NORMA IPC:

A Task-to-Task Communication System for Multicomputer Systems[1]

DRAFT 10/20/93 (*technical review still required*)

Bill Bryant (billb@osf.org)
Alan Langerman (alanl@osf.org)
Steve Sears (sjs@osf.org)
David Black (dlb@osf.org)
OSF Research Institute

Abstract

NORMA IPC is an in-kernel subsystem designed specifically to support task-to-task communication in multicomputer environments. The subsystem extends the Mach IPC subsystem so that tasks on separate multicomputer nodes may pass messages to one another with the same semantics that are supported by the standard IPC subsystem. This document provides a high-level description of the NORMA IPC subsystem as it is implemented in the NORMA 14 version of the OSF/1 AD operating system.

1 Introduction

Mach is a message-based operating system in which executing programs interact with their environments by sending messages to and receiving messages from other programs. The kernel subsystem that manages these interactions is referred to as Mach IPC.

Originally, Mach IPC provided communication between programs executing on the same host. Communication between programs on separate hosts was implemented via user-level services such as the NetMessage Server. Early versions of the operating system did not provide an in-kernel mechanism for host-to-host communication.

As part of an effort to extend Mach to non-shared memory multiprocessor platforms, Joseph Barrera, then at Carnegie Mellon University, designed and implemented NORMA IPC [1]. (NORMA is an acronym for No Remote Memory Access.) NORMA IPC replaces the relatively slow user-level network IPC systems with a faster kernel-based network IPC that is more in keeping with the high throughput and very low latency provided by multicomputers such as the Intel Paragon.

NORMA IPC extends the Mach IPC subsystem so that tasks on separate multicomputer nodes may pass messages to one another with the same semantics that are supported by the standard IPC sub-

[1]This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

system. This document describes NORMA IPC as implemented in NORMA 14 version of the OSF/1 AD operating system. In this version of Mach, IPC messages are still typed; NORMA will eventually be converted to support untyped Mach IPC.

Before we begin, it should be noted that the NORMA IPC subsystem is not an easy one to describe or understand; any adequate discussion of the subsystem must include a description of the standard Mach IPC subsystem, certain aspects of the Mach Virtual Memory Management subsystem, as well as the Mach External Memory Management model. These topics will be introduced as required.

This document uses the following conventions:

- Although the term NORMA is an acronym for NO Remote Memory Access, in this document, we use NORMA as short hand for NORMA IPC.

- We frequently refer to Mach tasks as though they are active entities; this is merely a convention. When we say that such and such a task is performing an operation, we mean that a thread within that task is performing the operation.

2 Standard Mach 3 IPC Features and Semantics

The standard IPC subsystem implements message-passing between tasks via the following entities: *ports*, *port rights*, and *messages*.

A port is a communication channel that is protected by the kernel; each port is implemented as a message queue that resides in the kernel's address space. When a task sends a message to a port, the kernel copies the message from the task's address space into the kernel's address space and places the message at the tail of the port's message queue. When a task invokes a receive operation on a port, the kernel removes the message that is at the head of the message queue and copies it into the task's address space.

A task can use a given port only if it has a right to do so; a task must hold a *send right* to a port to send messages to the port and a *receive right* to a port to receive messages from the port. Each task has a private area in which the kernel stores the task's port rights. This private area is referred to as the task's *port name space*.

Because ports are protected by the kernel, they provide a secure mechanism for naming kernel-implemented and user-implemented objects; consequently, the IPC subsystem is often used to implement client-server systems. In such systems, a server program creates and manages objects on behalf of its clients, and ports are used to represent the objects. Typically, a client invokes operations on an object by sending a message to the server on the port that represents the object. These interactions are referred to as *remote procedure calls* (RPCs).

The Mach kernel is itself a server that creates and manages objects such as threads and tasks. Each instantiation of these objects is represented by a port, and the system calls that manipulate the objects are implemented as RPCs [2].

In addition to implementing ports, the IPC subsystem also implements entities known as *port sets*. A port set is a group of ports that can be treated as a single unit; when a task receives a message from a port set, it does not know which port the message was delivered on. We will not discuss port sets further in this document.

2.1 Supported IPC Operations

In Mach 3.0, tasks generally invoke message-passing operations by calling the **mach_msg( )** interface. The IPC subsystem supports the following message-passing operations:

- A task may send a message without expecting a reply
- A task may send a message and wait for a reply. This type of operation can be used to implement synchronous RPCs.
- A task may expect a reply when sending a message, but not wait for the reply. This operation can be used to implement asynchronous RPCs.

2.2 Port Rights and Associated Semantics

Ports are accessed via *port rights*. Only one task may receive messages from a port; multiple tasks may send to a port. A port's *receive right* may be migrated to another task.

In addition to *send rights*, which allow a task to send an arbitrary number of messages to a port, the IPC subsystem provides *send-once* rights, which allow a task to send at most one message to a port. A task typically issues send-once rights to another task so that task may

- Reply to an RPC
- Send notification of an event.

For the most part, a task acquires send rights to ports by receiving them in messages. A task does inherit a small set of send rights to special ports from its parent task when it is created, including a send rights to the system's name server; a task may use this send right to acquire, via the name server, send rights to other server programs.

Port rights may be migrated between tasks. When a task *moves* a port right, it loses its right to the port; a task may move send rights and receive rights. When a task *copies* a port right, it retains its right to the port. Only send rights may be copied; receive rights cannot be copied because only one task may hold a port's receive right at any one time. The holder of a receive right can generate send or send once rights. The holder of a send right can generate new send rights; only the receive right holder may generate send-once rights.

When a task creates a send-once right, the kernel guarantees that the task will receive a message associated with the right: either a message from the task to which the right was sent, or, if the right is destroyed before it is used, the kernel sends the receiving task a send-once notification indicating that the send-once right has been destroyed.

A port's receiver may be interested in whether or not there are any send rights to the port held by other tasks, and it may choose to deallocate the port if there are no such rights being held. The kernel tracks existing send rights and send-once rights to a port. A port's receiver may request a NO-SENDERS notification; when the last send right has been deallocated, the kernel sends a NO-SENDERS message to the receiving task. Note that this notification refers only to send rights, not send-once rights. A task may receive a no senders notification while there are still active send-once rights. Note also that the kernel cancels a no-senders notification request if the port's receive rights are transferred to another task.

Sections XXX and XXX describe how NORMA implements port right migration and how it tracks send rights across the NORMA domain.

2.3 Messages

Mach places no restrictions on the type of data or amount of data that can be transmitted in a message. A message may contain its data within the body of the message; messages of this type are referred to as *inline* messages. There is a limit to the amount of data a message can transmit inline.

A message may also transmit data *out-of-line* by passing pointers to the data instead of passing the data itself. In this way, a task may send large ranges of its virtual address space to another task. A message may contain both inline and out-of-line data. And, as mentioned above, messages may be used to transmit port rights between tasks.

Each message consists of a message header and a message body. The message header contains the task's right for the port that the message is being sent on. The kernel uses this right to identify the message's destination. If the sending task expects a reply to the message, the message header may include a send right or a send-once right that the receiving task can use to send back the reply.

The message header also indicates whether or not the message body contains out-of-line and/or port rights. If the message body contains such data, the kernel must verify and translate the data before queueing the message on the destination port. All pointers to out-of-line data must be checked to ensure that they point to valid address space and all port rights must be checked to ensure that they represent valid ports.

When a user task invokes a message sending operation, the IPC subsystem copies the message from the task's address space to the kernel's address space. This operation is referred to as *message copyin*. It is during the copyin operation that the kernel verifies any port rights and/or out-of-line data pointers associated with the message.

The in-kernel version of a message is referred to as a *kmsg*. When a task invokes a receive operation on the port, the IPC subsystem copies the kmsg from the kernel's address space to the receiving task's address space. This operation is referred to as *message copyout*. If a message contains port rights, the IPC subsystem inserts the rights into the receiving task's port name space during the copyout. Out-of-line data is also inserted into the receiving task's address space during the copyout.

2.4 Message Ordering

Perhaps the most important semantic of Mach IPC is its message ordering guarantee; the subsystem ensures that if a thread within a task sends a sequence of messages to a particular port, that port's receiving task will receive the messages in the order those messages were sent. Note however, that this guarantee holds only for messages sent from a single thread. Mach makes no guarantee about the order in which a task receives messages from multiple threads that are sending messages at the same time.

3 Memory Managers

In Mach, a task's address space is implemented by an address map that maps *memory objects* to virtual address ranges. Each memory object represents a set of contiguous virtual pages. Memory objects are managed by separate *memory manager* tasks.

