A Fast and General Implementation of Mach IPC in a Network

H. Orman E. Menze III S. O'Malley L. Peterson *

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper describes an implementation of the Mach IPC abstraction on a network. Our implementation, called Mach NetIPC, is done in the context of the x-kernel, which provides a networking subsystem for Mach. The paper motivates the design choices we made, describes the x-kernel protocol graph that implements the design, and reports on the performance of the resulting system.

1 Introduction

Mach IPC is the communication abstraction of the Mach operating system. It supports a rich semantics: multi-part, typed messages are delivered reliably and in sequence to ports; tasks hold, and can transfer to each other, the right to send to, and receive from, ports; and tasks are notified when the status of ports they hold a right to changes (e.g., the holder of a send right is notified when no task currently holds a receive right). Although straightforward to implement in the microkernel of a single machine, extending Mach IPC semantics to a network environment—we call such a system Mach *NetIPC*—is problematic. Multiple host architectures, individual node failures, lost messages, network partitions, and variable network delays all combine to make the implementation of NetIPC a difficult proposition.

This paper describes an implementation of Mach NetIPC in the context of the x-kernel—a protocol implementation environment that has been incorporated into Mach [4, 7]. Our implementation of Mach NetIPC is not the first; there have been several earlier implementations of the Netmsgserver [5], and an in-kernel implementation tailored for NORMA architectures [1]. Because of our use of the x-kernel, however, our implementation has the following distinct advantages.

- The x-kernel supports the decomposition of complex protocols into "microprotocols" that are small, easy to understand, implement, debug, and tune. This has helped us to manage the inherent complexity of Mach NetIPC.
- The x-kernel supports configurable protocol graphs which make it possible to optimize the set of protocols used for different architectures, including tightly-coupled distributed memory multiprocessors, workstation clusters, and arbitrary collections of hosts connected by the Internet.
- The x-kernel protocol graph can be partitioned across multiple protection domains. This allows partitioning of the Mach NetIPC protocols between the microkernel and a user task, depending on how one wants to trade performance against trust. In other words, when building a secure version of Mach where microkernel minimization is important, one can place all the network protocols in a user task; when building a high performance version of Mach (perhaps for a multiprocessor) one can locate the Mach NetIPC protocols in the kernel.

*The work presented here was supported under DARPA grant DABT63-91-C-0001.

This paper is organized as follows. Section 2 motivates our design by examining various issues related to a network environment. Section 3 then describes the x-kernel protocol graph that implements Mach NetIPC and Section 4 reports on the performance of our implementation. Finally, Section 5 discusses the relationship between our implementation and proposed changes to Mach IPC, and Section 6 presents some concluding remarks.

2 Design Issues

This section identifies the difficulties in extending Mach IPC over a network, and reports the design choices we made to address them. The discussion motivates the protocols described in the next section.

2.1 Decentralization

An obvious problem in extending Mach IPC to run over a network is that information about ports—in particular, what tasks hold send and receive rights—must be maintained in a distributed fashion. In this context, the key question is what information does any single node need to maintain, and does this limit how large the system can grow.

A goal of our design is that it scale arbitrarily. That is, rather than assume that the set of tasks communicating with each other are restricted to a limited set of nodes that are known *a priori*, or restricted to a single network (e.g., a LAN), we assume that Mach NetIPC can be used across an arbitrary collection of nodes. Stated more precisely, if we consider nodes A and B to be *linked* when a task on node A holds a send right a port and a task on node B holds a receive right to the same port, then we expect the transitive closure of all linked nodes to be arbitrarily large—tens or hundreds of thousands—and possibly spread over the Internet.

Note that this definition of scalability is orthogonal to how Mach NetIPC scales across two other dimensions: (1) the number of tasks running on different nodes that all hold rights to any one port, and (2) the number of ports that tasks on any one node will hold rights to. In both of these cases, we would consider anything over a few hundred to be unusually large. This seems to be a fundamental limitation of Mach IPC, and is not specific to its implementation in a network.

The main implication of this goal is that we cannot expect to maintain a single, complete database of port rights, even if that database is physically replicated throughout the network. Instead, we can only expect each node to maintain information about those ports to which its tasks hold rights. In particular, each node that holds a send right knows the node that currently holds the receive right for the port (but not about the other senders), and the node that holds the receive right keeps track of all the nodes that currently hold a send right. A second implication is that our implementation cannot depend on the existence of a network broadcast facility.

2.2 Node Failure

It is important for Mach NetIPC to survive in the face of simple node failures, meaning that nodes that were communicating with the failed machine must eventually remove all state associated with the communication, including all port rights. We define a simple failure as one that exhibits crash semantics, that is, a node fails silently without transmitting any extraneous messages [10]. To meet this requirement, we have designed a failure handing system that is consistent with the fully decentralized communications model presented in the previous section.

We have adopted a "retry until reboot" strategy that dictates the equivalence of node reboot with node failure notification. When a node fails, all outstanding connections with that node will continue to retry until they detect a reboot. A confirmed reboot will cause deletion of all port rights associated with the node. The system assigns a unique identifier to each incarnation of a node, and that identifier is communicated to interested parties. The incarnation number of a remote node is used to provide an index of liveness for local database items and communication channels.

An additional mechanism, an administrative shutdown, is needed to handle the problem of node failure without reboot (e.g., removal of the machine to a warehouse). Using this mechanism, a systems administrator could inform Mach NetIPC of a node failure, causing the software would take care of all required notifications.

