Kernel to Kernel Transport Interface for the Mach Kernel

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. Central to this new design is a separate, well-defined subsystem responsible for reliably conveying byte streams from one node to another. 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

CHAPTER 1 Introduction

This paper specifies a transport interface to be used by Mach kernels operating independently in a multicomputer to send messages and data from node to node. Although designed for multicomputer class machines such as the Intel Paragon, the design is amenable to a loosely coupled cluster of computers that are cooperating to form a multicomputer.

The objective of this interface is to free the higher level consumer of transport specifics such as flow control, channel contention, etc. Recognizing that there are and will be several different transport media, the transport interface is sufficiently abstracted to allow different or even multiple transport implementations.

The primary consumer of this transport will initially be the NORMA subsystem. However, there are other projects that will benefit from a generalized kernel to kernel transport, so the interface is sufficiently abstracted to provide effective functional layering and relieve the consumer of transport issues.

The primary model for the transport favors a receiver-pull implementation of data movement, requiring the receiving node to request the data from the sender as space becomes available. There is nothing in the interface that precludes a sender-push style interaction, but there is some bias to the former. The target architectures this interface is designed for are primarily high bandwidth, low latency interconnects common in HPC computing.

The RPC interface presented in this document has been developed by Intel SSD. This part of the document is authored and copyrighted by Intel Corporation.

CHAPTER 2 Interface Specification

Conceptually, the transport mechanism is a connection oriented protocol. Due to the complexity of a Mach message, a datagram style interface becomes unwieldy and difficult to use.

There are two classes of calls in the interface: a bulk data mover, and an RPC interface.

Both sets of interfaces have two types of calls: those that are made by the kernel to the transport, and upcalls made from the transport into the kernel.

The first type of calls deal with sending and receiving data over the transport. It is a design goal that the transport is responsible for moving opaque data across the transport medium, with no requirement to examine the data being moved.

The second type of calls are used by the transport to obtain and release system resources, or to alert the higher layer that data has been sent or received. Another design goal is for the transport to obtain system resources such as buffers from a higher level that is "system aware," relieving the transport layer from complex resource issues.

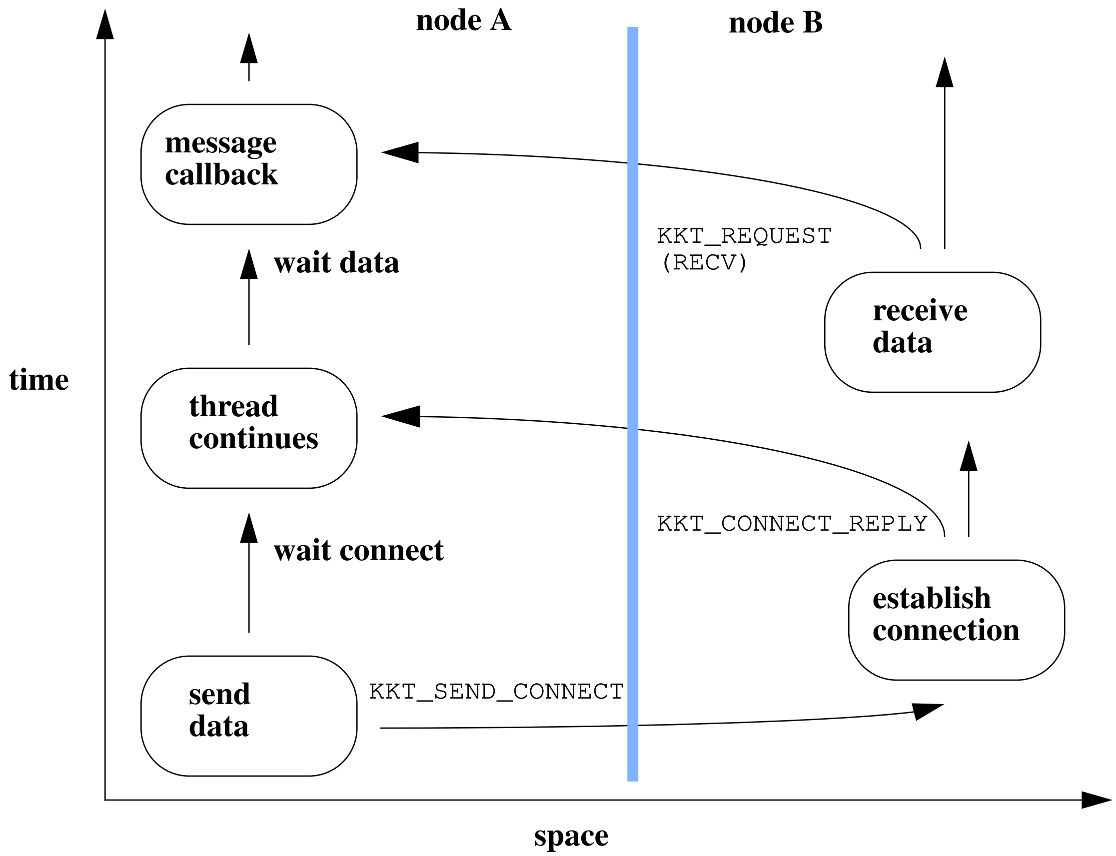

Figure 1: KKT interactions: time vs. space

2.1 Argument Description

channel — Used to obtain a class of services from the transport. There is a predefined set of channels that can be used in various ways by both the sender and receiver. Each channel has a pre-allocated set of resources associated with it.