A given memory manager is responsible for performing pagein and pageout operations on the memory objects it manages. For example, when a task references a virtual page that is not resident, the kernel resolves the page fault by issuing a request to the appropriate memory manager; the memory manager then furnishes the page. If, at a later time, the page's data is modified and the kernel's page replacement algorithm targets the physical page for reallocation, the kernel issues a pageout request to the memory manager; the memory manager then writes the page to secondary storage.

Memory objects are represented by ports and the interactions between the kernel and the memory managers are implemented as RPCs. The kernel requests a pagein by sending a message to the data's memory manager via the port that represents the data's memory object. The memory manager furnishes the page by sending it in a message to a port that the kernel holds a receive right to.

Memory managers are separate user-level tasks and the kernel cannot depend on them to respond to pageout requests in a timely manner. This situation is especially problematic when the kernel runs low on memory and must reallocate physical pages quickly. To prevent the system from failing due to resource shortage, the kernel uses a trusted *default pager* to pageout data that, for one reason or another, is not being paged out by its memory manager. The default pager is also responsible for managing data contained in anonymous regions of memory. This includes data contained in user stacks and in regions of dynamically allocated memory.

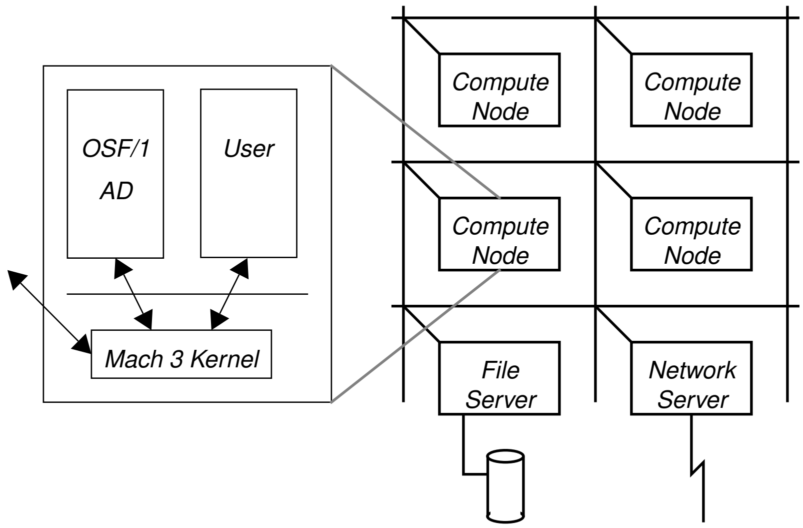
4 The Role of NORMA IPC

The NORMA IPC subsystem is a key component in the extension of the Mach operating system to multicomputer environments. As mentioned above, standard Mach IPC provides naming, synchronization, security and communications abstractions for the rest of the Mach kernel. NORMA IPC extends Mach IPC abstractions and semantics across multiple nodes, in effect providing a single IPC space to multicomputer environments. The extension of Mach IPC to all nodes of a multicomputer is a primary requirement for the success of Mach as a multicomputer operating system. [3].

NORMA IPC is in use in various locations. The OSF Research Institute's OSF/1 AD operating system uses NORMA to implement a distributed version of UNIX that provides a single-system image across all the nodes of the underlying multicomputer. Other research organizations use NORMA in research projects in distributed applications, process migration, and load balancing.

4.1 NORMA IPC and OSF/1 AD

The architecture of OSF/1 AD is derived directly from the target architecture, in which large numbers of nodes are connected via a high-speed message-passing interconnect. Each node is a full-fledged general-purpose processor with substantial memory. A subset of the machine's nodes have peripheral devices attached to them. The Mach microkernel runs on all of the nodes, providing generic task/thread management, memory management, communication services, and access to devices. Specialized user-space servers provide UNIX functionality such as file service, process management, and networking. Disk and networking devices are managed by servers that are typically co-located on the node attached to the device [4]. See Figure XXX.



As indicated in the figure, a user-level task running on a compute node must interact with other nodes to perform I/O. Also, compute node kernels must interact with server nodes to perform paging operations. NORMA IPC is the medium through which these interactions occur. It is not an overstatement to say that NORMA IPC is fundamental to the performance and correctness of OSF/1 AD. [3].

5 NORMA Port Name Space

In the current implementation, NORMA IPC constructs a “domain” during system initialization. The domain consists of all the nodes within the multicomputer. Within this domain, each node has a specific number that identifies it.

A port becomes a NORMA port only when rights to it have been migrated to another node in the NORMA domain. For example, when a task residing on Node A passes send rights to a port to a task residing on Node B, that port becomes a NORMA port.

Each NORMA port is identified by a unique number known as its *uid.* The high order bits in this number identify the port’s *root* node, the node on which the port was originally created. A uid’s low order bits specify the port’s id on the root node.

Each kernel maintains a table that names all NORMA ports it is aware of. This table does not necessarily name all currently active NORMA ports in the domain.

In standard Mach IPC, a port is a data structure that includes a queue containing messages that have been sent to the port and are pending delivery. In NORMA, a port is represented by such a data structure on all nodes that have rights for the port. On the node that contains the port’s receive right, the data structure is referred to as the port’s *principal*. On a node that holds one or more send rights to the port, the data structure is referred to as a *proxy*.

6 Overview of a NORMA IPC Send and Receive Operation

The following paragraphs provide an overview description of what happens when task A on node A sends a message to task B on node B. For the sake of argument, assume that the message body contains in-line data and that the message header and the message body together have a length of four pages. Assume the message contains simple data; that is, it does not include port rights or pointers to out-of-line data.

1. Task A composes the message and issues a call to **mach_msg( )**.

2. The call traps into the kernel. The kernel copies the message from task A’s address space into the kernel’s address space.

3. The kernel determines that the message is destined for a remote port and invokes the NORMA IPC entry point.

4. NORMA converts the message to NORMA format. Section XXX describes the NORMA format.

5. NORMA sends the message to the receiving node a page at a time, waiting for the receiving node to acknowledge receipt of the each page before sending the next page of the sequence. Since the message is contained in four pages, it will be transmitted in four pieces.

6. The receive side collects the pieces as they come across. When all the pieces are received, NORMA places the message on the destination port’s message queue.

7. When the receiving task (task B) is ready to receive the message, it issues a call to **mach_msg( )**; NORMA converts the message from NORMA format to standard IPC format, then the IPC subsystem copies the message from the kernel’s address space into the address space of task B.

7 NORMA Packets

NORMA transmits messages in units referred to as *packets*. The use of this term in NORMA is somewhat confusing it is a term that is already in use with respect to networking systems. In NORMA, a packet is a unit of data that contains a kmsg header and data, or a page of kmsg data. Each packet is prepended with a NORMA message header.

When NORMA is ready to transmit a given NORMA packet, the packet is passed to the transport module where may be divided into smaller units for transmission over the network. In the rest of this paper, when we use the term packet, we will be referring to a NORMA packet.

If a kmsg contains inline data and its length (kmsg header + kmsg body) is less than or equal to one virtual page, NORMA can send the kmsg in a single packet. Otherwise the message must be sent in multiple packets. Section XXX describes the format of a packet in detail.

8 The NORMA Modules

The NORMA send-receive cycle is managed by the following set of modules:

- The Output Module
- The Flow Control Module
- The Transport Module
- The Input Module

8.1 The Output Module

The Output module has two somewhat separate tasks: prepare outgoing messages for transmission across the network, and feed NORMA packets to the Transport Module for transmission. The structure of this module is analogous to the structure of a device driver; it has a top half, which provides the entry point from the standard IPC subsystem to NORMA, and a bottom half which responds to "interrupts" from the Flow Control and Transport modules. We will have more to say about the nature of these interrupts in Section XXX.

8.2 The Flow Control Module

This module is responsible for maintaining the NORMA IPC protocol. When the Output module has finished preparing a message for the network, it calls the Flow Control Module to indicate that it has a message ready to send. The Flow Control Module then controls the manner in which the message's packets are sent. When it is ready to send a given packet, the Flow Control Module calls the Output Module's "lower half" to pass the packet down to the Transport Module.

During a receive operation, the Transport Module hands the Flow Control Module the incoming packet for processing. If the packet contains a Control message, the Flow Control Module invokes the proper routine to handle the message. If the packet contains a kmsg piece that is acceptable, the Flow Control Module passes the packet up to the Input Module for further processing.

8.3 The Transport Module

The Transport Module provides the interface between NORMA and the kernel's network device driver. When the Output Module invokes the Transport Module during a send operation, it processes the current packet into one or more pieces suitable for the device driver to handle. When invoked to receive a packet, the Transport Module transfers the packet from the device driver into memory allocated to NORMA, then invokes the Flow Control Module.

