Network Server Design

MACH Networking Group

August 31, 1989

1. Introduction

The network server is responsible for extending the local MACH Inter-Process Communication abstraction over the network which interconnects MACH hosts. We have designed and implemented a new version of the network server in a way in which we hope makes it more efficient, easier to understand, more modular and easier to extend. In particular we intend to use this new network server to experiment with various ideas in the area of distributed systems networking, such as:

- evaluating the merits of various protocols for network interprocess communication — in particular, we want to examine:
  - connection-oriented versus connectionless protocols,
  - request-response protocols for remote procedure calls, and
  - the use of special-purpose protocols depending on the size or the destination of the data to be transported;
- evaluating various models for the distribution of network functions between the operating system kernel and user processes, and how these two components should interact;
- security in the network environment;
- multicast groups, and associated delivery problems; and
- copy-on-reference operations over a network.

This document describes the design of the network server and details the interfaces to the modules that are used to implement it. It is intended for implementers rather than users. This document reflects the current state of the implementation, and does not constitute a commitment for future developments.

In the next section the overall design of the network server is explained. The subsequent sections describe the structure of individual modules.

2. Overall Structure

2.1. General Operation

The set of all network servers on a network cooperate to provide IPC (*Inter-Process Communication*) between processes on different hosts on that network. They achieve that cooperation by exchanging *Network Server Messages*. Some of these messages contain the data of IPC messages that are to be transported across the network, while others are used by the network servers to communicate information about the status of the operations in progress, and to maintain a consistent view of the location of the ports used in the network environment. A small *dispatcher header* is used to distinguish between these various network server messages.

The network servers maintain a space of *Network Ports* and each network server maintains a mapping between ports local to its host and network ports. Each network port is represented by a *Network Port Identifier* which contains information to locate the receiver and owner for the network port and information which allows the security of the Mach port abstraction to be maintained in the network environment. (See [2] for further details.)

2.2. Control Structure

The server is structured as a collection of threads sharing the same address space. Each thread is used to perform one specific task asynchronously with the other threads. Typically, there is one thread for each "wait point" in the system. Wait points are most often waits for IPC messages. The thread is awakened whenever something happens at this wait point, performs all operations pertinent to the current event, and goes back to sleep. Should the service of an event require a further wait, another thread is signalled and asked to continue operations after that wait. In general, one thread should not have more than one wait point. Appropriate locking facilities are used where threads must share access to data structures. There are a fixed number of threads in the system:

Timer Used by all the other threads whenever they need to schedule some action to take place at some given time in the future. The most common use of this facility is to schedule packet retransmission.

IPC Send Waits for messages to arrive on any of the ports that are local representatives of remote network ports. Upon reception of such a message, performs the necessary translations and initiates transmission on the network. This thread does not wait for the transmission to complete; subsequent actions will be taken by the timer thread or by a transport receive thread.

IPC Re-Send Is awakened when some other thread or some external condition indicates that a message previously handled by the IPC send thread should be re-sent either to the original or to another destination. It essentially performs the same functions as the IPC send thread, but might take advantage of some work already done by the former. This thread is called into action when a port is not found at the expected location, when a remote port is blocked, or when network errors occur.

Transport Receive (One or more per each transport protocol.) Waits for packets to arrive from the network interface. Processes these packets, perhaps assembling multiple packets into contiguous data and perhaps matching incoming packets with previously queued outgoing transmissions. Passes IPC and other network server data received from remote network servers on to higher level modules for further processing. As suggested above, this thread may use the services of the timer thread to schedule retransmissions of packets and to determine when a transmission should be aborted.

Notify Messages Waits for notify messages from the kernel to arrive on the server's notify port. These messages indicate changes in the status of local ports (death, movements of rights or local IPC message accepted). Takes appropriate action in each case to update the server's records and may destroy a port, transfer access rights to a network port to a remote network server or signal the IPC re-send thread to retry a message transmission.

Name Service Handles requests from other tasks for network name service. Allows names to be looked up on this host, on a specific remote host or by broadcasting to all hosts.

Key Management Handles messages from the external *Key Distribution Server*. These messages tell the network server to use new encryption keys for remote hosts.

Other Services A number of other threads may be used to provide other services included with the network server, such as monitoring and specialized name services. In each case, the thread is waiting for requests on a particular service port.

In addition, if the operations to be performed upon reception of a network message are too long, the Transport Receive threads may hand the message to other specialized processing threads, in order to remain available for new network messages. Note that it is impractical to allocate one thread to each IPC message in transit due to resource limitations.

2.3. Code Structure

The code is distributed between several modules, each pertaining to some specific set of related operations or the management of some data structure. The main modules are:

Port Records Operations for the handling of port records. The main data structure used is a database of *port records*, which maintains a mapping between local ports and network port identifiers, as well as keeping general status information on the ports themselves. Almost every module uses this database to obtain information, and a number of modules modify the information to reflect new situations that they have detected.

Transport Protocols Provide the complete transport mechanism for a block of data of arbitrary size over the network. There are several such modules implementing several protocols; some guarantee reliable delivery, others don't. Clients of the transport protocols transmit messages by making a function call into the transport module; for all incoming messages a transport protocol calls a dispatcher function to deliver the message.

Dispatcher Dispatches incoming network messages assembled by the transport protocols to the appropriate module according the dispatcher type contained in the network message header. The handler functions are directly called by the dispatcher.

IPC Message Handling Provides operations for receiving local IPC messages and handing them to the transport layer, and receiving messages from the transport layer and handing them to the local user processes. To do this it must translate IPC messages to and from a format appropriate for transmission on the network. In particular it must translate ports, identify out-of-line data and, if necessary perform byte-swapping (this corresponds to the ISO presentation layer roughly). It also handles the blocking mechanism for network ports.

Port Operations Provides all port translation functions for the IPC module. Also handles transfers and deletion of port access rights due to the reception of notify messages from the kernel and from remote network servers.

Port Checkups Periodically verifies the information maintained in the local port records by consulting other network servers.

Port Search Implements the sequence of operations needed to update the information kept about a remote port when it is believed that the current information is incorrect. The checkups module calls this module when it finds an inconsistency in the information maintained about a port.

Key Management Maintains a mapping between remote hosts and the keys that should be use to encrypt secure messages. Also responsible for interfacing with the key distribution server to obtain new keys for remote hosts.

Crypt Provides functions to encrypt and decrypt network messages.

Network Name Service Provides a simple, host-directed network name look up service.

Memory Management Provides allocation and deallocation functions for all memory objects used in the network server.

Read/Write Lock Provides functions implementing read/write locks with multiple users.

Locked Queue Operations on shared (i.e. locked) queues of objects.

Timer Service Allows other modules to schedule actions after a specified interval.

Other, miscellaneous modules provide for the generation of unique identifiers, the initialization of the network server, the actual transmission of datagrams over the network, and other ancillary functions.

2.4. Data Representation

There are a number of circumstances during the processing of IPC messages when the server has to manipulate large amounts of data. In order to minimize overhead, large blocks of data are represented in a special format designed to limit the need for data copying and space allocation.

Any conceptually contiguous block of data may be stored internally as a set of separate segments of any size, and a special *sbuf* (*Segmented Buffer*) structure is used to identify the segments constituting one block. No special meaning is attached to how a block is segmented, i.e. the segments do not necessarily represent logical parts of a block of data. The segments reside in shared memory and are accessible by every thread. Data modification and data transfer between modules are accomplished by operations on *sbuf*s (`typedef struct {...} sbuf_t, *sbuf_ptr_t`).

For instance, if some data must be inserted at the head or the tail of some existing buffer, a new segment is allocated and placed at the right location in the *sbuf*. If some data inside a buffer must be modified (possibly changing its length), the *sbuf* is modified so that the target data spans complete segments only, and those segments are replaced in the *sbuf* by new segments containing the modified data.

Maximum efficiency for the allocation and deallocation of space for the segments cannot be attained with a general mechanism, consequently, space management is handled on a case by case basis by the modules that use *sbuf*s and is private to an individual module. The information needed to deallocate the space used by segments of a given *sbuf* is not kept in the *sbuf* itself, but is recorded by the module that created each particular segment. To simplify record-keeping, no segment may be referenced in more than one (public) *sbuf*. Typically, when an *sbuf* is passed between modules, the system provides a call-back procedure to signal the module that created the *sbuf* that it is no longer needed.

Special macros are provided to extract data from an *sbuf*, making its structure transparent to most modules. Special macros are also provided for inserting data, but for performance reasons, they cannot be made entirely transparent.

