Portable Systems Group

NT OS/2 Local Inter-Process Communication (LPC) Specification

Author: *Steven R. Wood*

Revision 1.5, February 17, 1989

.Begin Table C.

1. Overview

The NT OS/2 system is implementing the majority of the Application Program Interfaces (API's) using the Client/Server model where an application's call to an API is intercepted by a stub in the client process that packages up the parameters to the call and sends them to a server process that will actually implement the API. The Local Inter-Process Communication (LPC) package is the system facility that allows the stub procedure to communicate the data to the server process and wait for a response. The design of the LPC facility is guided by the fact that it will be used primarily to model a synchronous procedure call between two processes in the same memory domain.

1.1. Port Object

The primary data structure used to implement the NT OS/2 LPC mechanism is the Port Object. There are two types of port objects needed; a connection port and a communication port.

A connection port is created by a server process with a name. A server process usually maintains at least one thread that is listening for connection requests. A client process connects to the server process using the name of the connection port. Whenever a connection request is sent to a connection port, the server thread wakes up, examines the connection request and decides whether to reject or accept the connection. If the connection request is accepted, then the LPC facility creates two communication ports, one for the client and one for the server. The communication port objects have no names and cannot be inherited by child processes. Connection ports have names, but cannot be inherited.

A port object contains the following information:

- Flags that indicate which of the following queues are present.
- An optional queue for Connection messages.
- An optional queue for Request messages.
- A pointer to a connection port object.
- A pointer to a communication port object.
- Context value associated with a communication port.
- A zone segment that divided evenly into multiple chunks of a single size, which is the maximum message length. The zone segment consists of a singly linked list of free blocks, guarded by a spin lock.

- An event that is clear whenever a message is placed on the zone free list and the free list was empty at the time of insertion. Threads can wait on this event if they ever encounter an empty free list.
- An optional memory section handle that is mapped into both the client process and the server process address spaces. Location and size information is also recorded here so the memory can be unmapped later.

A connection port has a queue for Connection messages. It may also have a queue for Request messages if the **ReceiveAnyPort** parameter is specified on the call to **NtCreatePort**. The pointer to the connection port object points to the connection port object itself. The pointer to the connected communication port is NULL, as is the context value.

A server communication port does not have a queue for Connection messages. It may have a queue for Request messages if the **ReceiveThisPort** parameter is specified on the call to **NtCompleteConnectPort**. The pointer to the connection port object points to the connection port that the connection request came in on. The pointer to the communication port object points to the client communication port. The context value is set to the value specified on the call to **NtCompleteConnectPort**.

A client communication port does not have a queue for Connection messages. It has a small queue for Request messages that will be used to queue lost reply messages. The pointer to the connection port object points to the connection port that the connection request came in on. The pointer to the communication port object points to the server communication port. The context value is set to NULL.

The purpose of the optional memory section associated with a port is to optimize message passing. Since the maximum size of a message is fixed at connection port creation time, the intent is that for messages that are too big to fit in the message queue, the data is placed in the section object and the address of the data is passed via the message queue. The LPC mechanism has no knowledge of the format of the section object, it is only concerned with making the section visible to both the client and server processes whenever a connection is established. The memory is managed by the process that created the section. So the client process is solely responsible for managing the memory in the section object associated with the client communication port. Likewise the server process is solely responsible for managing the memory associated with the server communication port. The server will only need to provide a section object if it needs to perform callbacks to the client process.

1.2. Port Message Queues

Message queues are used for both Connection messages and Request messages. Each queue contains the following information:

- A linked list of messages that have been queued.
- A counting semaphore that is released whenever a message is placed in the queue.
- A serial number field that is used to generate unique message Ids as each message is placed in the queue.
- A maximum message size.

Port objects and queues for port objects are allocated out of non-paged pool memory. Pool quota for a pair of communication ports is charged to the client process that caused the communication ports to be created with a call to **NtConnectPort**. Pool quota for a connection port is charged to the server process. There is no quota charging done when a message is queued to a port, since the storage for messages is pre-allocated in the zone segment. The zone segment of the sender's port is used for request message allocation. The zone segment of the connection port is used for connection request message allocation.

The size of messages is fixed at the time the connection port is created. The size specified at connection port creation time is the maximum size of message the server is prepared to accept. The amount of data that is actually sent can be less than or equal to the message size. There is a limit on the size that can be specified on at port creation time, since space for queued messages is allocated out of pool memory in the system portion of the address space.

1.3. Port Creation

Port objects are created in two ways. A connection port is created by calling the **NtCreatePort** system service.