For example user generated Mach messages, and NORMA to NORMA messages used to maintain distributed state represent different classes of services.

channels are typically specified by a channel_t, which is of type usigned int.

handle — Connection identifier with the transport. Typically persists the duration of a single message transfer. It is allocated from and returned to the transport. Can be thought of as a connection endpoint. Handles are associated with channels. Handles are specified by a handle_t, which is opaque to the user and hence of type *void **

node — Used to define the destination node.

size — The size of the data being sent, in bytes.

data — The kernel virtual address of the data to be sent.

endpoint — Specifies a high level destination identifier which is opaque to the transport. Examples of an `endpoint` would be a DIPC UID or a remote memory object identifier. This value is opaque to the transport, which defines the data type as two elements of *void **.

info — Defined to be of type handle_info_t, which contains information the transport is willing to export to the user.

```
struct handle_info {
        node_name         node;
        kern_return_t     abort_code;
} *handle_info_t;
```

The *node* argument indicates which node, if any, the handle refers to. An unconnected handle will have a *node* of `NODE_NAME_NULL`.

The *abort_code* argument contains the reason why a handle was aborted (a handle abort operation is requested by a client of KKT). This argument is opaque to KKT and therefore can be used as a KKT client sees fit, with one exception. In the case that a handle has not been aborted, this code will be `KERN_SUCCESS`, so whatever states a client defines for *abort_code* must include `KERN_SUCCESS`.

request — Specifies a request to be performed by the transport. A request takes the form of a user-allocated request block:

A request block is filled in by the user with the exception of the *transport* field, which is opaque to the user and may be used by the transport to link requests onto a queue, and to maintain status bits. A `transport_t` is defined by the transport implementation.

```
typedef struct request_block {
        request_t         request_type;
        union {
                vm_offset_t       data_address;
                kkt_sglist_t      list;
        } data;
        vm_size_t         data_length;
        void              (*callback)();
        unsigned int      args[2];
        request_block_t   *next;
        transport_t       transport;
} *request_block_t;

typedef struct transport_request {
        short             type;
        short             status;
} *request_t;

typedef struct kkt_sglist {
        vm_offset_t       ios_address;
        vm_size_t         ios_length;
```

```
} kkt_sglist_t
```

Upon completion of the operation, the if *callback* is non-NULL it will be invoked with a transport return value, the handle, and the request_block. The *args* and *next* fields are opaque to the transport, reserved for use by the user. The callback routine is expected to execute in interrupt context.

data_address is a kernel virtual address which can be used by the transport directly. For most architectures, this limits the size of the request to one physical page or less, but this is not a requirement. *data_length* describes the size of the data.

node_map A node map is a transport dependent representation of a collection of nodes. It is an implementation dependent data structure that is used to store and retrieve nodes, to be used by higher levels for callbacks. The data type node_map_t is opaque to the user, of type *void* *.

2.2 KKT_CHANNEL_INIT

```
kkt_return_t KKT_CHANNEL_INIT(
        channel_t           channel,          /* IN */
        unsigned int        handles,          /* IN */
        unsigned int        receivers         /* IN */
        void                (*deliver)(),     /* IN */
        void                (*malloc)(),      /* IN */
        void                (*free)());       /* IN */
```

This routine is used to define the resources and install the upcalls for a channel intended for long-term data transfers. The upcalls are invoked when a message arrives on a channel, or when acquiring and releasing system resources.

KKT_CHANNEL_INIT must be executed only one time per channel, typically during system initialization. A null value for any argument is an error. If the system cannot allocate the necessary data structures due to resource shortages, KERN_RESOURCE_SHORTAGE will be returned.

Each channel may specify *handles* number of handles available for establishing communication with another node. These are the connection endpoints that must be acquired before a connection is established. *handles* specifies the total number of connection endpoints and makes no distinction between receive or send connections.

receivers specifies the number of handles that will be set to receive messages on the channel from remote nodes. If the local node transport does not have a handle available to receive a connection (i.e. *receiver* number of handles are currently processing requests from remote nodes), the transport may choose to queue the request or to refuse the connection and return KERN_RESOURCE_SHORTAGE.

It is an error to invoke **KKT_CHANNEL_INIT** with zero *handles* or *receivers*.

The **deliver* upcall identifies the main entry point into the system from the transport on a specific channel, initiated as the receive side operation in response to a **KKT_SEND_CONNECT** on the sending node. This upcall routine may called at interrupt level.

malloc identifies the main entry point into the system from the transport for a specific channel to obtain memory. This upcall routine is guaranteed to be called in thread context.