3. Port Records

3.1. Description

The port record module maintains all the data associated with ports. In particular it enables the network server to map local to network and network to local ports. An individual port record:

```
typedef struct {...} port_rec_t, *port_rec_ptr_t;
```

contains the local port, the network port, status information about the port, several fields used by the IPC module, several fields for security, a reference count and a read/write lock.

A network port identifier has the following structure:

```
typedef struct {
    long        np_uid_high;
    long        np_uid_low;
} np_uid_t;

typedef struct {
    netaddr_t   np_receiver;
    netaddr_t   np_owner;
    np_uid_t    np_puid;
    np_uid_t    np_sid;
} network_port_t, *network_port_ptr_t;
```

where the `np_puid` is the network port's *Public Unique Identifier* and the `np_sid` is the port's *Secret Identifier*.

```
typedef unsigned long netaddr_t;
```

is used to identify all network addresses within the network server.

3.2. Interface

```
boolean_t pr_init()
```

initializes the port records module.

```
void pr_reference(port_rec_ptr)
port_rec_ptr_t port_rec_ptr;
```

increments the reference count for the port record.

```
void pr_release(port_rec_ptr)
port_rec_ptr_t port_rec_ptr;
```

decrements the reference count for the port record, unlocks it and frees all memory associated with it if the reference count becomes zero.

```
void pr_destroy(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

logically destroys a port record by removing it from all tables and deallocating the local port associated with it. The actual space occupied by the port record is not freed until the reference count becomes zero, but this function performs one `pr_release` before exiting.

```
port_rec_ptr_t pr_np_puid_lookup(np_puid)
np_uid_t              np_puid;
```

looks up and locks a port record given a network port's public unique identifier.

```
extern port_rec_ptr_t pr_nportlookup(nport_ptr)
network_port_ptr_t    nport_ptr;
```

looks up and locks a port record given a network port.

```
portrec_ptr_t pr_ntran(nport_ptr)
network_port_ptr_t    nport;
```

looks up and locks a port record given a network port, creating a new port record and allocating a new local port if necessary.

```
extern port_rec_ptr_t pr_lportlookup(lport)
port_t                lport;
```

looks up and locks a port record given a local port.

```
portrec_ptr_t pr_ltran(lport)
port_t                 lport;
```

looks up and locks a port record given a local port, creating a new port record and allocating a new network port if necessary.

```
boolean_t nport_equal(nport_ptr_1,nport_ptr_2)
network_port_ptr_t     nport_ptr_1;
network_port_ptr_t     nport_ptr_2;
```

tests to see if two network ports are equal.

```
void pr_nporttostring(nport_str,nport_ptr)
char                   *nport_str;
network_port_ptr_t     nport_ptr;
```

returns in `nport_str` a printable representation of a network port.

```
lock_queue_t pr_list()
```

returns a list of all the local ports for which there is a port record. (See the section on locked queues for the definition of `lock_queue_t`.)

In general, all functions that return a port record lock that record before returning. Functions that take a port record as an argument require that record to be already locked on entry. The reference count normally reflects the presence of the port record in both the local and global tables, but no additional reference is taken by any of the above functions when returning a locked port record. Note that these functions will block on the lock if the record is already locked.

4. Transport Protocols

4.1. Description

Several transport modules co-exist within the network server, each of them implementing a different protocol. Some protocols under consideration are Delta-t, VMTP, NETBLT, TCP, UDP, and various multicast schemes. It is up to clients of the transport modules to choose the one that satisfies their needs best. All the transport modules deliver messages into the same dispatcher, and are accessed via separate entry points, grouped into a table similar to the Mach device table.

The organization of each protocol module is up to each implementor. Layered and non-layered approaches are both acceptable, as well as the use of external servers, not residing in the same address space as the network server. Implementors are encouraged to use the *sbuf* mechanism to represent and manipulate data internally, and to copy data only at the last level before the network interface. The timer module is available for handling periodic retransmission and other protocol functions needing timeouts.

4.2. Interface

Because of an ongoing revision of the implementation, there are currently two modes of operation for transport modules.

The first mode of operation specifies a simple *send* operation to transmit a message to a given destination. It is used by all modules except the IPC module, to exchange information with the corresponding modules in the network servers at other nodes.

The second mode of operation specifies a *request-response* interaction, in which one node (the *client*) sends a request to another node (the *server*), and then awaits the reception of exactly one response from that server. The transport protocol always supports the transmission of a 32-bit completion code in the response, and may optionally support the transmission of data in that same response. This mode of operation is used by the IPC module for the transmission of IPC message data; it is intended to be the standard for all transport protocols, and all other modules will be converted to use it.

Currently, each specific transport module only implements one or the other mode of operation; and can therefore be used either by the IPC module or by all other modules, but not both. In the future, all modules will be converted to the second mode of operations.

The following table is used to hold all the entry points for all the possible transport modules. Each transport module is assigned a specific number, to be used as an index into this table when invoking it, and to be used to identify it when it delivers a message to the upper level.

```
typedef struct {
    int          (*send)();
    int          (*sendrequest)();
    int          (*sendreply)();
} transport_sw_entry_t;
```

The entries in this table are filled by each transport module when it initializes itself. Any unused entry is set to the special function `transport_no_function`, which simply returns after emitting an error message. The `send` entry is used for the simple *send* interface. The `sendrequest` and `sendreply` entries are used for the *request-response* interface. Any protocol supporting that interface must implement `sendrequest`, but not necessarily `sendreply`, as described above.

In the following descriptions, `xxx_` is used as a generic prefix, to be replaced by the name of each transport module.

```
boolean_t xxx_init()
```

initializes the transport module and places the entry points in the transport switch table.

```
int xxx_send(client_id,trid,data,to,service,crypt_level,cleanup)
int                     client_id;
int                     trid;
sbuf_ptr_t              data;
```

```
netaddr_t                 to;
int                       service;
int                       crypt_level;
int                       (*cleanup)();
```

attempts to send message `data` to destination `to`. `client_id` is an ID used by the client to identify this message. `trid` is a transport-level ID used to identify a current request-response interaction; it should be 0 if not used. `service` is the kind of service required for this message; possible values are:

- `TRSERV_NORMAL`: normal transmission of a single message.
- `TRSERV_IPC`: transmission of an IPC message, "call-back" required. (OBSOLETE)
- `TRSERV_RPC`: transmission part of a request-response at the IPC level. Pairing information required and "call-back" required. Note that the "call-back" can be implicitly be the delivery of the response if the RPC succeeds. (OBSOLETE)
- `TRSERV_URGENT`: transmission of an urgent single message, to be delivered before other non-urgent transmissions in progress if possible.

`crypt_level` determines what kind of encryption should be used to protect the data. Possible levels of encryption include: `CRYPT_DONT_ENCRYPT` and `CRYPT_ENCRYPT`. `cleanup` is a function supplied by the client, to be called when the transmission is complete and the message data is no longer needed. Depending on the service requested, it may indicate a simple local completion, or participate in the "call-back" mechanism. It takes two arguments: the client ID, and a completion code.

`cleanup` returns 0 when all is well. `xxx_send` also returns a completion code to indicate the immediate local result of the call.

```
int xxx_sendrequest(client_id,data,to,crypt_level,reply_proc)
int             client_id;
sbuf_ptr_t      data;
netaddr_t       to;
int             crypt_level;
int             (*reply_proc)();
```

attempts to send a request containing `data` to destination `to`. `client_id` is an ID used by the client to identify this request-response interaction. `crypt_level` determines what kind of encryption should be used to protect the data. Possible levels of encryption include: `CRYPT_DONT_ENCRYPT` and `CRYPT_ENCRYPT`. `reply_proc` is a procedure to be called from the transport module to deliver a response.

`xxx_sendrequest` returns either `TR_SUCCESS` or a specific failure code. In the first case, a request-response interaction is initiated, that will terminate when `reply_proc` is called. The data supplied in the request must be kept valid by the caller for the whole duration of the request-response interaction. The reply procedure is guaranteed to be called exactly once, with the following arguments:

```
void reply_proc(client_id,reply_code,data)
int             client_id;
int             reply_code;
sbuf_ptr_t      data;
```

`client_id` is the ID given as argument to `xxx_sendrequest`. `reply_code` is a completion code supplied either by the transport module (in case of transport errors) or by the server process that handled the request. `data` may contain data supplied by the server in the response, or it may by null. This data is kept valid by the transport module only until `reply_proc` returns.

If `xxx_sendrequest` returns anything other than `TR_SUCCESS`, no request-response interaction is initiated, and `reply_proc` is never called.

The handler on the server side of a request-response interaction is invoked via the dispatcher module (see below). This handler must return a code that determines how the response is sent. If this code is anything other than `DISP_WILL_REPLY`, a response containing that code and no data is immediately sent back by the transport module. If this code is `DISP_WILL_REPLY`, no response is generated by the transport module, and the higher-level module must explicit call `xxx_sendreply` to terminate the request-response interaction:

