NORMA IPC Version Two: Architecture and Design

Open Software Foundation
Research Institute
One Cambridge Center
Cambridge, MA 02142
(617) 621-7340
norma-ipc@osf.org

Abstract. The Research Institute proposes a new architecture and design for the remote IPC facility integral to the Mach microkernel. Incorporating lessons learned from the original implementation of `NORMA IPC`, the second version has many goals, including enhanced robustness at all levels and much higher performance levels. This is a working document, which we update as the design evolves. The design will be used as a reference for the implementation and, where necessary, will be altered to reflect any changes that subsequently emerge.[1]

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

Table of Contents

List of Figures

CHAPTER 1 Introduction

1.1 Prerequisites

We assume that the reader is familiar with the existing Mach 3.0 microkernel, and especially the `VM`, `XMM`, `IPC`, and `NORMA IPC Version One` subsystems. This document does not review fundamental Mach concepts or commonly-known internals. Refer to section 1.6 for a list of companion documents necessary or helpful for reading alongside this one. It may also be helpful to refer to the current source code base (NORMA13, NORMA14, or later).

1.2 Purpose Of NORMA IPC

The `NORMA IPC` subsystem exists to extend Mach IPC semantics seamlessly across a cooperating computing domain. A `NORMA` domain contains independent nodes interconnected via a communications medium. (The `NORMA` acronym stands for NO Remote Memory Access.)

Unmodified Mach functions on uniprocessor and tightly-coupled shared-memory architectures, with a single kernel running on top of a single machine abstraction. By contrast, in a `NORMA` domain multiple kernels, each executing on top of an independent machine abstraction, cooperate to provide the illusion of a single Mach IPC space.

By virtue of providing a single IPC space, the existing Mach abstractions -- such as tasks, threads, memory objects, hosts and processor sets -- may be controlled transparently by applications running on any node within the domain.

The Mach eXternal Memory Management (`XMM`) subsystem uses `NORMA IPC` to provide fully coherent shared memory between nodes. Thus, tasks running on separate nodes may transparently share memory objects for reading and writing.

Higher-level software, such as OSF/1 AD, may then provide the abstraction of a single machine image spanning all cooperating nodes. Tasks (or even UNIX processes) may be spawned on any or all nodes, with or without load-levelling. Some systems may even choose to implement transparent task migration.

Because Mach is a communication-oriented microkernel, the local and remote IPC subsystems are crucial to the performance and robustness of the system as a whole.

1.3 Deficiencies Of NORMA IPC Version One

Today's system, although available in a reasonably robust commercial implementation, suffers several drawbacks. First and foremost, `NORMA IPC` was developed as a prototype and has been asked to provide levels of robustness and performance never originally envisioned. As a prototype, `NORMA IPC` successfully demonstrated the viability of extending Mach semantics across cooperating nodes.

1.3.1 Performance

Performance suffers from limitations on message transmission and protocol implementation. The underlying communications protocol assumes an unreliable transport and thus implements a stop-and-wait protocol to guarantee reliable data transfer. NORMA allows only one message at a time to be sent between nodes, although messages destined for separate ports have no inter-dependencies. Even within a message, the sender transmits one piece at a time and waits for the receiver to acknowledge it. There is no provision for transmitting large amounts of data at once.[1] This excessive handshaking throttles throughput even on unreliable media and is completely redundant when layering NORMA IPC on a reliable transport.

NORMA IPC does not degrade gracefully under load. The mechanisms controlling load are primitive and ill-suited to good performance under light or heavy load.

1.3.2 Robustness

In general, NORMA IPC suffers from message overload scenarios. Message storms directed at a single node cause many problems. Enough fixes have been applied to the current implementation that many storms are avoidable or survivable but this area remains a weak spot.

The NORMA IPC subsystem must examine the types of messages it carries to determine whether some require special treatment, say for pageout handling. This layering violation introduces irritating dependencies between NORMA and the rest of the Mach kernel, leading to long-term maintenance problems.

Finally, sufficient emergency work has been done on the current implementation that it is fragile and difficult to maintain.

1.3.3 Functionality

The following goals were not originally targeted by NORMA IPC:

SMP support	NORMA IPC is a uniprocessor-only subsystem. One implementation has been done that makes NORMA IPC multiprocessor safe, but the subsystem itself remains essentially single-threaded. There is today no version of NORMA IPC that is efficient on a tightly-coupled, shared-memory (SMP) multiprocessor.
Preemption	NORMA IPC is not preemptive.
Fault tolerance	NORMA IPC cannot survive the voluntary or involuntary removal of any nodes from a domain. For that matter, there is no practical provision for dynamically adding new (or reactivated) nodes to a domain.
Scalability	The subsystem allocates most of its data structures statically, configured for the maximum domain size ever expected. In a domain consisting of thousands or tens of thousands of nodes, NORMA IPC consumes too much memory to be generally practical.

[1] One vendor has implemented a bulk data transport optimization in Version One but other message transmission limitations within NORMA IPC still prevent high utilization of available hardware bandwidth.

The NORMA subsystem was never originally designed to handle any of these functional goals. While support for any or all of these needs could be retrofitted, the result would be of very poor quality.

1.4 Design Goals For NORMA IPC Version Two

The Open Software Foundation Research Institute has committed to a complete redesign and reimplementation of NORMA IPC to provide a new subsystem capable of meeting the following requirements.

1.4.1 Performance

In architecting this system, we have focussed directly on the requirements of high performance multicomputers (such as the Intel Paragon). While we believe the resulting design will be a reasonable fit to slower networks, we have made some design decisions that favor more sophisticated hardware that is often not to be found on slower speed networks.[2] This design will define a transport interface suitable for implementing efficient transport software in a variety of settings. Our performance goals include:

- Take advantage of high-speed interconnects and software transports
- Take advantage of reliable interconnects
- Minimize RPC traffic
- Full end-to-end flow control
- Avoid use of timeouts

Sustained high performance may only be achieved with smooth, end-to-end message flow control; otherwise, the system will repeatedly fall into message storms and load conditions that cause poor performance.

We avoid designing NORMA IPC with any dependency on timeout-based algorithms because experience has shown that timeout-based algorithms tend to scale poorly in large configurations.

1.4.2 Robustness

A more robust version of NORMA IPC depends on a new architecture for the subsystem. Responsibility for data movement must be separated from responsibility for extending the Mach IPC abstractions between nodes. The interactions between Virtual Memory and IPC must be completely mapped out to avoid the problems inherent in Version One. Memory consumption must be considered as an integral part of the design to avoid memory exhaustion problems and to permit the support of large domains.

1.4.2.1 Layering

We will separate the semantics and implementation of the software transport from the NORMA IPC layer. The NORMA IPC layer is responsible for extending the Mach IPC abstractions across nodes, including port right management and message ordering. The NORMA IPC layer uses a dis-

[2] For example, the buffer management design is a good fit for low-latency, high-throughput interconnects but when running over Ethernet mandates a copy operation to put incoming data in the right place.

tributed port right management scheme, known as the “principal/proxy” mechanism, a.k.a. “proxies”. This scheme minimizes inter-node communication.

The software transport layer has responsibility for carrying bytes from one node to another. The transport is fully flow-controlled. The transport is opaque to `NORMA`: only well-defined interfaces will be used to communicate between the two layers.

1.4.2.2 Virtual Memory Integration

The close coupling between virtual memory and interprocess communication caused unanticipated problems in `NORMA IPC Version One`, particularly with respect to pageout. In Version Two, resolving VM integration and pageout problems are primary design goals.

1.4.2.3 Memory Consumption

Memory consumption will be limited by transferring a message only when the receiver is ready to accept it. In Version One, an unsolicited message could force a receiving node to over-commit resources. In Version Two, memory consumption will generally be pushed back onto the sending node, so memory consumption and failure modes will be identical to uniprocessor Mach.

1.4.3 Functionality

`NORMA IPC Version Two` will include additional functionality beyond that provided by Version One.

1.4.3.1 Efficient Uniprocessor Support

`NORMA IPC` must provide efficient, multiplexed channels between nodes, capable of supporting simultaneous transmission of independent messages. Message setup and teardown must be as simple and efficient as possible, and should avoid the introduction of delayed operations implemented via auxiliary threads. For instance, when in thread context it should be possible to destroy a message and free up all associated resources without requiring the use of yet another thread.

If necessary, it should be possible to build a uniprocessor-only version of the `NORMA` subsystem that eliminates all SMP overhead.

1.4.3.2 Efficient SMP Support

The design and implementation of Version Two should permit efficient operation on shared-memory multiprocessors including up to 20-30 processors. Locking should be designed to reduce contention.

We assume that all underlying SMP hardware supports atomic access to individual cache lines, with a cache line width of at least 32 bits. In other words, it should be possible to read a `long` or smaller quantity while that location is being written, without taking a lock; the reader must obtain the old value or the new one but never a result consisting partly of the old value and partly of the new one.

1.4.3.3 Scalability

The new `NORMA IPC` must support domains of tens of thousands of nodes, assuming that each node in the domain has sufficient memory to allocate the data structures necessary for communicating with all current senders and receivers.

All data structures representing sending and receiving nodes will be allocated dynamically, so that NORMA IPC generally does not need to be configured for a specific domain size.

1.4.3.4 Multiple Transports

NORMA IPC supports one software transport interface. That transport may select various lower-level software transports and/or physical media for its connections but none of the details show through to NORMA. Routing and forwarding issues, if any, are entirely in the province of the transport.

1.5 Deferred Goals

1.5.1 64-bit and 32-bit Interoperability

This phase of the NORMA IPC rewrite project does not generally consider the need to interoperate within a domain between nodes with different native word sizes.

1.5.2 Differing Page Sizes

This phase of the NORMA IPC rewrite project does not consider the need to interoperate within a domain between nodes with differing VM page sizes.

1.5.3 Asynchronous VM Faults

This phase of the NORMA IPC rewrite project does not include enhancing the VM system to permit the asynchronous initiation of fault handling. However, this feature, as described in “Asynchronous Virtual Memory Faults” on page 66, is highly desirable.

1.5.4 Untrusted Pagers

This phase of the NORMA IPC rewrite project does not guarantee to support untrusted pagers. Although we intend to support untrusted pagers, time may not permit resolving every open issue associated with this problem. In particular, supporting untrusted pagers requires implementing asynchronous VM fault handling (see section 1.5.3).

1.5.5 Fault Tolerance

This draft of the design contains a placeholder, Chapter 7, for issues relating to fault tolerance and/or fault isolation. However, the first phase of the NORMA IPC rewrite does not consider these issues in detail.

1.5.6 Membership Services

Determining membership in a cluster is a critical service. However, the first phase of the NORMA IPC rewrite does not consider these issues in detail. A placeholder (Chapter 8) has been left for this portion of the design.

1.6 Related Documents

Two documents are essential reading along with this one. One document describes the original set of requirements imposed on the NORMA IPC rewrite project. The other document describes the transport interface (KKT) defined for remote procedure calls and bulk data transport between two cooperating kernels. NORMA IPC relies on the KKT interfaces.

NORMA IPC Version Two: Requirements. Alan Langerman, editor, Open Software Foundation Research Institute. August, 1993.

Mach Kernel to Kernel Transport Interface. Open Software Foundation Research Institute. January, 1994.

The reader may also find one or more of the following documents useful for understanding the origins and purposes of NORMA IPC.

NORMA IPC: A Task-to-Task Communication System for Multicomputer Systems. Bill Bryant, Open Software Foundation Research Institute. September, 1993.

A Fast Mach Network IPC Implementation. Joe Barrera III, Mach II Symposium, Usenix Association. November 1991.

Memory Exhaustion Via Reliable NORMA IPC Transport. Alan Langerman, Open Software Foundation Research Institute. February 1993.

A Fast and General Implementation of Mach IPC in a Network. Hilarie Orman, Mach III Symposium, USENIX Association. April 1993.

UNIX File Access and Caching in a Multicomputer Environment. Paul Roy, Mach III Symposium, USENIX Association. April 1993.

∇ Dejan’s task migration papers.

1.7 Terminology

Interconnect	The hardware medium connecting all nodes in a NORMA domain. Examples include Ethernet and the Intel Paragon mesh.
KKT	Kernel to Kernel Transport. The software layer responsible for implementing remote procedure call and bulk data transport between cooperating kernels.
NORMA or NORMA IPC	The layer(s) of software responsible for implementing Mach IPC abstractions across a multicomputer or cluster domain. This layer sits between the transport and local Mach IPC.
NORMA IPC Version One	Also, V1 or Version One. The first implementation of NORMA IPC.
NORMA IPC Version Two	Also, V2 or Version Two. The design and (coming) implementation of NORMA IPC analyzed in this document.
Origin node	The node on which a receive right is created.
Principal port	A receive right in NORMA IPC.
Proxy port	A place holder for a non-local port. The proxy may represent a migrated send or receive right.
Queue limit breaking	When a message is enqueued on a message queue even though the

	message queue already has messages enqueued up to its queue limit, the new message “breaks the queue limit”.
Residence node	The node on which a receive right currently resides. This node need not be the same as the receive right’s origin node.
Transport	The layer(s) of software and/or hardware responsible for low-level communications between cooperating kernels. The transport defines RPC and bulk data transport abstractions. See KKT.
∇	This symbol denotes a paragraph, section or chapter visibly in need of additional text or changes to existing text. As the design becomes more concrete, these symbols will refer to smaller portions of text (e.g., paragraphs or sub-sections rather than chapters).

∇ 1.8 Roadmap

Overview of design document.

You may find that, throughout this document, concepts and data structures are used before they are defined. We ask for your patience: please read through the entire document before commenting on areas that would otherwise appear incomprehensible. Your thoughts on how we can improve the organization and presentation of this tome are most welcome.

CHAPTER 2 Data Type Definitions

In this chapter, we summarize the changed and new data structures and data types introduced by NORMA IPC. We indicate changes to already-existing data structures by marking the changed fields in bold face.

2.1 node_name

A node name is transport-independent and defined to be of a fixed length across a NORMA domain. We define a type, `node_name`, that NORMA uses to contain a node name. This name is opaque to NORMA. The `node_name` is not used to directly index arrays or otherwise require pre-allocation of data structures.

```
typedef unsigned int       node_name;        /* at least 32 bits */
```

The transport is responsible for converting a `node_name`, if necessary, to whatever transport-dependent naming information is required. In practice, we define the `node_name` as a 32-bit quantity, allowing over 4 billion nodes in a single NORMA domain. If more routing information is required, the transport can use a `node_name` as an index, hint or address to find the additional information.

A `node_name` is unique; two different `node_name` values may not refer to the same node. Thus, NORMA may compare two `node_name` values for equality.

2.2 port_id

A port identifier is a 32-bit quantity that uniquely identifies a port created on a given node. A `port_id` by itself is not sufficient to uniquely identify a port within a NORMA domain.

```
typedef unsigned int       port_id;          /* at least 32 bits */
```

However, in conjunction with a `node_name`, a `port_id` specifies enough information to uniquely identify a port (see 2.3). We assume that a node always allocates a unique, non-repeating `port_id` for each instance of a port. In practice, we are not concerned with the problem of `port_id` rollover. For details, consult section 4.1, "Network Port Name", on page 20.

2.3 uid_t

The UID is the network-wide name for a port. It is a 64-bit quantity consisting of a `node_name` and a `port_id`.

```
typedef struct uid {
```

```
        node_name         origin;           /* node that constructed it */
        unsigned int      identifier;       /* unique node-relative id */
} uid_t;
```

A UID is constructed for a port when an associated right first becomes accessible to a remote node. The UID never changes, even when a receive right migrates from one node to another. Therefore, it is possible that the `node_name` of a receive right's UID does not match the `node_-name` of the right's residence node.

All UIDs generated by all nodes in the same NORMA domain have the same size and format.

2.4 handle_t

The `handle_t` is a 32-bit, opaque identifier used by the transport to identify an endpoint.

2.5 kmsg

The `kmsg` data structure is defined as follows:

```
typedef struct ipc_kmsg {
        struct ipc_kmsg   *ikm_next, *ikm_prev;
        vm_size_t         ikm_size;
        mach_msg_header_t ikm_header;
} *ipc_kmsg_t;
```

The kmsg is defined in this way because the header may be followed by an arbitrary amount of inline message data. Unlike the kmsg defined by Version One, this definition is precisely the same as that of the non-NORMA kmsg.

However, note that the `ipc_marequest_t` structure present in earlier versions of Mach has been eliminated. This structure existed only to support send notifications (the mach_msg SEND_-NOTIFY option). Send notifications have been deleted from the revised Mach specification.

2.6 meta_kmsg

The purpose of the `meta_kmsg` is to provide a placeholder for a remote `kmsg` in a port message queue. The receiving node will not obtain the real `kmsg` until the receiving thread is ready to process it.[1] Thus, the `meta_kmsg` contains the necessary information to retrieve a message from a remote node. However, because a port's message queue may contain a mixture of `meta_kmsg` and `kmsg` entries, a `meta_kmsg` must also cleanly overlay a `kmsg`. The `meta_kmsg` definition is:

```
typedef struct meta_kmsg {
        struct ipc_kmsg   *ikm_next, *ikm_prev; /* matches kmsg def */
        vm_size_t         ikm_size;         /* matches kmsg definition */
        mach_msg_bits_t   msgh_bits;        /* first ikm_header field */
        handle_t          ikm_handle;       /* overlays msgh_size field */
```

[1] Logically, the receiving NORMA request the real `kmsg` in the context of the receiving thread. However, the transport may have already buffered the `kmsg` on the sender or the receiver, waiting for the receiver to request the bytes comprising the `kmsg`.

```
        ipc_port_t          dest_port;          /* ... msgh_remote_port */
} *meta_kmsg_t;
```

We define a new bit in the mach_msg_bits_t structure, MACH_MSGH_BITS_META_KMSG, to distinguish between a meta_kmsg and the real thing. The definition of unused message bits changes accordingly.

The ikm_size and ikm_handle fields in the meta_kmsg comprise the necessary information for retrieving the remote message. The ikm_size describes the size of the inline kmsg and the ikm_handle is the transport handle needed to identify the remote data. Once the inline kmsg has been obtained, the receiver may walk the inline portion of the message to bring over any remaining out-of-line data.[2]

The destination port pointer is necessary for two reasons. First, it is necessary for various pieces of IPC code to examine a kmsg's destination port. For example, moving a port out of a port set requires extracting from the port set message queue all kmsgs (and now meta_kmsgs) that were bound for the port.

Second, the destination port's UID is not passed over the wire in the kmsg itself. The UID must somehow be retained until the kmsg arrives and the port pointer can then be stuffed into the kmsg header, where it belongs.

2.7 port

The port data structure definition differs only by the addition of pointers to NORMA- and XMM-specific data structures. However, locking rules change for some of the existing port fields and the use of the ip_nsrequest field changes slightly.

```
typedef struct ipc_port {
        [...]
        mach_port_mscount_t         ip_mscount;[3]
        mach_port_rights_t          ip_srights;
        mach_port_rights_t          ip_sorights;

        struct ipc_port             *ip_nsrequest;
        [...]
        struct ipc_pset             *ip_pset;
        mach_port_seqno_t           ip_seqno; /* locked by message queue */
        [...]
        struct ipc_mqueue           ip_messages;
        struct ipc_thread_queue     ip_blocked;
#if     NORMA_IPC
        norma_ipc_port              *ip_norma;/* NORMA-specific data */
#endif
```

[2] On a 32-bit architecture, the size of a meta_kmsg is 24 bytes. The size could be reduced to 20 bytes by noticing that pointers are always word aligned, so the bottom two bits of an ipc_marequest_t will always be zero. If we further constrain a handle_t to be 32-bit aligned, then we can form a union from the ipc_marequest_t and the handle_t, using the bottom two bits of the word to distinguish between the two cases. However, stealing the lower two bits of an ipc_marequest_t is non-portable.