The *free* upcall returns the memory allocated via the *malloc* upcall. Also must be called in thread context.

2.2.1 Channel Data Structure

The transport is expected to maintain a structure per channel that will store this (and other) information.

```
struct channel {
        channel_t           channel;
        void                (*deliver)();
        void                (*malloc)();
        void                (*free)();
        unsigned int        handle_count;
        queue_t             handle_free_list;
}
```

2.2.2 Return Value

```
KKT_SUCCESS                 /* channel initialized successfully */
KKT_INVALID_CHANNEL         /* illegal channel specifier */
KKT_CHANNEL_IN_USE          /* re-INIT of existing channel */
KKT_RESORCE_SHORTAGE        /* could not allocate resources */
KERN_INVALID_ARGUMENT       /* bogus parameter(s) */
```

2.3 KKT_HANDLE_ALLOC

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

Allocate a handle that will be used when passing data to the transport layer. The handle returned by this call represents a connection endpoint which is not yet associated with a destination.

The *channel* argument specifies a class of service to be used by the transport. There is a tacit assumption that there are a limited number of resources allocated to each class of service by **KKT_CHANNEL_INIT**, and that the call may block if it cannot allocate more. Using this call to request a handle is equivalent to requesting a class of service.

The *handle* out parameter is only filled in if the routine is successful. This will be used by all succeeding operations as the identifier upon which to send or receive data.

error is a user specified routine that will be invoked if the transport encounters an error during a transmit operation and the request does not have a callback associated with it. The error routine will be invoked for end -to-end types of transport errors, such as transmission failure or an unresponsive remote node. The error routine is also invoked when requests are aborted via **KKT_HANDLE_ABORT**. A NULL value is permitted but implies that requests currently queued to the transport that do not have a callback specified may fail without alerting higher levels. Error processing specifies that requests in progress during the failure will return the error reason; subsequent KKT operations will return `KKT_ABORT` or `KKT_HANDLE_TYPE`. Not specifying error may result in the loss of the transport error.

The calling routine has the option to wait until a handle becomes available, or to try again on a different channel or at a future time. The common case will block until a handle is available.

Finally, must_request will specify that no eager push of data will be allowed on this handle. If must_request is FALSE, then the transport will decide when the data can be effectively transferred to the remote node. Setting this parameter to TRUE allows the consumer of this service some control over when data movement will occur; data will not be transmitted until the receiver is ready for it.

2.3.1 The Transport Data Structure

The transport must maintain a handle data structure for each handle.

```
typedef struct transport_handle {
        channel_t          channel;
        boolean_t          connected;
        transport_handle_thandle;
        rdma_token_t       token;
        buffer_t           *buffer;
        struct request_block*request_list;
} *transport_handle_t;
```

2.3.2 Return Value

```
KKT_SUCCESS                /* handle was allocated */
KKT_INVALID_CHANNEL        /* illegal channel value specified */
KKT_RESOURCE_SHORTAGE      /* no handles available */
```

2.4 KKT_HANDLE_FREE

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

Return the specified handle to the transport, which may in turn may cause a thread waiting on a class of service to be awakened. This call will typically be executed from the data send completed callback on the final **`KKT_REQUEST`** or **`KKT_SEND_CONNECT`** invocation. Prematurely releasing the handle is undefined but may sever the connection. This routine can be called from interrupt level.

2.4.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE                  /* Handle argument is invalid */
```

2.5 KKT_SEND_CONNECT

```
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 */
```

This call initiates kernel to kernel communications and is synchronous with respect to the target node. The invoking thread will block until the target node has completed the delivery operation and returned a status. A connection is established and data made available at the same time. In the common case, the message will be able to be provided in it's entirety to the remote node, being completely described by *request*. If the message is too large or complex to send in a single call, the **KKT_REQUEST** call described below will utilize the connection established at this point.

On the remote node, the transport will upcall into the system with the appropriate channel **deliver* routine, which will typically hand off the message to a waiting thread but may process the request immediately. Status returned from the remote node via **KKT_CONNECT_REPLY** will be returned in *ret*; this would include insufficient resources, queue full, or other higher level return codes. *ret* must point to a buffer the size of two elements of kern_return_t in order to receive both a status and substatus. Both of these values are opaque to the transport. **KKT_CONNECT_REPLY** must be used on the receiver before further operations (such as **KKT_REQUEST**) may be used.

If a callback routine specified in *request* is a non-null function pointer it will be invoked when the data has physically been transferred off node. The callback will not be invoked if the transport returns `KKT_DATA_SENT`, indicating the data has been transferred to the remote node synchronously.

more provides a hint to the underlying transport that there will be more data coming as part of this transfer or not. This is useful where a single transport manages different transport implementations or media with different costs associated with them (e.g. a light weight transfer mechanism and a bulk data transfer mechanism).

2.5.1 Resource Consumption

In this transport model, the sender provides a kernel virtual address of the data to be sent. This requires the page presented to be in memory and wired in some way, allowing any conceivable transport device to immediately use it.

In order to manage and clean up the sending data, it is desirable to place the data into some temporary data structure such as an entry list `vm_map_copy_t`. This is required as clean up of the data may in fact be asynchronous, not occurring until the actual data has arrived on the remote node. The transport is not "system aware" and does not have knowledge of various data structures such as a `vm_map_copy_t`, requiring clean up to occur at a higher layer.

2.5.2 Return Values