8.4 The Input Module

The Input Module is responsible for processing packets that contain kmsg data. It interacts almost exclusively with the Flow Control Module. The Module's primary function is to assemble incoming packets into messages and queue those messages to their destination ports. The mechanisms by which these operations are accomplished are not simple; they are described in Section XXX.

8.5 The NORMA Flow Control Protocol

The transmission of packets between nodes is governed by NORMA's flow control protocol. This protocol is what is known as a *stop and wait* protocol; in its basic form the protocol works as follows: the sending node sends a packet, then waits for the receiving node to acknowledge receipt of that packet before sending the next packet destined for the remote node. The sending node will not send another packet to the remote node until the current packet has been successfully received.

Of course the protocol is a bit more complicated than this, due to circumstances related to a receiving node's ability to receive packets. A receiving node may not be able to receive a message when its memory resources are in short supply.

When a receiving node's memory resources drop below a certain level, the node temporarily stops accepting low-priority packets. The node will accept certain packets that are related to the default pager because these packets allow the node to reclaim resources. When the node has reclaimed enough memory, it again accepts all packets. The part of the protocol that governs this interaction is managed by NORMA's *suspend/resume* mechanism.

The NORMA IPC protocol is implemented through the use of *control messages*. Currently, the protocol is implemented using the following control messages:

- **CTL_ACK** - Indicates that the receiving node has received the current message. CTL_ACKs are accompanied by a *status* that specifies whether or not the receiving node had the memory resources to accept the message. If the receiving node has sufficient resources, it has received the message; the sending host may now send the next message. If the receiving node was low on resources, it has discarded the current message; the sending node saves the current message and *suspends* transmission of that message.

- **CTL_SYNC** - Used by the sending node when it has not received a CTL_ACK for the current message. Upon receiving a CTL_SYNC, the receiving node issues a CTL_ACK if it has received the message, or a CTL_NACK if it has not received the message.

- **CTL_NACK** - Indicates that the receiving node did not receive the current message. When the sending node receives a CTL_NACK, it resends the current message.

- **CTL_RESUME** - This control message may be sent by the sending node and the receiving node. A sending node that has suspended transmission will periodically issue a CTL_RE SUME to determine whether or not it may resume transmission. When a receiving node receives a CTL_RESUME, it sends back a CTL_RESUME if it has the memory resources available to resume accepting messages, or it ignores the control message if it does not have the memory resources.

We resume discussion of the NORMA flow control protocol in Section XXX.

8.6 The Send and Receive Cycle

Figure XXX shows how the modules interact with one another to manage the sending and receiving of a message packet. We assume that the packet is the first of two packets, and we assume that the remote node has enough memory to accept the message. The following describes the flow of control during the send and receive cycle:

1. The standard IPC subsystem determines that the current message is destined for a port on a remote node and calls the NORMA IPC entry point, thus invoking the Output module.

2. The Output Module prepares the message for transmission across the network then queues the message for transmission by calling the Flow Control Module.

3. When the Flow Control Module determines that it is time to send the message's first packet, it sets the packet's sequence number and passes the packet back to the Output Module.

4. The Output Module passes the packet's data, with a header, to the Transport Module.

5. The Transport Module divides the packet's contents into network driver-specific units and transmits those units to the remote node.

6. As the remote node receives the data, the network interrupt handler invokes the Transport Module, which copies the data into a NORMA-supplied buffer and passes the buffer to the Flow Control Module.

7. The Flow Control Module decides to accept the packet and passes it up to the Input Module for processing.

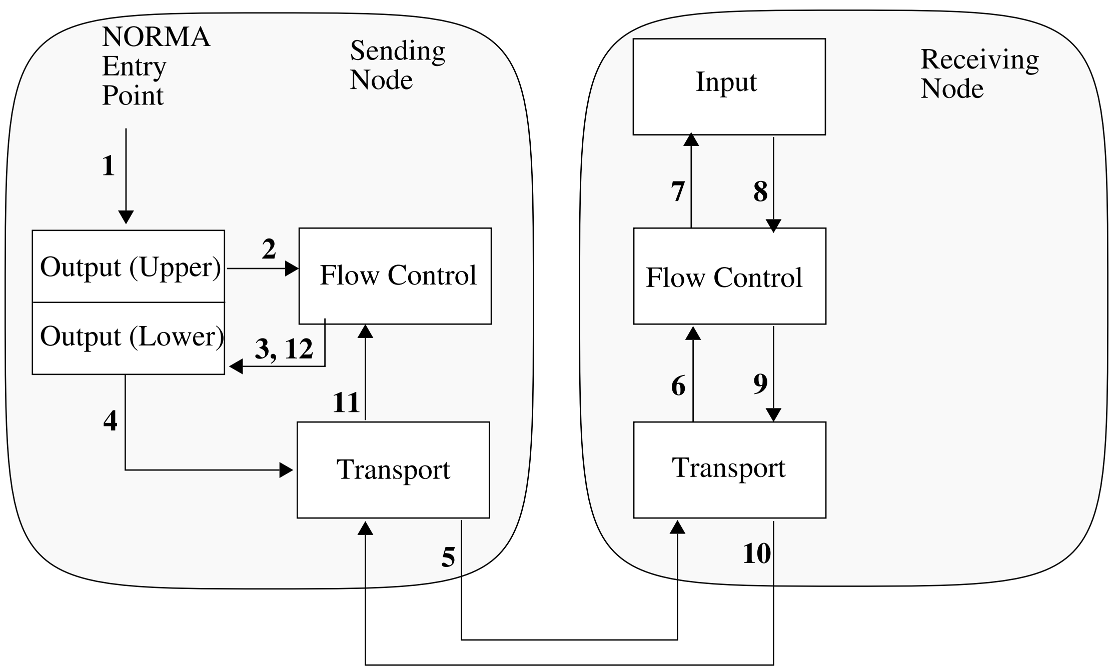
8. The Input Module acknowledges receiving the packet by invoking the Flow Control Module.

9. The Flow Control Module passes this message to the Transport Module by either invoking the Transport Module directly (if there are no other transmissions currently underway), or by queueing the message for sending.

10. The Transport Module invokes the network device driver.

11. The local Transport Module passes the Control message to the local Flow Control Module.

12. The Flow Control Module determines that the current packet has been received and begins transmission of the next packet in the sequence.



Module Interactions during a

Send and Receive Cycle

9 NORMA IPC Data Structures

NORMA uses the following data structures to manage its interactions with remote nodes:

- PCS Structure - The PCS (Protocol Control Structure) contains state associated with the connection between the local node and the remote node.

- Packet Structure - This structure contains state associated with the current kmsg being sent to the remote node.

- Packet State Structure - This structure is used to manage packet state with respect to NORMA's suspend/resume mechanism.

9.1 The pcs Structure

The pcs structure is the key element in the implementation of the NORMA protocol. Currently, each kernel maintains a static array of pcs structures, one per remote node in the NORMA domain. The pcs structure maintains two types of state: state associated with the current send operation and state associated with the current receive operation.

NORMA tags each outgoing packet with a sequence number so that the receiving node can check the order in which packets arrive. The pcs structure includes the sequence number of the last transmitted packet. When the remote node ACKs the packet, NORMA prepares the next packet for transmission and increments the sequence number in the pcs structure. Similarly, the receiving node uses the pcs structure to keep track of what packets it has and has not seen.

NORMA maintains a set of pcs queues that are used to manage connection states; these include a *send queue*, a *timeout queue*, and a *suspend queue*. When there are one or more kmsgs queued for transmission to a node, the pcs structure associated with that node is placed in the send queue. As described below, NORMA's *send* interrupt handler processes this queue periodically to transmit packets.

After NORMA transmits a packet to a remote node, it places the associated pcs structure in the timeout queue and waits for the remote node to ACK the packet. The timeout queue is processed by NORMA's *timeout* interrupt handler. If the ACK is not received before the time-out expires, NORMA sends the remote node a SYNC message.

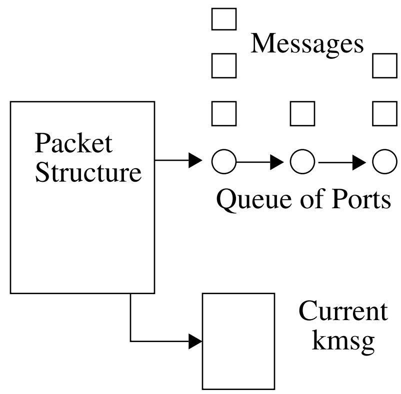
When a connection has been "suspended," NORMA places the associated pcs structure in the resume queue. NORMA periodically scans this queue and transmits RESUME queries to the remote nodes.

9.2 The Packet Structure

The packet structure is used to maintain state associated with a connection's current outgoing kmsg. Packet structures are dynamically allocated as needed. If there are no messages queued for a given remote node, its packet structure is deallocated.