```
int xxx_sendreply(trid,code,data,crypt_level)
int             trid;
int             code;
sbuf_ptr_t      data;
int             crypt_level;
```

`trid` is a transport-level ID supplied by the dispatcher with the request. `code` and `data` are the reply code and data to be used in the call to `reply_proc` on the client. `crypt_level` determines what kind of encryption should be used to protect the data. `data` can be null. If it is not, the data must be kept valid by the caller only until `xxx_sendreply` returns.

The procedure returns `TR_SUCCESS` or `TR_FAILURE`. It is an error to call `xxx_sendreply` for a terminated request-response interaction, including one terminated by the handler returning a code other than `DISP_WILL_REPLY`. Similarly, it is an error for the handler to return `DISP_WILL_REPLY` if the transport module in use does not implement `xxx_sendreply`.

Note that since multiple threads are involved, the reply procedure may be called before `xxx_sendrequest` returns. Similarly, it is acceptable for the server to call `xxx_sendreply` before returning from the handler procedure for the request, provided that this handler eventually returns `DISP_WILL_REPLY`.

The completion codes for transport operations are:

- `TR_CRYPT_FAILURE`: the transmission failed either because of a local encryption or a remote decryption failure — the local client module should try and get a new key for the destination host and then retry the transmission;
- `TR_SEND_FAILURE`: something went wrong with the transmission before any data could be sent.

- TR_FAILURE: something went wrong with the transmission within either the local or remote transport module; the error was detected too late to return TR_SEND_FAILURE.
- TR_OVERLOAD: the transport module is currently overloaded. No data was sent; the user should retry later.
- TR_SUCCESS: the transmission was successful but it was not possible to determine whether the message was accepted by the remote client.
- a client-specific completion code (see, for example, the codes returned by the IPC module).

Note that the transport-specific completion codes are in the same space as the client-specific completion codes; care must be taken to avoid collisions.

4.3. Specific Transport Protocols

4.3.1. Delta-t

Delta-t is a connectionless transport protocol in which each packet sent over the network is individually acknowledged by the destination network server before the next packet is sent. All the transport-level protocol handling is performed in the network server; network access at the IP level is achieved through the Mach network interface described below. Retransmissions are scheduled using the timer module. In order to detect duplicate packets, information about an incoming data packet is maintained by a network server either until the next packet in a sequence is received or the information has been held more that some minimum amount of time.

Delta-t currently implements the request-response interface without `deltat_sendreply`.

Delta-t is suitable for transmitting messages containing small amounts of data (less than a few packets in total). If the message is part of a RPC protocol in which the response to the request is expected shortly, then it is better to use a more specialized request-response protocol such as VMTP.

4.3.2. VMTP

VMTP (*Versatile Message Transport Protocol*) is the request-response protocol developed at Stanford by D. Cheriton and his group [1]. The VMTP module creates a single well-known server entity to receive all network requests, and keeps a pool of client entities to use on the client side to initiate transactions. Whenever `vmtp_sendrequest` is called, an unused client entity is pulled out of that pool; it is returned to the pool when the reply is delivered. The actual VMTP protocol handling module is in the kernel; it communicates with the network server using sockets.

There are currently two versions of the VMTP module. *vmtp1* implements the simple `send` interface and should not be used. *vmtp2* implements the full request-response interface, and is suitable for IPC messages.

4.3.3. TCP

The TCP module keeps a pool of TCP connections to the network servers on other nodes. Each connection is created when needed to transmit a request, and is kept open as long as possible to service further requests to the same destination. Open connections are recycled using a least-recently-used policy to limit their number. The protocol handling module is in the kernel; it communicates with the network server using sockets.

TCP currently implements the full request-response interface, and is suitable for IPC messages.

4.3.4. Datagram

The "datagram" transport protocol simply provides an interface to the UDP level. It allows unreliable datagrams to be sent over the network.

Datagram currently implements the simple `send` interface and cannot be used to transmit IPC message data.

4.3.5. Simple Request-Response

The simple request-response protocol permits the sending of a request over the network for which a response is expected. The data of the request is guaranteed to fit in one network datagram. This protocol will treat responses as acknowledgements to requests and inform its client either of the failure of the request (if no response was received after some number of tries) or of the success of the request in which case the response is returned to the client. It is assumed that a request can be handled without delay by the higher-level protocol and the response is supplied on return from the request call. Requests made using this protocol should be idempotent.

Although SRR is oriented toward request-response interactions, it implements the simple `send` interface and not the request-response interface. It cannot therefore be used to transmit IPC message data.

5. Dispatcher Module

5.1. Operation

The dispatcher module is responsible for invoking the correct handler procedure when some network message has been received. It is called by the various transport modules, examines the *dispatcher header*, and selects the appropriate routine to call according to the message type (incoming IPC, port death, and so on).

This module can optionally establish a separation between a network thread and one or more other threads used to process the messages at the network server level. This last scheme is to be used if the network thread would not be fast enough if it had to completely process each message before listening to the net again. Note that with this mechanism, care must be taken to avoid problems when a transport module deallocates the space used by incoming data before that data has been processed at the higher level.

To allow communication between machines with different data representation, the dispatcher header always use a standard representation. The rest of each message uses whatever representation is in use on the sending machine; a code for that representation is stored in the dispatcher header and made available to the message handler modules.

To allow future expansion, the dispatcher also checks a version number for each incoming message.

5.2. Interface

As discussed in 4.2, there are currently two transport interfaces, to which correspond two dispatcher interfaces. The following table holds a set of entry points for each message type.

```
typedef struct {
    int          (*disp_indata)();
    int          (*disp_inprobe)();
    int          (*disp_indata_simple)();
    int          (*disp_rr_simple)();
    int          (*disp_in_request)();
} dispatcher_switch_t;
```

All modules place entry points in the table for each message type and each type of handler procedure that they support and wish to receive messages for. Unused entries are set to the special procedure `disp_no_function`; the network server is organized in such a way that there are not collisions in the table.

`disp_indata`, `disp_inprobe`, `disp_indata_simple` and `disp_rr_simple` are called by transport modules using the simple `xxx_send` interface, and will eventually be eliminated.

`disp_in_request` is the sole entry point used with the request-response interface.

```
int disp_init()
```

initializes the dispatcher module.

```
int disp_indata(trid,data,from,tr_cleanup,trmod,
                client_id,crypt_level,broadcast)
int                    trid;
sbuf_ptr_t             *data;
netaddr_t              from;
int                    (*tr_cleanup)();
int                    trmod;
int                    client_id;
int                    crypt_level;
boolean_t              broadcast;
```

is the primary function used to dispatch incoming data in the simple `send` mode of operation. `trid` is a transport level identifier assigned by the transport module `trmod`. It should be used in the call to `tr_cleanup` which signals to the transport module that the higher-level module is finished with the data contained in the *sbuf* `data`. Other arguments are: `from`, the host that sent the data; `client_id`, an identifier assigned by the client module within a prior call to `disp_inprobe` (see below); `crypt_level`, the encryption level used to send data over the network; and `broadcast` whether the data was broadcast or not. `disp_indata` returns `DISP_FAILURE` is the dispatcher module did not find a higher-level routine to be called for the incoming message type or if the version number of the incoming message did not match the current version number of this implementation of the network server; otherwise it returns the value returned by the higher-level routine.

```
int disp_inprobe(trid,pkt,from,cancel,trmod,
                client_id,crypt_level,broadcast)
int                     trid;
sbuf_ptr_t              pkt;
netaddr_t               from;
int                     *((*cancel)());
int                     trmod;
int                     *client_id;
int                     crypt_level;
boolean_t               broadcast;
```

allows the first packet of a message to be dispatched to a higher-level probe routine. This allows the higher-level routine to decide before-hand whether to accept or reject an incoming message. If it decides to accept the message based on the probe packet, the it returns a `client_id` to allow it to later identify the whole incoming message. `cancel` (an out parameter) is called by the transport module if it is unable to deliver the complete message after a probe has been accepted. It takes as argument the `client_id` and a reason code. Other parameters are as for `disp_indata`.

```
int disp_indata_simple(client_id,data,from,crypt_level,broadcast)
int                     client_id;
sbuf_ptr_t              data;
netaddr_t               from;
int                     crypt_level;
boolean_t               broadcast;
```

is similar to `disp_indata` except that it is guaranteed that the data is processed at the higher-level within the same thread that made the call. Hence there is no need for a `tr_cleanup` call because, when the dispatcher call returns, the transport module knows that the data is no longer needed and can do the cleanup synchronously.