Our approach to node failure avoids the problems associated with incorrect failure notification and network partitioning. This is in contrast to systems that depend on timeouts (or heartbeat messages) to detect failure. In such systems a slow machine or one separated by a network partition may be mistakenly declared failed, which implies that the node must be killed (at a minimum, all of its network connections are severed) in order to ensure consistency. In normal operation using our approach, there is no possibility of a live node being declared dead, and thus, no need for the NetIPC code to kill a live node to insure the consistency of the port rights database. (Of course an incorrect administrative shutdown request could lead to a false notification, but we consider fixing this to be the responsibility of the issuer of the shutdown.) Our approach also inflicts no performance penalty in the common case where no nodes fail. Finally, the correctness of this approach is almost solely dependent upon the correctness of one simple protocol – an easily analyzed piece of code.

This failure model is different from that found in the Mach 3 NORMA IPC implementation[1]. The NORMA code assumes that there is a fixed and known collection of communicating nodes, that network partitions do not happen, and that there is no independent node failure (e.g. if one node in the group fails all nodes fail). While this approach has performance advantages, it does not scale well beyond distributed memory multiprocessors and tightly coupled workstation clusters. The Netmsgserver [5] uses reboot notifications and liveness checks to deallocate port rights, but the method is susceptible to inconsistent views arising from network partitions.

A disadvantage of our approach is a lack of timely notifications. We trade quick detection of probable failure via timeouts for certain notification of reboot via incarnation numbers. This leaves the options for fast fault detection and recovery to the realm of higher-level protocols. We believe that the administrative shutdown mechanism provided is sufficiently powerful to support such improved fault detection without requiring any change to the underlying Mach NetIPC implementation. In any case, we believe that Mach NetIPC is the wrong level to implement sophisticated failure detection mechanisms and is certainly the wrong level to act upon notifications generated by such mechanisms.

2.3 RPC: The Key to Efficiency

Discussions with Mach IPC designers indicated the importance of the Remote Procedure Call (RPC) concept in overall Mach client-server efficiency. Therefore, one of our first design decisions was to put an RPC protocol at the heart of Mach NetIPC. Previous x-kernel work had led to an efficient RPC protocol based on Sprite RPC [9, 3, 2]. This was an obvious starting point for our design. The more complex problem, however, was how to match this to Mach IPC; the communication abstraction does not have a direct match to RPC semantics, and there is some latitude possible in choosing a mapping.

Mach messages that use a send-once right for the reply port fit well with RPC, partly because there is no persistent state for the reply port. The receiver of the message can reply using the send-once right, and if this is one within one processing context, no database entries need be made. Similar consideration led to limiting the RPC definition to messages that do not carry additional port rights. This makes it possible to define a close match between the x-kernel implementation of the protocol and the microkernel support for RPC messages. A single thread is used on the initiating side for the request/reply, and a single thread is used on the responding side.

Non-RPC messages are handled efficiently as uni-directional messages; an empty reply message unblocks the sending thread. An eventual reply using the Mach reply port is handled as a separate, unrelated RPC (with a null reply) going in the opposite direction. This unification of the non-RPC case with the RPC case simplified construction of the protocol graph and the Mach NetIPC program.

2.4 Port Right Transfers

The design goals for port right transfers were fourfold: a correct algorithm, minimum overhead for the most common transfers, a deterministic algorithm, and the avoidance of any centralized database that would complicate overall robustness.

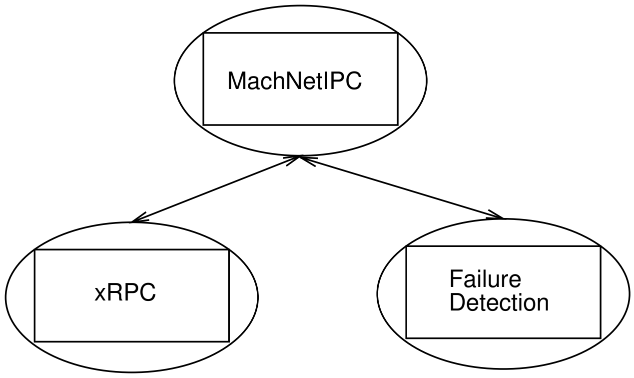

Figure 1: High-level view of protocol dependencies

A Mach port has one receive right and an unlimited number of send and send-once rights associated with it. These rights can be distributed arbitrarily across the network graph, and the method of distribution is to embed port rights in Mach messages sent between nodes. In most transfers, only the receiving node and the sending node participate in transferring the port right, and it is handled as a normal side-effect of message transfer. One exception occurs when an embedded send right has a receiver node that is neither the message sender nor the message receiver. Another occurs when an embedded receive right is associated with one or more sending nodes that are outside the message send/receiver pair. Each case invokes a multi-party protocol to keep all senders up-to-date on the current receiver for a port, and to ensure that the receiver knows all sending nodes for a port.

Moving a receive right involves coordination of the old and new receiver and notification of all senders of the new receiver address. Moving a send right involves only three parties: the old and new sender and the receiver.

3 Description of the Protocols

The design details of the protocols used in our Mach NetIPC implementation are presented in this section. This implementation consists of 15 protocols divided into three major groupings: Mach specific protocols (Mach NetIPC), failure detection protocols, and the x-kernel RPC protocols (xRPC). A coarse view of these groupings and their relationships is given in Figure 1.

3.1 Mach NetIPC

The Mach NetIPC implementation consists of four modules, each of which behaves as a protocol: the Mach message handler (MNIPC), the port manager (PORTM), and the port transfer protocols (RRX and SRX). These modules depend on the xRPC protocol group and failure detection mechanism for transport services and reboot notification, respectively. This section discusses the design of the first modules and how their control structure contributes to the overall efficiency of the mechanism.