In addition to maintaining the current kmsg's state, the packet structure also maintains a queue of ports that have kmsgs queued for the remote node. The figure below shows a simplified representation of a packet structure. As shown in the figure, the structure contains a pointer to the current message packet in transit to the remote node. When NORMA receives an ACK for the current packet, NORMA advances this pointer to the next packet. The next packet may be either of the following:

- The next part of the current kmsg.
- If the last packet was the end of a kmsg, NORMA begins transmission of the next message queued to the port, if there is one.
- If there is not another kmsg queued to the port, NORMA removes the port from the packet's queue and begins sending the first kmsg queued to the next port in the queue, if there is one.



9.3 Packet State Structure

For the most part, the packet state structure is used to maintain a connection's state with respect to the suspend/resume mechanism; that aspect of the packet state structure will be described when we resume discussion of the flow control protocol in Section XXX. However, the packet state structure is also the means by which the send interrupt handler finds a connection's packet structure, and so we must mention it here. Currently, each kernel maintains a static array of packet state structures, one per remote node in the NORMA domain.

9.4 How the Structures Participate in a Send Operation

When a task invokes a NORMA send operation, the Output Module converts the kmsg to NORMA format and "queues" the kmsg's destination port for sending. What actually happens in this queueing operation depends on whether or not there are kmsgs already queued for sending to the remote node.

If there are no kmsgs currently queued for sending to the remote our kmsg is destined for, NORMA performs the following operations to queue the port and initiate the sending of the kmsg:

1. Since there are no other kmsgs currently queued for our destination, NORMA allocates a new packet structure and associates it with the packet state structure that represents the remote node.

2. NORMA places our port on the packet structure's queue of ports.

3. NORMA removes the kmsg from the port's message queue and makes it the packet structure's current kmsg.

4. NORMA places the pcs structure associated with the remote on the send queue.

5. When the send interrupt handler (in the Flow Control Module) processes the pcs structure, the handler uses the pcs's remote node id to find the associated packet state structure in the

packet state structure array. The packet state structure points to the packet structure, which points to the current kmsg.

6. The send interrupt handler invokes the Output Module, which in turn invokes the Transport Module to send the kmsg's first packet.

7. The pcs structure is now placed on the timeout queue to wait for the remote to ACK the packet just sent. When the ACK is received (assuming it indicates that the packet was received successfully), NORMA will again place the pcs structure on the send queue.

Now suppose that there is already a transmission to the remote underway when we attempt to send the kmsg. In this case, NORMA has already allocated a packet structure. If there is a kmsg ahead of us that is destined for our port, the port is already queued to the packet structure. Otherwise NORMA queues the port.

10 NORMA Interrupt Handling and Related Issues

The NORMA subsystem is driven by two sorts of events: send operations, and hardware interrupts. When NORMA is not in the midst of a transmission, it is activated when a task initiates a send operation to a remote node. Once there are one or more transmissions underway, NORMA is driven by hardware interrupts. (By underway, we mean that one or more message packets have been sent and NORMA is waiting for the remote nodes to send back ACKs.)

Because these two mechanisms share access to the same data structures, and their operations may overlap with one another, NORMA must have a scheme for maintaining the integrity of the data structures. It boils down to this: if a send operation is underway and the kernel interrupts the operation to handle a NORMA-related interrupt, the kernel must prevent the interrupt handler from corrupting the send operation's state.

NORMA implements this protection scheme by maintaining the **netipc_lock** variable. When a thread is in the midst of initiating a send operation, NORMA takes this "lock" before attempting to queue the message and the port. (**netipc_lock** isn't really a lock, just a global variable; a thread "takes" the lock by being assigned the variable's owner.) Should a NORMA-related interrupt occur while the lock is held, the interrupt handler notices that the lock is "held," and "queues" an interrupt to be handled later.When is later? When the thread "releases" **netipc_lock**. The call that releases this lock causes NORMA to perform any interrupt activities that were attempted while the lock was held.

10.1 The NORMA Interrupt Handlers

There are three NORMA-related interrupt handlers: **norma_recv_intr( )**, **norma_send_intr( )**, and **norma_timeout_intr( )**. The first of these, **norma_recv_intr( )** is executed when the network driver receives NORMA-related data from the network. If the received data is a control message, the handler processes the message. If the received data is a valid kmsg packet, the handler passes it on to the Input Module. Depending on the packet's type, the Input module will process the packet at interrupt level, or arrange for the packet to be processed at AST level. See Section XXX for further discussion.

The **norma_send_intr( )** handler is invoked when the network hardware finishes placing the current NORMA packet on the network. When invoked, this handler iterates through the **pcs send queue**, searching for the next packet to send.

The **norma_timeout_intr( )**handler manages the **pcs timeout queue**. When a given packet has been placed on the network, the associated **pcs** structure is placed on the timeout queue. The timeout handler checks each **pcs** on the queue. For each **pcs** structure that has waited the requisite amount of time for its current packet to be ACKed, the handler arranges to send a CTL_SYNC message to the remote node. If there are no messages currently undergoing transmission, the handler causes the control message to be sent immediately. If there are messages pending transmission, it places the pcs structure in the send queue.

11 Preparing kmsgs For Transmission over the Network

Before it begins sending a given message, NORMA's send side processes the message so that the receive side can correctly assemble the message. There two aspects involved in this processing:

- Convert the kmsg to NORMA format.
- Divide multipage kmsgs into separate packets.

Both of these operations are performed by the Output module. The Output module converts a kmsg to NORMA format as follows:

1. It converts port rights in the kmsg header to their NORMA uid counterparts. This is necessary so that NORMA can determine the kmsg's destination.
2. If the kmsg's body contains port rights, either inline or in out-of-line arrays, the Output module converts those port rights to their NORMA uid counterparts. If a port right in a message corresponds to a port that does not have a NORMA uid (no rights to the port have been previously exported) NORMA creates a uid for the port and registers the port in the local node's uid table.
3. If the kmsg contains only inline data and its length (the kmsg header + the kmsg body) is less than or equal to a virtual page, no other processing is necessary.
4. Otherwise, the Output module places the kmsg into one or more *page list copy objects* (these are discussed in detail in the next section). The page list copy object is the mechanism NORMA uses to divide multipage kmsgs into single page packets for transmission over the network.

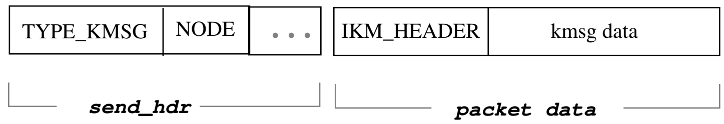
11.1 NORMA Packet Types

Each NORMA packet has two pieces: the packet's header (*send_hdr*) and the packet's data. The header includes the following information: the packet's type, and the node from which the packet comes. The receive side uses this information to determine how to handle the packet.

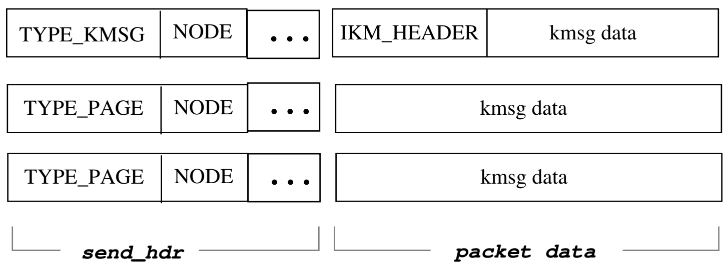
There are two packet types; a packet is either a NETIPC_TYPE_KMSG or NETIPC_TYPE_PAGE. A packet that is of type NET_IPC_KMSG always contains a kmsg header; this header specifies the kmsg's destination port and kmsg's length, and may also specify a reply port. The receive side uses the kmsg header's information to locate the kmsg's destination port and to determine if the kmsg is a single packet or multiple packet kmsg.

If a kmsg is contained entirely in one page (both the kmsg header and the kmsg body), NORMA transmits the kmsg in one packet and that packet is a NETIPC_TYPE_KMSG packet. Otherwise the kmsg is transmitted in multiple packets. The first of these packets is a NETIPC_TYPE_KMSG, the rest of the packets are NETIPC_TYPE PAGE. The figure below shows the format of the packets associated with a 1 page kmsg and a 3 page kmsg.

Packet Associated with a 1 page kmsg



Packets Associated with a 3 page kmsg



If a packet is of type NETIPC_TYPE_PAGE, its *send_hdr* includes information about packet's page such as whether or not this is the first or last page of the message. The receive side uses this information to decide how to assemble the kmsg and when to ACK the packet.