```
int disp_rr_simple(data,from,crypt_level,broadcast)
sbuf_ptr_t              data;
netaddr_t               from;
int                     crypt_level;
boolean_t               broadcast;
```

allows a transport-level protocol to make a simple request-response interaction with a higher level module. The higher-level module should process the request immediately and include the response on returning from the call. Note that this procedure is intended for request-response interactions within the simple `send` mode of operations, and not within the full request-response interface described above.

```
int disp_in_request(trmod,trid,data_ptr,from,crypt_level,broadcast);
int                     trmod;
int         trid;
sbuf_ptr_t         data_ptr;
netaddr_t         from;
int         crypt_level;
boolean_t         broadcast;
```

is the single function for dispatching in the request-response mode of operation. The arguments are similar to those of `disp_indata`. The data pointed to by `data_ptr` is valid only until this procedure returns. Any return value other than `DISP_WILL_REPLY` is interpreted by the transport module as a completion code to be returned immediately in a response to the client. `DISP_WILL_REPLY` means that the higher-level module assumes the responsibility to send a response, and the transport module should do nothing when this procedure returns (other than deallocate the space occupied by the data, if appropriate). Note that if the dispatcher returns `DISP_FAILURE`, that code is returned to the client in the normal way.

In subsequent sections of this document, functions which are called via the dispatcher module do not have their arguments described. The arguments are exactly as for the corresponding dispatcher function.

6. IPC Message Handling

The IPC Message Transfer module implements the upper layer of the mechanism used to communicate with remote ports. It relies on a separate transport module to provide the lower-level network transmission operations, and communicates with it using *sbuf*s. To maximize performance, the IPC module tries to identify messages that are part of a remote procedure call (*RPC*), and attempts to map the request-response structure of such calls into a request-response interaction at the transport level.

6.1. Normal Operation

The IPC module receives messages addressed to ports that are local representatives of remote ports. Upon reception of such a local IPC message, the IPC module

1. allocates a transaction record (`ipc_rec`) to keep information about the transfer in progress,
2. consults the port records to find the remote port corresponding to the local port,
3. generates an *IPC Header* to contain special information to be used by the remote network server,
4. translates the ports and out-of-line pointers in the message so that they will be intelligible on the receiving node, and

5. selects a transport protocol to use, and hands the whole message to the appropriate module via `xxx_sendrequest`.

Note that the breakup of the message into several packets is the task of a transport module and not the IPC module.

All these operations are performed on a message represented using an *sbuf*. In general, the segments contained in the *sbuf* are:

- the IPC receive buffer, containing the inline data;
- each out-of-line data section;
- the IPC header, allocated by this module; and
- any special data structures needed for the message translation process, such as accessibility maps or a network port dictionary.

In the remote network server receiving the message, all the component packets are assembled by the transport module, which calls the IPC module and hands it an *sbuf* representing the whole message. Typically segments in that *sbuf* are part of packet buffers in which the message was received. The module uses `ipc_rec` records to store information about current incoming messages. It performs all necessary translations (including byte-swapping and data type conversion), copies the message into a send buffer, and delivers it to its local destination.

The IPC module on the sending network server may not discard the message immediately after calling the transport module to initiate a transmission, because it may be necessary to effect a complete retransmission, possibly to a new destination and with different accompanying data. This is the case when some exceptional events, described in the next section, prevent the receiving network server from delivering the message to its ultimate destination. In addition, some operations pertaining to security must be performed by the IPC module on the sending node only when it is certain that the message has been correctly transmitted. For these reasons, the network server on the receiving node uses the request-response transport mechanism to return a completion code indicating if it was able to deliver the message, or what happened if it was not. Upon reception of this completion code, the sending network server may deallocate its `ipc_rec` and the message data, or undertake the appropriate recovery action in case of error, including initiating a new transmission.

6.2. Exceptional Events

The exceptional events are detected either at the transport level or by the IPC module in the remote network server. They are reported to the sending IPC module through the request-response completion code described above. The IPC module must then freeze the current transmission, and call another module to deal with that exception. That module may in turn request the IPC module to attempt to retry the transmission of any pending messages for a given destination port, possibly after changing some information in the port's record. In that case, the IPC module will restart processing of any affected messages as if that message had just been received on a local port. As an optimization, it could reuse some of the information already gathered, and stored in the record for this pending message.

Exceptional events include:

- Modification of a port record while a message for that port is in transit by some other thread running concurrently with the IPC sending thread (for example in response to a port death message). It simply triggers re-processing of the message.
- Network failure, detected by the transport module. The IPC module must decide to abort or retry the message, and find out if the port is dead.
- Crypt failure, detected by the transport module at the local or remote node when it does not possess the correct key to encrypt or decrypt the message. The IPC module calls the key management module to establish a new key.
- Remote port not found at the expected node, signalled by the receiving IPC module — a port search procedure is initiated (in another thread), and its completion will decide whether the transmission is to be restarted or aborted.
- Remote port blocked, signalled by the remote IPC module. The sending node suspends the message until further notice and indicates the situation in its port record. A `ipc_block` record for the sending node is allocated at the receiving node. When the port becomes unblocked, the list of those records is scanned and a special `PORT_UNBLOCKED` message is transmitted to each waiting network server, to indicate that transmission should be restarted. If this unblock message is lost then the port checkups module will discover that the port has become unblocked and will retry the message transmission.

To avoid blocking the message delivery thread in the receiving network server, all messages are always delivered using the `SEND_NOTIFY` option of `msg_send`. If the return from `msg_send` indicates that the local port has become blocked, no other messages will be accepted for that port until the kernel indicates that it has been unblocked. Appropriate marking and locking of the port record guarantees that no two threads can be in the situation of exercising the `SEND_NOTIFY` option on the same port at the same time. Note that this mechanism does not require the receiving network server to return a port blocked indication for a message accepted under the `SEND_NOTIFY` option, thereby allowing the actual message delivery to be performed in another thread after the dispatcher procedure has returned (the current implementation does not take advantage of this feature).

6.3. RPC Interactions

The IPC module offers an optimization for RPC interactions for which the user is willing to accept some semantic restrictions:

- the request message is followed by exactly one response message addressed to the reply port indicated in the request.
- the reply port is local to the node issuing the request.
- no new request is sent using the same reply port while awaiting a response on that reply port.
- the reply port is not deallocated while awaiting a response.
- the receive rights for the reply port are not transferred while awaiting a response.
- the reply port is not blocked or locked while awaiting a response.

If any of those rules is violated, the response is not guaranteed to be delivered, or may be delivered out of order; the behavior of the network server may vary from one instance of such an erroneous situation to the next. The user must specify the `MSG_TYPE_RPC` bit in the `msg_type` field of the message header for the request to indicate that he accepts those modified message semantics.

Whenever a valid request is received with the `MSG_TYPE_RPC` bit set, the network server on the server side uses `DISP_WILL_REPLY` if possible to delay the transmission of the completion code at the transport level. It keeps track of the pending request in a `ipc_rec`, and considers a new IPC message destined to the reply port as the response for the RPC interaction. Instead of sending this message with a new transport interaction, it places it in the response for the pending interaction. Because the transport interface does not provide an end-to-end acknowledgment that the data portion of a response was correctly handled by the IPC module on the client side, the network server on the server side must rely on the above assumptions for delivery of the response to the reply port.

This scheme also relies on the assumption that the response message will eventually be sent from the same node that received the request message, and that no other traffic involving the reply port takes place until that response is delivered. This assumption may easily be invalidated both by erroneous actions on the part of the client or server processes, or by normal operations such as request forwarding. Because resources in the transport module are tied up as long as the response has not been delivered, the IPC modules on both sides check for all events that may suggest that a response may not be forthcoming in the normal way. When any such event occurs, they force the transmission of a dummy response to terminate the request-response interaction, letting the real response, if any, proceed normally as a new single IPC message. The events causing such an abort include:

- transfer of receive or ownership rights for the reply port.
- transmission of a new message using the same reply port.
- reception of a message on the reply port on the client side, from a source other the expected server.
- reception of a message on the reply port on the server side, with the `MSG_TYPE_RPC` bit set.
- timeout at the network server on the server side (triggered by the port checkups mechanism).

Note that none of these aborts compromise the semantics of an RPC; they simply nullify the performance benefits of the RPC optimization when the situation is not simple. In addition, the network server itself never sets the `MSG_TYPE_RPC` bit when delivering a message to a local port, to avoid hidden forwarding problems.

6.4. Message Ordering

A weak ordering of message delivery is provided through the use of a queue of pending transactions for each remote network port. This queue operates in the following way:

- all outgoing RPC requests and single IPC messages are queued in the order in which they are received on the local port.
- as long as there are only RPC requests on the queue, each is transmitted as soon as it is queued; the system does not wait for a response before transmitting the next request.

- as soon as one single IPC message is queued, newer messages (IPC or RPC) are queued but not transmitted until that IPC message is successfully transmitted and dequeued (including any number of retransmissions).
- RPC responses are never queued; they are transmitted at once and never retransmitted.
- whenever the status of the network port changes, retransmissions are initiated as needed in the order in which records are on the queue.
- the local port is locked when the queue becomes too long and new messages cannot be transmitted.

This strategy guarantees that single IPC messages directed at the same destination port from the same node are strictly ordered. RPC's are naturally ordered simply because the client waits for the response before issuing the next request. There are no ordering guarantees for a single IPC immediately following the request for a RPC.

6.5. Interface

```
boolean_t ipc_init()
```

initializes the IPC module.

