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 (which are prefixed with "KKT" for KERNEL to KERNEL Transport), 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. xxx

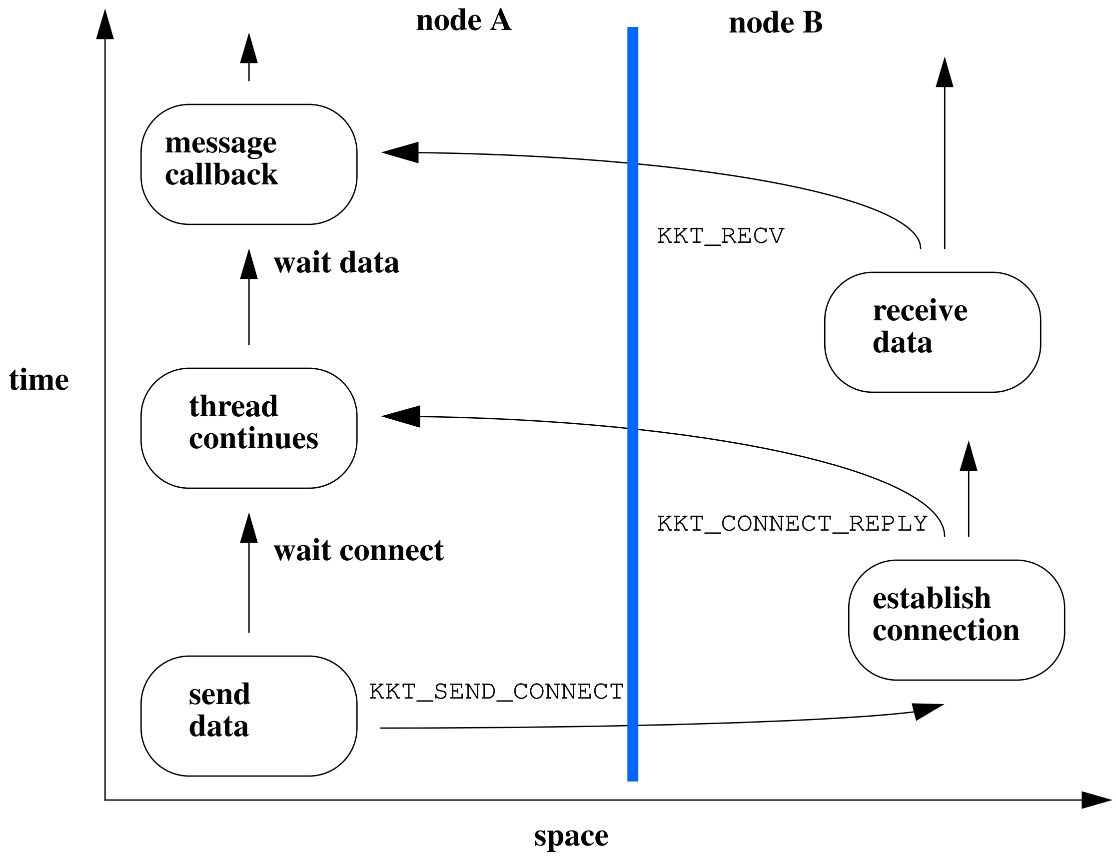


Figure 1: KKT interactions: time vs. space

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.

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.

node — Used to define the destination node.

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

data — The physical address of the data to be sent.

msg_callback — A simple vector provided to the send operation that will facilitate cleaning up any outstanding resources used for this operation.

```
typedef callback {
        void            (*callback)();
        unsigned int    arguments[3];
} *msg_callback_t;
```

At the completion of every send operation and after data has been transferred, the callback routine will be invoked. The sending thread will set the structure up. The callback routine is assumed to execute in interrupt context.

wait A boolean value to indicate whether the calling thread will block waiting for a resource to become available.

endpoint Specifies a high level destination identifier which is opaque to the transport. An *endpoint* is of type *endpoint_t* is a minimum of 8 bytes long. Examples of an `endpoint` would be a `NORMA` UID or a remote memory object identifier.

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_namenode;
} *handle_info_t;
```

2.2 KKT_CHANNEL_INIT

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

This routine is used to install the proper upcalls for a channel, and to define the resources the channel will manage. 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.

receive_handles specifies the number of handles available for accepting remote connections. This is the number reserved exclusively for receiving incoming connections. If the local node transport does not have a handle available to receive a connection, the transport may choose to queue the request or to refuse the connection.

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

The **deliver* upcall identifies the main entry point into the system from the transport on a specific channel, initiated as the receiver side operation in response to a **KKT_SEND_CONNECT** on the sending node. This upcall routine is 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             receive_handles;
        queue_t             send_handles;
}
```

2.2.2 Return Value

```
KERN_SUCCESS
KERN_RESORCE_SHORTAGE       /* could not allocate resources */
KERN_ERROR                  /* channel already in use */
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 */
        boolean_t           wait);              /* 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.

Finally, 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.

2.3.1 The Transport Data Structure

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

```
        struct transport_handle {
                channel_t           channel;
                boolean_t           connected;
                transport_handle_thandle;
                rdma_token_t        token;
                buffer_t            *buffer;
                queue_t             callback_list;
        }
```

2.3.2 Return Value