12 Sending Multiple Packet Messages

As mentioned above, the NORMA transmits kmsgs a page at a time. Here is how the NORMA send side and receive side cooperate to handle the transmission of a multiple packet message:

1. The send side divides the kmsg into multiple packets and sends the packets to the receive side one at a time.

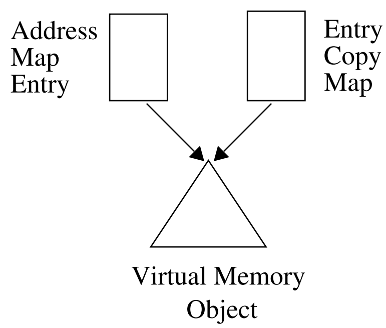
2. The receive side assembles the packets as they arrive.

3. When all of the kmsg's packets have been received and the kmsg has been completely assembled, the receive side queues the message to the destination port's message queue.

12.1 Virtual Memory Interlude: Copy Object Technology

Mach IPC has always allowed tasks to send large portions of their address spaces to one another. Such data is passed between the tasks out-of-line. While the data is in transit, the kernel must prevent it from disappearing should the sending task deallocate the memory before the receiving task has received the data. The kernel uses the *copy object* abstraction to handle this problem.

Suppose a task is sending a region of its address space to another local task. Within the sending task's address space, the region is represented by an address map entry which maps a VM object to a range of the task's virtual address space. When the task invokes Mach IPC to send the region out-of-line, the IPC subsystem creates a copy map and maps the VM object to the copy map (see Figure XXX). Now that the copy map references the VM object, the sending task can deallocate the region from its address space without effecting the transmission of the data. When the destination task receives the message, the IPC subsystem inserts the copy map into the task's address map.



Page List Copy Objects

The type of copy map used to pass out-of-line data between local tasks is referred to as an *entry-flavored* copy map. One of the features of this type of copy map is that the region's pages are not necessarily resident when the IPC subsystem creates the copy map or when it inserts the copy map into the receiving task's address space. Any pages that are not resident will be faulted in as needed when the receiving task attempts to access them.

When such a message is passed between tasks on separate nodes, the underlying network device driver cannot wait for non-resident data to be paged in; the device driver must have all the current packet's data resident. This requires a different form of copy object technology, in which the objects are referred to as *page list-flavored* copy objects. In page list copy objects, the copy object's pages are referenced via a *page list array*; the pages are wired into memory and so are always resident.

To send out-of-line data over the network, the NORMA send side converts the entry copy object that represents the data to a page-list copy object. During the conversion procedure, the kernel pages in the non-resident pages, then wires into memory all pages, thereby protecting them from the page-replacement mechanism.

A potential problem in converting an entry-flavored copy object to a page list copy object is that if the entry-flavored copy object represents a large range of virtual addresses, the system must consume a large amount of resources to wire the pages. Therefore, the kernel limits the number of pages that can be wired per page list copy object.

What happens during the conversion of an entry copy object that contains more pages than this limit? The VM subsystem provides a *continuation* mechanism whereby when the first set of pages in the copy object can be replaced by the next set of pages in the sequence. We will discuss this continuation mechanism shortly.

Object-Flavored Copy Objects

In addition to page list copy objects, NORMA employs a third flavor of copy object, the *object-flavored* copy object. Like entry-flavored copy objects, the object-flavored copy objects are pageable and they are used on the receive side to handle the assembly of large messages. The object-flavored copy object allows NORMA to receive large messages without consuming an unreasonable amount of memory.

VM Continuations

VM continuations are used to take care of two general problems: converting large pageable copy objects to page list copy objects, and discarding copy objects when they are no longer needed. Note that the VM continuation mechanism is not the same as the scheduler continuation mechanisms. The kernel uses the scheduler continuation mechanisms to conserve the amount of memory consumed by thread kernel stacks. The kernel uses the VM continuation mechanism to perform various VM operations in an incremental fashion.

Each copy object data structure includes two fields that are used to set up continuations: a field to specify the *continuation routine*, the other to specify one or more arguments to the routine. The easiest way to describe the setup and invocation of a VM continuation is by example. When the NORMA send side converts to page list form a region of out-of-line data that exceeds twenty pages, it can convert only twenty pages at a time. The conversion process proceeds as follows:

1. NORMA invokes the VM subsystem to perform the conversion.

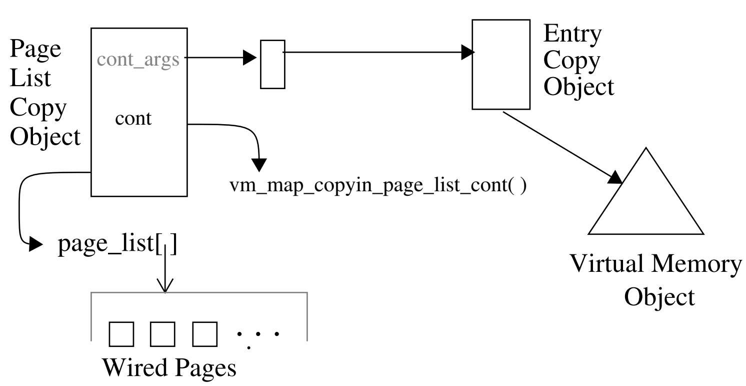
2. The VM subsystem allocates a new page list copy object

3. The VM subsystem inserts the first twenty pages of the old copy object into the page list array of pages, faulting in any non-resident pages and wiring all pages into memory.

4. The VM subsystem sets the page list copy object's continuation routine to **vm_map_copyin_page_list_cont( )** and sets the continuation's arguments to refer to the original entry copy object.

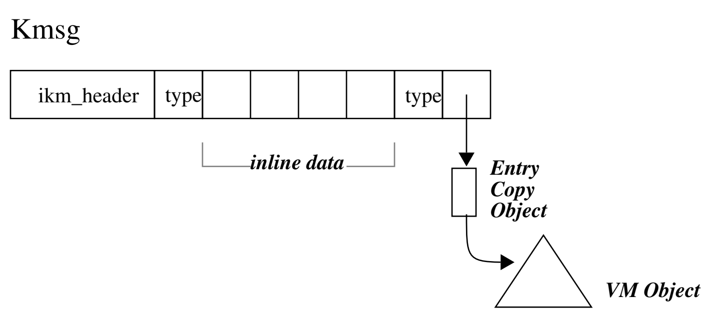
5. The VM subsystem returns the page list copy object to NORMA. As shown in Figure XXX, the page list copy object, through the cont_args field, continues to reference the original entry copy object.

When NORMA finishes sending out the page list object's pages, it discards the pages in the page array, invokes the page list's continuation routine (**vm_map_copyin_page_list_cont( )**), and so the next twenty pages of the entry copy object are converted to page list format.



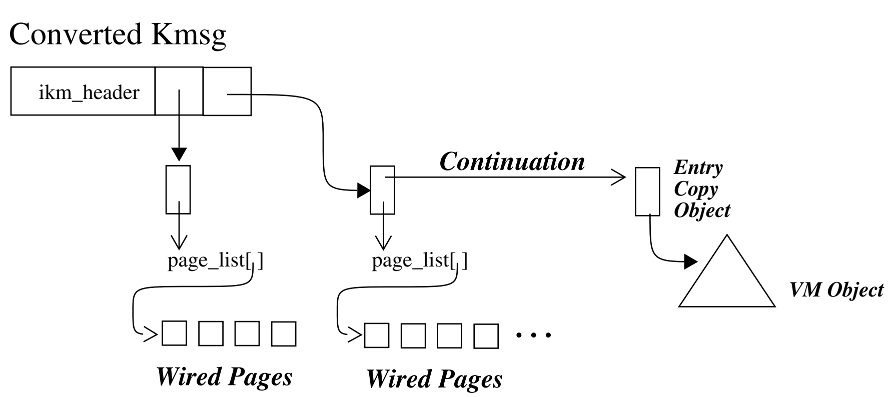
12.2 Preparing Multipage Kmsgs for Transmission

The following example describes how the NORMA send side prepares a multipage kmsg for transmission. To make the example interesting, the kmsg contains two sections of data. The first section contains four pages of inline data, the second contains a pointer to 100 pages of out-of-line data. The following figure shows the format of the kmsg as it is passed from the standard IPC subsystem to NORMA.



NORMA processes the kmsg body a section at a time. For the first section, NORMA allocates a new page list copy object, inserts the four pages into the object's page list array, and wires the pages. This page list does not require a continuation because the data it contains is less than 32 pages in length.

The second section is represented by an entry-flavored copy object that must be converted to a page list copy object. Since the out-of-line data is greater than 32 pages in length, the page list will have a continuation. The following figure shows the converted kmsg.