```
int ipc_in_request(trmod,trid,data_ptr,from,crypt_level,broadcast)
```

is called from `disp_in_request` when a RPC request or single IPC is received over the network by a transport module. It is the main entry point for incoming messages. See the dispatcher module for details about the parameters to this call.

```
void ipc_in_reply(client_id,code,data_ptr)
```

is the procedure used by the IPC module to receive responses from the transport module after a call to `xxx_sendrequest` for IPC message data. See the transport module for details about the parameters to this call.

```
ipc_in_unblock(client_id,data,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` when an unblock message is received from a remote network server. It will cause the message transmission to be retried. See the dispatcher module for details about the parameters to this call.

```
void ipc_msg_accepted(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

is called by the local port operations module when it receives a notify message from the kernel saying that a particular port is now unblocked. The IPC module will send `unblock` notification messages to remote network servers that are blocked waiting to send a message to the port.

```
void ipc_port_dead(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

is called by the local port operations module either when it receives a notify message from the kernel saying that a particular port is now dead or when it receives a message from a remote network server saying that a particular network port is now dead. The IPC module will clean up any data structures it has associated with the deceased port.

```
void ipc_port_moved(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

is called by the local port operations module when it receives a message from a remote network server saying that a particular network port has moved. The IPC module will abort any pending RPC's involving this port.

```
int ipc_in_abortreq(trmod,trid,data_ptr,from,crypt_level,broadcast)
```

is called from `disp_in_request` when a request to abort a pending request-response interaction is received over the network by a transport module. The data specifies which RPC is to be aborted. If it is still pending, a dummy response is sent at once; otherwise, this request is ignored. See the dispatcher module for details about the parameters to this call.

```
void ipc_in_abortreply(client_id,code,data_ptr)
```

is the procedure used by the IPC module to receive responses from the transport module after a call to `xxx_sendrequest` for a request to abort a pending RPC. See the transport module for details about the parameters to this call.

```
void ipc_retry(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

is called from other modules when a message transmission should be retried following some change in the port records. It will cause the retransmission to be executed in a special "resend" thread distinct from the one making the `ipc_retry` call.

```
void ipc_freeze(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

is called from other modules when the status of a port becomes such that no further transmissions should be attempted to that port. Transmission will be resumed when `ipc_retry` is called.

7. Port Operations

7.1. Description

The functions provided by the port operations module are called in one of the following three circumstances:

1. A message is received from the local kernel notifying the network server about a change in condition of a local port. These changes are:
   - the death of the local port;
   - the transfer of port access rights to another task (probably because the task holding the rights has died); and
   - the local unblocking of the port.
2. A message is received over the network notifying the network server of a change in the remote network port's condition. The possible changes are as for the local case except that they should be interpreted in the context of the remote port.
3. Access rights to a port are being transferred in a normal IPC message which is about to be sent to a remote network server or has been received from a remote network server.

The behavior of the port operations module depends on whether the port that it is handling must be treated securely or not. For instance, if send rights to a port are being transferred in an IPC message and the port is meant to be secure, then a token should be created for the port and transferred along with the network port identifier when the message is sent to the remote network server. Similarly, at the receiving end, the port operations module should store the token in the network port's record. However, if the port is not meant to be treated securely, then no transfer and storing of a token need be done.

In general the port operations module will often have to directly modify the port's records, it may retry or freeze the sending of an IPC message, it may initiate a port search, or, when port rights are being transferred, it will provide the information that must be sent to the remote network server. Conversely, it must process this information when the information is received from a remote network server. In addition, when it is considering a secure port, it may have to generate or check a token for the port or it may have to transfer or check the key that represents receive or ownership rights to the port.

7.2. Interface

```
boolean_t po_init()
```

initializes the port operations module.

```
po_check_ipc_seq_no(port_rec_ptr, host_id, ipc_seq_no)
port_rec_ptr_t          port_rec_ptr;
netaddr_t               host_id;
long                    ipc_seq_no;
```

checks that the incoming IPC sequence number of a message is greater that the last sequence number received for the network port with port record `port_rec_ptr` from the network server on machine `host_id`. This check is only done for secure messages and ensures that complete IPC messages cannot be replayed by a malicious party.

```
typedef struct {...} secure_info_t, *secure_info_ptr_t;
```

is used to hold the key representing receiver or ownership rights to a network port.

```
long po_create_token(port_rec_ptr, token_ptr)
port_rec_ptr_t          port_rec_ptr;
secure_info_ptr_t       token_ptr;
```

creates a token for a port. Stores the token in `token_ptr` and returns the random number used to construct the token.

```
void po_notify_port_death(port_rec_ptr)
port_rec_ptr_t          port_rec_ptr;
```

triggers handling of a local port death. Marks the port's record as deleted, sends out an unreliable port death notification messages and does other local cleanups.

```
void po_port_deallocate(lport)
port_t                  lport
```

deallocates a port but retains send rights to it. This allows the network server to transfer receive or ownership rights to a port to a local task using the notification mechanism of the kernel.

```
int po_translate_lport_rights(client_id,lport,right,security_level,
                destination_hint,port_data)
int                     client_id;
port_t                  lport;
int                     right;
int                     security_level;
netaddr_t               destination_hint;
pointer_t               port_data;
```

is called by the IPC module to pack up the data that needs to be sent to the host `destination_hint` in order to transfer the access rights `right` to port `lport`. The data that needs to be sent depends on the `security_level` of the transfer. The data is packed into the space pointed to by `port_data` and the size of the network port data that has been created is returned as the function's result. The `client_id` is an identifier remembered by the port operations module so that it can match up a subsequent `po_port_rights_commit` (see below) with this call of `po_translate_lport_rights`.

```
po_port_rights_commit(client_id, completion_code, destination)
int                     client_id;
int                     completion_code;
netaddr_t               destination;
```

informs the port operations module that a transfer of rights to a remote network host has either succeeded or failed. The `client_id` allows the port operations module to match this call with a previous call of `po_translate_lport_rights`. The `completion_code` can be one of `PO_RIGHTS_XFER_SUCCESS` and `PO_RIGHTS_XFER_FAILURE`. The `destination` names the remote network server to which the port rights were actually transferred. It may be different from the `destination_hint` passed to `po_translate_lport_rights`.

```
int po_translate_nport_rights(source,port_data,security_level,
                lport,right)
netaddr_t               source;
pointer_t               port_data;
int                     security_level;
port_t                  *lport;
int                     *right;
```

is called by the IPC module when it receives access rights to a remote network port in a message from a remote network server. The access rights are contained in the data pointed to by `port_data` and were received from the network server on host `source` and at `security_level`. The port data received is handled according to what access rights are being transferred and the local port corresponding to the network port that was transferred is returned in `lport`. In addition the actual right transferred is returned in `right` and the size of the port data that was processed is returned as the function's result.

```
int po_handle_ro_xfer_request(request,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an incoming transfer of receiver or ownership rights.

```
int po_handle_ro_xfer_reply(client_id,reply,from,broadcast,crypt_level)
```

is called by `disp_rr_simple` to handle the response to a transfer of receiver or ownership rights.

```
int po_handle_ro_xfer_hint(request,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an unreliable notification of a transfer of receiver or ownership rights.

```
int po_handle_nport_death(hint,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an unreliable notification of the death of a network port.

`int po_handle_token_request(request,from,broadcast,crypt_level)`

is called by `disp_indata_simple` to handle an incoming request for a token of receiver/owner authenticity.