```
KKT_SUCCESS        /* Connection made, data ready to be sent */
KKT_DATA_SENT      /* data sent to remote node; no callback */
KKT_NODE_DOWN      /* destination node is not responding */
KKT_INVALID_HANDLE/* handle is invalid */
KKT_HANDLE_TYPE    /* handle is of wrong type (could be aborted) */
KKT_INVALID_REQUEST/* illegal operation requested */
```

2.6 KKT_CONNECT_REPLY

```
          kkt_return_t KKT_CONNECT_REPLY(
```

```
                    handle_t          handle,           /* IN */
                    kern_return_t     status,           /* IN */
                    kern_return_t     substatus);       /* IN */
```

Respond to a **KKT_SEND_CONNECT** request. The sender on the remote node will block until the receiver responds with this call. The return values of *status* and *substatus* will be written into the location pointed to by the **ret* parameter specified in the **KKT_SEND_CONNECT** call invoked on the remote node.

KKT_CONNECT_REPLY may only be used once, on a newly-received handle. Subsequent attempts to use this call on a handle will result in an error, KKT_HANDLE_TYPE.

2.6.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE          /* handle is invalid */
KKT_HANDLE_TYPE             /* handle is of wrong type or aborted*/
```

2.7 KKT_REQUEST

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

A non-blocking operation which enqueues a *request* to the transport. The request is typed with the proper operation to execute, REQUEST_SEND or REQUEST_RECV. The user must provide a kernel virtual address in the data_address field of *request* where the data will be sent from or received into.

A request will take one of two forms, depending on the REQUEST_SG bit in type field: either as a single datum request, or as a vector request. If REQUEST_SG is set, data.list is a pointer to a list of kkt_sglist elements. The number of elements that will be processed in a kkt_sglist list is determined by the size of the request; which in this case is the sum of the individual list elements to be sent.

Upon completion, the request will be dequeued, and if the callback is a non-NULL function pointer, the callback routine will be invoked.

This call may return with KKT_DATA_SENT if it is a send operation and the transport was able to immediately send the data. Or, a receive operation may return with KKT_DATA_RECEIVED if the data is present on the node. In either case, the callback routine will not be invoked even it has been specified. Return codes always refer to the entire request; there are no partial successes.

2.7.1 Request types

User requests (request_type->type):

```
#define  REQUEST_SEND      1
#define  REQUEST_RECV      2
#define  REQUEST_SG        0x100
```

2.7.2 Return Value

```
KKT_SUCCESS
KKT_DATA_RECEIVED           /* data xfer'd; no callback forthcoming */
```

```
KKT_DATA_SENT            /* data sent to remote node; no callback */
KKT_NODE_DOWN            /* destination node is not responding */
KKT_ERROR                /* something BAD happened */
KKT_INVALID_HANDLE       /* handle is invalid */
KKT_HANDLE_TYPE          /* handle is of wrong type or aborted */
KKT_INVALID_REQUEST      /* illegal operation requested */
```

2.8 KKT_HANDLE_ABORT

```
kkt_return_tKKT_HANDLE_ABORT(
        handle_t         handle,           /* IN */
        kkt_return_t     reason);          /* IN */
```

This call will abort all operations outstanding on a handle, including remote operations that may be pending. All callback operations will be executed with a return value of *reason* supplied. The remote handle will execute all pending callbacks with *reason* supplied to them. Upon completion, both handles are unconnected and using them will result in a return value of `KKT_ABORT` or `KKT_INVALID_TYPE`.

2.8.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE       /* handle is invalid */
KKT_HANDLE_TYPE          /* handle is of wrong type or aborted*/
KKT_HANDLE_BUSY          /* abort happened, but transport in
                                an indeterminate state */
```

2.9 KKT_HANDLE_INFO

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

Get transport specific information from the transport about a specific handle. The available information is returned in info. Note that this call is valid for aborted handles, to obtain the error code.

2.9.1 Return Value

```
KKT_SUCCESS
KKT_ABORT                /* handle has been aborted */
KKT_INVALID_HANDLE       /* handle isn't valid */
KKT_HANDLE_TYPE          /* handle is of wrong type */
```

2.10 KKT_HANDLE_MIGRATE