```
KKT_SUCCESS
KKT_ERROR                          /* unusual failure */
```

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_SEND** 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_ARGUMENT               /* 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 */
                vm_offset_t         data,             /* IN */
                vm_size_t           size,             /* IN */
                msg_callback_t      msg_callback,     /* 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 *data*. If the message is too large or complex to send in a single call, the **KKT_SEND** 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.

data provided in this call is a physical address which can be used by the transport directly. For most architectures, this limits the size of the transfer to one physical page, but this is not a requirement. The *size* parameter describes the size of the data.

If the callback is a non-null function pointer it will be invoked when the data has physically been transferred off node. The callback function will be invoked with the handle, a return code, and the user supplied parameters. A null *msg_callback* parameter is valid. The callback will not be invoked if the transport returns `KKT_DATA_SENT`, indicating the data has been transferred to the remote node.

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.

The value returned by this function is a transport dependent value.

2.5.1 Resource Consumption

In this transport model, the sender provides the physical 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 Value

```
        KKT_SUCCESS        /* kmsg enqueued - MACH_MSG_SUCCESS */

Transport errors:
        KKT_DATA_SENT      /* data sent to remote node; no callback */
        KKT_NODE_DOWN      /* destination node is not responding */
        KKT_ERROR          /* something BAD happened */
```

2.6 KKT_CONNECT_REPLY

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

Respond to a **KKT_SEND_CONNECT** request. The sender on the remote node will block until the receiver responds with this call. A return value of *kr* will be returned to the remote node.

2.6.1 Return Value

```
KKT_SUCCESS
KKT_INVALID                /* handle argument is invalid */
```

2.7 KKT_SEND

```
kkt_return_t KKT_SEND(
        handle_t          handle,            /* IN */
        vm_offset_t       data,              /* IN */
        vm_size_t         size,              /* IN */
        msg_callback_t    msg_callback);     /* IN */
```

The caller of this routine presents wired memory to the transport to be sent to the remote node. In the typical sequence of events, the callback mechanism will be invoked as data is sent, allowing the sender to free wired pages and to send more. The underlying transport mechanism may refuse to accept another transfer request if some internal maximum is reached.

Information such as the destination node and higher level connection identifiers have already been established by **KKT_SEND_CONNECT** and are unnecessary.

The callback mechanism is identical to **KKT_SEND_CONNECT**. The callback will not be invoked if the function returns KKT_DATA_SENT, indicating data has been transferred to the remote node

2.7.1 Return Value

```
KKT_MSG_SUCCESS   /* data enqueued - MACH_MSG_SUCCESS */

Transport errors:
KKT_DATA_SENT     /* data sent to remote node; no callback */
KKT_NODE_DOWN     /* destination node is not responding */
KKT_ERROR         /* something BAD happened */
KKT_NO_SPACE      /* transport cannot accept request */
KKT_INVALID       /* handle is invalid */
```

2.8 KKT_RECV

```
kkt_return_t KKT_RECV(
        handle_t          handle,            /* IN */
        vm_offset_t       data,              /* IN */
        vm_size_t         size,              /* IN */
        msg_callback_t    msg_callback);     /* IN */
```

A message is received by the transport which calls the **deliver* routine specified in the channel structure, which in turn will awaken a waiting thread. The awakened thread will then receive the data from the transport and begin processing it. Must be executed in thread context.

data is the physical address of an available, wired page. Memory is always pre-allocated by the receiver. The size of the data is first obtained from the **deliver* upcall; subsequent data sizes must be determined from the data.

A *msg_callback* is specified to allow the receive side of the operation to reserve space in advance of it being sent. The callback is not invoked until all of the data associated with the receive operation has been received. A null *msg_callback* value is legitimate if no callback is desired.

Notice that the sending node may take an arbitrary amount of time to send the data, especially if the data has been paged out. This is deemed acceptable as we are dealing with system objects at this point, which will respond in a timely manner.

KKT_RECV will not block. The callback routine is the notification that the data has been received.

A return value of KKT_DATA_RECEIVED indicates that the data has been returned into *data* and the callback routine specified in msg_callback will not be invoked. This facilitates transports that are able to buffer up data in advance of a request and allows the upper layer to avoid an unnecessary context switch. KKT_DATA_RECEIVED is a superset of KKT_SUCCESS.

2.8.1 Return Value

```
        KKT_SUCCESS

Transport errors
        KKT_DATA_RECEIVED /* data xfer'd; no callback forthcoming */
        KKT_NODE_DOWN     /* destination node is not responding */
        KKT_ERROR         /* something BAD happened */
        KKT_INVALID_HANDLE/* handle is invalid */
```

2.9 KKT_MSG_CNTL

```
kkt_return_t KKT_MSG_CNTL(
        channel_t          channel,         /* IN */
        node_name          node,            /* IN */
        unsigned int       type);           /* IN/OUT */
```

This is the moral equivalent of ioctl(). It has no defined behavior at present, but is anticipated to perform transport specific operations.

2.10 Dynamically Adding/Deleting Nodes

Does the Transport figure out valid nodes & how to get at them, or do the above layers need to help out?

2.11 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 also havel all pending callbacks executed with *reason* supplied to them.

Upon completion, both handles are unconnected and using them will result in KKT_UNCONNECTED error.

2.11.1 Return Value