12.3 Send Side Page List Continuation Thread

The invocation of a page list continuation may cause the invoking entity to block if data must be faulted in to refill the page list. Consequently, the continuation cannot be invoked at interrupt level. Instead, NORMA maintains a kernel thread that processes page list continuations related to the sending of messages. We will refer to this thread as the *NORMA copy continuation* thread (in the code it is referred to as **netipc_vm_map_copy_continuation**).

When NORMA detects that the last page of a page list has been ACKed, and that the page list has a continuation, it places the associated packet structure onto the kernel's **netipc_continuing_packet_list** queue and wakes the NORMA copy continuation thread. When it is scheduled for execution, the thread invokes continuations for all packets that are currently on the queue, then blocks when all queued packets have been processed.

13 Receiving Messages

On the receive side of a connection, NORMA performs three operations on each kmsg:

1. The kmsg is assembled;
2. The kmsg is placed on the destination port's message queue;
3. The kmsg is converted from NORMA format to standard IPC format when the message is delivered.

The receive side of NORMA is more complicated than the send side because, in addition to performing the operations outlined above, the receive side must ensure that packets are assembled into their kmsgs in the correct order and that kmsgs are delivered to their ports in the correct order. As will be seen, there are a number of subtleties to the system that make packet and kmsg ordering difficult to guarantee. This section describes the issues that must be addressed by the receive side and describes as well the NORMA idle thread mechanism, an optimization to the message delivery path that allows NORMA to deliver certain kmsgs directly to threads that are waiting to receive them.

13.1 Message Assembly

The receive side of NORMA maintains two queues to manage incoming packets. The *incomplete message queue* holds kmsgs that are in the process of being assembled, and the *complete message queue* holds messages that are complete and ready to be queued on their destination port queues. We'll have more to say about the complete message queue in Section XXX.

The first packet of any multiple packet kmsg contains the kmsg header. The receive side determines the kmsg's length from the header and decides how to assemble the kmsg. There are two *paths* through the message assembly code: a *fast path* for kmsgs that are under a certain length, and a *slow path* for larger messages. At the time this document was written, the fast path handles kmsgs that are less than or equal to 20 pages.

Fast Path Assembly

The fast path assembles kmsgs using wired memory. Incoming data are placed in a wired page which is then inserted into a page list copy object. The fast path processes packets at interrupt level. If a kmsg exceeds the length of one page list, the fast path allocates a second page list and sets a continuation for the first page list so that the second page list can be located when the kmsg is delivered.

Slow Path Assembly

The slow path assembles kmsgs using pageable memory and the actual assembly is performed by a kernel thread instead of at interrupt level. The slow path was implemented to allow NORMA to support the transmission of very large messages without consuming all of a receiving node's memory resources.

The receive side processes slow path packets as follows:

1. The packet is received at interrupt level and is placed on NORMA's **netipc_slow_page_-queue**.

2. NORMA wakes the *slow path* kernel thread and returns.

3. The slow path kernel thread processes each packet on the slow page queue. For each packet on the queue, the thread locates the packet's kmsg on the incomplete message queue and inserts the packet's page into the pageable copy object.

13.2 Packet ACKing

One of the more complicated aspects of the NORMA receive side involves the ACKing of packets. Due to NORMA's stop and wait protocol, a node will not send the next packet of a multiple packet kmsg until it has received an ACK for the previous packet. The receive side must make sure that the current packet is appended to the kmsg before the next packet arrives.

For the purposes of this discussion, we classify packets into the following types:

1. kmsg packets (NETIPC_TYPE_KMSG)

2. fast path page packets (NETIPC_TYPE_PAGE)

3. slow path page packets (NETIPC_TYPE_PAGE)

kmsg Packet ACKing

When NORMA receives a packet of this type, it must determine whether or not the kmsg's destination port is actually located on the local node. (Obviously, the sending node believes it has sent the kmsg to the proper node, but the port may have migrated to another node; if so, the kmsg must be redirected. Section XXX describes what happens if the port has migrated.)

In order for NORMA to check the kmsg's destination, it must be able to lock various data structures and so this operation cannot be performed at interrupt level. Instead, NORMA sets an Asynchronous System Trap (AST) and performs the check at AST level. If the kmsg's destination is in fact local, NORMA issues the ACK from AST level, indicating that the send side may transmit the next packet.

Fast Path Packet ACKing

Since packets of this type are appended to their kmsgs at interrupt level, they are ACKed at interrupt level.

Slow Path Packet ACKing

These packets fall into two categories: packets that do not contain a kmsg's last page, and packets that do contain a kmsg's last page. As described above, slow path packets are queued on NORMA's **netipc_slow_page_queue** at interrupt level and are appended to their messages by a single kernel thread. Because pages are placed on the queue in the order they are received, NORMA can ACK slow path packets at interrupt level safely; the next packet in the kmsg cannot be queued ahead of the current packet.

However, it is not safe to ACK a slow path packet at interrupt level if that packet contains the last page of its kmsg. When the kernel thread processes such a packet, it determines that the kmsg is complete and attempts to place it on the destination port's message queue. In the time between when the packet is received and when the kernel thread queues the kmsg to the port, the kmsg's source node (the node from which the kmsg came) could send a single packet kmsg that would be entirely processed at interrupt level and queued to the port ahead of the slow path kmsg. To prevent races of this kind, NORMA delays ACKing the last packet of a slow path kmsg until the kernel thread is ready to queue the kmsg to the port.

13.3 The Complete Message Queue

The complete message queue, in conjunction with the packet-ACKing mechanism described above, provides the means by which NORMA guarantees the correct order in kmsg delivery. As might be expected, single packet kmsgs are placed directly on the complete message queue from interrupt level. In the case of a multiple packet kmsg, NORMA places it on the complete queue when it has received and incorporated the kmsg's last packet. When a kmsg is ready for the complete queue, NORMA places it at the end of the queue.

Kmsgs are then transferred from the complete queue to their destination port queues. NORMA performs this operation at AST level, beginning with the first kmsg on the complete list, then working its way to the end of the list. If there are two or more kmsgs destined for the same port, they will have been place on the complete queue in the proper order and so will be placed on the port queue in the proper order.

13.4 Message Delivery

When NORMA has finished assembling a kmsg, it places the kmsg onto the destination port's queue. At this point, the kmsg is still in NORMA format. Before it can be delivered to a receiving thread, the message must be converted into a form that the standard IPC subsystem can process. This conversion takes place when a thread invokes the IPC subsystem to receive the message. At that time, the IPC subsystem calls the NORMA subsystem (via the **norma_ipc_finish_receiving()** routine) and NORMA converts the message.

First NORMA checks to see if the sending node has supplied a send right or a send-once right for a reply port in the kmsg header, and if so, *receives* the right. This operation involves the following:

1. If the reply port's uid is not in the local NORMA uid table, insert the uid and create a proxy for the port.
2. Perform port right bookkeeping; see Section XXX for a description of send right and send-once right accounting in NORMA.

If the kmsg is a single packet kmsg that contains no inline port rights, no other processing is necessary. If the kmsg is a single packet kmsg and contains inline port rights, they must be *received* in the same manner as reply ports are received.

Delivery of Multiple Packet kmsgs

The pages of such a kmsg are at this point contained in a page list copy object (fast path), or in an object-flavored copy object (slow path). Such kmsgs are referred to as *fragmented* kmsgs; before handing them to the standard IPC subsystem, NORMA converts them to *contiguous* kmsgs.

The following paragraph may be incorrect!

The conversion process requires that the kmsg's underlying copy object be a page list copy object. If the kmsg was assembled in the slow path, NORMA first converts the object-flavored copy object to a page list copy object. Since the object-flavored copy object contains more pages than can be contained in a page list, the conversion operation will generate a page list copy object with a continuation.

13.5 The NORMA Idle Thread

In the standard IPC subsystem, if a thread invokes a receive operation on a port that has no messages currently queued, the kernel blocks the thread, then wakes it up when a message is delivered to the port.

The NORMA idle thread is an optimization to the IPC subsystem that allows a thread that is waiting to receive a message to spin on the processor if there are no other runnable threads currently available for execution. If the NORMA receive interrupt handler processes a message destined for the port while the thread is spinning, NORMA can deliver the message directly to thread, thereby saving the overhead of a context switch.

The NORMA idle thread mechanism works as follows: a thread invokes a standard IPC message receive operation on a port that does not have any messages currently queued to it. The kernel determines that there are no other threads currently scheduled for execution, sets the port's message queue to function as the *NORMA handoff queue*, then places the thread in an idle loop.

As the thread spins in the loop, the network hardware may trigger a NORMA receive operation that furnishes the awaited message. When the receive interrupt handler receives a packet that either contains a complete kmsg, or is the last packet of a complete kmsg, it checks the kmsg's destination to see if it matches the port that is currently functioning as the handoff queue. If there is a match, the receive interrupt handler places the message on the handoff queue and exits.