`int po_handle_token_reply(client_id,reply,from,broadcast,crypt_level)`

is called by `disp_rr_simple` to handle the response to a request for a token of receiver/owner authenticity.

8. Port Checkups

8.1. Description

The port checkups module does a periodic probing of other network servers to find out whether the status of network ports has changed. In particular, it is the default way in which the network server finds out about the death of a network port or the fact that receive or ownership rights have moved to a different network server. These special conditions can also be detected as part of the normal transmission of IPC messages across the network. The port checkups routine should only be called periodically and when the network server is otherwise idle; in other words it is of low priority.

The checkups module needs to be able to look at the port records in order to examine a “aliveness” parameter associated with each port record. The aliveness parameter is decremented by the port checkups module every time it is called. Only when it goes below some predetermined value, is a checkup performed for the port. Moreover, the aliveness parameter is updated to `fully-alive` when the IPC module has successfully sent a message over the network to the port. In other words, if the port is in regular use then no checkup is done for it.

When the checkups module actually decides to send a checkup request to find out about ports, it constructs an sbuf for each network server that it must query. An sbuf contains the ports in which it is interested for which it believes the remote network server is responsible. To transmit and receive checkup information across the network, the port checkups module uses the simple request-response transport protocol. After making a request by calling `srr_send`, the checkups module will either receive a checkup reply or a failure notification from the transport module.

On receiving a checkup request, the checkups module looks at each port contained in the request. If the information about the port that the requester sent does not match the information held locally, then the port in the checkup packet is marked as being “bad”. The checkup reply packet is then sent back to the requester.

On receiving a checkup reply, the requester examines all the ports in the reply and for those ports with a “bad” status it calls the port search module. It is up to the port search module to find out more about the status of the port. If no response was received to the checkup request then the checkups module must call the port search module for each port in the checkup request in order to resolve the port’s status (e.g. to determine whether it is dead).

One other function of the checkups module is to determine whether there exist any tasks with send rights to each port the network server knows about. This is in order to extend the MACH "no-senders" notification message into the network environment. The checkups module can determine that a network port has no senders if there has been no interactions (the reception of either an IPC message or a checkup request) involving this port for some period of time (typically some number of checkup rounds). If the network port has no senders then the checkups module can deallocate send rights to the corresponding local port and destroy the associated port record.

In addition the checkups module is responsible for handling hints received saying that a remote network server has just restarted. For such a hint the checkups module calls the port search module for each port that had the restarted network server as its owner or receiver.

8.2. Interface

`boolean_t pc_init()`

initializes the checkups module.

`int pc_do_checkups()`

is called by the timer module to perform a checkup.

`pc_handle_checkup_request(request,from,broadcast,crypt_level)`

is called by `disp_indata_simple` to handle an incoming checkup request.

`pc_handle_checkup_reply(client_id,reply,from,broadcast,crypt_level)`

is called by `disp_rr_simple` to handle an incoming checkup reply.

`void pc_send_startup_hint()`

is called on start-up to send out a hint saying that this network server has just restarted.

`int pc_handle_startup_hint(hint,from,broadcast,crypt_level)`

is called by `disp_indata_simple` to handle an incoming network server restart hint.

9. Port Search

9.1. Description

The port search module is called when some other module (probably either the port checkups module or the IPC module) determines that the information held about a network port is no longer correct. The task of the port search module is to update that information, in particular it may determine that the network port is dead.

The search procedure is basically as follows:

```
query network server believed to be the receiver;
if receiver responds with useful information
then believe it
else {
    query network server believed to be the owner;
    if owner responds with useful information
    then believe it
    else broadcast a request for information
}
```

The response to a port search query can be one of:

- port here, in which case the port search concludes successfully;
- port here but receive or ownership transferred, in which case the port search concludes successfully with the port record updated to reflect the new owner or receiver;
- port not here but receive and ownership rights transferred, in which case the port search continues by querying the new receiver;
- port dead, in which case the port search concludes and the port is destroyed locally; or
- port not known, in which case the port search continues by resorting to a broadcast query.

In addition, a query may receive no response in which case the port search continues by resorting to a broadcast query. To actually transmit port search queries and responses the port search module uses the simple request-response transport protocol.

The port search module is also responsible for authenticating a new receiver or owner for a network port if the identity of the new receiver or owner was obtained as a result of a broadcast search for the port. This authentication is only necessary is the port is being handled securely.

9.2. Interface

```
boolean_t ps_init()
```

initializes the port search module.

```
ps_do_port_search(port_rec_ptr,new_information,new_nport_ptr,retry)
port_rec_ptr_t          port_rec_ptr;
boolean_t               new_information;
network_port_ptr_t      new_nport_ptr;
int                     (*retry)();
```

is called to begin a port search for the network port recorded in `port_rec_ptr`. If the caller has `new_information` about the port (either the possible identity of a new receiver or owner for the port) then that new information is contained in the network port pointed to by `new_nport_ptr`. `retry` is a function supplied by the client to be called if the port search concludes successfully. It takes as its only parameter the `port_rec_ptr`.

```
int ps_handle_request(request,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an incoming port search query.

```
int ps_handle_reply(client_id,reply,from,broadcast,crypt_level)
```

is called by `disp_rr_simple` to handle an incoming reply to a port search query.

```
int ps_handle_auth_request(request,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an incoming request for authentication of a receiver or owner.

```
int ps_handle_auth_reply(client_id,reply,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an incoming reply to a request for authentication of a receiver or owner.

10. Key Management

10.1. Description

The key management module maintains a table which maps remote hosts to keys. When it has to send a message securely over the network, the IPC module checks that the key management module has a key for the message's destination. The actual encryption is done at the transport level when the message data has been placed in packets.

If the key management module has no key for a particular remote host, or the key that it possesses is obsolete, then it must call upon the local KDS (*Key Distribution Server*) to do a key exchange. The local KDS uses a central KDS to perform the key exchange. After the key exchange is complete, the key management module should retry the suspended IPC message.

10.2. Interface

```
boolean_t km_init()
```

initializes the key management module.

```
typedef struct {...} key_t, *key_ptr_t;
```

is used to hold an encryption or decryption key.

```
boolean_t km_get_key(host_id, key_ptr)
netaddr_t               host_id;
key_ptr_t               key_ptr;
```

looks up the key for the `host_id`. If there is a key it returns `TRUE` and places the key in `key_ptr`.

```
boolean_t km_get_ikey(host_id, ikey_ptr)
netaddr_t               host_id;
key_ptr_t               key_iptr;
```

looks up the inverse key for the `host_id`. If there is a key returns `TRUE` and places the key in `ikey_ptr`.

```
km_do_key_exchange(client_id, client_retry, host_id)
int                     client_id;
int                     (*client_retry)();
netaddr_t               host_id;
```

is called by a client module to get a key exchange done for `host_id`. When the key exchange succeeds, the key management module calls the function `client_retry` with the parameter `client_id` to inform the client that there is now a key for the host.

```
km_kds_connect(server_port, kds_port)
port_t                  server_port;
port_t                  kds_port;
```

is called by the local KDS to register its port (`kds_port`) with the network server.

```
km_use_key_for_host(server_port, host_id, key)
port_t                  server_port;
netaddr_t               host_id;
key_t                   key;
```

is called by the local KDS to tell the network server to use `key` for all future communication with `host_id`.

In the above two calls the `server_port` should always be a special port which is known only to the network server and the local KDS. The network server is responsible for starting the KDS and passing send rights to this special port to the KDS.

11. Crypt

The crypt module is responsible for the actual encryption and decryption of packets that are to be sent out over the network and received over the network.

11.1. Interface

```
typedef struct {...} netipc_t, *netipc_ptr_t;
```

points to an Internet packet encapsulated in a MACH IPC message.

```
int crypt_encrypt_packet(packet_ptr, crypt_level)
netipc_ptr_t            packet_ptr;
int                     crypt_level;
```

encrypts the packet pointed to by `packet_ptr` at the encryption level given by `crypt_level`. Returns either `CRYPT_SUCCESS` or `CRYPT_FAILURE` if there is no key for the remote host.

```
crypt_decrypt_packet(packet_ptr, crypt_level)
netipc_ptr_t            packet_ptr;
int                     crypt_level;
```

decrypts the packet pointed to by `packet_ptr` at the encryption level given by `crypt_level`. Returns either `CRYPT_SUCCESS`, `CRYPT_FAILURE` if there is no key for the remote host or `CRYPT_CHECKSUM_FAILURE` if the decrypted checksum is incorrect.

12. Network Name Service

12.1. Description

The network name service module provides a simple name service that is sufficient to boot-strap a higher-level name service that will provide a distributed and replicated user-level name service. The network name service is host-directed; that is requests for name look ups are sent to specific hosts and are not broadcast.

12.2. Interface

```
boolean_t netname_init()
```

initializes the network name module.