The main function of this protocol group is to represent Mach messages in a form that carries enough information to be usable on a remote machine. The basic Mach message is augmented with a header that describes the sending machine architecture, the type of the message (RPC request, RPC reply, RPC-request-no-reply, or port management), a message sequence number (unique to the sending node), and the network representation of the Mach local and remote ports. The network port representation is globally unique, based on the IP address of the node that creates it. Simple Mach messages have this header prepended, and no other modification is made by the sender.

Complex messages—those containing embedded ports or out-of-line data—have a list of network port descriptors and a byte array consisting of the concatenation of all out-of-line data added to the

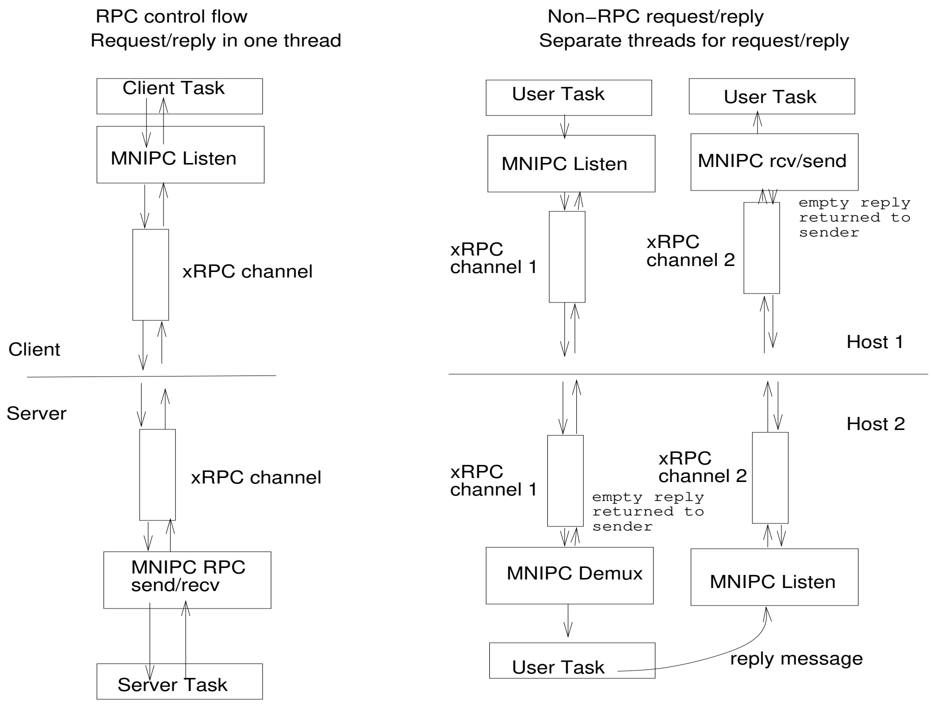

Figure 2: Matching Mach messages to *xRPC* sessions

Mach message. In addition, embedded port rights are replaced by their index in the network port list.

A major design goal was to get a close match between a Mach IPC port used for a Mach RPC (send/receive options) and the use of an xRPC channel. In fact, all actively used Mach ports that utilize our network services have a mapping to such a channel. The challenge was to remove as much overhead as possible from the main path, thus tying the performance of Mach NetIPC as closely as possible to that of xRPC. To do this, we needed to carefully define which Mach messages were acceptable as RPC's. The definition we have used is that a simple message with a send-once right as the reply port is an RPC that will receive "fast-path" treatment. These messages match the RPC semantics because the server side is stateless. That is, because the send-once right is consumed by the reply, the server side retains no state information that would have to be coordinated with the client.

When the Mach NetIPC module recognizes a message as an RPC request, it attaches a header marking it as such and sends it to the xRPC protocols using *xCall*, a blocking send/receive interface of the *x*-kernel. For this to preserve the sequencing semantics of Mach IPC, and for the reasons stated above, we require that the reply port of the Mach message specify a send-once right. When the *xCall* completes, the reply message is translated from network form to local form, and then delivered to the send-once right. This is illustrated in the lefthand side of Figure 2. In this diagram, a single thread is used to receive the Mach message and process it down through the protocol graph to the network; the same thread returns the reply. Similarly, on the server machine, a single thread receives the message from the network, delivers it to the server task, and returns the reply.

The send-once reply ports have a lightweight network representation that is simply a 64-bit identifier generated as a monotonic counter on the originating node. In theory, no identifier is needed because the reply is associated with the *xCall* reply message parameter, but not all RPC requests result in usable RPC replies, as explained below. No other database operations are made for send-once rights in RPC's in the normal case.

On the server side of the RPC, the request message is received for dispatching from xRPC by the