```
        KKT_SUCCESS
```

2.12 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.

2.12.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_PARAMETER
```

2.13 (channel->deliver)()

```
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 */
        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 will 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.

If the transport was able to send the data in the message, *inline* will be TRUE. This indicates that a **KKT_RECV** is not needed to receive the data. Data provided in this manner is contained in a transport resource and must not be deallocated. This data is persistent until **KKT_CONNECT_REPLY** is issued, at which time the pointer will be undefined. *data* is a kernel virtual address and does not require any further translation by the kernel.

Enough information is forwarded in this call to allow the upper layer to allocate resources and receive arbitrary amounts of data from the sender, throttled by the receiving node.

2.13.1 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**.

2.14 (*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.

2.14.1 Return Value

```
KKT_SUCCESS
KKT_RESOURCE_SHORTAGE
```

2.15 (*channel->free)()

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

NORMA routine for returning memory allocated to the transport layer. Will be invoked by the transport layer. There can be one *free* routine per channel, to match the channel *malloc* routine.

2.16 User Defined Callback

```
void    (*callback)(
        kkt_return_t       kr,               /* IN */
        handle_t           handle,           /* IN */
        unsigned int       arg1,             /* IN */
        unsigned int       arg2,             /* IN */
        unsigned int       arg3)             /* 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 first argument, *kr*, is always defined to be the transport return code. This value will indicate the success or failure of the transport operation.

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

And finally, the arguments supplied by the user in the callback structure are opaque to the transport and are supplied at this point. These values will uniquely identify the operation in question.

2.17 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_RECV**. 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_MSG_RECV**.

Once the send data has been wired down, **KKT_SEND** 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.

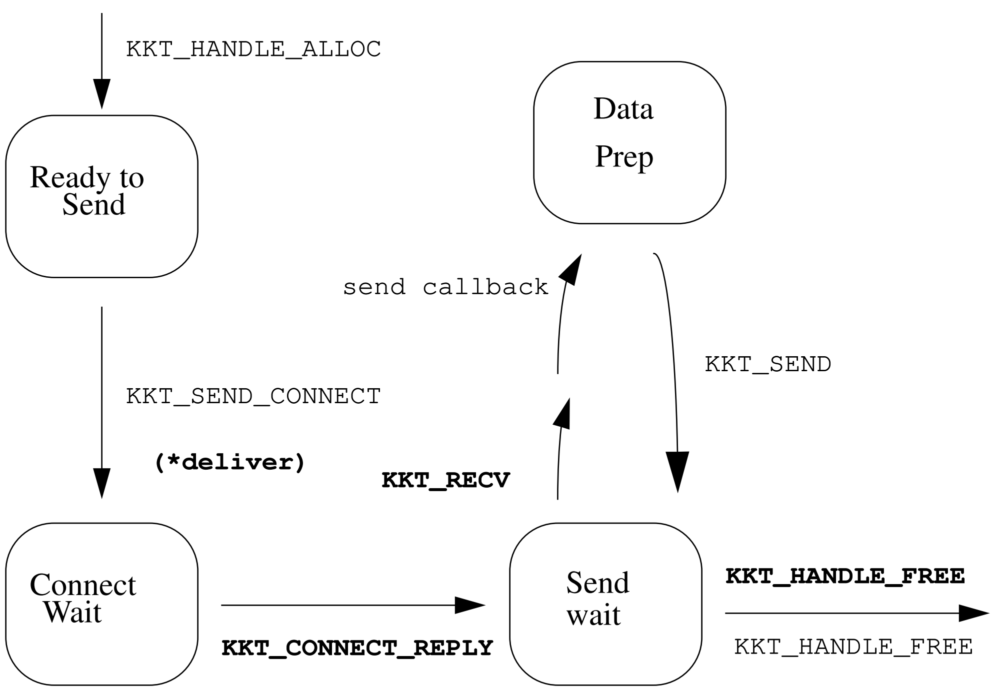


Figure 2: State transitions (Receive side in bold)

2.18 KKT_MAP_ALLOC

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

Used to obtain a node map from the transport when a node callback needs to be established. This facilitates a general purpose node-to-node callback mechanism that is necessary during resource shortages. This mechanism is intended to allow the receiver to contact the sender when resources become available, lessening dependence on timeout based retries.

By convention, individual subsystems utilizing kernel to kernel transport mechanisms will allocate global node maps at initialization time when memory resources are plentiful. This call is not guaranteed to succeed when resources are scarce.

A *map* structure is used to maintain a list of nodes in a manner that is meaningful to the transport, making the data opaque to higher levels. The actual representation of nodes within a node map is left to the transport. All of the transport interfaces that deal with a *map* are local to the node and do not negotiate in any way with remote nodes.

2.18.1 Return Value

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

2.19 KKT_MAP_FREE

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

Release the *map* back to the transport. Although this is typically done when the map is empty, this is not a requirement.

2.19.1 Return Value

```
KKT_SUCCESS
KKT_INVALID_PARAMETER /* bogus map argument */
```

2.20 KKT_ADD_NODE

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

2.20.1 Return Value

```
KKT_SUCCESS
KKT_RESOURCE_SHORTAGE     /* insufficient memory to allocate */
KKT_NODE_PRESENT          /* node already in map */
```

2.21 KKT_REMOVE_NODE

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

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.