```
nn_remove_entries(port_id)
port_t                  port_id;
```

removes all entries for the local port `port_id` from the local name table.

```
typedef char netname_name_t[80]
```

```
kern_return_t netname_check_in(ServPort,port_name,signature,port_id)
vport_t                 ServPort;
netname_name_t          port_name;
port_t                  signature;
port_t                  port_id;
```

checks in the port `port_id` under the name `port_name` protected by `signature`.

```
kern_return_t netname_look_up(ServPort,host_name,port_name,port_id)
port_t                  ServPort;
netname_name_t          host_name;
netname_name_t          port_name;
port_t                  *port_id;
```

looks up `port_name` at host given by `host_name`. Returns in `port_id` the port found.

```
kern_return_t netname_check_out(ServPort,port_name,signature,port_id)
port_t                  ServPort;
netname_name_t          port_name;
port_t                  signature;
```

checks out the port checked in under `port_name`. The `signature` must match the `signature` supplied to the `netname_check_in` call.

```
kern_return_t netname_version(ServPort,version)
port_t                  ServPort;
netname_name_t          version;
```

returns in `version` some version identification for the network server.

```
int nn_handle_request(request,from,broadcast,crypt_level)
```

is called by `disp_indata_simple` to handle an incoming request for a network name look up.

```
int nn_handle_reply(client_id,reply,from,broadcast,crypt_level)
```

is called by `disp_rr_simple` to handle an incoming response to a request for a network name look up.

13. Memory Management

13.1. Operation

The memory management module is responsible for allocating and deallocating various objects used by the different modules, such as port and message records, buffers, and so on. It attempts to use knowledge of the types of objects required to achieve good performance. It tries to reduce the load placed on the MACH virtual memory system.

13.2. Interface

```
boolean_t mem_init()
```

initializes the memory management module.

```
int mem_clean()
```

attempts to free as much unused space as possible to reduce the paging load on the operating system; it is potentially slow.

```
pointer_t mem_allocobj(objtype)
int                    objtype;
```

allocates one instance of an object of the given `objtype` and returns its address, or 0 in case of failure.

```
void mem_deallocobj(ptr,objtype)
pointer_t              ptr;
int                    objtype;
```

deallocates an object of `objtype` previously allocated using `mem_allocobj`.

```
pointer_t mem_alloc(size,aligned)
int                    size;
boolean_t              aligned;
```

allocates a memory area of arbitrary size; it returns 0 in case of failure.

```
void mem_dealloc(ptr,size)
pointer_t              ptr;
int                    size;
```

deallocates memory previously allocated by `mem_dealloc`.

14. Read/Write Locks

The read/write locks module provides locks which can have multiple readers and signals threads waiting for a lock when it becomes free.

14.1. Interface

```
typedef enum {PERM_READ, PERM_READWRITE} rw_perm_t;
typedef enum {NOBLOCK = 0, BLOCK = 1} rw_block_t;
typedef struct lock {...} *lock_t;

lock_t lk_alloc()
```

allocates a read/write lock.

```
void lk_free(lock)
```

frees a read/write lock.

```
int lk_lock(lock, perm, block)
lock_t                 lock;
rw_perm_t              perm;
rw_block_t             block;
```

locks the `lock` for type `perm`. If `block` is true, then this calls blocks waiting until the lock can be obtained, otherwise the function returns 0 if the lock cannot be obtained.

```
void lk_unlock(lock)
lock_t                 lock;
```

unlocks the `lock`.

15. Locked Queue Module

The locked queue module provides functions to manipulate items on queues. When a queue is accessed it is always locked before being manipulated.

15.1. Interface

```
typedef struct {...} *lock_queue_t;
typedef struct queue_item {struct queue_item *next} *queue_item_t;
```

```
lock_queue_t lq_alloc()
```

allocates and initializes a new locked queue.

```
void lq_init_queue(queue)
lock_queue_t            queue;
```

re-initializes the already allocated `queue`.

```
void lq_prequeue(queue, item)
lock_queue_t            queue;
queue_item_t            item;
```

inserts `item` at the head of `queue`.

```
void lq_insert_in_queue(queue, test, item, args)
lock_queue_t            queue;
int                     (*test)();
queue_item_t            item;
int                     arg;
```

inserts `item` in the "correct" position on `queue`. The correct position is determined by calling the user-supplied function `test` on `item`, `arg` and the members of `queue` until it returns `TRUE`.

```
boolean_t lq_remove_from_queue(queue, item)
lock_queue_t            queue;
queue_item_t            item;
```

removes `item` from `queue` if `item` is present on the queue. Returns `TRUE` is `item` was deleted from `queue`, `FALSE` otherwise.

```
boolean_t lq_cond_delete_from_queue(queue, test, item)
lock_queue_t            queue;
int                     (*test)();
queue_item_t            item;
int                     arg;
```

performs the user-supplied function `test` on `item`, `arg` and on successive elements of `queue`. If it returns `TRUE`, then the current element of the `queue` is deleted.

```
boolean_t lq_on_queue(queue, item)
lock_queue_t            queue;
queue_item_t            item;
```

checks to see if the `item` is on `queue`.

```
queue_item_t lq_dequeue(queue)
lock_queue_t            queue;
```

if `queue` is not empty remove and return the queue item which is at the head of it.

```
queue_item_t lq_blocking_dequeue(queue)
lock_queue_t            queue;
```

if `queue` is empty, a wait is done until it is non-empty. Removes and returns the queue item which is at the head of `queue`.

```
void lq_enqueue(queue, item);
lock_queue_t            queue;
queue_item_t            item;
```

inserts `item` at the tail of `queue`.

```
queue_item_t lq_find_in_queue(queue, fn, args)
lock_queue_t            queue;
int                     (*fn)();
int                     arg;
```

returns a `queue_item_t` which is found by applying the user-supplied function `fn` to successive elements of `queue` and `arg` until `fn` returns `TRUE`.

```
void lq_map_queue(queue, fn, args);
lock_queue_t            queue;
int                     (*fn)();
int                     arg;
```

applies the user-supplied function fn to each successive item of queue and arg.

In addition to the above routines, a number of equivalent routines are provided that do not acquire or release the queue lock when invoked, to be used in situations where global lock management is needed to avoid deadlock. Those routines are prefixed with lqn_ instead of lq_.

Finally, the network server also uses doubly-linked lists for some queues. using the same macros used in the Mach kernel for that purpose.

16. Timer Module

The timer module accepts requests from other modules for events to be scheduled at some time in the future. When the event's deadline expires the timer module calls the user-supplied function associated with the timer.

16.1. Interface

```
boolean_t timer_init()
```

initializes the timer module.

```
struct timer {...} *timer_t;
```

```
timer_t timer_alloc()
```

returns a new timer.

```
void timer_start(timer)
timer_t                 timer;
```

starts up timer.

```
void timer_stop(timer)
timer_t                 timer;
```

stops timer.

17. Miscellaneous

17.1. Unique Identifier Generator

Simply generates locally unique identifiers (*UID*s). The identifiers generated are unique with high probability.

17.1.1. Interface

```
void uid_init()
```

initializes the UID module.

```
long uid_get_new_uid()
```

returns a new UID.

17.2. Sbuf

The sbuf module provides macros that manipulate *sbuf*s.

17.2.1. Interface

```
void sbuf_printf(where, sb_ptr)
FILE                    *where;
sbuf_ptr_t              sb_ptr;
```

is the only exported function of the sbuf module. It prints out the contents of the *sbuf* pointed to by `sb_ptr`.

17.3. Network Interfaces

Under Mach the interface to the network is an IPC interface with a filter inside the kernel determining which network packets are to be received by the network server. Currently, many transport modules still use BSD Unix sockets to access network protocol implementations in the kernel.

17.3.1. Interface

```
int netipc_receive(pkt_ptr)
netipc_ptr_t            pkt_ptr;
```

waits to receive a packet from the kernel. Checks the packet's UDP checksum before returning to the caller.