[3] Given optional NMS detection, can these fields also be optional? Beware time trade-off!

```
#if     NORMA_XMM
        xmm_ipc_port            *ip_xmm; /* XMM-specific data */
#endif
} *ipc_port_t;
```

2.7.1 No More Senders

We define two flavors of port where before only one existed. One flavor of port supports all semantics *including* No More Senders (NMS) detection. The second flavor of port supports all semantics *except* NMS detection. The second flavor does not track send and send-once rights throughout the network, thus significantly reducing the messaging and maintenance cost of port migration. Distinguishing between the two flavors is done by examining the port's `ip_nsrequest` field: when the field is set to -1, the port does not permit NMS detection.

2.7.2 Locking

∇ Port data structure locking is modified slightly so key fields are available from interrupt level to permit delivery of a `meta_kmsg` or `kmsg` from interrupt level to a blocked receiver. (The presence of one or more blocked receivers suffices to guarantee that port exists, is valid, has no queue limit problem, etc.) This only appears to require access to the port's sequence number and blocked receivers queue, plus port set. Failure to deliver a message directly to a thread results in an enqueue operation in an auxiliary thread context.

2.8 norma_ipc_port

This structure contains the information necessary to extend a local port for remote operations.

```
typedef struct norma_ipc_port {
        unsigned int            nip_is_proxy:1,         /* flags */
                                nip_is_special:1,
                                nip_forward:1,
                                nip_migrate_state:3;
        uid_t                   nip_uid;                /* network name */
        node_name               nip_dest_node; /* proxy: residence node */
        long                    nip_transit;
        long                    nip_remote_sorights;
} norma_ipc_port;
```

This structure is generally governed by the port lock of its parent port structure. Once allocated, it is not destroyed until the entire port is freed.[4]

2.9 xmm_ipc_port

The following structure contains the information necessary to extend a local port for remote XMM operations.[5]

[4] The **`ipc_port_release`** macro becomes a function call.

```
typedef struct xmm_ipc_port {
        long                    xip_object_refs;
        struct ipc_port         *xip_object;
}
```

This structure is only allocated when a port represents an object managed by XMM. This structure remains allocated until the entire port is freed.

2.10 dipc_port_name

The `dipc_port_name` structure stores the relationship between a port structure (principal or proxy) and its network name. This structure is an element of the `dipc_port_name_table`.

```
typedef struct dipc_port_name {
        struct dipc_port_name   *next;
        ipc_port_t              port;
        unsigned int            ref_count;
} dipc_port_name;
```

The UID can be found from the port pointer, of course.[6]

A `dipc_port_name` is reference counted to determine when it is possible to remove the entry from the table.

∇ 2.11 thread

The thread structure changes slightly if we choose to support the current RPC abort semantic resulting in non-volatile memory. Locking rules for accessing the thread's IPC-related data structures -- the ith_next, ith_prev, ith_state, msize, kmsg, and ith_seqno -- change.

```
struct thread {
        [...]
        /* IPC data structures */
        struct thread           *ith_next, *ith_prev;
        mach_msg_return_t       ith_state;
        union {
                mach_msg_size_t   msize;   /* max size for recvd msg */
                struct ipc_kmsg   *kmsg;   /* received message */
        } data;
        mach_port_seqno_t       ith_seqno; /* seqno of recvd message */

        struct ipc_kmsg_queue {
                struct ipc_kmsg   *ikmq_base;
```

[5] If the `xmm_ipc_port` does not expand, it may be preferable to include these fields directly in the port structure to save space.

[6] Implementation detail: to speed lookup, two levels of indirection can be eliminated at the cost of expanding the structure to include a duplicate copy of the UID. The speedup may be insignificant, however.

```
        } ith_messages;
        [...]
};
```

TBD.

CHAPTER 3 Message Delivery & Flow Control

3.1 Semantics

For NORMA IPC to succeed, the following cases must be handled correctly, some with great efficiency:

M1. Simple, small inline messages.

M2. Pageout messages containing one or a cluster of pages.

M3. Simple, small inline messages with an interesting “chunk” of OOL data, e.g., disk I/O requests of 256K-1M.

M4. Small inline messages containing a very small number of send and/or receive rights.

M5. Arbitrarily complex messages with arbitrarily large inline data, any combination of inline and/or OOL send and receive rights, and any amount of OOL data.

Of course, cases M1 - M4 are subsets of case M5. We have accumulated enough experience to recognize that cases M1 - M4 are interesting scenarios that are very common in an OSF/1 AD system. Cases M1 - M4 should be handled as efficiently as possible.

3.1.1 Ordering

In all cases, NORMA IPC must obey Mach IPC message ordering semantics. Mach defines message ordering only from a single sending thread to a single port; messages from the thread must be received on the port in the same order in which they were sent. Today, NORMA IPC Version One actively alters message order in certain very unusual cases. This behavior is highly undesirable and so Version Two should completely preserve message ordering. It is important to note that Mach IPC makes no guarantee about message ordering with respect to messages originating from different threads, or with respect to messages sent by the same thread to different ports.

3.1.2 Queue Limits

Mach IPC also defines a port queue limit semantic: every port has a queue limit, N, beyond which sending threads should block rather than enqueue their messages. This queue limit is not always enforced. Internally, the kernel may set a flag, MACH_SEND_ALWAYS, on a message, in which case the kernel ignores the target port’s queue limit when enqueuing the message. In this instance, we say that the message breaks the queue limit. The following special cases do not obey queue limit semantics:

Q1. Asynchronous, kernel-generated messages (particularly pageout and XMM).

Q2. Kernel-generated messages using a stolen user thread context (examples include pagein and some notifications).

Q3. Kernel-generated replies to messages targeted at the kernel (e.g., Mach system calls that use messages instead of traps). These replies originate in the context of a user thread.

Q4. Notifications:

- dead name
- message accepted
- no more senders
- port deleted
- port destroyed
- send once
- send timeout

Messages generated in these cases are enqueued on the target port's message queue without regard to the current queue limit. In pathological pageout cases, we have observed 1500 or more pageout messages, each including a page of OOL data, queued to a local pager port. However, in the local case, Mach is able to page out pages from pageable zones, double-page pageouts pending for an external pager, and otherwise take actions that alleviate the effects of essentially unlimited message queue size. We point out, however, that there is no mechanism that guarantees that local Mach IPC won't encounter fatal resource exhaustion.

All other Mach IPC semantics must be preserved, as well. We call out the above issues because they have proven particularly troublesome in the past.

3.2 Flow Control

Many words have been written about flow control in a NORMA domain. Flow control can't be treated as an exclusively transport-to-transport issue. Another mechanism must exist involving higher-level constructs. Flow control could be implemented with strict CSP synchronization, having the transport block the calling thread until the receiving thread is ready to accept data. Unfortunately, it is not easy to restructure Mach IPC to obey strict sender/receiver synchronization. A mechanism on top of the transport is necessary to implement flow control.

3.2.1 Transport Flow Control

The transport is fully flow controlled and optimized for sending data between nodes. Associated with the transport is a limited resource the kernel can use to further limit the amount of data sent to a node.

The transport is initialized with a number of handles, which are assigned to be 'send' or 'receive' handles. This can be visualized as a pool of connection endpoints associated with a class of service (a channel). In normal operation, a kernel would designate a smaller number of receive handles than send handles. When a thread wants to send data to a remote node, a send handle must be obtained from the transport. If a handle is unavailable, the operation may block (at the caller's option) until one becomes available. This limits the number of send operations that can be in progress at a time on a channel.

When a message is sent to a remote node, the receiving transport must allocate a receive handle to establish a connection. Currently, the transport determines the exact algorithm for waiting for the handle to become available, but regardless, only a finite number of receivers will service requests on a channel at any given time.

This provides a first order flow control mechanism between nodes. If all of the receive handles are waiting for slow threads to accept the kmsg and transfer the data, no further requests can be made to the node on that channel until the channel is released back to the transport.

3.2.2 Mach Message Flow

In this section, we analyze the progress of messages between nodes in an abstract sense, without concerning ourselves in great detail with the contents of those messages. In the following section, we analyze the details of message data transfer for cases M1 - M5.

Attempting to apply strict sender/receiver synchronization to Mach IPC causes problems in the pageout path; for all asynchronous kernel messages where the kernel thread is not expected to block; and for user operations that trigger an IPC as a side-effect (e.g., notifications), which may introduce unanticipated dependencies (deadlocks) in server and client code. See cases Q1 - Q4 (section 3.1.2). Although a CSP-like model would make flow control very easy, a Mach implementation would require solutions for all the issues we just listed. Possibly all of these issues can be overcome, but we suspect this is a very large tar pit.

Extending the port queue limit semantic to include mandatory queue limits for cases Q1 - Q4 has the same problems as strict sender/receiver synchronization.

Our solution relies on distributed port queues to convert the distributed problems of cases Q1 - Q4 into local problems. Thus, the semantic and resource issues associated with these cases will be no worse in the distributed case than they are in the local case. At the same time, a receiving node will be able to prevent resource exhaustion when targeted by many sending nodes.

Assume for now that the messages under discussion are simple and well-behaved. In general, when sending a message a sender attempts to enqueue its message on the receiver's message queue, directly. The receive right, or principal, has the same queues of pending messages and blocked senders found in local IPC. The sender hands the message to the transport, which conveys it to the receiving node, where the receiving transport hands the message up to `NORMA`. The sender blocks until the message has been enqueued on the principal's message queue. If the principal's queue limit is exceeded, the sending thread remains blocked until the principal's queue drains and the message is finally permitted to be enqueued. By "drains", we mean that one or more receiving threads remove one or more messages from the queue, thereby permitting an additional message to be enqueued on the principal's message queue. (The mechanism whereby the sender remains blocked and retransmits the message or is awakened is an important detail that will be examined later.) We understand this model fairly well: it is directly analogous to the way in which messages are enqueued for local Mach IPC.

However, this simple model is not robust when dealing with queue limit-breaking messages. As we have already explained, blocking the sending thread is undesirable in these cases. However, pushing the message onto the principal's queue gives rise to the resource exhaustion problems made infamous by `NORMA IPC` Version One. When a queue limit-breaking message (one with MACH_SEND_ALWAYS enabled) encounters a principal with backlogged messages, we define the correct behavior to be this: the message is enqueued on the local proxy port and the sender is permitted to continue. When a queue limit-breaking message encounters a proxy with any queued messages, the message is enqueued on the proxy regardless of the state of the principal's message queue. When an ordinary, queue limit-respecting message encounters a proxy with a non-zero message queue, the message's sender blocks until the queue drains. The principal is always responsible for remembering that a blocked proxy exists with pending messages, and then for requesting those messages from the sender when resources become available. Queue limits are enforced as described in the following table.

	MACH_SEND_ALWAYS	!MACH_SEND_ALWAYS
Local IPC	Ignore Limit; Always Enqueue Message	Obey Limit; Block Sender
Remote IPC	Obey Remote Limit; But Enqueue Message Locally	Obey Remote Limit; Block Sender

These rules push message queue limit problems back to the sending node, reducing the distributed problem to a local problem. Message ordering is preserved as follows:

- Local IPC remains unchanged, hence ordering for messages sent from a thread to a co-located port works trivially.
- No interactions between local and remote threads enqueueing messages matter, as Mach IPC does not guarantee message ordering between threads.
- Remote IPC permits MACH_SEND_ALWAYS messages to be enqueued immediately, as in the local IPC case. Message ordering between MACH_SEND_ALWAYS messages generated by the same thread is maintained by the proxy message queue.
- Ordinary, !MACH_SEND_ALWAYS, messages generated by remote IPC obey Mach message ordering because the sending threads are blocked until their messages can be enqueued. There is no way for a sending thread to generate a second message until the first one is enqueued.
- A thread can't possibly violate message ordering by generating a !MACH_SEND_ALWAYS message followed by a MACH_SEND_ALWAYS message, because the thread blocks until the first message is enqueued.
- A thread can't possibly violate message ordering by generating a MACH_SEND_ALWAYS message followed by a !MACH_SEND_ALWAYS message because the first message is immediately enqueued and the thread blocks attempting to send the second message until the queue drains.

Besides message ordering, additional benefits derive from this model, as follows. Cases Q2 - Q4 behave from the standpoint of threads as if the IPC were local. Thus, Mach semantics are maintained and resource exhaustion issues reduce to those issues already present in the local IPC case. Unlike Version One, flow control applies for these messages when sent between nodes, and they do not arrive at the receiver faster than the receiver can consume them. Version Two will be as robust in the distributed case as in the local case when confronted with all cases Q2 - Q4.

Port queue limits also become trivial to enforce; if the message would exceed a queue limit, and the message being sent is subject to queue limit enforcement, the sender is blocked instead of being permitted to continue.

∇ 3.3 Resource Consumption

Distributing port message queues, in conjunction with the use of a `meta_kmsg` in place of a `kmsg`, suffices to prevents a node from being overwhelmed with messages originating remotely. Many messages sent to the same port will back up on the senders' nodes, while many messages distributed across many ports will only cause the receiver to consume `meta_kmsg` structures. Because a `meta_kmsg` is small relative to a `kmsg`, we do not regard `meta_kmsg` consumption as a problem in practice. Nonetheless, Mach IPC semantics are maintained, because message ordering is preserved and the sender is decoupled from the receiver. (See Chapter 7 for a discussion of the impact of this messaging model on fault tolerant systems.)

CHAPTER 4 Message Representation

A `kmsg` has two possible representations: local format and network format. The local format is the way in which a Mach message is represented when communicated between two tasks on the same node. This format is constructed by the kernel when transferring a message from user-mode into kernel-mode. When sent off-node, a local message must eventually be converted into network format before transmission. The entities represented in local and network format are:

- inline data
- inline port rights (message header as well as message body)
- out-of-line port rights
- out-of-line memory

At message copyin time, local Mach IPC copies the entire inline kmsg from user space into a (wired) kernel buffer. If the message is simple, the algorithm terminates. Otherwise, Mach scans the message and converts user references to ports and OOL memory into an internal representation. A port name is converted into a pointer to a port, an `ipc_port_t`. An out-of-line memory pointer is converted into a pointer to a `vm_map_copy_t`, which in turn describes the OOL region. With the exception of the local and remote ports in the message header, the contents of the message remain untouched after the copyin process completes until the message is received.

At message copyout time, local Mach IPC reverses the process, translating internal port and VM pointers back into user port names and pointers to OOL memory. Translation is performed for ports in the message header as well as any ports and OOL memory pointers in the message body.

The local IPC copyin path is an appropriate place for converting a message to network format. A message bound for a remote node can be detected easily. All of the necessary information is already present, and some economies may be realized by converting the message immediately rather than first representing it in local format and later converting it to network format. There can be a race when the message is bound for a remote port that migrates to the local node while the message is being sent, but this race is thought to be rare. It makes no sense to optimize for this case. By immediately converting the message, the burden of allocating the resources necessary to send the message is placed directly on the sender. The sender blocks if required resources aren't available.

The local IPC copyout path is an appropriate place for converting a message from network format while copying it out to the user. Local format can be avoided.

By recognizing network-format messages in the copyin and copyout paths, the burden of translating messages is placed directly on the senders and receivers of `DIPC` messages. These entities are precisely those that should assume this burden; alternatives such as performing the translation in the bowels of `DIPC` are less desirable because the burden may be shifted from sender and receiver to asynchronous DIPC threads.

A fully-converted, network-format kmsg has the MACH_MSG_NETWORK bit set in its header.

4.1 Network Port Name

4.1.1 UID

We retain the existing notion of a DIPC UID as a port identifier, because a UID provides a location-independent way of identifying a port in a DIPC domain. The existing design of cramming a node number and a unique identifier into 32 bits does not have sufficient bits available for future expansion. Thus, we expand the UID to 64 bits, consisting of a 32-bit node identifier (node_name) concatenated with a 32-bit port identifier (port_id). (See section 2.3, “uid_t”, on page 8.)

When a receive right is first used by NORMA IPC, its port is assigned a UID. This UID follows a receive right and all of its send and send-once rights as they migrate throughout the DIPC domain. Because rights may migrate, to send a message to a receive right it is necessary to specify a destination node (also a node_name) along with a UID. Thus, a proxy port contains the UID matching its principal port and a destination node_name specifying the principal’s residence node. (See section 2.8, “norma_ipc_port”, on page 11.) A principal port need only contain its UID.

A UID is assigned to a port for several reasons. Once a UID has been assigned to a port, it remains associated with the port until the port is destroyed.

4.1.1.1 Receive Right UID Assignment

When creating a send or send-once right from a receive right, to be sent to another node, the port representing the receive right -- the principal -- is assigned a UID so that messages originating from a remote send or send-once right may be directed to the principal.

When moving a receive right for the first time from its origin node to a new residence node, the port is assigned a UID that will follow the receive right forever more. The migration algorithm will eventually leave behind on the origin node a proxy forwarding port, which will retain the UID. (See section 6.7, “Receive Right Migration”, on page 48.)

4.1.1.2 Send Right UID Assignment

A send right inherits its UID from the creating send or receive right. The UID is retained in the proxy port. This UID is sent along with every message so that the receiver may identify the port for which the message is bound.

4.1.1.3 Send-Once Right UID Assignment

A send-once right inherits its UID from the creating send or receive right. The UID is retained in the proxy port. This UID is sent along with every message so that the receiver may identify the port for which the message is bound.

4.1.2 dipc_port_name_table

When a port is known, its UID is easy to determine: the UID is stored in the port structure. However, determining the port given a UID requires a lookup table. To store the relationship between a DIPC UID and a port, we define the `dipc_port_name_table`. This table consists of a series of `dipc_port_name` entries (section 2.10, page 12), with fast lookup based on UID value. Ports and their associated UIDs are entered into the table for several reasons.

• A principal, of course, must have its UID installed in the `dipc_port_name_table` so that incoming messages may be delivered.

• A send right proxy must also have its UID installed in the table. A second send right arriving on the same node may then be coalesced with the existing right. Coalescing must be done so that all send rights to the same receive right will have the same name. This is accomplished simply by looking up the UID to find the port data structure that already represents the first send right.

• A proxy representing a send-once right need not be entered in the `dipc_port_name_table`. Send-once rights do not require coalescing. Eliminating send-once rights from the table optimizes the use of send-once rights (a very common case) and substantially reduces the table's memory requirements.[1]

The `dipc_port_name_table` is organized as a hash table for fast lookup by UID value. The table will be organized as an array of buckets containing linked lists of UIDs that hash to the same bucket.

Installing a `dipc_port_name` consumes a reference on its related port. The `dipc_port_name` itself also has a reference count because lookup operations may not directly acquire the port lock to update the port reference count. Because a lookup operation may take place from interrupt level, the table lock must be acquired at splnorma which prevents acquiring the port lock.

We define the following operations on the DIPC UID table and UIDs: create, install, retrieve, compare and destroy. Before detailing these operations, we first analyze the locking constraints on the use of the `dipc_port_name_table`.

4.1.2.1 Table Locking

The table is locked by a global lock, the `dipc_port_name_table_lock`. This lock must always be acquired at splnorma. If lock contention proves to be a problem, the global table lock may be split into per-hash chain locks.

The port lock may be held while acquiring the `dipc_port_name_table_lock`. However, it is not possible to first acquire the table lock and then a port lock, because port locks may not be acquired at elevated spl.