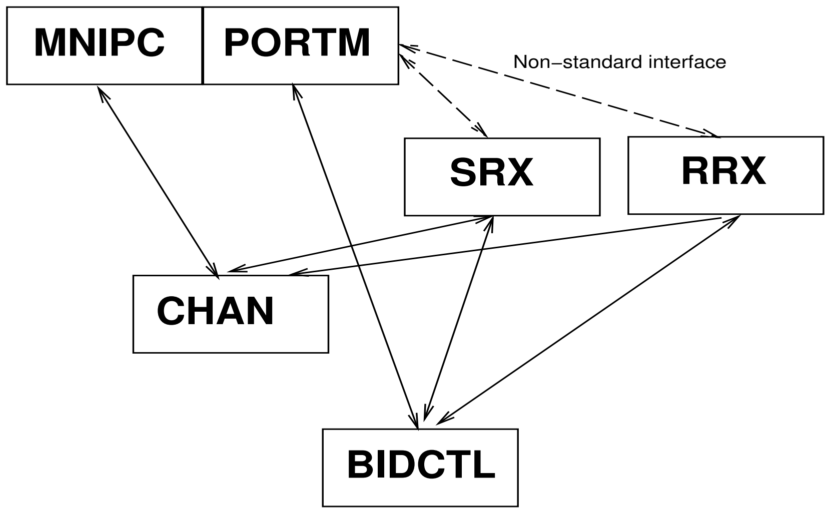

Figure 3: Mach NetIPC immediate protocol neighbors

xCallDemux x-kernel interface routine. A Mach port for the reply is assigned from a cache of pre-allocated ports. The request is translated from network form to local form, and it is delivered using the *mach_msg* options for RPC (MACH_MSG_RCV and MACH_MSG_SEND). This can complete either by a timeout or by a successful message receive. In the most efficient case, a non-complex Mach message is received with a reply port of MACH_PORT_NULL. It is translated to network form and sent to the network. This work is accomplished without any thread switches.

In the less efficient case, either a timeout occurs or the message is complex. In this case, an empty xRPC reply message is returned. The reply port is entered into the network port database, and, if a timeout has occurred, a thread is started to listen for the Mach reply. The originating side, upon receipt of the empty message, similarly enters the reply port into the network database. When the reply message arrives, it is matched against the database to locate the Mach reply port, and a *mach_msg* operation completes the transaction. This is illustrated in the righthand side of Figure 2. In this case, two threads are used on each node: one for the request, and one for the reply. In addition, separate channels in the xRPC protocol stack are used; an empty reply message completes the transaction in each direction.

When Mach NetIPC receives a Mach message that is not "RPC-style," it assigns an xRPC channel for the remote Mach port, attaches a non-RPC marker to the header, sends the message, and does not look at the reply, because it should be empty. The empty reply serves as acknowledgement of message receipt, allowing any locked resources to be released. This is also illustrated by each half of the righthand side of Figure 2.

Our initial implementation did not use the lightweight representation of send-once rights, and the request and reply were treated as separate transactions. In this case, a port database entry was made for each send-once right, and a separate thread handled the reply. The result was that tests using send-once rights used 25% more time than tests using send rights. This necessitated the implementation of a lazier approach to send-once rights, which greatly improved performance. However, it complicated what was an otherwise straightforward implementation.

Mach NetIPC has two modules that are invoked for every message: one that handles the semantics of the data portion of Mach messages and one that maintains the port database. The latter module is an impostor x-kernel protocol: during initialization it replicates the Mach NetIPC object structure and changes its name. Thereafter, only messages or transactions involving the network representation of Mach port rights will go through the second module, known as PORTM, the port maintenance module. The first Mach NetIPC module, the data module, communicates with *portm* via a set of subroutine calls for creating, modifying, and deleting network port rights. When a port manipulation involves more than two machines, PORTM will use the SRX or RRX protocol to transfer the send or receive right, respectively. These four protocols and their relationships are illustrated in Figure 3.

The port maintenance module supports port locking (to limit port mobility while messages are in transit), no-more-senders, and port death. Port death occurs either when a port is destroyed or deallocated on a functioning node, or when a node holding a receive right fails.

To support no-more-senders notifications, a receiving node keeps a per port list of the addresses of all sending nodes. In this list, a "network make-send" count is maintained for each sender. This count reflects the number of send rights to the port that have been transferred to that particular node. When a sending node reports a no-more-senders condition for a port, it includes its "network make-send" count. The receiving node can compare the count in the message with the count in its database to discard stale messages with lower counts. The synchronization in the port transfer protocols can result in such stale no-more-senders messages, even with an ordered network. When all sending nodes have reported no-more-senders, and the send-once count is at zero, the receiver can destroy the port. This is, of course, a re-enactment of the accounting done by the local microkernel.

This organization suggests that the port maintenance module could be a separate protocol, and this is true but for one caveat. The port maintenance module performs operations on Mach port right names, and these must be the same names used by MNIPC. Therefore, the two modules must coexist in the same Mach task.

3.2 Transferring Port Rights: SRX and RRX

The port transfer protocols are send-right-transfer (SRX) and receive-right-transfer (RRX). They work in concert with PORTM to assure that the holder of a send right always knows the address of the receiver, and the holder of a receive right always knows the addresses of the senders.

When the port right is being transferred between two machines, and these machines constitute both the receiver and the only sender, then the port transfer protocols are not invoked. But when three or more machines are involved, the transfer protocols are needed to ensure that a consistent view exists between all parties. The transfer protocols achieve this by keeping the port locked (*i.e.*, unavailable for sending messages), until all necessary parties are either informed of the transfer or have rebooted.

Multi-party transfers are initiated and directed by the node holding the right to be transferred. At any point in the transfer protocol, the right is either clearly identified with exactly one node (the old or new node), or with *either* the old or new. In the first case, a reboot of the owner node destroys the right in transit. In the latter case, a reboot of *either* node destroys the right. If the right is a send right, a reboot of the receiver node will also destroy the right. This protocol should be amenable to analysis for correctness with respect to node failure because the state of the right is never strictly dependent on more than two nodes at a time.

Port rights that are transferred using these protocols arrive at their destination and are entered in the database, but they are not fully activated until a Mach message containing a reference to the right is delivered to the machine. When the message arrives, the port manager assumes responsibility for watching the health of the remote senders or receivers, and it informs the port transfer protocols that the message has arrived. Because several messages carrying send rights to the same port may be in transit to a machine, the port transfer protocols have responsibility for the rights until all such messages have arrived. This is why a message identifier is needed in the network header.