During each pass through the loop, the idle thread checks to see if a message has been placed on the handoff queue. If so, the thread breaks from the loop and returns the kmsg to the IPC subsystem.

13.6 kmsgs Bound for the Kernel

As described above, NORMA processes the complete message queue from AST level and places the kmsgs it finds there on their respective port queues. This in not true, however, of kmsgs that are destined for the kernel. Delivering a kernel-bound kmsg may invoke a kernel operation, which may in turn block to wait for resources. Consequently, kernel-bound kmsgs cannot be queued at AST level, but must be queued in thread context.

NORMA handles kernel-bound kmsgs via the *kserver* mechanism. When NORMA processes the complete messages queue at AST level, it transfers all kernel-bound kmsgs to the **kernel_kmsg** queue. This queue is processed by NORMA's kserver threads.

14 Send Right Transmission in NORMA

The standard IPC subsystem is frequently used to transfer send rights and send once rights between tasks. NORMA must also be able to transfer send rights and send once rights between tasks, and preserve the IPC semantics associated with these rights.

14.1 Send Right Management in the Standard IPC Subsystem

The standard IPC subsystem keeps track of a port's send rights and send-once rights as follows: the port data structure includes an srights count and an sorights count. When the port's receiving task copies a send right to another task, the IPC subsystem increments the port's srights count. When a task destroys a send right, the kernel decrements the port's srights count. Similarly, when the task creates a send once right for another task, the kernel increments the port's sorights count and when a task consumes an so right, the kernel decrements the receiving task's sorights count.

Each time the kernel decrements a port's send right count, it checks to see if there are no more send rights out in the world (the port's srights count is zero). If there aren't and the port's receiving task has requested no-more-senders notification, the kernel sends the task the notification.

14.2 Send Right Tracking in NORMA

This section is incomplete.

Like standard MACH IPC, NORMA IPC tracks a port's send and send-once rights. This discussion focuses on send right tracking, but for the most part, send-once rights are tracked in a similar manner.

According to IPC semantics, any task that has a send right to a port may copy or transfer that right to another task. In NORMA, the following scenario can occur: a port's receiving task copies a send right to a task on another node, and that second task copies that send right to a third task on yet another node. NORMA implements send right tracking so that the principal does not need to be involved each time remote tasks copy send rights to other remote tasks.

NORMA tracks send rights by maintaining *send right counts* and *transit counts*. We will explain the use of these counts by example. Let's suppose that a port's receiving task is about to copy a send right to a remote task. Here's what happens:

1. NORMA increments the principal's send rights count by one, and its

14.3 No Senders Notifications

This section is incomplete.

The goal of NORMA with respect to the no senders IPC semantic is to prevent the premature issuing of such notifications. The no senders mechanism is implemented in the regular IPC code, and is triggered when the IPC subsystem decrements a port's send rights count; if the count drops to zero, the IPC subsystem generates a no senders notification.

When this happens to proxy, the standard IPC subsystem invokes NORMA to send the notification to the principal's node. At this time, the proxy's node sends its extra transits back to the principal's node so that the principal can properly account for them.

A no senders notification cannot be issued on the principal's node unless its transit count indicates that there are no outstanding send rights in the NORMA domain.

On the principal

In NORMA, a port has several manifestations. Associated with the receive right is the port's principal. When a task has receive rights to a port, the kernel Send rights

14.4 Transit Vector Mechanism

In working with NORMA at the OSF RI, we noticed that NORMA spent a significant amount of time processing yield_transit messages. This was because such messages contains transit counts for only one port. We implemented an optimization whereby a node sends yield counts in batches, collecting counts destined for the same node and sending them together. This optimization is referred to as the *transit vector mechanism*.

15 Receive Right Migration

The mechanism used to migrate receive rights is referred to as a *pull* mechanism; the sending node merely sends an indication that it is migrating the receive right; upon receiving this notification, the destination node (the target of the migration operation) pulls the right across.

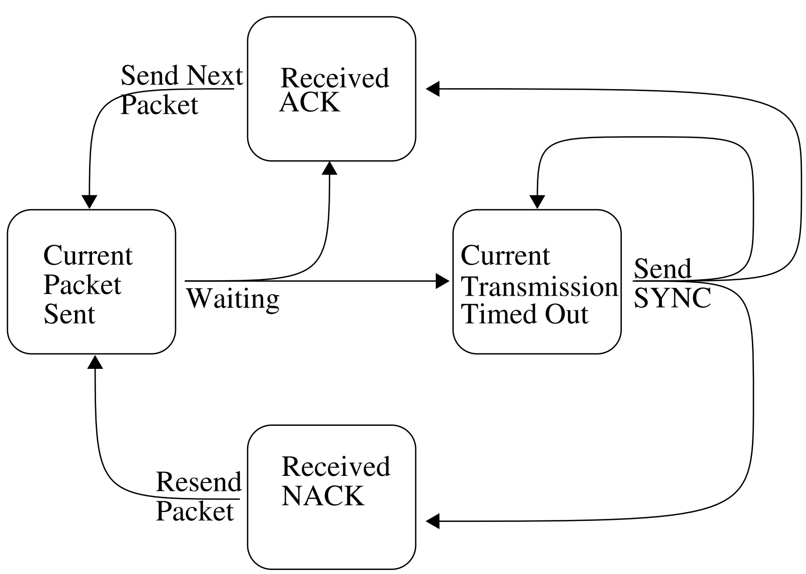
While a receive right is in transit, NORMA must preserve the order in which messages are sent to the port. Until the target node issues a call to pull the right, the right's previous node continues to receive any messages sent to the port. When the target node receives the right, it then issues a call to the original node to migrate the messages that have accumulated.

Before it sends the pull request to the original node, the target node allocates a port data structure for the port and creates two queues for the port, a regular message queue and a special message queue, known as an *atrium*. The atrium is used as a repository for the migrating messages. Should any tasks send messages to the port before all messages accumulated on the other node have been migrated across, they will be placed in the regular message queue.

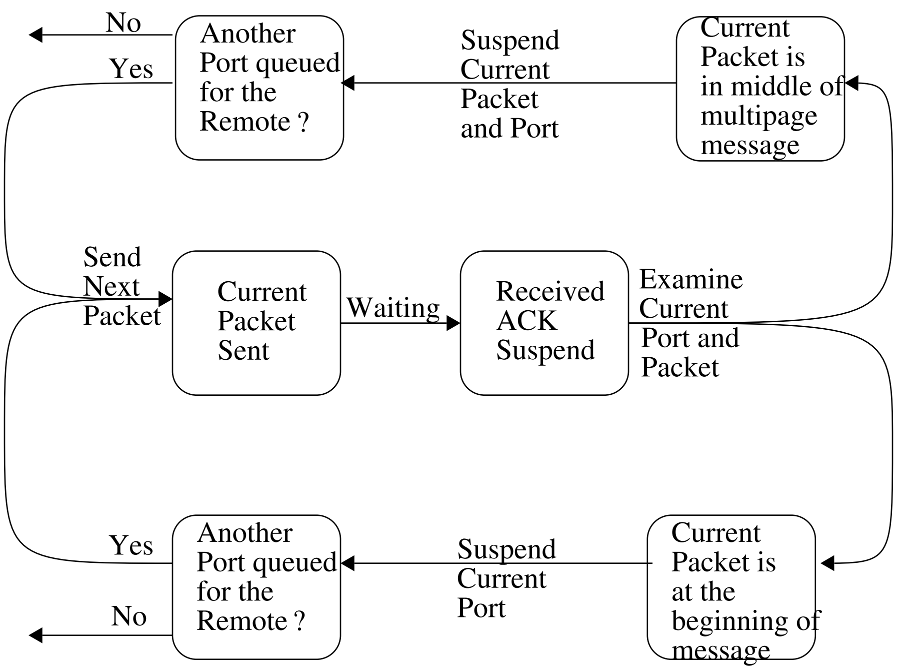
When the original node receives the pull request, it places forwarding information in the port's data structure up so that messages sent to the port are redirected to the proper node. The original node then begins to send the messages it has accumulated during the transit time to the target node. When the target node finishes receiving all of the messages (which it collects in its atrium message queue), it consolidates its atrium with the regular message queue by placing the messages in the atrium ahead of the messages in the regular queue. This operation preserves message ordering.

16 The Flow Control Protocol, Revisited

This section provides a series of diagrams the illustrate the finer points of the flow control protocol. The following figure provides a pictorial representation of NORMA's stop and wait protocol. As shown in the figure, NORMA sets a time-out for each transmission; if the remote node does not respond with an ACK before the time-out expires, the local host sends a SYNC.