4.1.2.2 dipc_uid_allocate

A UID is created when a receive right first enters DIPC. Necessarily, this allocation is relative to the current node. An entry in the `dipc_port_name_table` is created.

---

[1] Unfortunately, an attempt to send a message using a send-once right that encounters a queue full condition on the remote node must enter the send-once right into the UID table for wakeup processing. The implementors may decide whether all rights, including send-once rights, will be entered in the UID table for convenience. Alternately, the send-once right UID could be entered into the table at blocked senders time.

```
dipc_return_t
dipc_uid_allocate(
                    ipc_port_t          port);
```

Context: *Thread*

Side effects:
- Port reference count is increased by one
- Port’s `nip_uid` is set
- A new table entry is created with reference count of 1

Return values:

DIPC_SUCCESS	operation completed successfully
DIPC_DUPLICATE	entry already exists in table

This routine may block for memory allocation; therefore, the caller may not hold simple locks across this call. This routine raises processor priority to splnorma and acquires the table lock.

It is possible for two threads to race entering a UID for a port. Because it is not possible to keep the port lock across a call to this function, one thread may discover that a UID has been installed. This thread will receive the DIPC_DUPLICATE error status, and should then use the UID installed in the port.

4.1.2.3 dipc_uid_install

On occasion, it is necessary to add an explicit UID to the current node’s UID table. This case arises when migrating send and receive rights.

```
dipc_return_t
dipc_uid_install(
          ipc_port_t          port,
          uid_t               uid);
```

Context: *Thread*

Side effects:
- Port reference count is increased by one
- Port’s `nip_uid` is set
- A new table entry is created with reference count of 1

Return values:

DIPC_SUCCESS	operation completed successfully
DIPC_DUPLICATE	entry already exists in table

This routine may block for memory allocation; therefore, the caller may not hold simple locks across this call. This routine raises processor priority to splnorma and acquires the table lock.

See section 4.1.2.2 for a discussion of DIPC_DUPLICATE.

4.1.2.4 dipc_uid_lookup

Retrieve the port based on the UID. Return with a reference on the `norma_port_name` entry. The port data structure is guaranteed to remain available subsequently, although the port itself may still become dead.

```
dipc_return_t
dipc_uid_lookup(
        dipc_uid          uid,
        ipc_port_t        *port)
```

Context:	*Thread or Interrupt*
Side effects:	- Table entry reference count is increased by one
Return values:	
DIPC_SUCCESS	operation completed successfully
DIPC_NO_UID	no such entry in the table

When successful, this operation returns with a pointer to the UID's port. However, the port reference count remains unaltered; only the increased `norma_port_name` reference count prevents the port from being deallocated. Use **`dipc_uid_port_reference`** to adjust the port and `norma_port_name` reference counts.

If this function returns DIPC_NO_UID, no such port exists any longer on this node. The caller may infer that the port is dead or dying.

The table entry reference acquired by this routine may only be disposed in thread context, as it may be necessary to deallocate the table entry and then the port.

This routine raises processor priority to splnorma and acquires the table lock.

4.1.2.5 dipc_uid_port_reference

After translating a UID to a port pointer, it is necessary to take out a reference on the port.

```
dipc_return_t
dipc_uid_port_reference(
        ipc_port_t        port);
```

Context:	*Thread*
Side effects:	- Port reference count is increased by one
	- Table entry reference count is decreased by one
	- The table entry may be deallocated
Return values:	
DIPC_SUCCESS	operation completed successfully

This routine does the following:

1. Acquire port lock
2. Increase port reference count

3. Raise priority to splnorma and acquire table lock
4. Decrement `norma_port_name`'s reference count
5. If the reference count reaches zero:
   - Deallocate table entry
   - Decrement port reference count (matches reference acquired by **`dipc_uid_allocate`** or **`dipc_uid_install`**)
6. Release table lock and lower priority
7. Release port lock

The port is guaranteed to exist with a reference count of at least one when this routine completes. However, the table entry may be deallocated if this routine consumes the last reference to the `norma_port_name`.

The caller of this routine should check whether the port is alive before continuing.

4.1.2.6 dipc_uid_destroy

Attempt to reclaim a UID table entry.

```
dipc_return_t
dipc_uid_destroy(
        ipc_port_t          port);
```

Context:	*Thread*
Side effects:	- Table entry reference count is decreased by one - The table entry may be deallocated - The port reference count may be decreased by one
Return values:	
DIPC_SUCCESS	The table entry was found and its reference count was decremented; the entry may have been deallocated
DIPC_NO_UID	The port had no associated table entry

This routine decrements the table entry's reference count and, on reaching zero, deallocates the table entry. If the entry is deallocated, a reference is released on the port. This reference matches the one acquired by **`dipc_uid_allocate`** or **`dipc_uid_install`**. The caller should already hold a reference on the port to prevent the port from being deallocated by this routine.

4.1.3 Converting a Port Right's Name to a UID

A port right's name is converted to a UID by first translating the port right's name into a pointer to the associated port structure and then looking up or creating the port's UID. The resulting UID is inserted into the message in place of the original port right name. When preparing a port name to be sent off-node, additional activities may be required, such as maintaining transit count information for no more senders ports. These cases break down as follows:

- Converting a send right

- Converting a send-once right
- Converting a receive right

4.1.4 Converting A Send Right To Network Format

In general, when converting a send right, the port's name in the message is replaced with its UID. The send right may already be a proxy for a remote receive right, in which case we take the following steps:

1. For a port tagged for no more senders detection, update transit counts. (May require obtaining more transits.) Decrease the local send right count by one.
2. The port already has a UID assigned, which is used to replace the send right's name in the message.
3. If this is the last send right on this node, clean up any left-over data structures.

Alternately, the send right may be associated with a local receive right:

1. If the port is marked for NMS detection, update transit counts; also, reduce local send right count by one.
2. If the receive right has never before been exported to a remote node, assign a new UID to the port.
3. Use the port's UID to replace the send right's name in the message.

4.1.5 Converting A Send-Once Right To Network Format

In general, when converting a send-once right, the port's name in the message is replaced with its UID.[2] As with converting a send right, two cases arise. In the first instance, the send-once right may already be a proxy:

1. A UID already exists for this port, which is used to replace the send-once right's name in the message; decrease the local send once right count by one.
2. If the port has no other uses, it may be removed.

In the second instance, the send-once right refers to a local receive right:

1. For a port tagged with NMS detection, reduce local send once right count by one and increase count of remote send once rights by one.
2. If the port has never before been exported, assign a new UID to the port.
3. Use the port's UID to replace the send-once right's name in the message.

4.1.6 Converting A Receive Right To Network Format

Preparing a receive right for remote transmission is a more complex activity than preparing

∇ [2] Consider optimizing send-once right handling for the common RPC case. For instance, DIPC might maintain a cache of ports that only require adding a UID to represent a single send-once right.

either a send or a send-once right. In this case, it is also necessary to convert to network format any messages queued on the migrating receive right.

1. If the port has never before been exported, assign it a UID.
2. Walk the local message queue, converting messages to network format. This process is recursive: a message may contain ports that in turn have more queued messages. We don't expect a recursive walk to be a problem in the ordinary case. If absolutely necessary, it is possible to code this walk in a non-recursive fashion.
3. Port becomes "remote" and is marked as "in-transit".
4. Use the port's UID to replace the receive right's name in the message. With untyped IPC, there is sufficient space between the type descriptor (which has a 32-bit field at the end that can be used) and the port name (itself 32 bits) to contain the uid_t.
5. Side-effect: because the port appears to be remote, messages received on the port between copyin time and actual migration time will be stored in network format.

Thus, during the later phases of receive right migration, all messages involved will already have been converted to network format.

4.2 Converting a Kmsg's Port Rights to Network Format

On a machine architecture where the size of a UID -- 64 bits -- is less than or equal to the size of a port name, conversion should be done in place. The ports in the kmsg header, the ports in the message body and any out-of-line ports are converted from a port name to a `DIPC` UID by first looking up the local port and then extracting or constructing the UID. Because the UID is small enough relative to the port name, the port name may be overwritten with the UID.

Because the size of a UID -- 64 bits -- exceeds the size of a port name / `ipc_port_t` on some architectures, we consider at length the problem of representing ports as UIDs in network-format kmsgs. We begin by considering the problem of when to convert a port name or ipc_port_t into a UID.

In an outbound message, ports may be found in the following places:

- Kmsg header
- Kmsg body
- Out-of-line port array

These observations apply to send, send-once and receive rights.

4.2.1 Converting Ports In The Message Header

A Mach message header contains two ports, `msgh_local_port` and `msgh_remote_port`. The remote port is always present -- at send time, it is the name of the destination port. The optional local port specifies the reply port for an RPC. These ports are part of the `kmsg` header, as well, and must be converted to network format.

On those architectures where the size of an `ipc_port_t` is less than the size of a `uid_t`, the local port name will be converted to a uid_t and stored in the message header using the existing msgh_local_port field plus the next field in the message header, msgh_seqno. The sequence number has no use while a message is in transit. However, the receiver must be careful to convert the local port name's uid_t back to an ipc_port_t before stuffing the sequence number into the message header.

The remote port name will be converted to an ipc_port_t, and will remain as an ipc_port_t throughout the message transmission process. The UID of the remote port will be conveyed to the target node out-of-line but along with the kmsg proper. In other words, the UID will be maintained as a local variable and eventually handed to the transport as part of the message transmission sequence. The receiver will obtain both kmsg (or meta_kmsg) and UID from the transport. At the time the receiver looks up the target UID, it places the resulting ipc_port_t in the kmsg (or meta_kmsg) header.

Thus, the kmsg structure needs to expand only to accommodate the UID of the local port.

4.2.2 Converting An Inline Port

The type descriptor for a port (in conjunction with the port name) is sufficiently large to contain a 64-bit UID. A port name may be converted in place without requiring the allocation of additional memory.

4.2.3 Converting An Inline Port Array

Each port in an inline port array has its own type descriptor. The type descriptor (in conjunction with the port name) is of sufficient size to contain a 64-bit UID. Thus, a port name may be converted in-place to a UID without requiring additional memory.

4.2.4 Converting An Out-of-Line Port Array

An array of OOL ports is an array of 32-bit port names, which must be converted to 64-bit UIDs. The local IPC copyin path already expects to allocate wired kernel memory to contain the ipc_port_t's resulting from copying in the message's out-of-line port array(s). We simply increase the size of the intended memory allocation so that the resulting region is large enough to contain UIDs rather than port names / ipc_port_t's. After the names have been copied in from the user task, the port array is walked *backwards* to convert each name to its UID. Thus, the names can be expanded in place, using the same buffer to first copyin the port names and then convert them to network format.

Note that this behavior matches the behavior of local IPC, which already allocates (wired) kernel memory when copying in port arrays. Additionally, by converting all ports at the outset, control flow is simplified and DIPC need not support on-the-fly UID conversion, UID conversion buffers, or other constructs.

4.3 Out-of-Line Memory

Representing out-of-line memory carried in a message must meet the following constraints:

1. The memory must always be pageable, except for that portion actively being processed by the transport.
2. Memory must be represented in such a way that it can be conveyed

quickly from node to node.

3. The representation of OOL memory must be compact.
4. Ideally, copyin processing should produce only one representation of OOL memory.

4.3.1 Analysis of Copy-On-Write Semantics

When moving OOL data, Mach defines the semantic as a logical copy. Modifications by the receiver to the received data should not be visible to the sender. Similarly, modifications by the sender to the buffer containing the sent data should not be visible to the receiver. In practice, this semantic is implemented by copy-on-write (COW) processing. Unfortunately, COW processing is expensive.

In Version One, most message send operations avoided COW processing because the sending thread blocked until the associated OOL data was sent to the receiving node -- and the data were always sent immediately. Because the sender was blocked until the receiving node obtained the data, it was not necessary to mark OOL data for COW.

However, blindly sending the data is precisely the worst behavior with respect to best system performance and overall robustness. In Version Two, the data will generally stay on the sending node until the receiving *thread* (not node) accepts the data. We cannot block the sender waiting for the receiving thread, so it becomes necessary to mark the sender's data COW -- in some circumstances. The following analysis details the cases in which COW processing may be avoided. The cases are broken down based on Mach IPC features.

4.3.1.1 RPC

For the sending side of a remote procedure call, COW processing may be avoided. The sending thread is blocked until the receiving thread responds, so any OOL data contained in the sender's message may be left volatile. To identify remote procedure calls, we introduce the MACH_RPC_MSG bit in the message header. This bit can be set by the programmer or by MIG.

An important special case exists when an RPC is aborted. If the RPC is aborted while still in the send-side processing, no further consideration need be given to the state of OOL memory. However, if the RPC is aborted on the receive-side, it is necessary to fix all OOL memory regions in the kmsg being sent to be COW, if the kmsg has not already been sent off-node. Otherwise, it may be possible for the sending thread to modify OOL data before the data are completely sent. (See section 5.10, “Aborted Message Send”, on page 43.)

4.3.1.2 Device Operations

While device operations may be asynchronous, we nevertheless define all device operations as operating on volatile memory for the duration of the operation. To avoid COW processing, device operations implemented as Mach message calls will be converted to system call traps as part of the Version Two project.

The system call traps avoid performing COW manipulations of the specified OOL regions. Each trap is responsible for packaging its requested operation as a message and then feeding the message into the IPC system without triggering COW activity.

4.3.1.3 Asynchronous Message With Deallocate

Any OOL memory sent with the deallocation option need not be marked COW because the memory is no longer visible to the sender.

4.3.1.4 Asynchronous Message With Flush

Setting the flush option on a message (see “Message Flush Bit” on page 55) causes the sending thread to block until the receiving thread has accepted the message. In this instance, OOL memory will arrive on the receiver before the sending thread could possibly modify it. Therefore, there is no need to mark the OOL memory sent in a flush message as COW. Note that this behavior is nearly equivalent to the Version One behavior, where the sending thread blocked until the data was sent off-node.

4.3.1.5 All Other Asynchronous Messages

All remaining asynchronous messages must have their OOL memory regions marked for COW so that the sender’s modifications to those regions do not become visible to the receiver.

4.3.2 OOL Representation

Out-of-line memory regions will be represented by existing `vm_map_copy_t` technology, with the modification that all ool memory regions (including an ool port region) will be linked together at copyin time. By linking all regions together, the message does not need to be walked again during the transmission processes. Message walking will be confined to two places: copyin and copyout processing.

4.4 Receiving a Network Format Message

The local Mach IPC copyout path is modified to understand network format messages, which can be detected by examining the MACH_MSG_NETWORK bit in the message header. The copyout path walks the inline kmsg, converting ports and ool regions as they are encountered.

∇ 4.4.1 Converting A UID To A Port Right Name

A UID is generally converted to a port right name by looking up the UID in the norma_port_-table to find an existing port or by creating a new port. For a port with NMS detection enabled, it is necessary to maintain transit count information.

The resulting port is translated into a right name and inserted into the message in place of the existing UID.

4.4.1.1 Receiving A Send Right

[TBD]

4.4.1.2 Receiving A Send-Once Right

[TBD]

4.4.1.3 Receiving A Receive Right

[TBD]

4.4.2 Converting A Network Format Kmsg's Port Rights

4.4.2.1 Converting Ports In The Kmsg Header

The destination port has already been converted from a UID to an ipc_port_t by the time the message reaches the copyout logic. The normal copyout logic takes over, replacing the port pointer in the message header with a port right name.

The reply port must be converted from a UID to a port right name.

4.4.2.2 Converting An Inline Port

The copyout path walks the entire kmsg, converting all inline port UIDs into port right names, following the rules outlined for receiving rights in section 4.4.1, page 29.

4.4.2.3 Converting An Inline Port Array

Each port in an inline port array has its own type descriptor. The type descriptor (in conjunction with the space allocated for the port name) contains the 64-bit UID. The UID may be converted to a port right name and the new name overwrites the original UID.

4.4.2.4 Converting An Out-of-Line Port Array

When the copyout path recognizes an out-of-line port array, it kallocs sufficient memory to hold the out-of-line memory region created by the copyin path. Copyout then invokes the necessary `DIPC` logic to obtain the region from the sending node. Copyout then walks the port array, forwards, converting each UID into an `ipc_port_t` and overwriting each UID with the resulting `ipc_port_t`. The copyout path then frees up the remaining kalloc'd memory. The array of `ipc_port_t` entries is then fed to the existing copyout logic that converts it into an array of port names and maps it into the user's address space.

4.4.3 Receiving Out-of-Line Memory

When the copyout path recognizes an out-of-line memory object, it invokes the necessary `DIPC` logic to receive an out-of-line region from the sending node (see chapter 5). The data are received directly into the receiving task's address space.

CHAPTER 5 Message Transmission

This chapter describes the mechanics of transmitting a NORMA IPC message between kernels. This encompasses all of the activities that occur between when the sending thread finishes converting the message to network format on the sending node and when the receiving thread begins to convert the message from network format to local format on the receiving node. The format conversions themselves are described in Chapter 4. For clarity of presentation, the transmission of port rights and out of line port arrays are discussed elsewhere (see chapter 4).

5.1 Overview

The basic sequence of operations for transferring a NORMA IPC message between kernels is:

1. The sending thread calls **`KKT_SEND_CONNECT`** to send the `kmsg` to the receiving node.
2. The transport implementation of **`KKT_SEND_CONNECT`** invokes the **`deliver`** upcall on the receiving node with information about the `kmsg` being sent, but not the contents of the `kmsg` itself.
3. The NORMA implementation of the **`deliver`** upcall invokes a thread that uses this information to enqueue a `meta_kmsg` on the receiving port's message queue.
4. Completion of the `meta_kmsg` enqueue releases the sending thread to continue execution by replying with **`KKT_CONNECT_REPLY`**.
5. The receiving thread dequeues this `meta_kmsg`, extracts the information it contains (including the size of the `kmsg`), and frees the `meta_kmsg`.
6. The receiving thread allocates the memory required to hold the `kmsg`, and calls **`KKT_REQUEST`** with a *receive* request to receive the `kmsg` into this memory. The network bit is set in this `kmsg` to indicate that it is in network format.
7. Completion of `kmsg` transmission on the sending node invokes the upcall specified in the `request_block` argument to **`KKT_SEND_CONNECT`**; this upcall causes the `kmsg` on the sending node to be freed by an internal thread.
8. Completion of `kmsg` reception on the receiving node invokes the upcall specified in the receive `request_block` argument to **`KKT_REQUEST`** on the receiving node; this upcall causes the receiving thread to continue execution (it will proceed to convert the `kmsg` from network format to local format).

This sequence of operations is used to successfully send a `kmsg` of moderate length that does not contain receive rights, out of line memory, or out of line port arrays. Other cases cause the sequence to change in the following ways:

- If the `kmsg` is short, the transport may deliver the entire kmsg instead of just information about it. Among other things, this obviates the need for the receiver to retrieve the `kmsg` from the remote node.
- If there is a waiting receiver, the `meta_kmsg` (or `kmsg` in the previous case) may be delivered directly to the receiver, bypassing the port's message queue.
- The enqueue operation may fail because the port doesn't exist or has been migrated to another node. The reason for the failure is returned to the sender as the result of **`KKT_SEND_CONNECT`**, and changes the sender's behavior.
- The enqueue operation may fail because the port's queue is full or because the receiving node does not have the resources to complete the enqueue operation successfully. In addition to returning the reason for the failure to the sender, the receiving node is obligated to notify the sending node when the conditions are right to retry the operation.
- If the `kmsg` is longer than one page, it may not be possible to transfer the entire message using the above sequence of operations. This is because each page of the message requires a separate physical memory descriptor which implies the pages are wired. The implementation limits the amount of wired memory consumed by any send/receive operation.
- If the kmsg contains one or more receive rights, regions of out of line memory, or out of line port arrays, additional interactions between the sending node and receiving thread are required to transfer these entities. In this case, the interactions will often involve an internal thread on the sending node.