2.21.1 Return Value

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

2.22 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.

CHAPTER 3 RPC Interface

3.1 Overview

This section describes interfaces to microkernel-internal facilities that transport control and data across node boundaries. The facilities are separated into two complimentary functional units that, when used together, provide a flexible, robust, and high performance data transport mechanism for multicomputer systems.

One functional unit of the transport interface is designed to support low-latency exchange of control information between nodes. Implicit policy choices by the interface are deliberately kept to a minimum, as are the feature set and abstractions exported by the interface. Reasonable constraints on the interface enable the underlying implementation to be simple and fast.

The other functional unit of the transport interface is designed to support high-bandwidth transfer of data between nodes. The interface presents the abstraction of full-duplex connected transfer engines, each providing for transfer rate mismatches on either end of the connection. Each endpoint may be retargeted to any node within the multicomputer system.

3.2 Microkernel RPC

The *RPC* (Remote Procedure Call) facility is an event driven, reliable, lightweight protocol used to exchange untyped procedure call and return value data between nodes within a multicomputer. The interfaces are designed to be used to support low-latency exchange of control information between nodes. Using the RPC interfaces, microkernels can arrange to send an RPC request to a node, be notified of the arrival of an RPC request from a node, invoke a local service routine with data supplied by the requesting node, respond to the requesting node with a reply containing untyped return data, and be notified of the arrival of an RPC reply.

Notifications of RPC events are accomplished by interrupt-level callbacks. Polling interfaces to detect RPC events and local changes of RPC state are available and may be used alone or in combination with the callback mechanisms.

Sending and receiving RPC requests and replies are asynchronous events with respect to the caller. The RPC interfaces are used to schedule the exchange of RPC requests and replies between nodes involved in an RPC. Mechanisms internal to the RPC facility synchronize and flow-control the actual exchange of RPC requests and replies. The RPC facility guarantees reliable, time-ordered delivery of RPC data (both requests and replies) for all RPC exchanges.

The policies used for composing procedure call arguments, decomposing return values, RPC event notification, and RPC service dispatch are not the responsibility of the RPC facilities. It is the responsibility of software calling the RPC interfaces to select one or more polices, specifically for

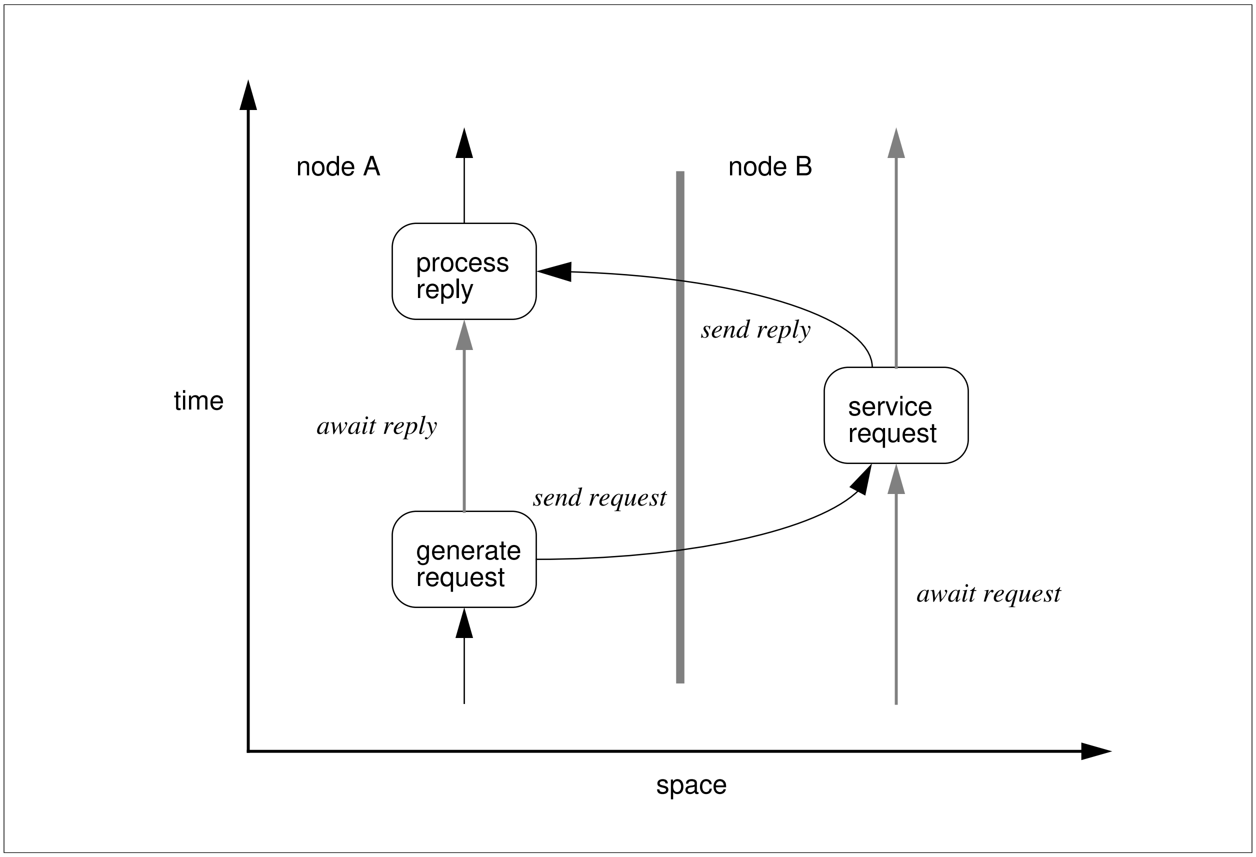
RPC event notifications and RPC service dispatch. For example, software using the RPC facilities may choose to poll for an RPC reply from a node after making one kind of request, and use the callback notification mechanisms for another RPC request.