```
kkt_return_tKKT_HANDLE_MIGRATE(
        node_name        node,             /* IN */
        unsigned int     old_handle,       /* IN */
        handle_t         handle);          /* IN */
```

Given that *old_handle* is a valid handle on node *node*, move the receive connection from node *node* to the current node and associate it with *handle*.

Notice that *old_handle* must be quiescent but connected to a sending node - the sending node has enqueued the data to be sent to the underlying transport, but the receiving node has not yet initiated reception of the data. If a message is currently being sent, or if *old_handle* is not yet connected, an error will be generated.

The new handle, *handle*, must be newly-allocated and unused for any other purpose.

2.10.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE          /* illegal handle value (for either handle) */
KKT_HANDLE_TYPE             /* handle has been used for something else */
KKT_HANDLE_UNUSED           /* remote handle is not connected */
KKT_HANDLE_BUSY             /* remote handle is transmitting */
KKT_HANDLE_MIGRATED         /* remote handle has already been
                             * migrated */
```

2.11 (*channel->deliver)()

```
void (*channel->deliver)(
        channel_t         channel,          /* IN */
        handle_t          handle,           /* IN */
        struct transmit   *transmit,        /* IN */
        boolean_t         inline);          /* IN */
```

The **deliver* routine is the central entry point the transport will use to enter the kernel when it has data to deliver. It may be called from interrupt level. This routine acts as a server to decide which path the incoming data will take. It is called exactly once each time a newly allocated handle is invoked on a node.

The *size* field conveys the size of the data sent by **KKT_SEND_CONNECT** on the remote node.

The *transmit* parameter is a pointer to a transport provided buffer as follows:

```
struct transmit {
        unsigned int      *ulink;  /* user available field */
        endpoint_t        endpoint;/* same as endpoint param */
        vm_size_t         size;    /* same as size param */
        unsigned int      buff[];  /* data if valid */
}
```

The ulink field can be used to store things like a queue pointer or a handle, at the discretion of the user. This provides a general purpose mechanism to enqueue data from interrupt level that could be consumed in thread context later. This structure and all of the elements contained therein are transient and only valid until **KKT_CONNECT_REPLY** is issued, at which time the pointer will be undefined. *transmit* is a kernel virtual address and does not require any further translation by the kernel.

If the transport was able to send the data in the message *inline* will be a TRUE and valid data is contained in the transmit buffer in *buff*. This indicates that a receive request is not needed to receive the data. Data provided in this manner is contained in a transport resource and must not be deallocated and must be copied if it is to persist beyond the execution of **KKT_CONNECT_REPLY**. *size* indicates the size of this data. *endpoint* is the user defined endpoint identifier.

2.11.1 Transport Action

Upon receipt of a connection request, the transport must allocate a data structure that will store the necessary state for maintaining a connection. This will include whatever transport dependent tokens or handles are necessary. The transport will then provide to the upper layer a unique handle to be used for communicating on the connection. Receive handles were established at initialization time by **KKT_CHANNEL_INIT**.

2.12 (*channel->malloc)()

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

Routine provided by the kernel that the transport can invoke to obtain memory. Memory allocation is not done in the transport, which does not have system knowledge.

2.12.1 Return Value

Pointer to allocated storage, else NULL implies a resource shortage.

2.13 (*channel->free)()

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

Routine provided by the kernel for returning memory allocated to the transport layer. It will be invoked by the transport layer. There can be one *free* routine per channel, to match the channel *malloc* routine.