The port transfer protocols use the standard x-kernel interface for utilizing the transport protocols. However, the interface to the PORTM module is non-standard. This is because the port transfer protocols notify the PORTM module about changes in port states, and these changes are outside the normal sort of data delivery that protocols use.

3.3 Reboot Detection: BootId and BootId Control

As described previously, machine reboots define the notion of failure in our system. Reboots are detected by changes in the incarnation number associated with a machine. Unique identifiers for the incarnation number of a node are handled through two protocols: BootId (BID) and BootId Control (BIDCTL). The BootId protocol is tightly integrated with the xRPC protocol subgraph. It is responsible for detecting boot identifier mismatches. Mismatches occur when a message arrives with an unexpected boot identifier for either the local or remote machine. The BootId Control protocol handles generation and maintenance of the boot identifier database, probing for changes in boot identifiers, notifying local protocols about reboots, and accepting requests from local protocols to enter or remove a remote machine from the database.

When a node reboots, it changes its incarnation number and broadcasts the new number. Communicants receiving this broadcast request confirmation of the reboot using a nonce to protect against delayed or spurious broadcasts. Upon receiving confirmation, Bootid Control informs interested local protocols of the reboot. Communicants that do not receive the broadcast will learn of the change when they attempt to communicate with the rebooted node, because the BootId protocol will reject traffic that uses the old number and will initiate the confirmation process for the new number. Note that the initial broadcast message accelerates reboot detection for local machines, but the correctness of the algorithm does not depend on it.

Guaranteeing that an incarnation number is not re-used is important to this design. We are using the system time to generate the number, but this is slightly unsatisfactory because clocks can be inaccurate (even "stuck", especially during the boot procedure). This problem can be ameliorated by storing the previous incarnation id in some form of non-volatile memory, but ideally a true random number generator device would be available.

3.4 The RPC Protocol Subgraph: xRPC

Several protocols in a complex configuration comprise the protocol grouping that we call xRPC, the transport service of Mach NetIPC. The graph that we have used during our development and testing is given in Figure 4. The protocols are described in this and the following sections. The transport service is designed to provide efficient delivery in ethernet and inter-network topologies. While it is specially designed to support RPC interactions, it is also appropriate for uni-directional streams.

3.4.1 RPC Transport Service: CHAN

The CHAN protocol is the RPC transport protocol that directly underlies Mach NetIPC. It has the desirable properties of:

- Multiple client-server channels (a pair of nodes can have many simultaneous channels between them)
- Acknowledgement of a request is packaged with the reply data
- Duplicate message elimination ("at most once" semantics)
- Data sequencing on a channel matches Mach sequenced message queues
- Minimal state required for each channel
- Message size up to $2^{32} - 1$ bytes (Note that lower protocols may not support the full size)

The client (or initiating) side supports an *open* operation by assigning a new channel identifier and initializing the state. It also registers interest in the destination with the BootId Control protocol. When a message is sent over the channel, CHAN enters the sending state and expects either an acknowledgement or a reply (data plus ack). If neither arrives in a timely manner, a request for an ack will be sent. If the message is small, it will be retransmitted along with the request for an ack.

The server (or remote) side accepts the first message on a new channel by registering interest in the sender with BootId Control and by initializing its state. The message is delivered "upward", and the protocol enters the state of waiting for a reply from the server.

When the reply is eventually sent from the server back to the client, the server expects an ack from the client. A symmetric request for ack from the client is carried out if a new request (implicitly

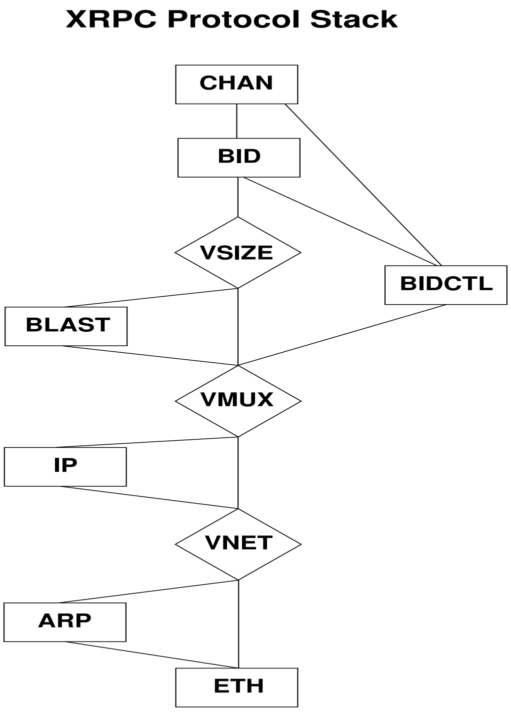

Figure 4: The transport and reliability protocols of Mach NetIPC

acking the reply) is not received in a timely manner. If an explicit ack (with no new request) is received by the server, the channel enters an inactive state.

Inactive channels are persistent objects: they remain available for reuse between the same nodes. This minimizes the overhead of starting up a channel and eliminates the complexity necessary to reliably tear down a channel (cf TCP). Channel objects are destroyed upon notification of a peer rebooting.

3.4.2 Virtual Protocols: VSIZE, VMUX, and VNET

Below BootId and CHAN, a series of *virtual protocols* [7] are used to select the appropriate transport protocols, based upon various properties of the message or connection. The first virtual protocol, VSIZE, dynamically directs outgoing messages to either VMUX or BLAST depending upon the size of the message. At connection open time, VSIZE opens both VMUX and BLAST connections and determines the optimal packet size of the VMUX connection. In the graph in Figure 4, the optimal size for VMUX is the Ethernet maximum packet size. VSIZE will send short packets which do not need fragmentation directly to VMUX, avoiding the BLAST protocol. BLAST itself implements a simple blast algorithm with selective retransmission for sending large packets between nodes. This algorithm is tuned for the case where both machines are on the same local net. Note that since BLAST has a maximum packet size it determines the maximum packet size supported by this implementation of the Mach NetIPC (currently 32K bytes).