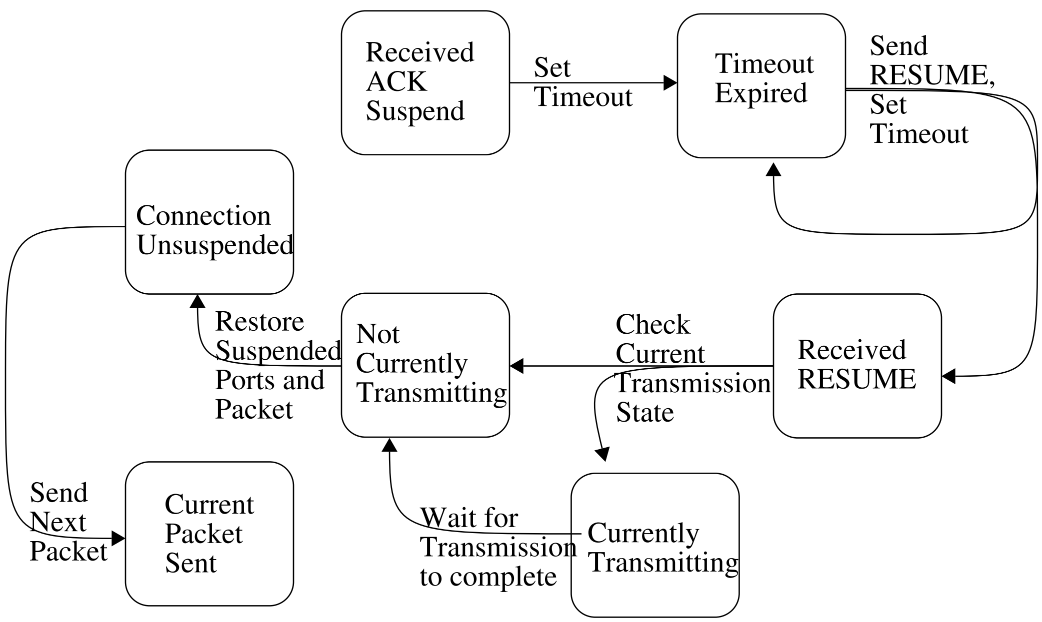
The following figure depicts how NORMA responds to a suspend request. This figure assumes that



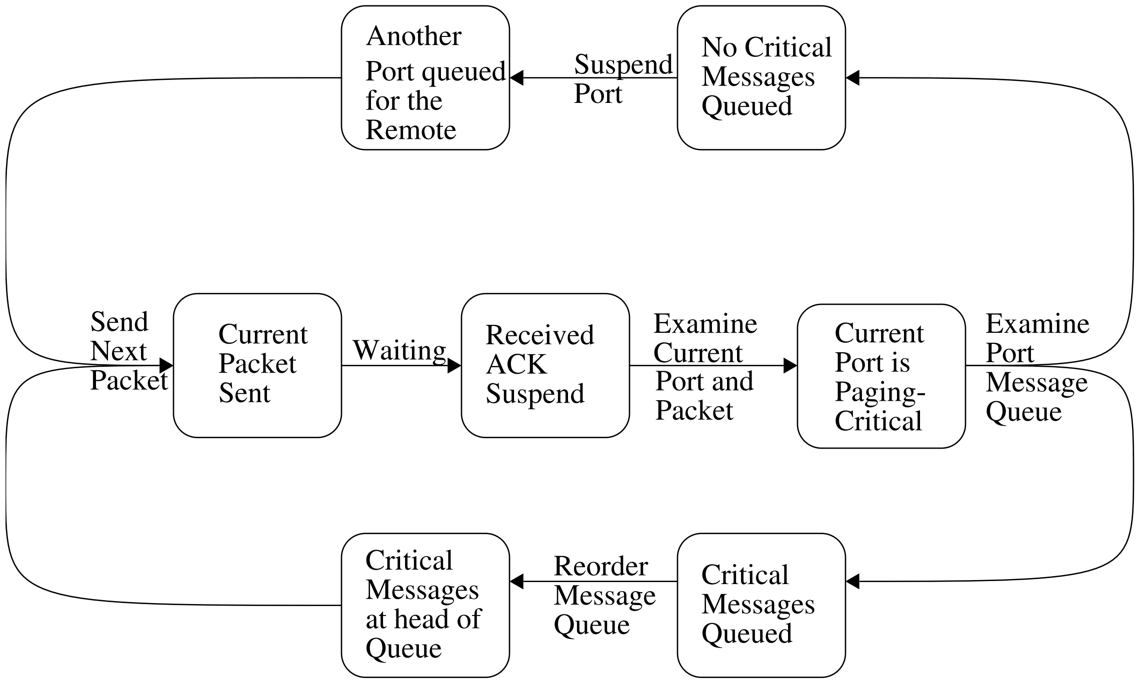
the current port is not a paging-critical port (we'll discuss the paging-critical instance shortly). As shown in the figure, if NORMA is in the middle of sending a multipage message, it performs two operations; it suspends the current packet, then it suspends the current port. (We'll discuss how these suspend operations are implemented shortly.) If the current packet is at the beginning of the message, NORMA just suspends the port. In both cases, the local node continues sending messages to the remote node, but it should be noted that NORMA does not send other messages from the port that are behind the suspended message. Once the current port is suspended (and the current packet, if required), NORMA advances to the next port queued for transmission to the remote and begins sending the first message queued to that port.

When a local node has had its transmission to a remote node "suspended", it is the local node's responsibility to periodically query the remote node to determine when it may resume "normal" transmission. This is effected by means of a time-out; periodically, the local node sends a RESUME control message to the remote node. The remote node ignores these messages as long as it continues to be short on memory. When it again has adequate resources, the remote responds to a RESUME by sending back a RESUME. The following figure depicts this interaction and what happens on the local node when it receives a RESUME message from the remote.

As shown in the figure, when the local node receives a RESUME message, NORMA restores the connection's suspended state and continues transmitting messages. We'll discuss the restoration operation shortly.



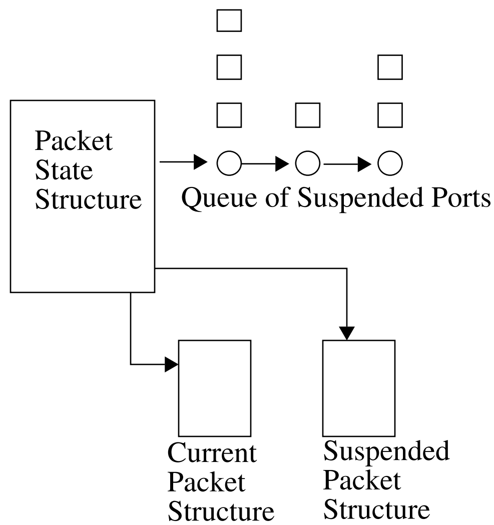
As mentioned before, a node continues to transmit messages to a remote node even if its connection to the node has been "suspended." This is because the remote node may be using the local node for default paging activities and the local node may have messages queued for the remote node that will allow the remote node to reclaim resources. In its current implementation, NORMA makes the remote node responsible for determining whether or not a given message is paging-critical; the local node continues sending messages. The following figure depicts how NORMA manages paging-



critical ports when a connection is in a suspended state. As shown in the figure, NORMA suspends a paging-critical port if it has no paging-critical messages queued. If the port does have such messages queued, NORMA shuffles those messages to the head of the queue. Technically, this operation violates the Mach IPC message-ordering semantic, but in practice it is absolutely necessary to do so and the violation does not cause problems because the queued messages do not depend on one another. It should be noted that NORMA will restore a suspended paging-critical port if the kernel queues a paging-critical message on it.

16.1 The Packet State Structure, Revisited

NORMA uses the Packet State structure to maintain a connection's state with respect to the suspend/resume mechanism. The figure below shows a Packet State structure that is associated with a connection that has been "suspended." The connection became suspended while NORMA was in the middle of transmitting a multiple packet kmsg, hence the suspended packet structure. As shown in the figure, the Packet State structure is also used to maintain the queue of suspended ports.



17 The Future of NORMA IPC

The current version of NORMA IPC was implemented as a prototype and suffers from many problems. Currently, the OSF RI is planning to rewrite NORMA IPC as part of the development of OSF/1 AD, Rev 2.

References

[1] Barrera, J. A Fast Mach Network IPC Implementation. In *Proceedings of the USENIX Mach Symposium*, Nov. 91

[2] Open Software Foundation, *Design of the OSF/1 Operating System*, *Release 1.2*, 1993

[3] Langerman, A. NORMA Rewrite Requirements. OSF RI Internal Document, Sept, 93.

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