2.14 User Defined 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 */
```

The *callback* routine is supplied by the user when invoking asynchronous data movement operations. The user callback is invoked by the transport when either the data associated with the callback has been physically sent/received, or when the transport has determined the data cannot be sent/received. Examples of the latter case exist when a send operation has been initiated and the remote node has gone down.

The *error* routine is supplied when allocating a handle and is invoked when an error condition arises and the current request does not have a callback specified.

The first argument, *kr*, is always defined to be the transport return code. This value will indicate the success or failure of the transport operation. If the transport encounters an error on an operation, kr will represent the error to the callback in the current request. If no callback is specified, the error routine specified on the handle will be invoked with the same return code; if there is no error routine, the return value will be dropped and the request will be dequeued. All remaining requests queued on the handle will be processed in the same manner with kr set to `KKT_ABORT`. All subsequent operations on the handle will return `KKT_ABORT` or `KKT_INVALID_HANDLE`. Note that a handle may also be aborted explicitly, via **`KKT_HANDLE_ABORT`**; in that case, **`KKT_HANDLE_INFO`** returns an error code supplied by the caller that aborted the handle.

handle is always returned as the handle the operation has occurred on.

request is the request block that was supplied to initiate the operation. There is room in the request block to supply pointers and data that is opaque to the transport and can be used to identify the current operation and initiate further operations.

2.15 Message State Transitions

Once a thread has acquired a handle, it may send data and establish a connection via **`KKT_-SEND_CONNECT`().** The remote node will set up a receive handle and establish the necessary state to maintain the connection. The transport will then invoke the local **deliver* routine specified for the channel the message is being delivered on.

The transport provides a receive handle to the **`*deliver`** routine, at which point a persistent connection has been established. The **`*deliver`** routine will examine the endpoint to establish a more meaningful connection between kernel subsystems. The request may be met at interrupt level, or more commonly a thread would be awakened to deal with the incoming data. Once a complete connection has been established, a **`KKT_CONNECT_REPLY`** is issued with a return code for the waiting **`KKT_SEND_CONNECT`** on the remote node.

The awakened thread allocates *size* bytes of memory and receives the message into it by invoking **`KKT_REQUEST`** specifying a **REQUEST_RECV** operation.The remote data is associated with the handle, which is used to request it from the remote node.

The receiver may elect to sleep and wait until the data has arrived, or it may allocate more memory anticipating forthcoming data. The callback routine will not be invoked until all of the data asked for has been received - i.e. there are no partial transfers.

The receiving thread may now examine the data and discover that there is more to come on the remote node. Once again it will allocate memory and invoke **`KKT_REQUEST`** `to receive data.`

Once the send data has been wired down, **`KKT_REQUEST`** is once again invoked to transfer the data off node. The remote memory has been allocated and the transport has had an opportunity to make this an efficient transfer.

Once all of the data has been transferred, the handles on both sides of the connection will be returned to the transport layer by the **msg_callback* routine.

If the connection is prematurely terminated the behavior is undefined.

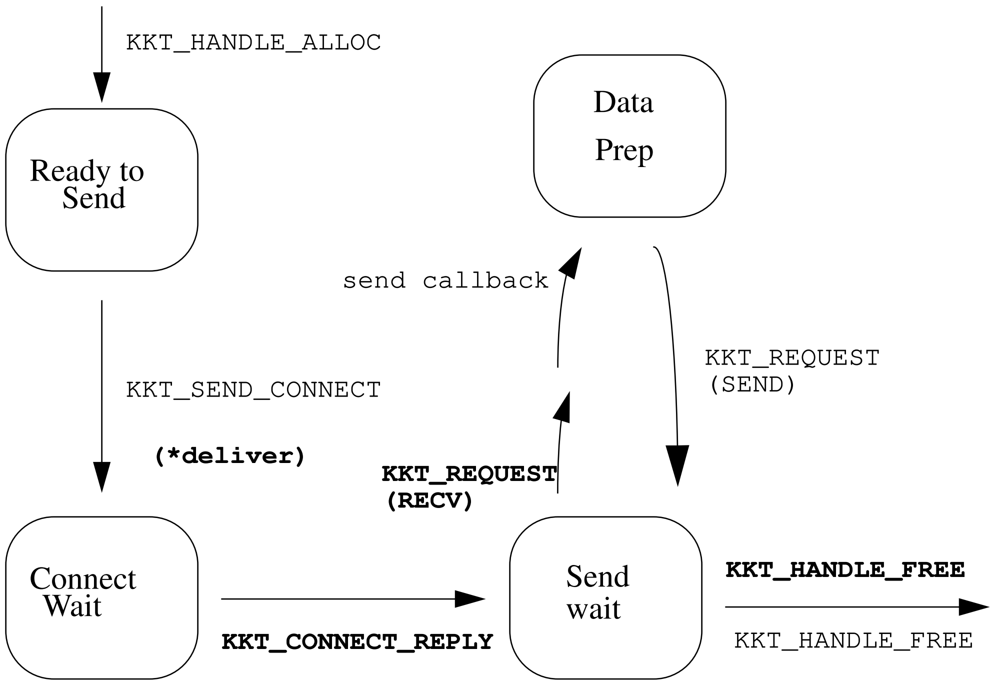

Figure 2: State transitions (Receive side in bold)

2.16 RPC Interface

The KKT interface provides a set of simple but powerful RPC interfaces. Although conceptually similar to the bulk data facilities, there are some subtle differences. For example, handles are not interchangeable between types of operations.

2.17 KKT_RPC_INIT

```
kkt_return_t KKT_RPC_INIT(
        channel_t          channel,           /* IN */
        void               (*rpc_deliver)(),  /* IN */
        unsigned int       send_handles,      /* IN */
        unsigned int       receive_handles,   /* IN */
        unsigned int       max_send_size)     /* IN */
```

Establish a channel to be used for RPC operations. The ***channel*** parameter must be unique among all KKT channels. *send_handles* and *receive_handles* refer to the number of handles allocated for the transmission of messages. RPC handles are a scarce resource; if not enough handles are available, an error will be returned and the call will fail.

The **rpc_deliver* upcall identifies the main entry point into the system from the transport on a specific channel, initiated as the receive side operation in response to a **KKT_RPC** on the sending node. This upcall routine may be called at interrupt level.

The *max_send_size* parameter specifies the largest message that will be transmitted on this channel. The actual allowable size is implementation dependent; if *max_send_size* is larger than the implementation will support, KERN_RESOURCE_SHORTAGE will be returned by the underlying transport.

2.17.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_CHANNEL         /* illegal channel specifier */
KKT_CHANNEL_IN_USE          /* attempt to re-initialize existing channel */
KKT_RESOURCE_SHORTAGE       /* could not allocate resources or
                                   max_send_size too big */
KKT_INVALID_ARGUMENT        /* illegal argument value(s) */
```

2.18 KKT_RPC_HANDLE_ALLOC

```
kkt_return_t KKT_RPC_HANDLE_ALLOC(
        channel_t           channel,          /* IN */
        handle_t            *uhandle,         /* OUT */
        vm_size_t           size)             /* IN */
```

Allocate a send handle on channel *channel*. The handle will be returned in *uhandle* if the call is successful. *size* must be less than or equal to max_send_size set up by **KKT_RPC_INIT**.