The second virtual protocol, VMUX, bypasses the IP protocol in cases where the target machine is reachable on the local area network. The advantages of using IP are well-known: nearly a million other hosts are potentially accessible on the Internet. However, the IP service is unnecessary for connections between hosts on the same local network. When a VMUX connection is opened, VMUX attempts to open VNET. If that open succeeds, the target host is accessible on the local area network and all data packets are directed to VNET, bypassing IP. If the VNET *open* fails, an *open* is performed on IP and all data packets are directed to IP.

The third virtual protocol, VNET, isolates the rest of the protocol graph from the existence of multiple physical network links. VNET is configured with ⟨physical network, address resolution protocol⟩ pairs. VNET is opened using the IP address of the remote host, which it attempts to match with one of the physical networks below it using IP network masks. If it finds a match, VNET opens a connection with the appropriate physical network. In the configuration shown in Figure 4 there is only one physical network (ETH) and one address resolution protocol (ARP).

The graph presented in Figure 4 has three limitations. First it limits the maximum size of a Mach IPC message. This is obviously unacceptable. Second, BLAST lacks any form of congestion control and hence is not well suited to sending large messages over the Internet. Third, it does not provide efficient support in situations where a user sends several non-RPC messages to the same port. All of these problems can be solved by including a stream oriented protocol such as TCP in the graph. We are currently working on a new graph in which TCP will be used for larger messages than BLAST will support, any large message over the Internet, and situations where the Mach IPC protocol has detected that the use of a port would benefit from streaming. Except for the detection of streaming, the addition of TCP should not require any modifications to the higher level protocols.

3.5 Locality Issues

All of our code which is concerned with manipulation of Mach IPC data structures (data conversion, port allocation, mach_msg operations, etc.) resides in a user task, much like the Mach Netmsgserver. There are two problems that derive from this. One is that send-once rights cannot be matched with their corresponding receive rights. This leads to a "ghosting" state in moving receive rights, wherein the old receiver waits for local kernel notification that all send-once rights associated with the port are used. Only when this occurs can it can stop forwarding messages sent via send-once rights to the new receiver. A similar situation occurs when a send-once right is moved between nodes twice; in this case the right acquires two global names, both of which will be consumed in sequence when the right is used to send a message. This is an inefficient way to use a right, although such a circumstance is reportedly rare.

When moving a receive right, much of the port state must be retrieved from the microkernel via interface calls, and the queue of messages must be completely transferred to the new node before allowing additional enqueuing. Within the microkernel, the port state could be accessed more quickly, and the queue could be controlled more directly by blocking active senders.

4 Performance Results

With all protocols located in user space, the RPC roundtrip time for a 50 byte simple reply/request operation, using a standard Mach kernel running on a 25MHz DecStation 5000 Model 200, is 6.3 milliseconds. Of this time, the overhead due to the Mach NetIPC module alone, is 2.3 milliseconds. This is about twice as high as expected, based on our experiences in implementing other protocols, and we expect to reduce the number to about a millisecond. In contrast, the same test program, when run with the Netmsgserver, has a 30 millisecond roundtrip cost.

When the xRPC and the failure detection protocols reside in the Mach microkernel, the RPC roundtrip time is reduced to 4.8 milliseconds. The time saving over the case where all protocols reside in user space is largely due to avoiding the Mach device interface to the ethernet driver. For longer messages, up to the network maximum transmission unit (MTU), the time is expected to increase linearly. When the MTU is exceeded, the partitioned interface should demonstrate a more pronounced advantage over the "all-user-space" implementation, due to a reduced number of Mach IPC calls across the user/microkernel boundary. For the partitioned configuration, the cost of the crossing to the microkernel to send and receive data from the lower-level protocols is estimated at 16% of the total roundtrip time.

Of the time spent in Mach NetIPC, approximately 10% is due to Mach interface calls to determine the type of reply right being used, to manipulate reference counts for ports, and to request/field Mach notifications. We estimate that another 15% is spent in lock operations; most x-kernel operations require acquisition of the master lock to protect common resources, and send rights are protected by locking while messages are in transit through those rights. At least another 20% is estimated to be due to overhead of the cthreads package: scheduling and activating a thread in the client and server.

The NORMA implementation cites 2.5 milliseconds for RPC roundtrip time [1] for i486 PC's. While it is difficult to compare expected performance between the PC and a DecStation, we expect the DecStation to be at least as fast. For comparison, we note that the NORMA code is entirely resident in the microkernel, and we estimate that .8 milliseconds of our roundtrip cost would be eliminated if our Mach NetIPC code were similarly resident. This still indicates Mach NetIPC would have best case 4.0 millisecond roundtrip time, for 50 byte messages. Our code is not yet fully optimized, and we would expect to be fully comparable with the NORMA implementation within the coming months. This is based on observing that our transport service takes well under 2 milliseconds for 50 byte messages and that the processing required for simple messages is not high.

A detailed timing breakdown for the partitioned configuration is given in Figure 5. Each box shows the transit time used in an RPC between two machines. Where two times are given, the first represents the time used in the "down" direction, and the second represents the "up" direction. The RPC times for a Mach RPC and a CHAN RPC, using 50 bytes messages, are shown at the side.