The rest of this chapter describes additional details in the basic transfer sequence, followed by the various modifications. The complete description of receive right transfer can be found in Chapter 6.7.

5.2 Message Transfer Details

This section describes implementation details of the basic message transfer sequence presented in the previous section

5.2.1 Deliver Upcall

The deliver upcall has two arguments of interest: the *handle*, and the *data* pointer. The basic course of action is to place *handle* into `data->subhead` and queue *data* on a kmsg service queue using `data->head` as a link pointer. A service thread will awaken, dequeue the request, verify the port, acquire a `meta_kmsg`, and deliver the `meta_kmsg` on the appropriate port queue.

The UID of the destination port is transmitted in the endpoint parameter. The destination port is located by looking up the NORMA UID in the UID table. If the lookup operation fails, the operation terminates and the sender is informed of the error. A `meta_kmsg` is allocated from a non-pageable zone; if one is not available, memory resources are checked which may result in a resource shortage error being returned to the sender.

The remaining two arguments, *size* and *handle*, are entered into fields of the `meta_kmsg` to be enqueued on that port. The destination port is entered into the `meta_kmsg` as well. The initial *size* argument is always the size of the inline kmsg that was transmitted. Using the *handle*, the receiver is able to request the inline kmsg and determine how much more, if any, of the kmsg needs to be transferred.

5.2.2 Enqueueing a meta_kmsg

Enqueuing a `meta_kmsg` on a port queue is done from thread context as the appropriate locks must be acquired first, hence the **`deliver`** upcall hands the data off to a **`norma_message_delivery`** thread for the enqueue operation. This thread translates the port from *endpoint* and proceeds to enqueue the `meta_kmsg` on the port. The standard function used to enqueue a kmsg is **`ipc_mqueue_send`**. For a `meta_kmsg`, we instead use a new routine, **`ipc_enqueue_meta_kmsg`**. This routine recognizes many of the same error cases handled by **`ipc_mqueue_send`**, complicated by the presence of a `meta_kmsg`.[1] Once the enqueue completes, this thread invokes **`KKT_CONNECT_REPLY`** to indicate this fact to the sending node.

There is one case in which the `meta_kmsg` will not be enqueued on the destination port's message queue. If the destination port is in a port set, then the enqueue takes place on the port set's message queue.

5.2.2.1 Enqueueing to a kernel port

If the destination port's receiver is the kernel, then the `meta_kmsg` is enqueued on a special `norma_kernel` port. Each node has exactly one `norma_kernel` port, created at boot time. One or more **`norma_kernel_delivery`** threads execute infinite loops receiving messages from the `norma_kernel` port and executing the requested kernel operations. Although it is referred to as a port, the behavior of the norma_kernel_port more closely approximates a port set; port queue limits for various kernel ports are respected, and service threads queue up on the kernel port. These threads block in the usual way (e.g., in **`ipc_mqueue_receive`**) when there are no messages on this port.

5.2.3 Dequeuing a Meta-Kmsg

A `meta_kmsg` is dequeued like any other kmsg in **`ipc_mqueue_receive`**. The receiving thread immediately checks to see if this is a `meta_kmsg`, and calls a synchronous version (wrapper) of **`KKT_RECV`** to retrieve the actual `kmsg` (including allocating memory for it). The destination port is then copied from the `meta_kmsg` to the `kmsg`. Control flow then returns to **`ipc_mqueue_receive`** at the point where the destination port is extracted from the `kmsg`.

5.3 Short kmsg transfer

If the kmsg is short, the transport may decide to deliver the entire kmsg as part of the **`deliver`**

[1] Implementation detail: given that a meta_kmsg shares the same layout as the beginning of a kmsg, and is marked with MACH_MSGH_BITS_META_KMSG, it may be better to alter ipc_mqueue_send to recognize the few special cases that apply for a meta_kmsg.

upcall instead of providing information about it. This results in the following sequence of operations:

1. The sending thread calls **KKT_SEND_CONNECT** to send the `kmsg` to the receiving node.
2. The transport implementation of **KKT_SEND_CONNECT** invokes the **deliver** upcall on the receiving node with the contents of the `kmsg`.
3. The NORMA implementation of the **deliver** upcall invokes a thread that uses this information to initialize and enqueue a `kmsg` with the delivered contents. The network format bit is set in this `kmsg`.
4. Completion of the `kmsg` enqueue releases the sending thread to continue execution; **KKT_SEND_CONNECT** provides a return value that the data was sent and the sending thread frees the `sent kmsg`.

The resulting kmsg is then treated like any other kmsg that was delivered from a remote node. In particular, this means that if the kmsg contains other data (e.g., out of line data), the kmsg itself can still be delivered using this optimized path.

5.4 Direct Message Delivery at Interrupt Level

In one important instance, it is possible to deliver an incoming `kmsg` or `meta_kmsg` directly from interrupt level, without resorting to the use of a **norma_message_delivery** thread. If a blocked receiver exists, the message may be delivered directly into the thread without observing the usual port protocols. That is because the existence of a blocked receiver is sufficient to guarantee that "liveness" of the port, and further implies that no queue limit violation can result.

[TBD: describe mqueue locking.]

*[TBD: This text does not belong here. The service thread is guaranteed to run - it must be stack privileged and prepared to cope with resource shortages. If delivering a message will block due to insufficient resources, an error value will be returned to the sending thread on the remote machine in the **KKT_CONNECT_REPLY** mechanism.]*

5.4.1 **deliver* Interrupt Pseudo Code

```
if (there is a waiting thread AND
 the inline_data parameter is set AND
 a kmsg is available to put the data into) {
         Receive the data into the kmsg;
         awaken the thread;
         KKT_RPC_REPLY(NORMA_SUCCESS);
         return;
}
/*
 * Failed fast path: wake up a service thread
 */
Enqueue handle & size;
Awaken service thread;
```

5.4.2 Interrupt Delivery

This is discussed in detail elsewhere in the document and is only mentioned here for completeness. While it is highly desirable to deliver a kmsg during the initial transport interrupt **deliver* call, interrupt-level delivery is only an optimization. For correctness, it must always be possible to deliver a `meta_kmsg` or kmsg from a thread context.

5.5 Enqueue Failures

Enqueueing a `meta_kmsg` or `kmsg` delivered by NORMA may fail because the port does not exist, exists, but is dead, or has migrated to another node. In all three cases, the enqueue does not occur, and an error code is returned via **`KKT_CONNECT_REPLY`** to indicate what happened. In the case of a migrated port, the sender must perform an additional transport RPC to the destination node to retrieve the new location of the port. This new location information will be cached by the sending node.

[TBD: Can we optimize away the additional RPC by jamming a return parameter back through CONNECT_RELPY/SEND_CONNECT in all cases, or just in the error case?]

5.6 Port Queue Full

When an enqueue operation determines that a port's message queue is full, the corresponding sender is blocked via a callback mechanism. This involves the following sequence of events:

1. The receiving node records the sender's node number in a callback map data structure linked to the port. The transport is responsible for defining and accessing this data structure.
2. A return code is provided to **`KKT_CONNECT_REPLY`** (and hence returned from **`KKT_SEND_CONNECT`**) indicating that the sending thread must block until the receiving node allows the send to be retried.
3. The sender marks the port remote blocked and does local queue full processing. If the local port is a send-once right, it must be entered into the UID table. If the kmsg is marked `MACH_SEND_ALWAYS`, it will be enqueued on the port and the thread will continue; else, the thread will queue itself on the ports' blocked send queue and sleep.
4. When a space becomes available in the receiver's message queue, the receiver retrieves a node from the callback map data structure, and sends an RPC to that node.
5. The RPC causes a norma service thread to try to send all of the queued messages on the port. After the kmsg queue is drained, the port is unblocked and any blocked senders will be awakened.
6. The response to this RPC indicates whether there will be a forthcoming send operation. A blocked sender may not be released if its message operation was aborted before the notification RPC arrives. If messages are waiting to be sent, step 4 is repeated.

The above sequence of events has a race condition inherent in it. Should space become available on the receiving node immediately after refusing the connection, a space available RPC could in fact race ahead of the QUEUE_FULL return code. This can easily happen if the sending threads priority is lower than the thread handling the RPC, or on a multiprocessor platform where there are competing CPUs. To guarantee the messages are correctly handled, a block and unblock count are maintained in the NORMA area of the port. In both cases, the threads will perform operations on the port with the port lock held. The unblock operation will increment the port unblock count; if the count is greater than the block count, it will do no further processing as this provides the information that it has raced ahead. The sending thread will increment the port block count and will be able to tell if it can retry the operation (because the unblock raced ahead) or if it must block.

The callback map data structure is a per-port data structure that is created as needed, and managed by the transport. Node numbers are recorded in this structure, with each node being recorded at most once. The transport is responsible for implementing routines for allocation and freeing of callback maps, as well as adding a node to and removing a node from a callback map.

If there are multiple senders on a node, then the queue of blocked senders is attached to the proxy on that node in the usual fashion. For consistency, if there are any blocked senders on the local node, the local node's number is also entered in the callback structure. Retrieval of the local node's number causes a local blocked sender to be awakened instead of sending an RPC to awaken a remote one. The use of a single mechanism to determine which blocked sender to awaken ensures a degree of fairness among competing local and remote blocked senders.

This design of recording node numbers at most once per port reduces space requirements on the receiving node over other data structure designs. This is valuable in cases where a single node is being bombarded by messages from a large number of nodes. This consideration of large fan in situations is also why an active callback mechanism is used in preference to a backoff timeout mechanism, as the latter results in much more inter-node traffic in such a situation.

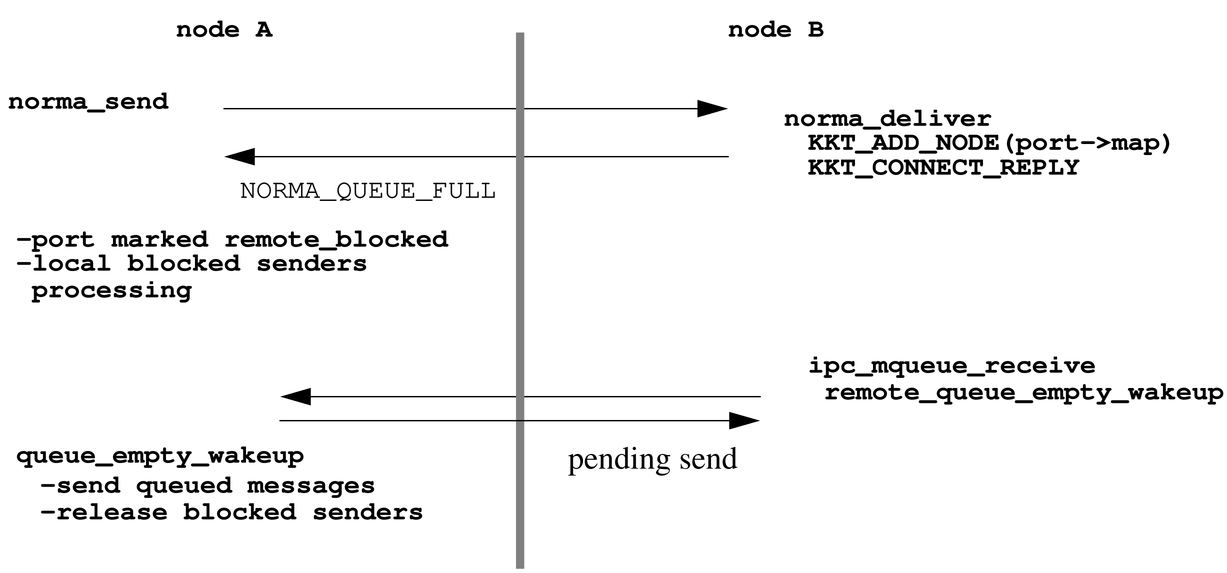

Figure 1: Queue Full Node Interaction Diagram

5.7 Resource Shortages

This section deals with the general problem of resource shortages on the receiving node that cause an enqueue operation to fail. There are two types of such shortages:

- Allocating a `meta_kmsg` or `kmsg` may fail.
- Allocating a callback map may fail, or allocation of memory required by insertion of a node into a callback map may fail. Some transports may guarantee that memory is never allocated for a node insertion operation (Paragon is an example).

In both cases, the enqueue operation fails and returns a resource shortage error code to the sender (See Figure 1); this causes the sender to block awaiting notification from the receiving node that resources have become available. If the sending thread would have blocked anyway due to a full message queue, it may have to go through both a resource shortage callback sequence and a message queue full callback sequence before enqueueing its message.

The notification mechanism for resource shortages is based on a fully pre-allocated callback map, the *static_node_map*. The full preallocation ensures that a node can always be inserted in this data structure without causing memory allocation. Both classes of resource shortages cause the sending node to be entered into this map, and a resource shortage error code to be returned to the sender.

Although resource shortages occur from more than one resource, the `meta_kmsg` is used to determine that resources are available on the node again. When a `meta_kmsg` is returned to the free pool, the `static_node_map` is checked to see if there is a node waiting. A **`remote_unblock_node`** rpc is sent to the waiting node, which in turn will enable the blocked ports and start sending kmsgs. Although there is a risk that the sending node will quickly saturate the receiver by immediately sending a large number of requests, it is relatively easy to fix this behavior if it is determined to be a problem

5.7.1 Sender Response to Resource Shortage

A resource shortage causes the **`KKT_SEND_CONNECT`** call to return a memory shortage error code to the sending thread on the sending node, which causes the sending thread obey local port queue full semantics. A thread that is sending a kmsg marked as `MACH_SEND_ALWAYS` cannot block, requiring that the `kmsg` is enqueued on the port; all other types of `kmsgs` will cause the sending thread to block on the port.

The *node_wait* structure is used to synchronize resumption of services:

```
typedef struct node_wait {
        node_wait_t         next;
        node_wait_t         prev;
        node_name           node;
        ipc_port_t          port_chain;
} *node_wait_t;
```

A global queue head, *blocked_node_queue,* is used to maintain the list of nodes the kernel is waiting on. This queue is only accessed from thread context, so only a lock need be acquired to manipulate the queue.

When a thread is notified of queue full, it will find or establish the `node_wait` structure for the remote node before issuing an RPC requesting a callback when the node has resources to enable receiving.

Although a node may be short of resources, there is no effort in this phase of the design to limit the number of send requests to a remote node. Each thread will discover that a remote node can accept or reject a connection request. The cost of a node lookup to determine the resource state of the remote node is heavy weight, especially when it is the exceptional case that a message cannot be delivered to the remote node.

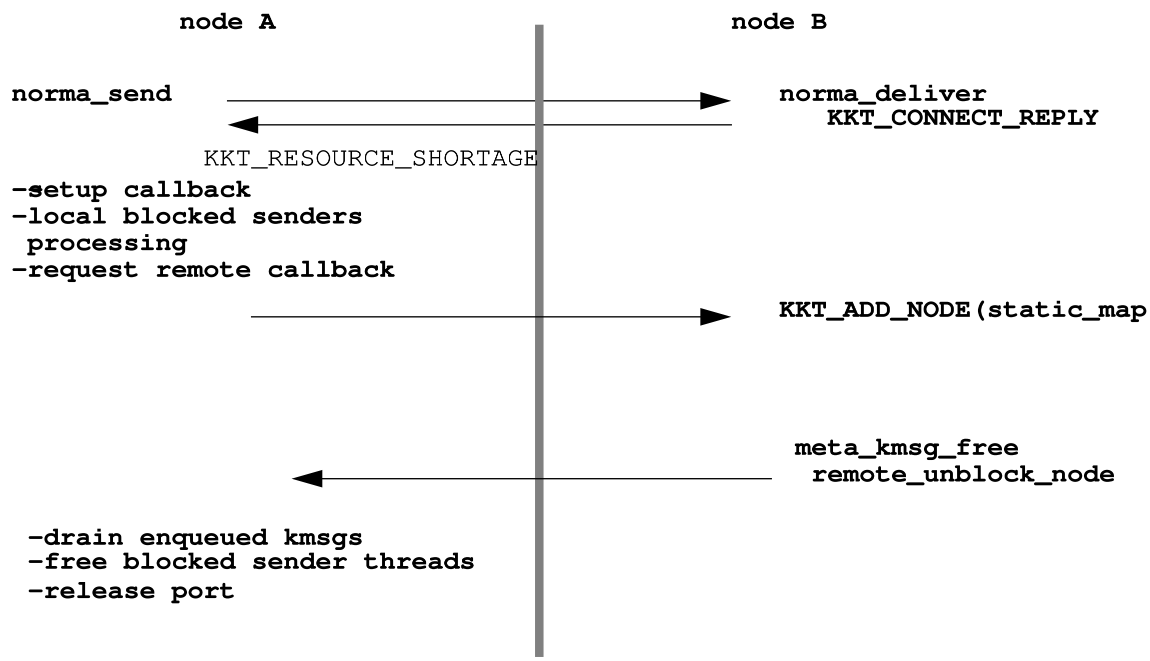

Figure 2: Resource Shortage Diagram

5.8 Transmitting Complex Messages

In the context of the send/receive engine, a complex message can be either a kmsg with out of line ports or data, or it can be an inline kmsg that is larger than a single physical page. If either criteria is met, a `msg_progress` structure is allocated and filled in during the **`copyin`** process.

The `msg_progress` structure is used to drive the prep/pin engine and to maintain the current position of the data being sent. A state machine is implemented to transition at various points in the transmission and prepping of the data. One structure is allocated for each `kmsg` being sent or waiting to be sent.

The transmit engine will maintain two sets of request chains that can be enqueued to the transport to achieve some pipelining of data. Requests are chained together by the user with a callback being located on the final chain element. If the request chain is N requests long, the transmit engine will be able to be prepare N pages while another N pages are being sent.

The `msg_progress` structure takes the form

```
struct msg_progress {
        ipc_kmsg_t        kmsg;    /* kmsg pointer */
        vm_offset_t       offset;  /* kmsg or send_region */
```

```
                vm_offset_t       ool_data;/* first ool region */
                vm_page_t         *ool_array;/* pointer to vm_page_ts */
                int               array_size;/* # elements in ool_array*/
                vm_map_copy_t     *send_region;
                int               send_reg_end;/* index into ool_array */
                int               send_last_size;
                int      pin_start;
                int      pin_end;
                int      prep_end;
                         prep_region;
                int      prep_region_end;
                req_chain*idle_chain;/* points to request chain */
                int      chain_count;/* number of request chains */
                page_errorpage_err_info;/* pointer to page_err_page */
                enum {
                         SENDING,
                         STALLED,
                         RESET_ARRAY,
                         SEND_PAGE_ERR,
                         PAGE_ERR_SEND_DONE,
                         MSG_DONE
                } state;
                simple_lock_tlock;
}
```

5.8.1 Large inline kmsg

If a kmsg is stored on more than one physical page, this is noted in the `msg_progress` structure by filling in the *kmsg* field. The first page will be sent via **`KKT_SEND_CONNECT`** and then nothing will occur until the callback is received. A pointer to the `msg_progress` structure is stored in the `request_block_t` given to the transport, which is always provided back as a parameter to the callback. All of the inline kmsg is located in wired kernel memory, so no prep/pinning operation need to occur on this data.

During the callback, the transmit engine will fill out a single chain of requests and hand them to the transport via **`KKT_REQUEST`**, the final request having a callback associated with it. As each page is placed into a `request_block_t` and handed to the transport, the `msg_progress->offset` field is updated.The entire inline kmsg will be sent before attempting to send OOL data.