```
int netipc_send(pkt_ptr)
netipc_ptr_t            pkt_ptr;
```

calculates the UDP checksum for the packet and then sends it to the kernel for transmission over the network.

17.4. IPC Message Receive

17.5. Interface

```
int netmsg_receive(msg_ptr)
msg_header_t            *msg_ptr;
```

does a non-blocking receive for a local IPC message.

17.6. Debugging

The network server keeps a log in memory of various events happening during its operation. This log, along with statistics on various operations, can be obtained via the *logstat* service exported by the network server. In addition, many operating parameters, including the level of debugging information written to the log, can be set using this same service.

17.6.1. Interface

Macros and procedures called within the network server

```
DEBUGn(condition,print_level,code,arg1,...,argn)
```

is a macro to be used to write a record containing the `code` and all the integer `args` into the log. `n` is a number between 0 and 6, indicating how many integers must be copied into the log. A log entry is only made if `condition` evaluates to `TRUE`. In addition, if `print_level` is greater or equal to the global `debug.print_level`, a message is printed on stderr.

```
DEBUG_STRING(cond,level,string)
DEBUG_NPORT(cond,level,nport)
DEBUG_NETADDR(cond,level,netaddr)
DEBUG_KEY(cond,level,key)
DEBUG_SBUF(cond,level,sbuf)
```

are similar to the `DEBUGn` macros, but are used to enter a string, a network port identifier, a network address, an encryption key or an sbuf into the log.

The `DEBUG` macros can be made to expand to nothing via a compile-time switch to avoid overheads at execution time. Each of those macros has an equivalent `LOG` macro that can be enabled or disabled independently; those `LOG` macros are intended for events that should always be entered in the log and are infrequent enough that the overhead involved is negligible.

```
ERROR((msg,format,args...))
```

is used to print out a message on stderr an make an entry in the log. The argument should be a valid set of arguments for `sprintf`, with the message string `msg`.

```
void panic(error_msg)
char                    *error_msg;
```

is called if something catastrophic happens. Prints out the `error_msg`, dumps the log and terminates the network server.

```
void ipaddr_to_string(output_string, input_address)
char                    *output_string;
netaddr_t               input_address;
```

translates the `input_address` IP address into a printable representation in `output_string`.

Procedures exported outside the network server: The following procedures can be called remotely by sending requests on a port checked-in as `NM_LOGSTAT` in the network server.

```
kern_return_t ls_sendlog(ServPort,old_log_ptr,old_log_size,
                         cur_log_ptr,cur_log_size)
port_t          ServPort;
log_ptr_t       *old_log_ptr;
unsigned int    *old_log_size;
log_ptr_t       *cur_log_ptr;
unsigned int    *cur_log_size;
```

is used to obtain both network server logs in the response message. The old and new logs correspond to the two alternating logs used to record events.

```
kern_return_t ls_resetlog(ServPort)
port_t          ServPort;
```

resets the log to zero size.

```
kern_return_t ls_writelog(ServPort)
port_t          ServPort;
```

causes the network server to write its log in a file `NMLOG` in its current working directory.

```
kern_return_t ls_sendstat(ServPort,stat_ptr,stat_size)
port_t          ServPort;
stat_ptr_t      *stat_ptr;
unsigned int    *stat_size;
```

is used to obtain a record with the vital network server statistics in the response.

```
kern_return_t ls_resetstat(ServPort)
port_t          ServPort;
```

resets all statistics counters to zero.

```
kern_return_t ls_senddebug(ServPort,debug_ptr,debug_size)
port_t          ServPort;
debug_ptr_t     *debug_ptr;
unsigned int    *debug_size;
```

is used to obtain a record with all the debugging flags used to control the operation of the `DEBUG` macros.

```
kern_return_t ls_setdebug(ServPort,debug_ptr,debug_size)
port_t          ServPort;
debug_ptr_t     *debug_ptr;
unsigned int    *debug_size;
```

is used to replace the record with all the debugging flags used to control the operation of the `DEBUG` macros.

```
kern_return_t ls_sendparam(ServPort,param_ptr,param_size)
port_t          ServPort;
param_ptr_t     *param_ptr;
unsigned int    *param_size;
```

is used to obtain a record with the network server control parameters.

```
kern_return_t ls_setparam(ServPort,param_ptr,param_size)
port_t          ServPort;
param_ptr_t     *param_ptr;
unsigned int    *param_size;
```

is used to replace the record with the network server control parameters.

```
kern_return_t ls_sendportstat(ServPort,port_stat_ptr,port_stat_size)
port_t          ServPort;
port_stat_ptr_t *port_stat_ptr;
unsigned int    *port_stat_size;
```

is used to obtain a record with the port record statistics.

17.7. Camelot Support

The Camelot Distributed Transaction Facility [3] requires special handling for IPC messages used in Camelot transactions. This handling is performed in a special Camelot module, not described, here, that behaves as an extra step in the translation process for incoming and outgoing IPC messages.

In addition, Camelot also requires some specialized name servers, also implemented in the Camelot module.

17.8. Kernel Netport Support

Certain Mach kernels provide an experimental feature, called *Netport* or `MACH_NP` with which Kernel port records may be flagged as corresponding to local representatives for remote network ports. Under certain very restricted conditions, the kernel may, upon processing a message destined for one of these ports, send the message directly to the remote node instead of handing it to the network server. This scheme results in improved performance by avoiding the overhead of invoking the network servers on both ends of the communication. Correctness is assured by having the kernel abort its transmission and reflect the message back to the network server as soon as a situation arises that is too complex for the Netport code to handle.

When enabled, all modules in the network server that modify network port records enter the correct information in the kernel port records to allow the Netport code to function.

17.9. Initialization

The network server initialization sequence takes care of detecting modules that require kernel support not present on the current node, and of setting the working parameters accordingly. These include:

- Access to a network interface. If there is no network, the network server degenerates into a simple local Name Server, as specified by the `conf_network` parameter.
- Netport support: controlled by the `conf_netport` parameter.
- VMTP support. The `transport_default` parameter is set to the index of the best transport protocol available.

17.9.1. Interface

```
boolean_t nm_init()
```

initializes the network server by calling and checking the error returns for all the module initialization functions.

A Compiling a Network Server

The various configuration options for the Network Server are all defined in the file `config.h`, which should simply be edited before compiling the system. The option settings in the file as distributed are suitable for a normal generic configuration.

The configuration options are:

NET_LOG enable the `LOGn` macros.

NET_DEBUG enable the `DEBUGn` macros.

NET_TRACE enable tracing of procedure calls, under control of a command line switch.

NET_PRINT enable printing from the `LOG` and `DEBUG` macros.

LOCK_THREADS do not allow more than one thread to run at any one time. To use only for debugging.

NM_STATISTICS enable normal statistics gathering.

NETPORT enable entering information in the kernel port records for use by the Netport option.

PORTSTAT enable port statistics gathering.

RPCMOD enable the RPC optimization, and the request-response transport interface. Should always be on.

COMPAT enable special operating mode for compatibility with the previous implementation (Mach 1.0 and 2.0) of the Network Server.

NOTIFY explicitly allocate a notify port, which is not created by default in newer versions of the Mach kernel.

CAMELOT include the Camelot module in the network server.

NM_USE_KDS use an external Key Distribution Server.

USE_VMTP include the VMTP transport module in the network server.

USE_DELTAT include the Delta-t transport module in the network server.

USE_CRYPT include the Crypt module in the network server.

USE_DES include the DES encryption module in the network server.

USE_MULTPERM include the “multiple permutations” encryption module in the network server.

USE_NEWDES include the “new” DES encryption module in the network server.

USE_XOR include the “exclusive or” encryption module in the network server.

USE_KEYMAN include the keymanager module in the network server.

USE_TCP include the TCP transport module in the network server.

USE_DATAGRAM include the DATAGRAM transport module in the network server.

USE_SRR include the SRR transport module in the network server.

Both `USE_VMTP` and `NETPORT` require kernel support that is not normally present. Normally, `USE_TCP`, `USE_SRR` and `USE_DATAGRAM` should always be enabled for the system to work.

In addition to the configuration options, the file `config.h` also contains all the definitions needed to compile the network server on various architectures. It is the only file that should be modified when porting the network server to a new architecture.

References

References

[1] Cheriton, D. *VMTP: A Transport Protocol for the Next Generation of Communication Systems*. In: **Proceedings of the ACM SIGCOMM 86 Symposium on Communications Architectures and Protocols**. ACM, 1986, pp. 406–415.

[2] Sansom, R. D., Julin, D. P., and Rashid, R. F. *Extending a Capability Based System into a Network Environment*. In: **SIGCOMM '86 Symposium: Communications Architectures & Protocols**, ACM SIGCOMM. 1986. *Also available as Technical Report CMU-CS-86-115.*

[3] Spector, A. Z., Bloch, J. J., Daniels, D. S., Draves, R. P., Duchamp, D., Eppinger, J. L., Menees, S. G., and Thompson, D. S. *The Camelot Project*. **Database Engineering**, vol. 9 (1986). *Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.*