RPC request and reply data are transported between nodes uninterpreted by the RPC mechanisms. Memory space for RPC requests must be allocated from the RPC facility. Data associated with the RPC request must be transferred to that buffer prior to scheduling the transmission of an RPC request. A pointer to the data for an RPC request that has arrived can be retrieved for use by the RPC service routine. RPC replies are sent from the same RPC request buffer. RPC reply data must be transferred to the request buffer prior to scheduling the transmission of an RPC reply.

Dispatch of a specific RPC service from an RPC request buffer is a function of the software using the RPC facilities. For example, the first word of the RPC request buffer might be used as an index into a table of function pointers to be indexed and called when an RPC request arrives.

An RPC exchange between microkernels within a multicomputer is assumed to be a trusted operation, requiring no additional authentication by microkernels using the RPC facilities. Any microkernel may make an RPC request to any other microkernel within the multicomputer after the RPC facilities have been initialized during system initialization.

The following illustration depicts the time and space interactions between two nodes performing an RPC:



RPC interactions: time vs. space

Each arc in the diagram above that crosses the node boundary represents a rendezvous between sender and receiver. The *send request* arc contains data for the RPC request from the requesting node, and the *send reply* arc contains data for the RPC reply from the node servicing the RPC request. The data that is transferred in either direction is also referred to occasionally as the *RPC payload*.

Software using the RPC interfaces must guarantee that the full cycle of request/service/reply must guarantee to complete; the interface defines no mechanisms for aborting or restarting any arc of the RPC interaction between an nodes.

3.0.1 RPC Abstractions

The RPC facility exports three abstractions that are described within this section.

3.0.1.1 RPC Handles

An RPC resource is identified by a *handle*. The total number of RPC handles available for use is configured during system initialization. Additional RPC handles cannot be configured after system initialization. An RPC handle is defined to be of type **`rpc_handle_t`**.

RPC handles must be explicitly allocated prior to the use of an RPC resource. RPC handles may be used any number of times, including zero, prior to being explicitly deallocated. RPC handles must be allocated from thread context within the microkernel. Allocation of an RPC handle from interrupt context may produce undefined behavior. Deallocation of an RPC handle representing an RPC resource with pending operations may produce undefined behavior. RPC handles may be safely deallocated from either thread or interrupt contexts.

No additional multiprocessor locks are required to be held to allocate or free an RPC handle.

3.0.1.2 RPC Groups

During system initialization, each RPC handle is configured into exactly one RPC handle *group*. The total number of RPC handle groups available for use is configured during system initialization. Additional RPC handle groups cannot be configured after system initialization. An RPC handle group is defined to be of type **`rpc_group_t`**.

An RPC handle cannot be placed in more than one group. RPC handles cannot be moved between groups. Only those RPC handles that have been configured into a group may be allocated. An RPC handle group may contain any number of RPC handles including zero. The maximum number of RPC handles within a group is the total number of RPC handles configured.

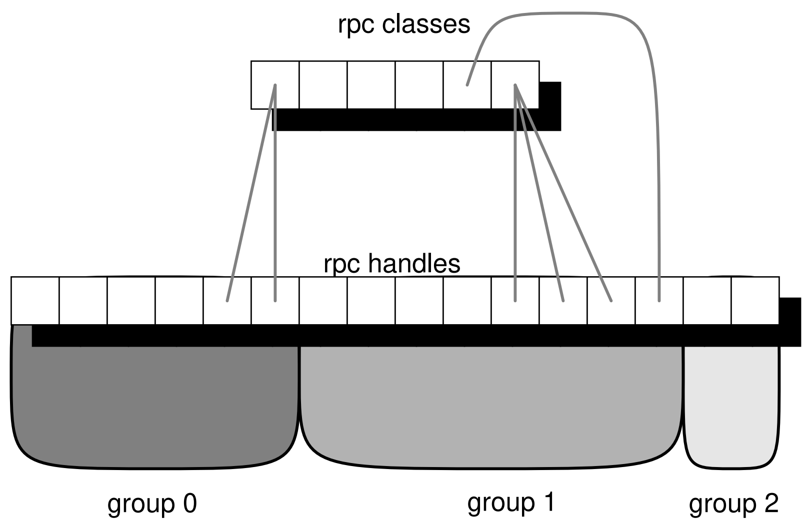
An RPC handle group is supplied when allocating an RPC handle. Deallocation of an RPC handle returns the handle to the group from which it was allocated.