Upon completion, the chains of `request_block_t`'s and `msg_callback` structure will be cleaned up and released back to the system and the sending thread will continue.

5.8.2 Additional Kmsg Contents

In addition to the `kmsg` body, a `kmsg` may also contain out of line memory or out of line port arrays. An out of line port array is treated at transmission time as an out of line memory region. These regions follow the `kmsg` body on the connection opened by **`KKT_SEND_CONNECT`**. Both the sending node and receiving thread execute similar code that scans the `kmsg` body to determine what to transfer next, and execute a **`KKT_REQUEST`** operation with the appropriate send or receive requests needed to effect the transfer. Callbacks are used to sequence the transfer and to prepare data for transmission. An out of line memory region containing ports is converted back into an out of line port array in the copyout path (see chapter 4).

5.8.3 Out of Line Data

Out of line (OOL) data transmission is synchronized by a state machine implemented within the `msg_progress` data structure defined above. Figure 3 below encapsulates the various state transitions as the occur in interrupt or thread context.

After the initial **`KKT_SEND_CONNECT`** completes successfully, if OOL data is present the following sequence of events occur:

1. `msg_progress->ool_array` is allocated, `send_region`, `send_reg_end`, and `send_last_size` fields are initialized, and the `state` is set to **SENDING**.
2. Up to 2*PREP_MAX pages are prepped and put into the `ool_array`.

Pointers to the `request_block_t` chains were placed into the opaque area of the `request_block_t` that was used to initiate the **`KKT_SEND_CONNECT`**, which enables them to be recovered when the callback is initiated by the transport.

When the callback is received, the **SENDING** state will be noticed and a number of pages will be pinned to be sent, the number being equal to the number of elements in the request chain. `vm_page_t`'s for the pages are found in the `ool_array` field of the `msg_progress` structure, making this a simple operation from interrupt context. Although the send engine will always try to send the maximum number of pages the request chain can send, an absent page will cause the sends to stall; however, so long as a single page can be pinned, it will be sent as soon as possible.

If a page of the OOL data has been removed from memory, the pinning operation will fail and the state will transition to **STALLED**. When this happens, the request is handed to the prep/fault thread. From thread context, the page can be faulted back in and prepped, then pinned and sent. Again, if the transmit engine cannot send all of the pages, it will send them one at a time if necessary to make forward progress.

The `ool_array` holds a small number of `vm_page_t`'s that have been prepped. The initial implementation will only prep a single region in advance of sending, which further limits the number of elements in the `ool_array`. When the end of a region is reached, `msg_progress` will transition to **RESET_ARRAY** state. The array may be reallocated if it is too small to deal with the largest number of pages, pointers are reset to the next OOL region, and prepping begins anew. This transition will always mean the pipeline has been stopped and must be restarted.

At the end of sending the `kmsg`, `msg_progress` will transition from **SENDING** to **SEND_PAGE_ERR**, which will cause the `page_error` information to be sent to the remote node, and transition to **PAGE_ERR_SEND_DONE**. This state which will clean up the outstanding request chains and transition to **MSG_DONE**.

MSG_DONE is the final state and will further clean up residual data structures. It will destroy the second request chain (if it exists), any `page_error` info, and queue the kmsg to be destroyed by a service thread; **MSG_DONE** is processed from interrupt level and cannot clean up the kmsg itself without risk of blocking.

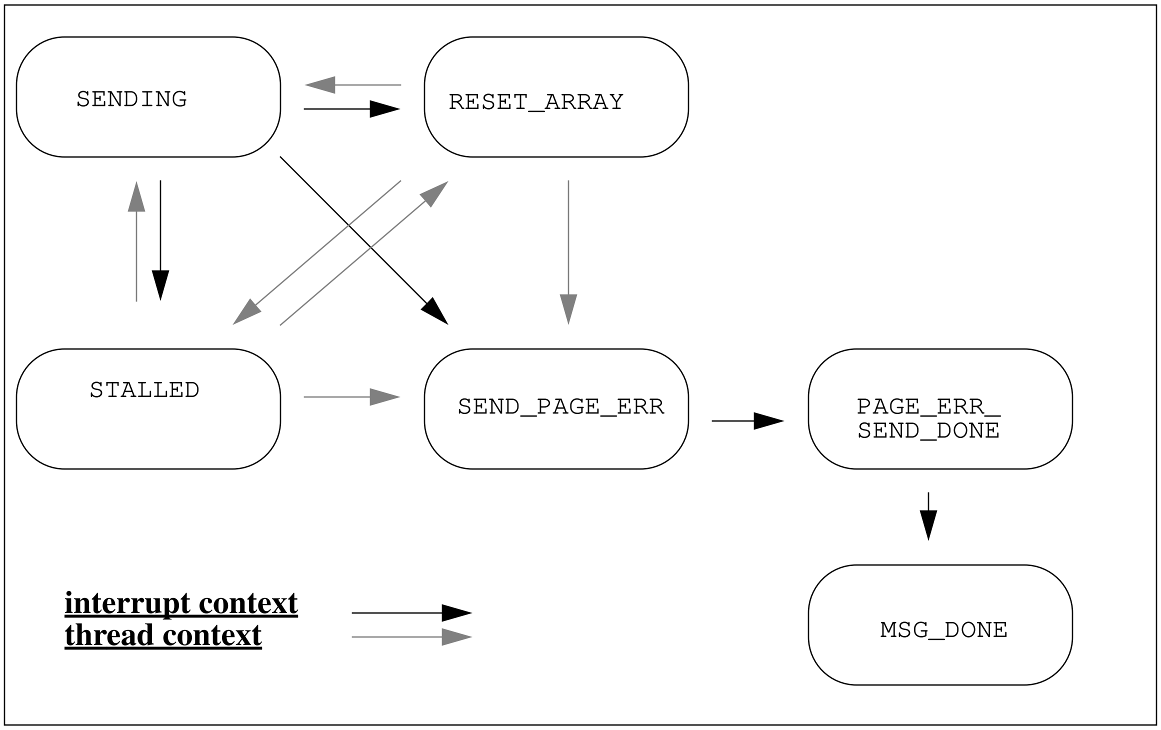

Figure 3: msg_progress State Transitions

5.8.3.1 NORMA OOL Error Propagation

An additional complication to sending out of line memory is that the pagein operation on a page that is not present may fail (e.g., disk parity error on pagein). In the local case, this would cause an exception when the receiver accessed that page because the pagein does not occur until the page fault caused by this access. In the remote case, NORMA must propagate this error to the remote node in order to mimic this behavior (receiver access causes an exception).

In the local Mach IPC case, the receiver of out-of-line memory takes a page fault when accessing a page not present in its address space. The kernel issues a **`memory_object_data_request`** call to the appropriate memory manager to satisfy the page fault. If an error occurs, the manager responds with **`memory_object_data_error`** to indicate the page cannot be retrieved, and that a memory failure exception should be raised in any client threads that access the page. Clients can catch these exceptions and retry their page faults. As a result, this mechanism is used to report transient errors as well as permanent ones.

The internal mechanism used by the kernel to record pager errors is to establish a fictitious page that is marked with an error condition. When any client tries to access the page, it receives the memory failure exception. The distributed implementation should work in the same way.

The sender is aware of a page on which pagein failed when the page is prepared to be sent to the remote node. When encountering such a page, the sender instead transmits a zero filled page. The receiver accepts such a page in the same way it would any other (present) page. By inserting a dummy page, we need not modify critical-path send-side and receive-side logic to recognize this rare error case.

For each erroneous page the sender accumulates the page offset and error information into a vector of `page_error` records. The definition of a `page_error` record is:

```
typedef struct page_error {
        vm_offset_t      region;  /* offset into kmsg */
        vm_offset_t      offset;
        kern_return_t    error;
} page_error;
```

By convention, at the end of sending each kmsg containing OOL data, the sender transmits a status word, `error_count`, summarizing the outcome of the transfer.

The status word is simply a count of `page_error` records representing non-present pages in the OOL region. When `error_count` is zero, all pages were present and no further processing needs to be performed by sender or receiver. Thus, in the typical case of a successful transfer, the only overhead required is a single status word per kmsg. See Figure 4.

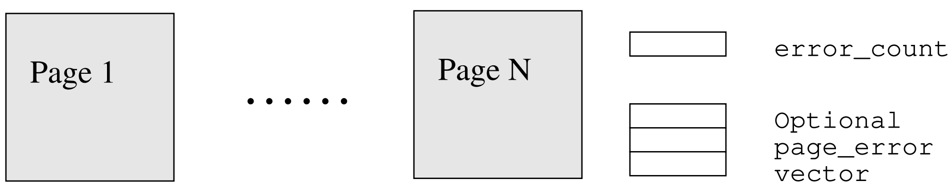

Figure 4: Layout of OOL Transmission with trailing page status

If `error_count` is non-zero, the sender also transmits the accumulated vector of `page_error` records. The receiver allocates a buffer for receiving the error vector. This buffer must be `error_count` * sizeof(`page_error`) bytes long. It is possible that the sender will transmit multiple pages of page_error vectors, which requires the receiver to issue multiple receive requests to the transport. The receiver accepts the vector from the sender, then walks the vector, locating the region and associated `vm_page_t` associated with each page offset specified in the vector. The real, zero-filled page originally transmitted by the sender is thrown away and replaced with a fictitious page. The fictitious page is marked with the error code specified in the vector. If the receiving task attempts to access an error page, a memory failure exception will be raised, identical to the local case.

It is unusual but possible that a dummy, zero-filled page has been evicted from memory and given to a memory manager before the receiver obtains the `page_error` vector. Therefore, the receiver's OOL memory error processing must be able to handle an absent page. However, erroneous pages are themselves quite rare and paging out the placeholder for an erroneous page is rarer still. There is no performance problem here.

[TBD: But there is another problem here. The current code frees absent/error fictitious pages when encountered on the assumption that the error might be transient and the pager will respond with an error again if the problem is persistent. Won't work in the remote case because the object is internal (get zero fill instead, oops). Maybe page needs yet another bit to say "permanent error".]

5.9 Out of Line Port Arrays

Out of line port arrays are sequenced for transmission using the same callback-driven KKT_-

SEND and KKT_RECV operations as out of line memory. Because out of line port arrays are sent from and received into wired kernel memory, neither pinning nor the out of line memory error mechanism is used.

5.10 Aborted Message Send

A **thread_abort()** operation that occurs during a **mach_msg()** raises an interesting case. Data being sent by **mach_msg()** will fall into one of two cases: memory is volatile (RPC, device operations) or memory is non-volatile (asynchronous messages).

The volatile case makes the guarantee that the memory will not be altered during the send operation. This is the most cost effective way to transmit data as there is no VM processing (such as copy-on-write) required.

However, if the thread is aborted via **thread_abort()**, we can no longer make guarantees about the memory that is in transit. Exception processing may occur, the thread may be resumed in a different place, or the task may be terminated. If transmission of the message has begun it will be completed, a guarantee made by Mach IPC. This brings us to the following implementation rules for distributed messages:

- A message that has not established a connection may be completely aborted; this case also includes a sender blocked due to a queue limit.
- A message that has established a connection must have copy-on-write processing initiated on the memory involved before exception processing is invoked.[2]

[2] However, we currently define that all volatile regions remain volatile at abort time. If the region was not originally volatile, it was marked for COW at copyin time, anyway. The only tough case is a true RPC: the region is treated as volatile during transmission. If the RPC is aborted on the send side, the region has not been transmitted and may be regarded as non-volatile by the sending task. If the RPC is aborted on the receive side, the region may have been transmitted, or will be transmitted at some future time. The region remains volatile until the sending task receives the reply to its RPC.

CHAPTER 6 Distributed Port Right Management

The overall goal of this area of the design is to introduce no additional inter-node communication to manage port rights over and above that already required to pass messages. This is accomplished by distributing state in such a way as to convert right management operations that would otherwise require inter-node communication to local operations. We present the design followed by an explanation of its rationale.

6.1 Design Overview

The distributed port right management design consists of six components:

1. Changes to port allocation interface to designate ports that require no more senders notifications.
2. Management of send rights with no more senders notifications.
3. Management of send rights without no more senders notifications.
4. Management of send once rights.
5. Management of dead name notifications.
6. Management of receive right transfer (migration).

In each of the right transfer cases, a right exists in one of three states, namely on the sending node, in transit (network state), and on the receiving node. The bulk of the design consists in describing the transitions between these states. A right starts out in the 'on sending node' state when the message is initially sent. It enters the 'network' state when the `NORMA` code on the sending node converts the message from local format to network format. It exits the 'network' state when `NORMA` code on the receiving node converts the message from network format to local format. Receive rights have an additional intermediate state, 'migrating' that occurs between the 'network' and 'on receive node' states because the conversion to local format is complex and requires additional state to be retrieved from the sending node.

6.2 No More Senders Interface Change

The state maintenance required to correctly deliver no more senders notifications in a distributed system requires additional interactions between nodes when rights are deallocated. Since send right deallocation is a relatively common operation, and the use of no more senders notifications is relatively rare, the Mach kernel's port allocation interface will be changed to explicitly identify ports that require no more sender notifications so that the common case of deallocating a send right for a port that does not require no more senders notifications can be optimized.

Specifically, the following changes will be made to the Mach kernel interface:

- The existing **`mach_port_allocate`** interface allocates a port on which no more senders notifications may not be used.
- A new interface, **`mach_port_allocate_with_nms`**, is introduced to allocate a port on which no more senders notifications may be used.
- Ports created by the kernel (e.g. those representing kernel objects) do not permit the use of no more senders.
- It is an error to request a no more senders notification on a port that does not permit it.
- The mps_srights field in the `mach_port_status_t` structure returned by mach_port_status is changed from a boolean to an integer. The integer value -1 is returned for a port that does not permit no more senders notifications indicating that it is unknown whether send rights exist. The existing boolean true and false values are encoded as the integer values 0 and 1.

These interface changes apply to both NORMA and non-NORMA systems. A reserved value (-1) in the ip_nsrequest field of a port indicates that the port does not permit no senders requests. The only impact on existing code is that the new **`mach_port_allocate_with_nms`** interface must be used to allocate a port on which no more senders notifications will be used. Forbidding the use of no more senders notifications on ports created by the kernel has no impact outside of NORMA as no more senders notification is not used by non-NORMA kernel subsystems.

[TBD: How about also creaming the mps_sorights field (-1 = don't know) in the mach_port_status_t structure for these common ports?. This eliminates the remote send once right count in the common case, and Alan thinks it'll also kill off the srights and sorights counts as well.]

6.3 Send Rights With No More Senders: Transits

A transit is the network representation of a send right, and can be thought of as a distributed reference count. All port structures (both principal and proxy) for a port that utilizes no more senders notifications keep both a count of send rights (on the local) node and a count of transits. On the principal, the transit count is a count of the number of send rights the principal believes exists outside the principal's node. On a proxy, the transit count is the corresponding count of the number of rights outside the principal's node that are represented by the send rights on the proxy. A simple implementation could demand that the transit and send right counts always match on a proxy, hence requiring the principal to be contacted every time a send right is created or destroyed on another node. By allowing these counts to differ, many cases of right creation and destruction can become local operations that do not require contacting the principal. To distinguish the different meanings of the transit count on the principal and proxy, the principal's transit count is always zero or negative (vs. proxy transit counts that are always zero or positive); this convention will be retained. Transit counts are not visible at the Mach kernel interface.

There are three critical invariants that govern the management of send rights with transits:

1. Any send right in the 'network' state must 'carry' at least one transit. Disposition of the transit(s) after the transfer is unspecified.
2. If one or more send rights for a port exist on a node other than the port's

principal node, the proxy on that node must have at least one transit.

3. At all times, the sum of transits held by proxies and by send rights in the 'network' state must equal the count of transits on the principal.

In support of the first invariant, the following conventions are employed:

- If the destination port in the message header is a send right, it carries one transit. This transit is returned to the sending node when the message is enqueued on the receiving node.
- If the reply port in the message header is a send right, it carries one transit. This transit is retained by the receiving node.
- A send right in a message body sent from the principal node carries two transits. These transits are retained by the receiving node.
- A send right in a message body sent from a proxy node carries one transit. This transit is retained by the receiving node.

The last two cases are distinguished by the representation of the send right in the body of the message itself.

[TBD: Make sure message representation design has and describes this bit in the representation.]

There are two inter-node bookkeeping RPC operations involved in transit maintenance, one to request more transits (**`norma_ipc_transit_request`**), and one to return excess transits (**`norma_ipc_yield_transits`**). **`norma_ipc_transit_request`** is used when a proxy has insufficient transits to convert a send right to network format; this RPC is directed at the principal node and returns a fixed number of additional transits (the constant 100 will be used in the initial implementation). This routine may also return an error code indicating that the principal has died (and hence the proxy should be destroyed). **`norma_ipc_yield_transits`** is used only when a proxy with a non-zero transit count is destroyed, and serves to return the proxy's transits to the principal.

Send rights are converted to their network forms as part of converting a local message to network format. If the sending node is the principal node for this port, there are two cases (destination port cannot occur because message would then be local):

1. The reply right: decrement local send right count, add -1 to transit count.
2. Any right in message body: decrement local send right count, add -2 to transit count, set bit in network representation indicating that this right carries two transits.

If the sending node is the proxy node for this port, then in all cases both the local send right count and the local transit count are decremented. If this would result in a situation in which the transit count would become zero without the local send right count becoming zero, the conversion does not take place. Instead, a **`norma_ipc_transit_request`** is sent to the principal to obtain more transits, after which the conversion is retried. If **`norma_ipc_transit_request`** replies that the port is dead (and hence returns no transits), one of the following two things happens:

1. If the right being converted was the destination right, the message is destroyed, and the sending thread receives an invalid destination error.
2. Otherwise the right is represented in the message using the reserved

value to indicate a dead port (NORMA_PORT_DEAD, converts to MACH_PORT_DEAD on reception).

In converting rights from their network representations, the destination right is handled differently from the other rights in a message. The destination right is converted at message enqueue time and the resulting transit is returned to the sending node. The principal's local send right count is incremented, but its transit count is not affected. Instead the proxy's transit count is incremented when the message enqueue operation (**`KKT_SEND_CONNECT`**) returns successfully. The destination right can only be converted from network format on the destination node. For all other rights, it is necessary to distinguish between the principal and proxy cases, and to be aware of how many transits are carried (one if reply, otherwise message representation indicates one or two). On the principal the local send right count is incremented, and the number of transits is added to the principal's transit count (recall that this count is negative, so this addition actually reduces the count). On a proxy, a proxy is created if one does not already exist (right and transit counts are initially zero), its local send right count is incremememented, and the number of transits is added to the proxy's transit count (increasing the count).

One important implementation detail: Local send rights hold references to the port data structure (principal or proxy). Transits do not.

[TBD: Message representation problem -- What does destination right of a queued for transfer message look like? Since it doesn't hold any reference to the local proxy, a pointer to the proxy is clearly wrong (and asking for error case code to dereference something it shouldn't). This occurs in both NMS and non-NMS cases.]

6.4 Send Rights Without No More Senders

No distributed state is maintained for this case. A send right is converted to its network form by decrementing the local send right count and substituting its network representation into the message. A send right is converted from network form by incrementing the local send right count and substituting its local form into the message. Port reference counts adjust in the same way because a local send right holds a port reference.[1]

6.5 Send Once Rights

Send once rights maintain a count of remote send once rights on the principal only. A send once right is converted to its network form by decrementing the local send once right count, and incrementing the remote send once right count if this is the principal node. A send once right is converted from network form by incrementing the local send once right count, and decrementing the remote send once right count if this is the principal. Port reference counts adjust in the same way because a local send once right holds a port reference. This simple technique works because unlike send rights, send once rights can only be created from the receive right, and deallocating a send once right causes a notification message to be sent to the corresponding receive right.