The time for processing and dispatching an ethernet packet interrupt were obtained by indirect measurement techniques, but the others were measured by using tests that removed one processing layer at a time, running hundreds or thousands of message per test session. The Mach NetIPC time is shown as 1.1 milliseconds for the initiating side, and 1.2 for the replying side. This division is a guess; the server side of most protocols does more work, because it must associate the incoming message with state information for the session. For Mach NetIPC this is a small percentage of the overall time.

5 Proposed Mach IPC Changes

Most of our work in achieving an efficient mechanism have been tied to the concept of "fast-path" messages, but the time penalty for avoiding this pathway can be high. Things that cause concern are operations that require touching many bytes of the message, operations like copying, transforming, checking. We made little attempt to optimize these operations, because where they are required they are inherently expensive. These concerns have lead us to suggest three changes to the Mach 3.0 IPC specification: data streams, ordered messages, and untyped data. These changes have been tentatively accepted in the ongoing process of creating a new common Mach interface specification [6]. Note that the software described earlier in this paper implements the original Mach 3.0 IPC definition and does not reflect these changes.

The proposed changes all relate the to the Mach message format. The current Mach message format is a flat unordered series of tagged data items. The supported base type tags are: unstructured, bit, boolean, integer_16, integer_32, char, byte, integer_8, real, string, string_c and several port right tags. The Mach message format also supports one dimensional arrays of these base types and allows any tagged item to be marked as out-of-line data. Normally a message sender creates a message template with tags; each time the template is used for sending, the data to be sent is copied into the template buffer after the appropriate tag. The tags provide enough information to allow the local and distributed Mach IPC implementations to perform the correct actions on embedded ports and out-of-line data segments [1]. The tags also provide enough information to allow the distributed Mach IPC implementation to perform the appropriate conversion on the user data (such as integers) when the data representation of the sending and receiving machine differ.

Avoiding copying is the grail which all network specialists seek. Header additions/deletions, fragmentation/reassembly, and format reconstitution usually result in copying. The *x*-kernel internal libraries are constructed to help avoid copying, but it has been difficult for us to match this to Mach messages, which must be contiguous or else use the out-of-line option. The first proposed modification to Mach IPC is to add data streaming: a third method of passing user data in a Mach message. As currently defined user data in a Mach message must either be copied into the

[1] Out-of-line data is "eagerly" copied between nodes, *i.e.* no cross-node memory objects are created.

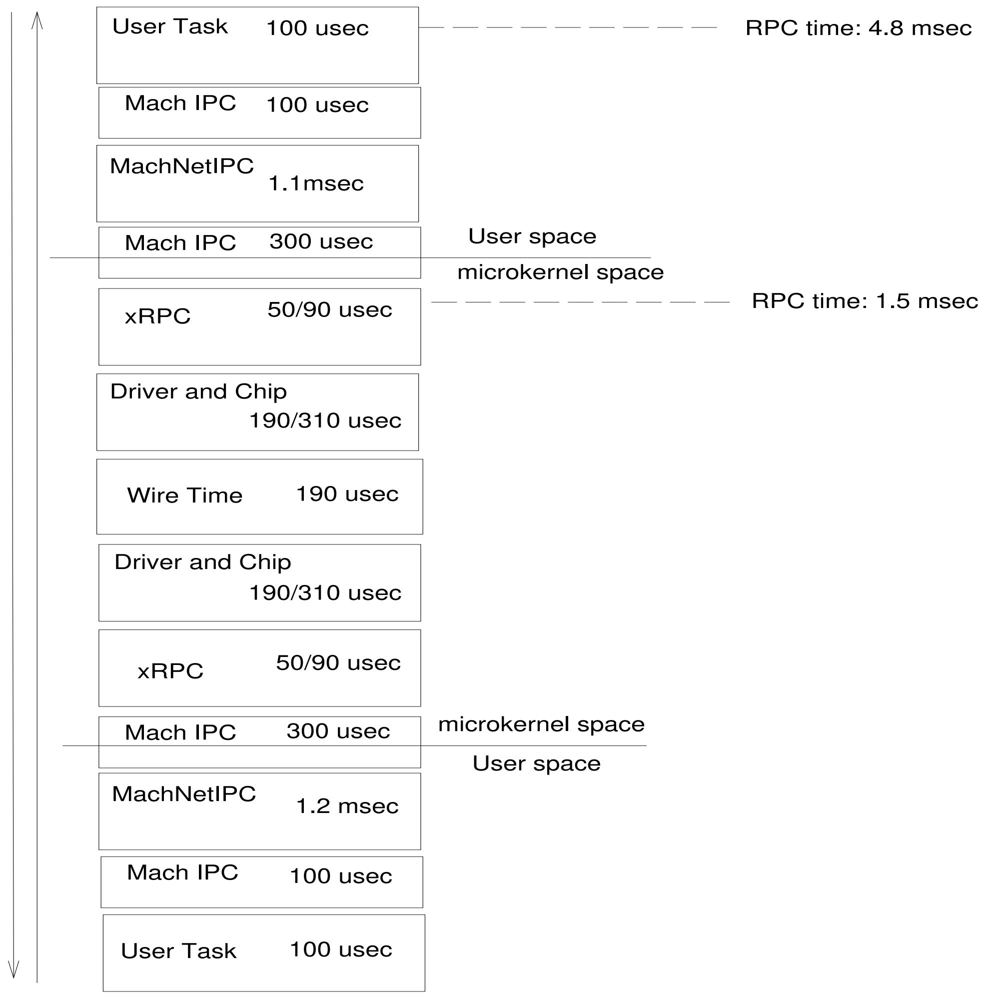

Figure 5: Timing model, protocols split between microkernel and user space

Mach message template —incurring an extra copy—-or passed as out-of-line data, in which case only the data address is written into the message template, but copy-on-write semantics will be used to transfer the data—incurring the copy-on-write overhead. Given that the break-even point for data transferred using copy-on-write is large (on the order of 4000 bytes), we propose that the Mach message format be modified to allow out-of-line data which is eagerly copied by the Mach IPC implementation rather than transferred using copy-on-write.