3.0.1.3 RPC Service Classes

RPC requests are addressed to a node within the multicomputer and to a service *class*. The total number of service classes available for use is configured during system initialization. Additional RPC service classes cannot be allocated after system initialization. An RPC service class is defined to be of type **`rpc_class_t`**.

To export an RPC class of service, an RPC handle and an RPC class must be supplied. An RPC handle cannot be within more than one RPC class concurrently. An RPC class may have any number of RPC handles (up to the total number of RPC handles configured) awaiting RPC requests.

The following illustration depicts 16 RPC handles, 3 RPC groups, and 6 RPC classes:



In the illustration above, six RPC handles have been placed in group 0, eight RPC handles have been placed in group 1, and two RPC handles have been placed in group 2.

Two RPC handles within group 0 are exporting service in class 0, three RPC handles within group 1 are exporting service in class 5, and one RPC handle within group 1 is exporting service in class 4.

RPC requests to a class that have no RPC handle exporting service are internally deferred until at least one handle exports service within that class. For example, in the illustration above, RPC requests to class 1 will be deferred until some local handle exports service in class 1.

Multiple RPC requests to a class compete for access to an RPC service. Should there be more RPC requests pending than a class can concurrently satisfy, internal flow-control mechanisms transparently provide fair access to all requests in some random order. Time ordering of requests is implicitly guaranteed to be preserved between RPC cycles that originate from the same handle on the same node.

All RPC operations support loopback operation; any transmission operation that can be directed to a remote node may also be directed to the caller's node.

3.0.2 RPC Interfaces

This section details all available RPC interfaces. All RPC type definitions are defined in the microkernel header file **<rpc_rdma/rpc.h>**.

3.0.2.1 RPC initialization

Each of the following are called during system initialization.

rpc_init

```
rpc_return_t rpc_init(nhandles, ngroups, nclasses)
int nhandles, ngroups, nclasses;
```

Initialize and configure the RPC facilities to use the supplied numbers of RPC handles, groups, and classes. This routine must be called exactly once during system initialization. A succesful call returns `RPC_SUCCESS`. If insufficient resources are available, `RPC_SHORTAGE` is returned.

rpc_group_alloc

```
rpc_return_t rpc_group_alloc(group, nhandles)
rpc_group_t group;
int nhandles;
```

Associate the supplied number of RPC handles with an RPC group. This call must be made exactly once for each group configured during system initialization. If successful, the call returns `RPC_SUCCESS`. `RPC_INVALID_GROUP` is returned if the group supplied is not a valid group. `RPC_HANDLE_SHORTAGE` is returned if an insufficient number of handles remain to configure into the group.

3.0.2.2 RPC handle management

Each of the following routines are used for the management of RPC handles and per-handle resources.

rpc_handle_alloc

```
rpc_handle_t rpc_handle_alloc(group, wait, argsize)
rpc_group_t group;
boolean_t wait;
int argsize;
```

Allocate an RPC handle from an RPC group. The `wait` parameter indicates that the caller is willing to be blocked if no RPC handles are currently available within the group. The `argsize` parameter indicates the maximum size in bytes of any RPC arguments (request or reply) that will be processed on the handle returned.

If at least one RPC handle is free within the group, the call returns an RPC handle. If no RPC handles are available within the group and wait is `TRUE`, the caller will be blocked until a handle becomes available. If `wait` is `FALSE` and no RPC handles are available for use within the group, the call returns the special handle `RPC_GROUP_EMPTY` with no handle allocated.

rpc_handle_free

```
void rpc_handle_free(handle)
rpc_handle_t handle;
```

Return an RPC handle to the group of RPC handles from which it was allocated. Memory for RPC arguments is implicitly deallocated with this call and should not be dereferenced after this call returns. This call should not be made while any RPC operations associated with the supplied handle are still in progress.

rpc_arguments

```
rpc_args_t *rpc_arguments(handle)
rpc_handle_t handle;
```

Return a pointer to a bufffer associated with the supplied RPC handle that is suitable for use for RPC requests and replies. This buffer is also referred to as the RPC payload or the RPC payload bay.

3.0.2.3 RPC operations

Each of the following routines schedules an RPC operation to occur.

rpc_send_request

```
void rpc_send_request(node, class, handle, callback, callarg)
nodeid_t node;
rpc_class_t class;
rpc_handle_t handle;
void (*callback)();
rpc_notify_t callarg;
```

Initiate an RPC request to the class of receivers on the specified node, using the arguments associated with the handle. The routine returns to the caller as soon as the send request has been scheduled for transmission.

If the callback is a non-null function pointer, the callback function will be called, passing `handle` as the first argument and `callarg` as the second, when the corresponding RPC reply arrives. The callback function cannot assume thread execution context.

rpc_recv_request

```
void rpc_recv_request(class, handle, callback, callarg)
rpc_class_t class;
rpc_handle_t handle;
void (*callback)();
rpc_notify_t callarg;
```

This routine associates the handle with a class of service for the reception of RPC requests, and expresses the calling node's interest in receiving RPC requests. The routine returns to the caller as soon as the receive request has been scheduled for reception.