6.6 Dead Name Notifications

We retain the algorithms used in Version One. Dead name notifications for the node on which

[1] If the relative lengths of the above two sections aren't a convincing case for considering deletion of no more senders on complexity grounds, the editors don't know what is.

the principal resides are unchanged. Dead name notifications for other nodes are attached to the proxy port structure on that node. When the first dead name notification is attached to a proxy, that proxy requests a NORMA dead name notification from the principal, using a **norma_ipc_dnrequest** RPC. A NORMA dead name notification is entered into the principal's dead name request structures like any other dead name notification request, except that a reserved value that can't possibly be a port pointer is used (0x22222222 in the current implementation). The port name contains the node number of the proxy that requested the NORMA dead name notification. When the port dies, the code that sends dead name notifications detects this value and calls NORMA code to actually send the notification. NORMA informs the node of the proxy that the proxy (identified by NORMA UID) has died by sending a **norma_ipc_dnnotify** RPC. This causes the NORMA code on that node to destroy the proxy and take the resulting appropriate actions (including delivery of dead name notifications registered with it). This sequence happens for each proxy that has a dead name notification attached to it (and hence has a NORMA dead name notification registered with the principal).

An interface change related to dead name notifications is lazy evaluation of port death. The formal change is that port death may be lazily evaluated with respect to message transfer, i.e., two tasks may successfully exchange a send or send-once rights to a port that is dead (receive right no longer exists) if no dead name notification was requested for the right involved. If a dead name notification was requested, the right becomes a dead name promptly upon destruction of the receive right (this is not guaranteed to be immediate -- tasks different nodes may be able to determine that the receive right was destroyed and the send right was successfully sent in a message afterwards because there is no guarantee of atomicity of the effects of destroying the receive right in the distributed case, nor should there be). In addition, attempting to send a message to the right or calling **mach_port_status** on it will result in it becoming a dead name if the receive right has been destroyed. This lazy evaluation is necessary to avoid having to contact the principal node when two proxy nodes transfer a send or send-once right between themselves.

6.7 Receive Right Migration

The receive right migration (transfer) design addresses the following areas:

1. Actual transfer of receive right state (including queued messages or meta-kmsgs) between nodes.
2. Effect on operations that occur during the migration.
3. Effect on operations occurring after the migration (including how the new location is discovered).
4. Cleanup of residual state when a migrated port is destroyed.

6.7.1 State Transfer

The state components to be transferred consist of:

- Queued messages
- Dead name notifications
- Rights and transit counts
- Other state associated with port (sequence number, make send count, etc. See existing NORMA IPC Version One implementation of

norma_ipc_pull_receive for details)

The transfer requires the use of an internal thread on the sender; this is used to handle a callback RPC to extract the state components from the sender and forward the queued messages. Operations on this port structure and the port structures the sender are blocked for the duration of the migration (e.g. no messages can be queued while this is in progress).

The queued messages are sent from sender to receiver using the existing message transfer mechanisms (these work for both messages and meta-kmsgs). This requires only that the receiving node understand that the message is being forwarded, and may have originated on a node other than the one forwarding it. It is not necessary to transfer blocked sender records from the sender to the receiver (although it may be an optimization to do so). Instead, it is sufficient to awaken all blocked senders with an indication that the port has moved when the migration is complete and have them attempt to send to the new node (and reblock if needed). This separates port migration from the effect of a blocked senders record shortage on the target node (at the potential cost of many inter-node messages if there are many blocked senders -- we assume that this is a rare case for now).

Dead name notifications are not transferred between nodes. Instead, the destination node of the migration is informed that the source requires a node to node dead name notification when the receive right is destroyed. This will invoke code on the source that causes any needed notifications to be delivered locally. The design and much of the implementation (e.g., how these node-to-node dead name notifications coexist with ordinary IPC dead name notifications) of this functionality can be transferred from the existing NORMA system. (See section 6.6, “Dead Name Notifications,” on page 47.)

Send and send once rights are not transferred. These counts represent rights that exist on the source node that are not accompanying the receive right being transferred. If transits are in use, one transit must be left behind if any send rights remain on the source (with the resulting transit count at the destination adjusted accordingly). If transits exist at the destination, they are cancelled against the receive right’s count of outstanding transits. The existing NORMA logic of preloading an arbitrary value into the transit (and sotransit) variables is no longer needed because operations cannot occur on these data structures during the migration. Other state associated with the port is moved directly as part of this (refer to NORMA IPC Version One’s norma_ipc_pull_receive implementation, which identifies this state).

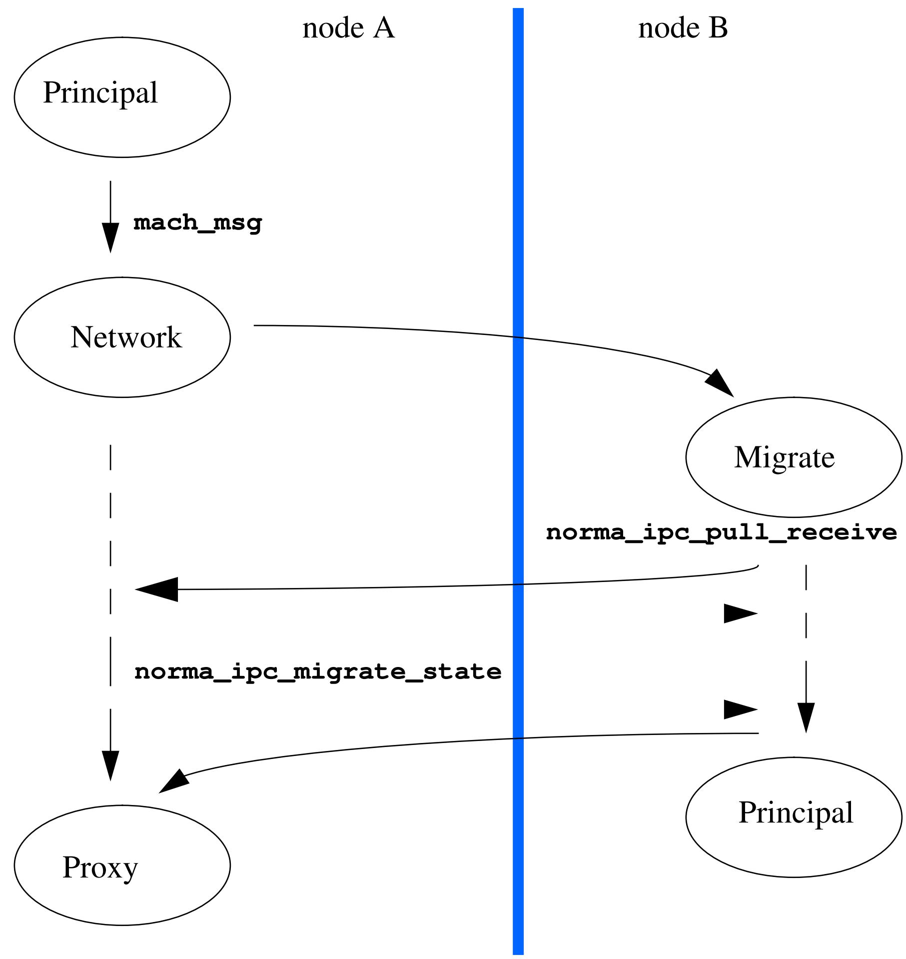

Figure 5: Migrating port state transition diagram

6.7.2 Detailed State Transfer Sequence:

(1) On sending node at message send time, the IPC copyin of the message containing the receive right puts the port into network state. Messages can be sent to such a port, but because no task holds the receive right, no messages can be received. The receive right's local message queue is scanned, and any messages not already in NORMA transport format are converted at this time. This conversion may allocate memory for data structures to describe out of line memory and buffers to hold UIDs for ports sent out of line. The IPC copyin logic treats a port in network state as remote, so all subsequent messages enqueued on the port will have been converted to NORMA transport format by the IPC copyin code.

(2) The IPC message containing the receiver right is transferred to the receiving node. It contains only the name of the receive right, and no other associated state.

(3) When the message is transferred to the receiver, the receiver creates a port data structure (or converts an existing proxy) to hold the receive right. This new structure is marked as migrating to block further enqueue operations.

(4) The receiver performs a **`norma_ipc_pull_receive`** RPC to the sender to cause queued messages to be transferred. The sender's initial receipt of this RPC puts the port into the migrating state in which it behaves as if its queue were full (even if it is not); ordinary senders block, but messages that ignore queue limits continue to be queued. The sender replies to the RPC at this point.

(5) The transfer of queued messages occurs via the same mechanisms used to transfer other Mach messages (e.g., only the meta-kmsgs need be transferred in most cases). Message ordering is preserved by transferring only one message at a time. Only a queue limit's worth of messages need to be transferred; messages beyond the queue limit must be MACH_SEND_ALWAYS messages from the sender's node that can remain queued on the proxy. If the port's queue limit was adjusted downward while the queue was full, stopping at the new (lower) queue limit could cause problems. The most straightforward solution to this is to forbid queue limit reductions when there are queued messages. This should not cause any problems (queue limits are usually set immediately after port allocation). The actual transferred message must be marked as migrating so that they will be queued on this port that is not accepting other messages, but the port rights within these messages do not require special treatment.

(6) Finally, port state other than queued messages (transits, remote send once right count) is transferred by the sender at this time, as a separate **`norma_ipc_migrate_state`** RPC to the receiver. This also indicates whether any MACH_SEND_ALWAYS messages are queued and whether there are any local blocked senders (if either is true, a blocked sender record naming the sending node must be created by the receiver).

(7) The receiver inserts the port state into the port data structure, marks it as a principal, changes its state to normal from migrating, replies to the RPC, and creates a NORMA dead name notification for the sending node. The sender then converts its principal to a proxy and removes this port from the migrating state, and restarts all blocked senders. It is important for the receiver to exit the migrating state first to avoid having the sender restart a blocked sender which immediately blocks again because the receiver is not ready.

Messages sent to a port while it is being migrated are treated as if they exceeded the queue limit; ordinary senders block, and MACH_SEND_ALWAYS messages are queued on the proxy queue; this is why the sender must indicate whether it has any such messages in step (6). Clearing the migrate bit in step (5) will unblock all such blocked senders with an indication that the port has moved, including the node number of the port's new location; this is identical to the behavior senders will experience if they send to the old location after migration.

6.7.3 Effects on other operations

For simplicity, other operations on the migrated port will be disabled during the migration operation (e.g., attempts to enqueue a message will use the existing logic to block the sender, possibly including declaring a resource shortage). Operations that only affect the destination node proxy may be allowed to proceed while the queued messages are being transferred; whatever is the easiest to code will be done. Operations need only be disabled during the actual transfer of messages and state between nodes; there is no problem, for example, with queueing additional messages before this transfer is commenced.

After the migration, the original principal remains on the source node to inform other operations that the port has moved. An attempt to enqueue a message results in returning new information about where the port can be found; the sender is expected to retain this information in its proxy to optimize future operations on this port. A level of indirection is needed on sending nodes to avoid having the port known by two names (based on old and new node). The port must have a single NORMA UID that does not change, but also has additional location information (a node identifier) stored in the proxy. This is equivalent to the use of norma_uid and destination node in the current NORMA code.

[TBD: Fix the 'Dejan' misfeature (named for discoverer, not guilty party). Migrate a receive right, then NORMA thinks the right is still on its original node when a send or send-once right is handed to a node that never had one before. This is horrible when migrating a task because the send once rights for RPC replies appear to be on the wrong node. The particular case of send once rights for RPC replies is probably fairly easy to fix, but the general case for rights in message bodies is horrible]

6.7.4 State Cleanup

Implicit in the description of how the port is located after a migration is that state which allows the port to be located remains on every node that has ever had the receive right until the port is destroyed. Such "forwarding proxies" must be explicitly destroyed when the receive right is destroyed in order to release their memory. The NORMA dead name notification mechanism is sufficient to implement this.

6.8 Port Data Structure Impact and Interface Changes

[TBD: Is everything here? Perhaps this belongs in another section?]

The following fields are present in the `norma_ipc_port` data structure to implement the functionality described in this chapter:

- A transit count for the transit logic. (See section 6.3, "Send Rights With No More Senders: Transits," on page 45.)
- A remote send once count for tracking how many remote send once rights exist. (See section 6.5, "Send Once Rights," on page 47.)
- A state field to indicate network and migrating states for receive right transfer (See section 6.7, "Receive Right Migration," on page 48.)

An attribute of this new design is that a proxy data structure can never hold a reference that causes a principal data structure to continue to exist even when no port rights exist on the principal node. The NORMA UID table provides the needed level of indirection to implement this; if a principal has been destroyed, the lookup of the destination port in an inbound message will fail.

In addition to the interface change to make no more senders support explicit in the port allocation interface (See section 6.2, "No More Senders Interface Change," on page 44.), there are two other changes to the Mach interface required by this design:

1. Reducing a port's queue limit will fail when there are queued messages. A port's queue limit may only be reduced when no messages are queued. This avoids a nasty problem in receive right migration (See section 6.7.1, "State Transfer," on page 48.)
2. The system is permitted to lazily evaluate port death as it affects send

and send once rights when a dead name notification has not been requested. (See section 6.6, “Dead Name Notifications,” on page 47.)

6.9 Distributed Port Management Background and Goals

Consider a port that represents an object (e.g., an open file) for which the implementor of the object (server) and user of the object (client) are on different nodes. The life of the port consists of the following mandatory and optional stages:

1. Creation. The port is created by the server on what we will call its origin node. The node where a port currently resides is called its residence node, or residence. For simplicity, assume that the origin and residence nodes are identical and the residence node has never changed.

2. Transfer to client. A send right is created and transferred to the client on a different node.

2a. (Optional) A dead name notification is requested on the send right. The first dead name notification request generated by a node for a particular send right causes a message to the residence node. Subsequent dead name requests from the same node are handled locally.

3. Use of the object. The client sends messages to the server using the object as its destination. This results in the client creating send rights that are absorbed by the server.

4. (Optional) Transfer to other clients. The client may transfer the send right to other clients, creating send rights that are not sent to the server.

4a. (Optional) Server deallocates receive right, generating dead name notifications. No dead name notifications are generated if no requests for dead name detection have been registered for the receive right.

5. Client deallocates the send right. An intelligent client may dispose of the send right by transferring it in its last message to the server, but often the right is deallocated via a kernel operation. We consider this stage to include any distributed state manipulation that is needed to cause the next stage to happen (if needed).

6. (Optional) No more senders (NMS) notification. When the last send right is gone, the server receives a no more senders notification if it had requested one. Use of NMS is believed NOT to be the common case.

Stages (1) and (6) are purely local operations involving the principal. Of the remaining four stages involving two nodes, stage (3) is the most important to optimize (as there may be many operations using the port), followed by a restricted form of stage (4) in which the first client and server are located on the same node. For instance, the OSF/1 AD filesystem does this for the credentials port. Stages (2) and (5) also merit attention, as they occur once for each port that does not exist for the life of the system, which is by far the common case. The primary goal of optimizing these stages is to avoid inter-node bookkeeping interactions and their resulting latency, even on a high speed interconnect. A secondary benefit is reduced interconnect utilization.

In descending order of priority, the optimization goals are:

[A] Stage (3): Creating a send right from a send right should not require contacting the holder of the receive right. An alternative is to combine this interaction with queueing of the message.

[B] Stage (4): In addition to [A], depositing a send right onto a node from a message should not require contacting the receiver, even if the task into which the send right is deposited already holds a send right to the same port.

[C] Stage (5): If the send right was consumed by the last message and an NMS notification is required, then an additional inter-node interaction is unavoidable. The optimization goal is to avoid the interaction in all other cases.

[D] Stages (2a) and (4a): Enabling dead name notification is already optional and not mandatory. When requested, extra messages are required to implement this functionality; these extra messages are unavoidable. The design and implementation should avoid paying any additional cost above that present in the existing implementation.

6.9.1 Transits -- Rationale and Justification

To motivate the design of the transit mechanism, consider the following situation:

•Node B holds a send right to a port whose receive right is on node A.

•Node B creates and then destroys a second send right to this port.

In principle, there is no need for node B to contact node A during this pair of operations, as they do not have any user visible effects on node A. After all, the Mach interface only exports whether send rights exist to the holder of the receive right, not how many there are. In practice, if node A has a count of rights that includes the number of rights on node B, then node B must contact node A to manipulate this count.

The basic idea behind the transit mechanism is that node B should keep two counts of the number of send rights, the number of rights node A thinks node B has, and the number of send rights that node B actually has; the former count is called a count of 'transits.' As long as node B has at least one transit, the actual number of send rights it has does not matter; hence node B can now create and destroy its second and subsequent send rights without contacting node A. In the absence of fault isolation concerns, there is no need for node A to keep a transit count -- it is a useful debugging firewall in that it prevents remote nodes from consuming send rights that they do not possess. Send once rights do not benefit from (and do not need to maintain) transit logic because they cannot be created on nodes that do not have the receive right.

The transit mechanism (described in Section 6.3) implements optimization [A] because when the right is transferred to a remote node, it carries two transits, one of which can be loaned to cover message sends, the other of which ensures that the node always has at least one transit for its local rights. Return of the loaned transit after message send avoids inter-node bookkeeping to acquire more transits for future messages. Optimization [B] is implemented by separating the right count from the transit count; if the right count decreases because two rights merge (e.g., two send rights merge resulting in one send right with two user references), nothing need be done with the transit counts. Optimization [C] is implemented by avoiding no more senders overhead in the common case. (See section 6.2, "No More Senders Interface Change," on page 44.)

CHAPTER 7 Fault Tolerance

∇ 7.1 Fault Isolation / Management

7.2 Fault Isolation Goals for V2

7.3 Behavior of Ordinary Messages

All parts not guaranteed to leave sending node when sender unblocks

7.4 Message Flush Bit

Mach semantic change

Implications for operation of message flush bit: message flush does not imply flushing of any previous or subsequent messages. For example, a message sent with the flush bit enabled has no way to guarantee that a previous message with an obnoxiously large OOL region has been moved to the receiver in its entirety.

7.4.1 NORMA behavior for flush

7.4.2 Transport behavior for flush

Shouldn't be any different, but we should say this explicitly.

CHAPTER 8 Membership Services

CHAPTER 9 Memory Pinning

Mach contains a facility to force virtual pages to remain resident, preventing the pageout daemon from paging them out. This facility, **vm_wire**, operates on a range of pages, faulting in any missing pages, removing them from the page queues used by the pageout daemon and setting a wire bit in each page's `vm_page_t` to indicate this state.

This mechanism is somewhat heavyweight in three ways. In the first place, the cost of wiring itself could be made somewhat cheaper. In the second place, it is impractical to wire large regions of a task's address space when sending out-of-line memory because there may be an arbitrary delay until the message is delivered. Third, Mach's **vm_wire** operation may only be invoked from within thread context. However, it would be very useful to be able to invoke an interrupt-level operation that, in effect, wired a page because we expect NORMA's transports to execute primarily at interrupt level.

We introduce a page *pinning* facility with the following desirable properties.