Another concern has been the expense of transferring ports embedded in a Mach message. The ports in the header, the remote and local ports, are handled through our "fast-path" mechanisms, but others must be located in the message and changed to a network representation. The second proposed modification to Mach IPC is to improve the data organization in the Mach message. The existing Mach message format allows an arbitrary sequence of tags. Ports and out-of-line data objects may appear anywhere in the message mixed in with typical user data such as integers and character strings. Hence, unless the user marks the message as simple, the Mach NetIPC implementation must interpret the entire Mach message, examining each tag for ports and out-of-line data segments. This expensive process must always be done for non-simple messages at the sending node, and it must be repeated at the receiving node. The sending node adds a list of network port descriptors and a list of out-of-line data arrays to each complex message. These must be integrated into the message at the receiving node. This processing can be minimized by ordering the Mach message. In the new proposed format all ports are at the beginning of the message, followed by all out-of-line data segments and streams, followed by user data.

Our final concern is that Mach messages have typed data that must be translated to the target machine architecture if they are transferred between machines with differing architectures. It is our contention that the proper place for the data transformation is at the point where the data will be used. The data typing is a matter of convention between the sender and receiver, and we believe that Mach IPC should be typeless except for ports. Thus, the third proposed modification to Mach IPC is move the responsibility for the marshaling and de-marshaling of user data from the Mach IPC mechanism to the message generator (MIG). This is accomplished by eliminating the base type tags that require conversion (bit, boolean, integer_16, integer_32, char, byte, integer_8, real, string, string_c) and only supporting the type tag *unstructured* and the various port right tags. As currently proposed, the sending MIG stub will be responsible for affixing a NIDL-like [8] architecture tag to data; the tag will be checked by the receiving MIG stub before unbundling the Mach message. This removes data representation from the responsibility of Mach NetIPC and makes it a strictly end-to-end issue. In particular, this change will not require that data items be marshaled when the sender and receiver are on the same machine.

When the effects of these three proposed modifications are combined, the resulting Mach IPC implementation should be significantly simpler and faster overall. Applications can avoid a copy on data items too small to take advantage of copy-on-write. The need to repeatedly scan messages looking for ports is eliminated and it is also much easier to tell if a specific message does or does not contain additional ports. User data is now treated as an undifferentiated byte stream which is neither examined nor modified by either the local IPC or Mach NetIPC implementation. The elimination of user type tags significantly simplifies the Mach NetIPC implementation and eliminates one copy in cases where data is sent between machines with different data representations. Finally the formatting of user data is now completely the responsibility of MIG which will make extending or changing the currently restricted user data representation format much simpler.

6 Conclusions and Further Work

This work demonstrates that modular protocol design can be used to decompose the Mach IPC mechanism into efficient protocol units. It is possible to operate over either LAN's or internetwork configurations without any additional coding. Placing the transport protocols into the Mach kernel yields a significant performance benefit, a fact that has been previously noted with respect to the Mach 2.5 kernel with TCP/IP [1]. The timing breakdown indicates that moving the port manipulation functions into the Mach kernel would make a large difference.

This implementation does not define the legal participants of a communicating network. It is likely that administrators would like to limit the communication to selected node lists and/or subnets. This could be achieved by adding an authorization protocol to be used in conjunction with the port transfer protocol.

Cryptographic methods for assuring message integrity and privacy are being developed as generic protocol components in separate research. Digital signatures modules developed as part of this work could be used to support the security of the authorization protocol.

For machines with special-purpose communication hardware, our approach is still valid. The xRPC stack could be replaced by any code that obeys the x-kernel interface conventions and utilizes the message library and thread locking conventions. The overhead of the x-kernel interface operations is already low, and it should be an acceptable cost for communication-intensive operations that work through Mach IPC.

Finally, an x-kernel that distributes processing among tightly coupled processors is an open issue. The current implementation is MP-safe in the trivial sense that if the master lock acquisition is honored, the x-kernel threads can be distributed to separate processors. To achieve a finer processing granularity, much of the locking must be reworked. This reworking would have to include the issue of locks within the Mach NetIPC database, as well.

7 Acknowledgement

Richard Schroeppel did the detailed timing breakdowns and improved the ethernet device driver, making it possible to get accurate latency measurements for RPC's.

References

[1] J. S. Barrera. A fast Mach network IPC implementation. In *Proceedings of the Usenix Mach Symposium*, pages 1–12, 2560 Ninth Street, Suite 215, Berkeley CA 94710, November 1991. Usenix Association.

[2] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984.

[3] N. Hutchinson, L. Peterson, S. O'Malley, E. Menze, and H. Orman. *The x-Kernel Programmer's Manual (version 3.2)*. Computer Science Department, University of Arizona, Tucson, Arizona, January 1992.

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

[5] D. P. Julin. Network server protocol specification (draft). Computer Science Department, CMU, August 1989.

[6] K. Loepere. Mach 3 kernel principles, draft proposed specification. Open Software Foundation, July 28 1992.

[7] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[8] Network computing architecture, version 2.0 specifications. Open Software Foundation, September 25 1992.

[9] J. K. Ousterhout *et al.* The Sprite network operating system. *IEEE Computer*, 1988.

[10] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983.