If callback is a non-null function pointer, the callback function will be called, passing `handle` as the first argument and `callarg` as the second, when an RPC request arrives. The callback function cannot assume thread execution context.

rpc_send_reply

```
void rpc_send_reply(handle, callback, callarg)
rpc_handle_t handle;
void (*callback)();
rpc_notify_t callarg;
```

Initiates an RPC reply associated with the supplied handle. The contents of the buffer associated with the handle will be scheduled for transmission to the original source node of the RPC request. The call returns as soon as the reply has been scheduled for transmission.

If callback is a non-null function pointer, the callback function will be called, passing `handle` as the first argument and `callarg` as the second, when the RPC reply has been logically departed from the node (meaning the handle may be used again or freed). The callback function cannot assume thread execution context.

rpc_send_reply_recv

```
void rpc_send_reply_recv(class, handle, callback, callarg)
rpc_class_t class;
rpc_handle_t handle;
void (*callback)();
rpc_notify_t callarg;
```

Initiates an RPC reply associated with the supplied handle and initiate another RPC receive request. This routine is equivalent to the combined function of the following two calls:

```
rpc_send_reply(handle, 0, 0);
rpc_recv_request(class, handle, callback, callarg);
```

`rpc_send_reply()` callbacks are not accessible and will not be generated through this interface.

3.0.2.4 RPC polling interfaces

Each of the following may be used to poll for changes in local state associated with an RPC handle.

rpc_send_ready

```
boolean_t rpc_send_ready(handle)
rpc_handle_t handle;
```

Return `TRUE` if the reply from an RPC request has arrived, otherwise return `FALSE`.
If a reply arrival callback notification had been requested, `rpc_send_ready()` will return `TRUE` if called from inside the callback function.

rpc_recv_ready

```
boolean_t rpc_recv_ready(handle)
rpc_handle_t handle;
```

Return `TRUE` if an RPC request has arrived, otherwise return `FALSE`.
If a request arrival callback notification had been requested, `rpc_recv_ready()` will return `TRUE` if called by the callback function.

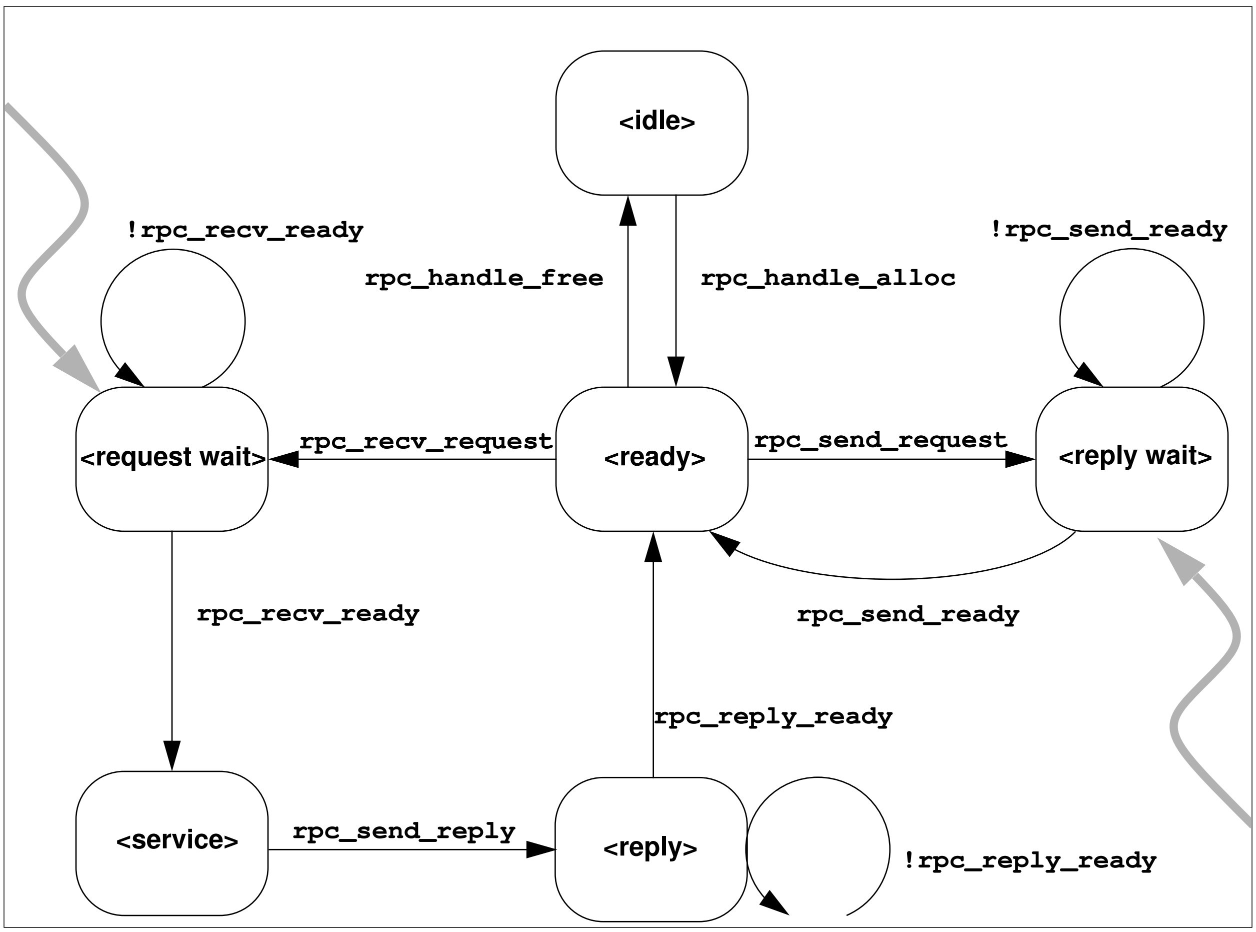
rpc_reply_ready

```
boolean_t rpc_reply_ready(handle)
rpc_handle_t handle;
```

Return `TRUE` if an RPC reply has departed, otherwise return `FALSE`.
If a reply departure callback notification had been requested, `rpc_reply_ready()` will return `TRUE` if called by the callback function.