- Page pinning and unpinning can be done from interrupt as well as thread context.
- Page pinning is a generic, machine-independent, portable VM-based facility.
- Although available from interrupt context, it is not necessary to burden existing VM operations on ordinary, non-pinned pages with any spl/splx processing, which is quite expensive.
- Preparing pages for pinning has a very low cost.
- Prepared pages remain pageable until actually pinned.
- When pages are present in memory, both preparing and pinning them have very low costs.
- Page pinning may be used in a general way by any interrupt-level activities, such as a NORMA IPC transport or a device driver

In general, a page must be prepared (or *prepped*) for pinning; then pinned; and eventually, unpinned. The preparation phase is necessary to indicate to the VM system that the page is subject to manipulation (i.e., pinning) from interrupt level, and hence additional synchronization (e.g., blocking interrupts) is necessary when manipulating the corresponding `vm_page_t`. Note that a prepped paged remains pageable until it is pinned. Because a prepped page may be paged out before it is used, it may be necessary to fault in a missing page and repeat this cycle. Once a page has been pinned, it will be ignored by the pageout daemon.

9.1 Data Structures

The `vm_page` data structure is modified by adding the following highlighted fields:

```
struct vm_page {
        queue_chain_t     pageq;   /* queue info for FIFO
                                    * queue or free list (P) */
        [...]
        unsigned int      wire_count:16,    /* how many wired down... */
        /* boolean_t */   inactive:1,       /* page is in inactive... */
                          active:1,         /* page is in active... */
                          laundry:1,        /* page is being cleaned... */
                          free:1,           /* page is on free list (P) */
                          reference:1,      /* page has been used (P) */
                          gobbled:1,        /* page used internally (P) */
                          limbo:1,          /* page prepped then stolen (P) */
                          :0;               /* (force to 'long'... */
        [...]
        vm_offset_t       phys_addr;        /* Physical address... */
        vm_prot_t         page_lock;        /* Uses prohibited... */
        vm_prot_t         unlock_request;   /* Outstanding unlock... */
        unsigned int      prep_count:16,    /* Page prep count */
                          pin_count:16;     /* Page pin count */
};
```

The prep and pin fields need to be counts to deal with the same page occurring in concurrent messages or multiple times in a single message. To deal with the potential of overflowing a 16 bit prep count (difficult, but possible), it's fairly easy to delay the prep operations that would cause the count to be exceeded; this situation is highly unlikely in practice, and hence not worthy of further design effort.

The current size of a `vm_page_t` is 48 bytes. This can be shrunk to 44 (has been in the 1.3 development branch). The prep and pin counts restore it to 48. The limbo bit is used for the additional synchronization required when a prepped page is paged out (interrupt routine must acknowledge that it has lost interest in the page).

9.2 Locking

The prep and pin counts are locked by a new global lock, the `vm_page_pin_lock`, that can be taken by interrupt handlers (and hence non-interrupt code must block interrupts to acquire and while holding it. This lock parallels the `vm_page_queues_lock`, except that the `vm_page_pin_lock` need not be taken if the prep and pin counts are both zero. Interrupt handlers may access the prep and pin counts when either is non zero, and may read the limbo bit; these interrupt handlers must take the `vm_page_pin_lock` (but no other locks) when doing so. Hence the `vm_page_pin_lock` locks against access by interrupt handlers, but when the prep and pin counts are zero, the page is in a state where it cannot be accessed by interrupt handlers. The `vm_page_pin_lock` must also be taken by any code that puts a page into limbo state. This is not an additional requirement because a page can only go into limbo if its prep and/or pin counts are non zero, hence the code already had to hold the lock to verify that the counts were non zero.

A key feature of this prep/pin design is that it can be verified that a page is not involved in prepping or pinning without blocking interrupts (a potentially expensive operation). At thread level, a thread holding the `vm_page_queues_lock` can read the prep and pin counts. (A union will be used to do the obvious overlay of a single unsigned int on the pair, but this is not shown above for clarity). If both are zero, then the page is not involved in prepping or pinning, and a bit can be set (e.g., busy) that will prevent the prep transition (which must hold the `vm_page_queues_lock`).

9.3 Algorithms

The prep count is the number of times this page has been prepared for use by an interrupt handler, and the pin count is the number of times a page is in use by an interrupt handler (and hence is not pageable). For a single message, pages are prepped in advance in thread context, pinned in interrupt context (by/for the transport in the case of NORMA), and both unpinned and unprepped in interrupt context when the transport is done. A page that is prepped but not pinned is still pageable, but the pageout demon must do more work to set the page up for pageout. This prep/pin sequence applies to both send and receive sides (except that the receive side will usually have to prefault as well as prep the pages involved). The limbo bit is used to indicate that the page cannot be pinned (e.g., because the page has been paged out). The limbo bit is also used for a page that had to be removed from its object while pinned (e.g., object termination). When the limbo bit is set, the page is removed from whatever page queue it was on (active/inactive) and placed on a special limbo queue. This queue is scanned by both **`vm_page_more_fictitious`** (called when more fictitious pages are needed) and the pageout demon (invoked when more physical pages are needed) and any pages with prep and pin counts of zero are removed from limbo state and freed. A global count will be added to count the number of real pages in limbo (and the pageout daemon consults this to determine whether to scan the limbo queue).

Pages that are prepped but not pinned are still pageable, but the setup for paging is more complex. It is necessary to allocate a new fictitious page and swap both page identities (fictitious page becomes physical page that was prepped, and prepped page becomes fictitious) and page to object relationships (prepped page is removed from object and enters limbo state, new page inserted) before paging out the new page (new `vm_page_t`, same physical memory). A small routine in `vm_resident.c` is required. The pageout daemon can check that the prep and pin counts are both zero without acquiring the `vm_page_pin_lock` (it must hold the `vm_page_queues_lock`), so in the common case (page not involved with an interrupt handler), blocking interrupts is not necessary. The pageout daemon sets `m->busy` (and obtains a paging reference) on a non-prepped page in order to prevent `it` from being prepped while it's being paged. The pageout daemon skips pinned pages (should reactivate them if found on inactive list because they are clearly being referenced).

All other modifications to the VM system are confined to **`vm_page_free`**; this routine causes a page to be freed to enter the limbo state if it is either prepped or pinned. If the page is only prepped, it becomes fictitious via an identity swap with a fictitious page (similar to pageout case). The most obvious case of this is object termination.

Rough pseudo code for the prep, pin, and unpin transitions follows:

prep:

```
Must hold vm_page_queues lock
also need vm_page_pin_lock if prep/pin counts != 0
lookup the page
if it's ok (!busy, !error, !fictitious, etc.)
        record vm_page_t
```

```
                  increment prep count
pin:
          Must hold vm_page_pin_lock only
          In interrupt context:
                  if page is not in limbo
                          increment pin count
                  else
                          decrement prep count
                          /* can't send this page */
                          Queue request to kernel internal
                          thread to fault in next page(s)
                          and prep as many more as can be prepped
unpin:
          Must hold vm_page_pin_lock only
          decrement prep and pin counts
```

A page that the VM system has manipulated out from under an interrupt handler always enters the limbo state. This requires active acknowledgment from the interrupt handler that it is finished with the page (pin and prep counts must go to zero before page can be freed). This eliminates any potential problems with an interrupt handler having a pointer to something it shouldn't. The limbo interaction is expensive -- it should be rare (if it's not, then things like how far the prep work runs in advance of use by the interrupt handler need to be tuned).

9.4 Comparison with Wiring

We considered whether the pin and prep mechanism could be combined with the existing page wiring mechanism, and decided that it could not be as the design stands for two reasons:

1. Wired pages are removed from the page queues. Prepped and pinned pages are not. The page still needs a wired count to count number of times it has been so removed.
2. Merging with the wired mechanism probably requires the `vm_page_queues_lock` to be taken at high spl. This is highly undesirable as the `vm_page_queues_lock` is extensively used.

9.5 Use of Memory Pinning by NORMA

For NORMA, the send side sequence of operations is:

(1) IPC copyin produces an entry list flavor `vm_map_copy_t`

(2) In sender's thread context, NORMA runs over entire out of line region (the entry list) trying to prep every page and making an array of pointers to the `vm_page_t`'s so prepped. The first few pages in the out of line region will be faulted in (this number is a settable/tunable constant). The first non-resident page encountered beyond the faulted in ones terminates this process (and a max size parameter should be provided to terminate it early in case over eager preparation sends too many pages into limbo). NORMA sends the kmsg, and may pin and send the first two pages (decision is dependent on state of memory resources; if memory is tight, wait for send completions before wiring this memory).

(3) At interrupt level, pin ahead of transport by using completion handlers. Basic idea is to stay a small number of pages ahead -- e.g., if transport has pages 4 and 5 queued, the completion

event from page 3 causes attempts to pin and queue 6 and 7 (in addition to unpinning 2 and 3). First pin failure (includes that caused by prep failure) causes end of this interrupt handling, and an internal NORMA thread is invoked to fault in the page on which the pin failed, prep it and whatever further pages can be prepped (may also fault ahead if desired). The send process then picks up where it left off.

For the initial design, it's ok for these internal NORMA threads to block -- pageout can't stumble in this way (probably need to mark vm_map_copy_t generated by pageout to indicate that pages are already wired, hence pin can't fail[1]). Allowing these threads not to block is required for correct support of unprivileged external memory managers (suppose all of these threads block on a manager that will respond on the first Tuesday of May, 2004), but support for these is not an initial design requirement.

The corresponding sequence of operations on the receive side involves three cases:

1. Overwrite, region has page aligned start and end.
2. Allocate new memory.
3. Overwrite unaligned.

Either source or destination has start or end that are not page aligned. The matching unaligned case (e.g., both off by 0x100) is included here. The first two cases are similar. A major difference from the send side is that the prep pass wants to fault in data (in case 2, this is data that NORMA will overwrite completely, and hence zero fill should be avoided). Need to prep in chunks here (fault in but not wire pages), and then have transport do the same pin and unpin logic as on the send side. Receiving thread is responsible for fault in and prep logic. Since both of these cases put the data directly in the desired destination memory, no ipc copyout processing is necessary (and a bit would be set in the kmsg descriptor to so indicate. In case (3), it's necessary to allocate memory in the kernel, put the data in there, and then use the existing **`copyout_overwrite`** logic to get it to show up correctly in the receiver's address space. This logic avoids excessive copying in the matching unaligned case.

[1] If the pin can't fail, why bother marking the vm_map_copy_t? Alternatively, if it is marked, why bother to pin?

CHAPTER 10 Interactions With Other Mach Subsystems

10.1 External Memory Management Interface

Pagers are no longer required to respond to all pageouts with memory_object_data_write_-completed. However, the first revision of NORMA IPC still requires that all pagers be trusted. Refer to “Paging Flow Control” on page 64.

∇ 10.2 Device Operations

A complete implementation of flow control requires modifications to the kernel’s device subsystem to eliminate two problems. The first problem is caused by the fact that each device includes a potentially unlimited queue of pending operations. If operations are queued faster than the device can execute them, then the queued operations consume wired memory for the data involved. Left unchecked, this device queueing problem can consume all available memory on a node. The problem is worst in situations where a device is being inundated by operations from multiple nodes. The second problem is caused by the fact that reply messages from device operations are sent by the kernel and hence ignore queue limits. In the presence of an eager transfer transport implementation and a fast device, the resulting memory can overload the receiving node’s ability to process it and cause undesirable paging (and resulting performance degradation). This overload problem has been observed with the existing `NORMA IPC Version One` system and high speed devices. A related desirable characteristic is that if the overload is caused by a network device with a protocol that allows packet drops, then it should be possible to recover by dropping packets instead of enforcing reliable delivery of all the packets involved in the overload.

10.2.1 Device Queueing

The device queueing problem is caused by generation of device operation requests faster than a device can satisfy them. If the source(s) of some or all of these requests are nodes other than the one to which the device is attached, then the requests (and their associated memory) pile up in the device node’s memory forcing it to page. Instead, the device node must be able to apply backpressure to the generating nodes to avoid having its memory overcommitted in this fashion. This backpressure must also be applied to local device operations because in the paging case, the overload can result from the local vnode pager queueing operations (pageouts originating from other nodes) to the device faster than the device can handle them.

The problem is addressed by a limit on the total number of outstanding device operations per device. In the current code base, this limit and the count of outstanding device operations is contained in the device structure (one exists per open device). If a future code base has a device structure per client that has opened the device (and hence an open device may have multiple such structures), it may be necessary to introduce a shared operation limit structure; this issue should be revisited if it proves to be a problem in practice on such code bases.

The device code is responsible for enforcing this limit by blocking threads that attempt to initiate operations that exceed it, and waking them up when it is ok to continue. This may result in blocking kserver threads executing operations on behalf of NORMA, which is satisfactory for devices which guarantee to complete pending requests. For example, a disk device provides this guarantee. For devices that implement blocking reads (e.g., serial line devices such as ttys), blocking a kserver thread is not reasonable. Instead, the best solution is not to limit operations for such devices. Network reads (via filter mechanisms) must also bypass this limiting mechanism entirely.

The value of the limit is one of the parameters of the device. It could be set either by a flavor of **device_set_status and device_get_status** (have to pick the flavor value carefully so as not to cause future problems with the ioctl-derived encoding of flavors), or by a new **device_set_operation_limit** call (and corresponding get call). The default value should be part of the device configuration -- either in the `dev_ops` vector, or a mandatory flavor of **dev_info** in the driver.

10.2.2 Input Overload

The second problem is in input overload. Some devices and collections of devices are capable of generating input faster than the application or system component involved can absorb it. The memory involved can pile up on the node containing the application or system component bottleneck causing memory overcommitment on that node. High bandwidth networking and parallel filesystems are both capable of causing this.

Most of the input overload problem is solved by the use of receiver pull message transfer in the new NORMA system. This moves the potential overload problem to the device node; if an application node gets overloaded with input data, it's because the application asked for all of it, and it is not the system's responsibility to protect misbehaving applications from themselves.

Network devices allow one further response to this situation. If the network device is one that can drop data with impunity, then the return of QUEUE_FULL or RESOURCE_SHORTAGE from the destination node of the NORMA message should cause the input message to be discarded (and its memory freed) rather than queued. Having NORMA expose these error codes to the code that sends messages generated by network filters is not feasible, hence another option to **mach_msg_send_from_kernel** is required to cause the message to be discarded in these cases.

[TBD: Fix return codes to match rest of document]

10.3 XMM

10.3.1 Pageable XMM Pages

The Mach VM system protects itself against unresponsive external pagers by allowing a page that is waiting to be sent to this pager to be *double paged* to the default pager. Once the data reaches the default pager, the page can be freed. In effect, this means that out-of-line pages contained in messages to a pager need to be pageable. When memory_object_data_return() is called to send a page to the pager, the page is contained in an object-flavor copy object, and it is pageable.

Within XMM, there are two paths the page can take. If the XMM stack is entirely local, with the SVM object on the same node as the page, the page is passed along still in the object-flavor copy object until it is sent in a message to the pager; it can be double paged the whole time, even while waiting in the pager port's message queue. The other case is that the SVM object resides on another node. In this case, the page remains pageable on the original node up until the time it is placed in a message to go to XMM on the SVM object's node. Since the kernel is the receiver of the message, it can be assumed that the page will be sent off of the original node within a reasonable time.

The node with the SVM object that receives the pageout message can do one of three things with the page:

1. drop it, if another node has a copy;
2. supply it to another node that has requested access; or,
3. send it to its pager.

The first two cases present no problem, since the page is either freed or immediately sent to another node that is waiting for it. The third situation is vulnerable to unresponsive pagers in the same way as the pageout daemon. In the current system, XMM expects the page in page list format; the page itself is either loose or busy and therefore cannot be double paged. This problem can be addressed by having XMM convert the copy object to object or entry list format when it determines that the page is destined for a pager.[1]

XMM has no inherent dependency on the particular format of out-of-line pages. It currently handles either object or page list format but could easily accommodate an entry list with some minor modification. There is also one instance where XMM makes a copy of a page and then immediately sends the copy to the pager (**svm_copy_page**). This new page is currently placed in a page list copy object, and it is not pageable. To allow double paging, **svm_copy_page** needs to be modified to create an object-flavor copy object instead.

10.3.2 Paging Flow Control

The use of XMM decouples pagers from normal IPC flow control mechanisms, as follows. A kernel or user task sending messages directly to a remote user-level entity, e.g., a pager, will be flow controlled by Mach IPC. The queue limit mechanism prevents the accumulation of messages on any single port. The meta-kmsg enqueueing mechanism guarantees that memory will not accumulate across ports because the active participation of a receiving thread is required to bring across the resources associated with a message. However, although these rules apply to inter-kernel traffic, as well, unmodified XMM nevertheless circumvents the intended flow control.

Messages sent from one kernel to another are received on the in-kernel, NORMA pseudo-port. This port is subject to the same queue limit mechanism as any other port, but this port is also backed up by a relatively large number of threads ready to receive incoming messages. In the case of XMM, the kserver threads stand ready to pick up incoming pageouts and convey them into XMM. If XMM absorbs the messages, no further action is required. However, if XMM forwards the pageout requests to a user-level pager located on the same node, messages can accumulate on

[1] The problem may also disappear entirely if a page received by NORMA is handed to XMM in a pageable format (object or entry list). However, we expect that kernel-targeted messages will be received in non-pageable memory for efficiency.

the pager's ports without bound. This accumulation results because messages originating from the kernel are sent with MACH_SEND_ALWAYS enabled, and when the user-level target is on the local node, queue limits are ignored. Thus, when many nodes are paging out, it is possible for message generation to outrun the ability of a pager to keep up.[2]

The good news is that this problem is not fatal. If the node accumulates pages, the kserver threads will block trying to allocate new memory to receive incoming messages. However, this blocking behavior happens late in the game, after the node has become swamped. It is important to apply back pressure as soon as possible.

There are several possible solutions to this problem. These solutions rest on a simple observation: it is necessary to directly couple kernel outputs to pager inputs so that Mach IPC can perform flow control.

1. When sending pageout messages from XMM to a co-located pager, in the context of a kserver thread, disable MACH_SEND_ALWAYS. Kserver threads are throttled back when a pager is slow. Pages will not arrive from off-node faster than they can be discarded. However, two unpleasant implications derive from this solution. First, pagers must be trusted to dispose of their pageouts. Second, incoming kernel-bound messages may be delayed when kserver threads are consumed by a slow pager.
2. Provide a new way for XMM to receive its messages. For instance, create a pool of dedicated XMM threads that have a special message reception attribute. The message reception code recognizes this attribute as follows: inline kmsg processing remains unmodified, so that a thread continues to allocate memory for an inline kmsg (blocking, if necessary) and then brings over the message from the sending node. However, all out-of-line memory reception is bypassed and must be done explicitly by the receiving thread. With the inline kmsg in hand, XMM has sufficient information to decide how to process the pageout. If the pageout is forwarded to a user-level pager, the message can be enqueued as an inline kmsg with an OOL portion on another node.
3. Modify EMM/XMM interfaces to "steal" the pager's threads. Instead of using kserver threads to process incoming messages, if XMM can somehow use the pager's threads to process messages related to its objects then the pager's threads will be directly coupled to the kernel-generated pageouts. Mach IPC queue limits will apply directly.

Solution #1 is the simplest, although we continue to investigate solution #2.

∇ A very real concern remains: even with the application of Mach IPC flow control mechanisms, will paging be as smooth under load as with NORMA IPC Version One's explicit paging flow control implementation?

[2] Converting XMM to use direct transport RPCs rather than Mach messages does not eliminate this paging flow control problem. XMM can still enqueue messages to a pager faster than a pager can process them.