KKT_RPC_HANDLE_ALLOC will block if a handle is not available when the call is made.

2.18.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_CHANNEL         /* illegal channel specifier */
KKT_INVALID_ARGUMENT        /* bogus parameter(s), e.g., size is
                                   larger than the pre-declared
                                   buffer size */
```

2.19 KKT_RPC_HANDLE_BUF

```
kkt_return_t KKT_RPC_HANDLE_BUF(
        handle_t            handle,           /* IN */
        unsigned int        *userbuf)         /* OUT */
```

Given an RPC handle, return the buffer in *userbuf*. The size of the buffer was specified when the handle was allocated.

2.19.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE          /* illegal handle value */
KKT_HANDLE_TYPE             /* not an RPC handle */
```

2.20 KKT_RPC_HANDLE_FREE

```
kkt_return_t KKT_RPC_HANDLE_FREE(
        handle_t             handle);         /* IN */
```

Return the specified RPC handle to the transport, which may in turn may cause a thread waiting on an RPC handle to be awakened. This call will only be executed after the completion of an RPC, in response to a **KKT_RPC_REPLY** on a remote node. It is an error to free a handle that is receiving a message - only handles that are allocated need be freed.

2.20.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE         /* illegal handle value */
KKT_HANDLE_TYPE            /* not an RPC handle */
KKT_INVALID_REQUEST        /* attempt to free a receive-side handle */
```

2.21 KKT_RPC

```
kkt_return_t KKT_RPC(
        node_name        node,            /* IN */
        handle_t         handle)          /* IN */
```

Send an RPC message to a remote node *node*. This call will block until the remote node responds with **KKT_RPC_REPLY**. The data transferred is contained in the buffer obtained by **KKT_R-PC_HANDLE_BUF**. This call should fail if the remote node cannot be reached.

Unfortunately, not all transport implementations provide desirable semantics; some failures may only be observed on callbacks, others (such as an attempt to send an RPC to a node that doesn't exist) may simply result in a hung thread.

2.21.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE         /* illegal handle value */
KKT_HANDLE_TYPE            /* not an RPC handle */
KKT_NODE_DOWN              /* illegal or unavailable node */
KKT_ERROR                  /* lower-level transport error */
```

2.22 KKT_RPC_REPLY

```
kkt_return_t KKT_RPC_REPLY(
        handle_t         handle)          /* IN */
```

Reply to an RPC that was received on handle. This call will implicitly deallocate the handle, making it available to receive another RPC message on this channel. The buffer to be transmitted was received in the ***rpc_deliver** routine.

2.22.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_HANDLE         /* illegal handle value */
KKT_HANDLE_TYPE            /* not an RPC handle */
KKT_ERROR                  /* lower-level transport error */
```

2.23 (*channel->rpc_deliver)

```
void (*chan->rpc_deliver)(
        channel_t        channel,         /* IN */
        rpc_buf_t        rpc_buf)         /* IN */
```

The **rpc_deliver* routine is the central entry point the transport will use to enter the kernel when it has data to deliver. It may be called from interrupt level. This routine acts as a server to decide which path the incoming data will take. It is called exactly once each time a KKT_RPC call is executed to this node.

The *channel* parameter conveys the channel the request was made on.

The *rpc_buf* parameter is a pointer to a transport provided buffer as follows:

```
typedef struct rpc_buf{
        unsigned int      *ulink;  /* user available field */
        handle_t          handle;  /* RPC handle */
        unsigned int      buff[];  /* data buffer */
} *rpc_buf_t;
```

The ulink field can be used to store a queue pointer, at the discretion of the user. This provides a general purpose mechanism to enqueue data from interrupt level that could be consumed in thread context later. This structure and all of the elements contained therein are transient and only valid until **`KKT_RPC_REPLY`** is issued, at which time the pointer will be undefined. *rpc_buf* is a kernel virtual address and does not require any further address translation. It is also a kernel resource and should not be deallocated. *buff* is a copy of the buffer the sender had available.

Upon receipt of an RPC request, the handle the request is received on is unavailable until a **`KKT_RPC_REPLY`** is executed, which will implicitly rearm the handle for reception.

2.24 Node Map Interface

The KKT interface is designed to support a a large number of architectures and to relieve the higher layers from platform specific mechanisms. A platform dependent data structure is the `node_name`, used to identify a specific node within a multicomputer or cluster. A `nodename` could be represented by a simple integer or something as complex as an IP address. The actual representation of the data is platform dependent and generally opaque to higher levels.

Situations arise whereby `node_name`s must be grouped together. Although it is possible for this need to be met by using a simple array, the transport itself may have an efficient way of storing and manipulating this type of data structure. This introduces the concept of a **`node_map`**, used to store collections of `node_names`.

The consumer of this service must obtain a `node_map` from the transport, and then will add to or extract from the map `nodenames`. This is useful for establishing callbacks or logically grouping some number of nodes together.

The implementation of a `node_map` is platform dependent, and may be dynamic. Provision is made for maps that must be allocated in their entirety when the `node_map` is obtained to prevent resource shortage problems at critical times. For example, the `DIPC` subsystem uses a `node_map` to establish callbacks on ports that are experiencing resource shortages; if a `node_map` cannot be allocated when it is needed, the `node_name` is added to a global map that was set up during system initialization.

2.25 KKT_MAP_ALLOC