3.0.3 RPC State Transitions

This section contains an illustration of the RPC state machine.



If a callback is requested for `rpc_recv_request()`, the callback will be invoked when `rpc_recv_ready()` returns `TRUE`.

If a callback is requested for `rpc_send_request()`, the callback will be invoked when `rpc_send_ready()` returns `TRUE`.

If a callback is requested for `rpc_send_reply()`, the callback will be invoked when `rpc_reply_ready()` returns `TRUE`.

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 */
        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 */
        vm_offset_t         data,               /* IN */
        vm_size_t           size,               /* IN */
        msg_callback_t      msg_callback,       /* 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_SEND(
        handle_t            handle,             /* IN */
        vm_offset_t         data,               /* IN */
        vm_size_t           size,               /* IN */
        msg_callback_t      msg_callback);      /* IN */

kkt_return_t KKT_RECV(
        handle_t            handle,             /* IN */
        vm_offset_t         data,               /* IN */
        vm_size_t           size,               /* IN */
```

```
        msg_callback_t    msg_callback);    /* IN */

kkt_return_t KKT_MSG_CNTL(
        channel_t         channel,          /* IN */
        node_name         node,             /* IN */
        unsigned int      type);            /* 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 */
        boolean_t         inline);          /* 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 */
```

APPENDIX B

Sample RDMA Implementation

B.1 KKT_HANDLE_ALLOC

```
t = (obtain a local handle structure from the channel);
if (t not a valid handle)
        return (KKT_ERROR);
t->channel = (channel structure address);
*handle = t;

return (KKT_SUCCESS);
```

B.2 KKT_HANDLE_FREE

```
rdma_handle_free(t->rdma_handle);
/*
 * Clean up rest of transport handle structure...
 */
(Free the handle structure back to the local pool);
```

B.3 KKT_CHANNEL_INIT

```
/*
 * Map channels to groups & classes. Assumes the existence
 * of an array that is filled out with the necessary goodies.
 */
for (i = 0; i < NGROUPS; i++) {
        rpc_group_alloc(i, channel_init[i].handles);
        channel = channel_array[i];
        channel->deliver = deliver;
        channel->malloc = malloc;
        channel->free = free;
        channel->group = channel_init[i].groups;
        /*
         * Set up the receivers from the number of
         * receive handles we allocated.
         */
        for (recvs = 0; recvs < channel_init[i].recv_hndls; i++)
                handle = rpc_handle_alloc(channel->group,
                                          TRUE,
                                          ARGMAX);

                rcp_recv_request( i,
                                  handle,
                                  channel->deliver);
```

```
}
```

B.4 KKT_SEND_CONNECT

```
        t->rpc_handle = rpc_handle_alloc(channel, wait, 32);
        t->token = rdma_token(t->dma_handle);
        args = rpc_arguments(t->rpc_handle);
        args[0] = endpoint;
        args[2] = size;
        args[3] = t->token;
        rpc_send_request(node, t->chan, rcp_handle, completion_routine);
        /*
         * Wait for remote node to respond with kkt_connect_reply
        */
        sleep()...

        t->connect = TRUE;
        /*
         * don't hang on to scarce resource
        */
        rpc_handle_free(t->rpc_handle);
        t->rpc_handle = NULL;
        /*
         * give the transport the data
         */
        rdma_send(t->rdma_handle,
                  data,
                  size,
                  transport_callback,
                  opaque);
...message is dequeued...
```

B.5 KKT_CONNECT_REPLY

```
handle->rpc_buffer[0] = kr;
return rpc_send_supply(handle);
```

B.6 KKT_SEND

```
/*
 * An RDMA token was forwarded in the request and put in
 * the transport_handle structure.
 */
rdma_send(t->rdma_handle,
          data,
          size,
          transport_callback,
          opaque);
```

B.7 KKT_RECV

```
rdma_recv(t->rdma_handle
           data,
           size,
```

```
            transport_callback,
            opaque);

return (KERN_SUCCESS);
```

B.8 (channel->deliver)()

```
t = (allocate local handle);
/*
 * The local handle for receiving must be available, as are
 * the necessary rpc_handle and rdma_handle. If this blocks
 * or fails we have problems...
 */
t->dma_handle = rdma_handle_alloc(channel, FALSE);
t->token = rpc.argument.xxx;
rdma_connect(t->rdma_token, t->rdma_handle);
t->connect = TRUE;

(channel->deliver)(rpc.argument.xxx,
                   rpc.argument.xxx,
                   handle,
                   rpc.argument.flush,
                   (rpc.argument+nnn));
```