. In particular, we were faced with Ethernet cards that produced five percent packet loss rates between two machines on the same wire; when combined with a hardware latency around a millisecond, it makes no sense to use timeouts of hundreds of milliseconds, as the BSD TCP implementation does [Leffler et al. 89].

3 Tradeoffs

The primary tradeoff between this implementation and the netmsg server implementation is speed versus generality. This implementation has been primarily designed to support distributed memory multiprocessors; it therefore assumes a fixed number of homogeneous nodes, low network latencies, no independent failures, and no network partitions. In return, it provides fast message delivery with small space costs and timely notifications for events such as port death.

It is possible to obtain both fast IPC with nearby nodes and good connectivity with far-away machines by using the fast IPC implementation among a cluster of nodes and then running a netmsg server on one of the nodes. In this configuration, the netmsg server acts as a gateway. Messages sent within the cluster will be handled entirely by the fast IPC implementation; message sent from a node in the cluster to a machine outside of the cluster will be sent first via fast IPC to the node running the netmsg server, and then to the remote machine by way of the remote machine's netmsg server.

4 Evaluation

The described implementation is considerably faster than the netmsg server, and comparable to the fastest RPC systems described in the literature. A simple Mach RPC using the netmsg server between two i486 PCs running Mach 2.5 (with in-kernel TCP/IP) takes 9.2 milliseconds. In contrast, a simple Mach RPC using the described implementation between the same machines takes 2.5 milliseconds. This represents almost a fourfold improvement over the netmsg server, and is comparable with RPC times for V (2.5 milliseconds) and Sprite (2.8 milliseconds), as measured between sun 3/75 workstations, and for Firefly RPC (2.7 milliseconds) running on a five processor Firefly [Schroeder & Burrows 89].

The described implementation also provides comparable performance to that of proprietary message passing systems on distributed memory multiprocessors. For example, a simple Mach RPC between two 16 Mhz i386s using the iPSC/2 interconnect takes 0.9 milliseconds; this compares to 0.7 milliseconds for NX, Intel's proprietary operating system, running on the same hardware, despite the considerably simpler semantics of NX message passing.

5 Conclusion

A Mach IPC implementation has been described which has been optimized for clusters of processors connected by a fast network. By avoiding the complexities introduced by ill-behaved hosts and networks, adopting optimizations demonstrated in previous fast RPC work, and developing new techniques to avoid copying, the new implementation performs competitively with other RPC systems and considerably faster than the netmsg server implementation, while preserving full Mach IPC semantics. This implementation will serve as a cornerstone for efficient Mach kernel support for distributed memory multiprocessors.

References

[Black 91] Black, D. L. Page Lists and EMMI Work in Progress. Unpublished, June 1991.

[Cheriton & Zwaenepoel 83] Cheriton, D. and Zwaenepoel, W. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the Ninth Symposium on Operating System Principles.* ACM, 1983.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the USENIX Mach Workshop*, pages 101–121, October 1990.

[Julin & Sansom 89] Julin, D. P. and Sansom, R. D. Issues with the Efficient Implementation of Network IPC and RPC in the Mach Environment. Unpublished, September 1989.

[Leffler et al. 89] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3 BSD Unix Operating System.* Addison-Wesley, Reading, MA, 1989.

[Ousterhout et al. 88] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[Sansom 88] Sansom, R. D. *Building a Secure Distributed System.* PhD dissertation, Carnegie Mellon University, May 1988.

[Schroeder & Burrows 89] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. In *Proceedings of the Twelfth Symposium on Operating System Principles.* ACM, 1989.

[van Renesse et al. 88] van Renesse, R., van Staveren, H., and Tanenbaum, A. S. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):23–34, oct 1988.