10.4 Asynchronous Virtual Memory Faults

Wiring memory in the context of a NORMA transport thread rather than in the context of the original, sending thread introduces the twin possibilities of deadlock and thread exhaustion. Deadlock may arise when a page to be faulted in resides on the node that is the target of the send operation. This deadlock might be eliminated by introducing separate classes of transport threads. However, there is still the possibility that all threads in the transport class for outgoing operations could be consumed by pending faults.

To guarantee correct operation, we introduce asynchronous virtual memory fault handling.[3] Today, fault handling consumes the thread taking the fault. The asynchronous fault facility allows a page fault to be initiated by a calling thread without synchronously awaiting fault resolution.

Another benefit of asynchronous faults is the correct operation of the system in the face of malicious or slow user-level pagers. Requests for pages managed by such a pager will consume one fault context structure per OOL region dropped into the transport, but the transport threads will remain available for other work.

∇ Last, asynchronous VM fault handling will fix a current bug in uniprocessor Mach.

10.5 Server Dependencies

A requirement for external, default pagers is that they do not create an unlimited number of threads in response to incoming pageout requests.

[TBD: expand]

[3] This feature may not be implemented in the first phase of the NORMA IPC rewrite.

CHAPTER 11 Mach Interface Changes

11.1 No More Senders

11.1.1 mach_port_allocate

Ordinary ports allocated with `mach_port_allocate` no longer provide no more senders detection. See section 6.2, “No More Senders Interface Change”, on page 44.

11.1.2 mach_port_allocate_full

A new interface, `mach_port_allocate_full`, must be used to allocate a port that supports no more senders. See section 6.2, “No More Senders Interface Change”, on page 44.

11.1.3 mach_port_status

The mach_port_status call returns -1 for the count of send rights and -1 for the count of send-once rights when invoked on a port that does not support NMS detection. See section 6.2, “No More Senders Interface Change”, on page 44.

11.2 Message Queue Limit Behavior

Reducing a port’s queue limit fails when there are queued messages. A port’s queue limit may only be reduced when no messages are queued. See section 6.7.1, “State Transfer”, on page 48.

11.3 Port Death

The system is permitted to lazily evaluate port death as it affects send and send once rights when a dead name notification has not been requested. See section 6.6, “Dead Name Notifications”, on page 47.

11.4 Message Header Bits

11.4.1 User-Visible Bits

MACH_MSG_RPC

MACH_MSG_VOLATILE

MACH_MSG_SYNC (a.k.a. FLUSH)

MACH_SEND_NOTIFY has been eliminated from the revised Mach specification and hence we remove all supporting code and data structures.

11.4.2 Kernel Internal Bits

MACH_MSGH_BITS_NETWORK

```
#define  MACH_MSGH_BITS_META_KMSG   0x04000000/* internal use only */
#define  MACH_MSGH_BITS_UNUSED      0x03ff0000
```

11.5 MIG

11.6 Node Naming

We change or abolish the following user-level Mach interfaces to eliminate or minimize the propagation of node name information to user-level tasks. Node names are replaced with host or host_priv ports, where possible.

1. routine	**task_set_child_node**(	target_task:	task_t;
		child_node:	int);

ABOLISHED. This routine does not exist in V2. It exists in V1 only as a historical artifact.

2. routine	**norma_node_self**(	host:	host_t;
		out node:	int);

Retain in current form. Because **norma_get_special_port** must exist, we define **norma_node_self** to be able to answer the question, “what node am I?”

3. routine	**norma_task_create**(	target_task:	task_t;
		inherit_memory:	boolean_t;
		child_node:	int;
		out child_task:	task_t);

Replace node name with host_priv port.

4. routine	**norma_get_special_port**(	host_priv:	host_priv_t;
		node:	int;
		which:	int;
		out port:	mach_port_t);

Retain in current form. Norma_get_special_port is used to solve the problem of bootstrapping node communications: there must be a way for one node to obtain another node’s host port (and other special ports).

5. routine	**norma_set_special_port**(	host_priv:	host_priv_t;
		which:	int;
		port:	mach_port_t);

No change required.

6. routine	**norma_port_location_hint**(	task:	task_t;
		port:	mach_port_t;

out node: int);

Possibly this routine could return a host_t. Uses of this interface remain to be detailed.

We also include the NORMA IPC Version One internal interfaces to show where node name information is currently propagated within the implementation. These interfaces may not be retained in Version Two; for instance, they may be replaced by transport-level RPCs. In any case, these routines may continue to use node numbers where appropriate as they are all invisible to the user.

```
7. routine      norma_ipc_pull_receive(host_paging:      host_paging_t;
                                       uid:              int;
                                       dest_node:        int;
                                       out stransit:     int;
                                       out sotransit:    int;
                                       out nsrequest: mach_port_move_send_once_t;
                                       out pdrequest: mach_port_move_send_once_t;
                                       out seqno:        short;
                                       out qlimit:       short;
                                       out moved_msg_count: int);
```

Receive right migration changes significantly in V2. See section 6.7, “Receive Right Migration”, on page 48.

```
8. routine      norma_ipc_stransit_request(host_priv:   host_priv_t;
                                       uid:              int;
                                       out stransit:     int;
                                       out retarget_node: int);
```

[TBD: Transit management being revamped and simplified.]

```
9. simpleroutine  norma_ipc_node_dnrequest(host_priv:   host_priv_t;
                                       uid:              int;
                                       node:             int);
```

[TBD: Dead name management under discussion.]

```
10. simpleroutine norma_ipc_node_dnnotify(host_priv:    host_priv_t;
                                       uid:              int;
                                       source_node:      int;
                                       release_forward:  boolean_t);
```

[TBD: Dead name management under discussion.]

```
11. routine     norma_ipc_node_set(    host_priv:        host_priv_t;
                                       uid:              int;
                                       node:             int);
```

ABOLISHED. This interface is a paging back-channel. It is unnecessary with improved right migration.

CHAPTER 12 Comparison With Version One

12.1 Correctness

12.1.1 Message Ordering

Use proxy port queues to simulate the behavior of port queues in local Mach IPC.

12.2 Robustness

Analysis of failure modes common in V1 that are redressed in V2.

12.2.1 Message Ordering Violations

12.2.2 Virtual Memory Exhaustion

12.2.3 Many-to-One Message Overload

12.2.4 One-to-One Message Overload

∇ 12.2.5 Paging Flow Control Analysis

Elimination of memory_object_data_write_completed constraint.

NORMA IPC Version Two has been designed to eliminate the requirement for paging flow control imposed on Version One. This chapter discusses the need for explicit paging flow control in Version One and explains why it is no longer needed in Version Two.

Discuss required device subsystem modifications.

Discuss pager constraints.

In V2, port queue limits will be obeyed as follows.

Send right and receive right co-located:

- MACH_SEND_ALWAYS means that the message will always be forced onto the receiver's queue (this is the regular Mach semantic) and the sender will be allowed to proceed immediately

- Otherwise, port queue limits are obeyed (this is the regular Mach semantic)

Send right remote from receive right:

- MACH_SEND_ALWAYS means that the meta-kmsg will be enqueued on the receiver's queue IF the queue has not exceed the port queue limit; otherwise, the message will be enqueued on the sender's proxy queue without regard to queue limit; in both cases, the sender will be allowed to proceed immediately

- Otherwise, the sender attempts to enqueue its meta-kmsg on the receiver; if the receiver's queue is full, the sender is blocked (this is the regular Mach semantic)

In the case of a remote send right, the meta-kmsg enqueued on the receiver's message queue describes the real kmsg, which continues to live on the sending node until a thread actually "receives" the meta-kmsg. When the receive happens, the data are brought over from the sender. Fault isolation implications are discussed elsewhere. (See section 7.3, "Behavior of Ordinary Messages", on page 55.)

For pageouts originating from the kernel, which are MACH_SEND_ALWAYS, a burst of pageouts will consume resources on the sending node without killing the receiving node; pages will be brought over only as the pager manages to chew them up. However, there are two important caveats to be observed:

1. The vnode pager can't create unlimited threads. It doesn't today, but any future multi-threaded vnode pager must use a fixed set of threads. Otherwise, the pager can cause thrashing by continually creating threads while trying to keep up with an essentially infinite request queue.

2. Interrupt completion processing in the device subsystem must be changed slightly. Rather than permit unlimited asynchronous I/O to the disk, which can't sink data as fast as the pageout and vnode pager threads can generate it, after a time the device subsystem must block threads when no more asynchronous I/Os can be handled. A fixed number of operations will be permitted for each device, regardless of operation type (synchronous/asynchronous, read/write). When these operations have been consumed, requests for asynchronous I/O will be forced to be synchronous, instead. Normal behavior will resume as operations complete.

Given this design, today's paging flow control can be eliminated. Flow control will originate from the disk and be propagated backwards to the vnode pager threads, thence to message queues and eventually causing resources to accumulate on the sending node(s) rather than the receiver. Resource consumption on the sending node(s) is not a problem in the sense that it is identical to the resource consumption of a uniprocessor Mach kernel that overruns an external pager.

When compared with V1 paging flow control, the "window" for pageouts will be effectively managed by each disk involved in paging rather than today's sender-limited window. Thus, V2 "paging flow control" will offer a greater opportunity for increased throughput to multiple disks. V2 flow control solves the problem of unlimited user-to-user messages by using the port queue limit. This feature subsumes the functionality originally proposed in September for "message queue throttling". Additionally, V2 flow control solves the general problem of unlimited kernel-to-user messages by pushing those messages back onto the sending node WITHOUT causing the sending threads to block -- thus preserving Mach semantics and eliminating the need to rewrite kernel code that depends on sending messages without blocking.

12.3 Port Right Management

The existing NORMA implementation:

•Always transfers one transit with every send right except the destination right in the message header.

•Keeps transit counts with the receive right as well as holders of send rights.

•Relies on the global message ordering from the stop and wait transport to avoid associating a transit with the destination right in the message header (this is not correct, but very hard to break in practice).

•Also implements send once right transit logic for code symmetry reasons (there is no other good reason for this).

•Requires bookkeeping operations to return transits for all ports.

The new transit design is superior in all of these areas.

The existing `NORMA` implementation also contains some confusing code that:

•retries origin node if port is not initially found

•resets location of migrated port in origin node before destroying port.

This code is no longer necessary because operations on the port during migration will be blocked (e.g., message enqueues). This means that if a principal or forwarding proxy cannot be located for the port on a node where the receive right is known to have existed at some point, the receive right can be assumed to have been destroyed. The existing `NORMA` implementation employed this mechanism to deal with races caused by allowing message to be enqueued while a port was being migrated. Eliminating this code eliminates the need for the norma_ipc_node_set operation and its special treatment in the current `NORMA` code.

Receive right migration: When transits are in use, the notion of *borrowing* a transit eliminates all special-case code that existed in `NORMA IPC Version One` for transferring queued messages.

12.4 Performance

12.5 Important Implementation Details

Proxy (and other) data structures can be represented considerably more compactly.

Remote send once rights need not appear in the `NORMA` port table except for debugging purposes.

CHAPTER 13 Implementation Issues

13.1 Implementation Process

We expect to divide the project into many smaller tasks. Where possible, a task or collection of tasks will correspond to an identifiable subsystem that may be unit-tested before being integrated with other subsystems. We also expect to push less-critical tasks towards the end of the project time-line. For instance, user-level interface modifications may be postponed, and so may certain performance optimizations.

13.2 Platform

The initial `NORMA IPC Version Two` implementation will target the Intel Paragon. While some development and debugging may be done on Intel DOTs or even on PCs, the only transport we will implement is KKT layered on top of Intel's RPC/RDMA. No plan currently exists to implement an Ethernet or cluster-oriented transport.

13.3 Robustness Testing

Testing for robustness is divided into unit tests, functional tests, system regression tests, and system stress tests.

13.3.1 In-Kernel Unit Tests

Where possible, we will write in-kernel unit tests for new pieces of `NORMA IPC`. These tests will be compiled into the kernel itself (but only in a non-production kernel). We will boot the kernel to a point just before loading the first task, then break into the kernel and invoke the test driver(s) directly from the kernel debugger.

These tests will be confined to a single-node. We will introduce loopback facilities wherever we can to test both message send and receive sides. For instance, we will introduce a debug-only user-level Mach call, **`norma_debug_proxy_allocate`**, that allocates a send right based on a receive right supplied as an argument. The receive right is entered into `NORMA` as a side-effect, and the send right is created as a proxy for the receive right. Thus, it will be possible to send and receive messages on a single node that traverse `NORMA IPC`.

13.3.2 Lightweight Performance Tests

At selected milestones, we will perform simple performance tests design to test whether our design and implementation will yield acceptable results. For instance, we will measure simple message transfer by timing our in-kernel unit test; we will do the same thing for out-of-line data transfer.

13.3.3 Functional Tests

User-level “little shop of horrors”, including migration.

13.3.4 Regression Tests

RI NORMA and XMM tests

SSD EATs and SATs. PROBLEM: effective use of EATs and/or SATs requires the use of an Intel TNC-based server, which uses typed IPC. NORMA no longer supports typed IPC!

13.3.5 Stress Tests

SSD EATs and SATs. PROBLEM: effective use of EATs and/or SATs requires the use of an Intel TNC-based server, which uses typed IPC. NORMA no longer supports typed IPC!

AIM

VSX

13.3.6 Performance Characterization

There will be a final phase to characterize the performance of the beta-level or production-level `NORMA IPC Version Two`.

13.4 Dependencies

We have identified the following dependencies critical to the success of the implementation project. These dependencies are:

- Availability of an official NMK16 release by project start.
- Availability of an official NMK17 release as the ultimate code base target.
- Availability of an untyped IPC version of AD Rev 1.0.
- Availability of an official release of the Intel RPC/RDMA transport, with the following sub-dependencies:
  - This release may not include the ability to loopback RPC/RDMA requests in software. RI therefore may have to modify the Intel transport to support loopback. This feature is fundamental to our test plan.
  - This release may not have a satisfactory callback mechanism. The RI may have to modify the Intel transport to support a more general callback mechanism. This feature is fundamental to a correct implementation.
- Either EATs or SATs or both require the use of TNC, but we do not have a plan to obtain a version of the TNC server that supports untyped IPC.
- An adequate SMP testing strategy has yet to be defined.
- The RI Paragon must be upgraded to support message coprocessor functionality. Currently, the RI’s machine only uses one processor per node. Intel agrees that the theoretical throughput available in this configuration is around 30MB/sec from node to node. The platform must offer a much higher theoretical throughput if RI is to satisfactorily demonstrate that `NORMA IPC Version Two` achieves its performance goals. Note: we assume that the Intel

RPC/RDMA transport automatically “does the right thing” and uses the message coprocessor if it is available.

APPENDIX A NORMA KKT Summary

This appendix summarizes the use of KKT by NORMA.

A.1 Remote Procedure Calls

NORMA defines the following RPCs.

A.1.1 TRANSIT_REQUEST

A.1.2 YIELD_TRANSITS

A.1.3 norma_ipc_pull_receive

A.1.4 norma_ipc_migrate_state

A.1.5 RESOURCE_AVAILABLE

A.1.6 UNBLOCK SENDER

A.2 Transport Upcalls

NORMA depends on the following transport upcalls.

A.2.1 *deliver

APPENDIX B

NORMA IPC Threads

B.1 norma_message_delivery

The NORMA *deliver upcall obtains an incoming `meta_kmsg` or `kmsg` from the transport. The upcall determines the destination port and, if it exists, attempts to deliver the message directly from interrupt level. If the fast delivery path isn't possible, the message is handed off to a **`norma_message_delivery`** thread to be delivered as if the message originated on the local node.

The delivery thread attempts to enqueue a message following all of the protocols governing port operations, checking for aliveness, queue limit violations, blocked receivers, etc.

This thread has a wired stack and is guaranteed to run.[1]

B.2 Transport Thread

B.3 Kserver Thread

[1] Does this thread require VM privilege?

APPENDIX C KKT Interface Listing

Calls from a kernel subsystem into the transport:

```
kkt_return_t KKT_HANDLE_ALLOC(
        channel_t           channel,            /* IN */
        handle_t            *handle,            /* OUT */
        void                (*error)(),         /* IN */
        boolean_t           wait);              /* IN */

kkt_return_t KKT_HANDLE_FREE(
        handle_t            handle);            /* IN */

kern_return_t KKT_CHANNEL_INIT(
        channel_t           channel,            /* IN */
        unsigned int        handles,            /* IN */
        unsigned int        recv_handles,       /* IN */
        void                (*deliver)(),       /* IN */
        void                (*malloc)(),        /* IN */
        void                (*free)());         /* IN */

kkt_return_t KKT_SEND_CONNECT(
        handle_t            handle,             /* IN */
        node_name           node,               /* IN */
        endpoint_t          endpoint,           /* IN */
        request_block_t     request,            /* IN */
        boolean_t           more,               /* IN */
        kern_return_t       *ret);              /* OUT */

kern_return_t KKT_CONNECT_REPLY(
        handle_t            handle,             /* IN */
        kern_return_t       kr);                /* IN */

kkt_return_t KKT_REQUEST(
        handle_t            handle,             /* IN */
        request_block_t     request);           /* IN */

kkt_return_t KKT_HANDLE_ABORT(
        handle_t            handle,             /* IN */
        kkt_return_t        reason);            /* IN */

kkt_return_t KKT_HANDLE_INFO(
        handle_t            handle,             /* IN */
        handle_info_t       info);              /* OUT */
```

Calls from the transport to the kernel (indirect via the channel structure):

```
        void (*channel->deliver)(
                channel_t          channel,          /* IN */
                handle_t           handle,           /* IN */
                endpoint_t         endpoint;         /* IN */
                vm_size_t          size,             /* IN */
                vm_offset_t        *data);           /* IN */

        unsigned char * (*channel->malloc)(
                channel_t           channel,         /* IN */
                vm_size_t           size);           /* IN */

        kern_return_t (*channel->free)(
                channel_t          channel,          /* IN */
                unsigned char      *mem,             /* IN */
                int                size);            /* IN */
```

Callback support routines:

```
        kkt_return_t KKT_ADD_NODE(
                node_name          node,             /* IN */
                node_map_t         map);             /* IN */

        kkt_return_t KKT_REMOVE_NODE(
                node_name          *node,            /* OUT */
                node_map_t         map);             /* IN */

        kkt_return_t KKT_MAP_ALLOC(
                node_map_t         map);             /* OUT */

        kkt_return_t KKT_MAP_FREE(
                node_map_t         map);             /* IN */

User supplied callbacks:
        void (*handle->error)(
                kkt_return_t       kr,               /* IN */
                handle_t           handle,           /* IN */
                request_block_t    request);         /* IN */

        void (*request->callback)(
                kkt_return_t       kr,               /* IN */
                handle_t           handle,           /* IN */
                request_block_t    request);         /* IN */
```

NORMA IPC Issues

Issue	Description	Status	Owner
Pin/unpin engine review			
	...errors		
	...unaligned transfers		
KKT request interface			
fast interrupt delivery path review			
port migration review			
blocked senders review			
Port/UID Conversion	UID Format	Open	!!!!!
RPC OOL optimization	Reply to an RPC w/OOL is COW!	Open	
norma_port_location	Define host_t interface & add to spec	Open	
XMM/paging flow	Allow XMM to do inline ONLY msg recv	Open	
Kmsg / Port Release	Decrementing port ref counts		
OOL Memory	Describe entire OOL delivery path	Closing	
- IPC Format	Use entry lists (document)	Closing	alan
- Reception	Overwrite and alignment issues	Closing	
- Volatility	MIG; MACH_MSG_RPC/VOLATILE	Closing	
Vnode Pager Reqs	Requirements necessary for pfc	Open	
Message Reception	Describe entire reception path	Open	