```
kkt_return_t KKT_MAP_ALLOC(
        node_map_t        *map,             /* OUT */
        boolean_t         full);            /* IN */
```

Used to obtain a `node_map` from the transport. The *full* parameter specifies that *map* must be completely allocated.

2.25.1 Return Value

```
KKT_SUCCESS
KKT_RESOURCE_SHORTAGE      /* insufficient memory to allocate */
```

2.26 KKT_MAP_FREE

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

Release the *map* back to the transport. There is no requirement that *map* be empty when this is invoked.

2.26.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_ARGUMENT       /* bogus map argument */
```

2.27 KKT_ADD_NODE

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

Add a node to a list of nodes being maintained by *map*.

node is the new node to add to the list. If the node is already present the call will return an error.

map is an opaque node_map_t that was allocated by **KKT_INIT_MAP**.

2.27.1 Return Value

```
KKT_SUCCESS
KKT_RESOURCE_SHORTAGE      /* insufficient memory to allocate */
KKT_NODE_PRESENT           /* node already in map */
KKT_INVALID_NODE           /* illegal value for node */
```

2.28 KKT_REMOVE_NODE

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

Remove a node from a list of nodes being maintained by the transport in the `node_map_t` pointed to by *map*.

node will be filled in with a node_t that was previously inserted into *map* by **KKT_ADD_NODE** if the call is successful.

`KKT_RELEASE` is returned when the last node is returned, leaving an empty map. This return code indicates that *node* is valid, but map is empty and should be deallocated. `KKT_RELEASE` should be interpreted as `KKT_SUCCESS` with an advisory warning.

When there are no nodes in the map to return to the user, `KKT_EMPTY` is returned.

2.28.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_ARGUMENT        /* bogus argument */
KKT_MAP_EMPTY               /* no nodes in map */
KKT_RELEASE                 /* node is valid, map is empty*/
```

2.29 Multicast

It is desirable for work in progress and anticipated future optimizations to have a multicast capability available at this level of the system. TBD.

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 */
        boolean_t         must_request);    /* IN */

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


kkt_return_t KKT_CHANNEL_INIT(
        channel_t         channel,          /* IN */
        unsigned int      handles,          /* IN */
        unsigned int      receivers,        /* 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 */

kkt_return_t KKT_CONNECT_REPLY(
        handle_t          handle,           /* IN */
        kern_return_t     status,           /* IN */
        kern_return_t     substatus);       /* 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 */

kkt_return_t KKT_HANDLE_MIGRATE(
        node_name         node,             /* IN */
        unsigned int      old_handle,       /* IN */
        handle_t          new_handle);      /* IN */
```

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

```
void (*channel->deliver)(
        channel_t         channel,          /* IN */
        handle_t          handle,           /* IN */
        struct transmit   *transmit,        /* IN */
        boolean_t         inline);          /* IN */

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

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

```
Node Map routines:
kkt_return_t KKT_ADD_NODE(
        node_map_t        map,              /* IN */
        node_name         node);            /* IN */

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

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

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

RPC routines:

```
kkt_return_t KKT_RPC_INIT(
        unsigned int      channel,          /* IN */
        void              (*rpc_deliver),   /* IN */
        unsigned int      send_handles,     /* IN */
        unsigned int      receive_handles,  /* IN */
        unsigned int      max_send_size);   /* IN */

kkt_return_t KKT_RPC_HANDLE_ALLOC(
        unsigned int      channel,          /* IN */
        handle_t          uhandle,          /* OUT */
        vm_size_t         rpc_buf_size);    /* IN */
```

```
        kkt_return_t KKT_RPC_HANDLE_BUF(
                handle_t            handle,             /* IN */
                void                *userbuff);         /* OUT */

        kkt_return_t KKT_RPC_HANDLE_FREE(
                handle_t            handle);            /* IN */
        kkt_return_t KKT_RPC(
                node_name           node,               /* IN */
                handle_t            handle);            /* IN */

Callback support routines:
        kkt_return_t (*chan->rpc_deliver)(
                channel_t           channel,            /* IN */
                rpc_buf_t           rpc_buffer);        /* 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 */
```