A pair of communication port objects is created whenever a server process accepts a connection request to a connection port. These are called the client port and the server port. When the client process sends a request to its port, it appears in the server port's request queue. When the server process sends a message to its port it appears in the client port's message queue. The client port and server port are bound together internally. The client port handle is valid only to the client process and the server port handle is valid only to the server process.

The following API calls are defined for ports:

NtCreatePort - used by server process to create a connection port
NtConnectPort - used by client processes to connect to a server process
NtListenPort - used by server process to listen for connection requests
NtAcceptConnectPort - used by server process to accept or reject a connection request
NtCompleteConnectPort - used by server process to complete the acceptance of a connection request
NtRequestPort - used to send a datagram message
NtRequestWaitReplyPort - used to send a message and wait for a reply
NtReplyPort - used to reply to a particular message
NtReplyWaitReplyPort - used to reply to a particular message and then wait for a reply to a previous message
NtReplyWaitReceivePort - used by server process to wait for a message from a client process
NtImpersonateClientOfPort - used by server thread to temporarily acquire the identifier set of a client thread

The following is an overview of how the API calls are used by the client and server processes:

Server Process:

Process initialization calls **NtCreatePort** to create a connection port object.

The main server thread then blocks in a call to **NtListenPort**. Whenever it returns it means that a new client process has called **NtConnectPort**. The server examines the connection request and calls **NtAcceptConnectPort** to either accept or reject the connection request. If it wants to accept, then it can also specify a section object for use when issuing callbacks to the client. **NtAcceptConnectPort** returns the server communication port handle to the server and causes the client process to return from its call to **NtConnectPort** with the client communication port handle. The server's main thread then goes back to the **NtListenPort** call to block waiting for another new client connection request.

One or more server request threads are blocked in **NtReplyWaitReceivePort**, waiting for a message to any of the connected server communication port objects. When **NtReplyWaitReceivePort** returns, the server will have a message, along with a unique identifier for the client thread that sent the message, a unique identifier for this particular message and the context value associated with the communication port that was the target of the request.

After processing a message, a server request thread will use the **NtReplyWaitReceivePort** call to send a reply back to the previous message received (based on the unique message identifier) and then wait for another request.

Client Process:

Process initialization will allocate a section object for passing information to the server process. **NtConnectPort** will then be called to create a port handle that is connected to the server process. **NtConnectPort** will return with an error if called incorrectly or if the server rejects the connection request.

Otherwise, when it returns, the client will have a valid communication port handle that is connected to the server process. **NtConnectPort** will block if there is no server thread waiting in a corresponding call to **NtListenPort**. A side effect of **NtConnectPort** is to make the client's section object visible in the server process' address space (most likely at a different virtual address than the client's). The base address of the client's section object in the server's address space will be returned to the client.

Whenever a thread in the client process wants to send a request to the server, it will call **NtRequestWaitReplyPort**, which will send the request and wait for a reply.

Callbacks:

The server can call back to the client process by using the **NtRequestWaitReplyPort** service, specifying a particular message identifier. This will unblock the client process, which is waiting for a reply to that message. The client process will examine the message and determine that it is a request instead of a reply. The client will process the request and send a reply using the **NtReplyWaitReplyPort** service. This service will send the reply back to the server and then block waiting for the original reply. If during the process of handling the callback request from the server, the client process calls **NtRequestWaitReplyPort** to send another request to the server, then the server will perform similar logic when it notices that it received a request instead of a reply.

2. Inter-Process Communication System Services

2.1. NtCreatePort

A server process can create a named connection port with the **NtCreatePort** service:

```
NTSTATUS
NtCreatePort(
   OUT PHANDLE PortHandle,
   IN POBJECT_ATTRIBUTES ObjectAttributes,
   IN ULONG MaxConnectionInfoLength,
   IN ULONG MaxMessageLength,
   IN BOOLEAN ReceiveAnyPort
   )
```

Parameters:

PortHandle - A pointer to a variable that will receive the connection port object handle value.

ObjectAttributes - A pointer to a structure that specifies the name of the object, an access control list (ACL) to be applied to the object, and a set of object attribute flags. *ObjectAttributes* Structure - See the Object Manager Specification for a detailed description of the fields in this structure. If the *ObjectName* field is not specified, then an unconnected communication port is created rather than a connection port. This is useful for sending and receiving messages between threads of a single process. If the *SecurityDescriptor* field is not specified, then any process will be allowed to access this port. The *Attributes* field must be zero or OBJ_CASE_INSENSITIVE, as none of the other standard values are relevant for this call. Connection ports cannot be inherited, are always placed in the process's handle table and are exclusive to the creating process.

MaxConnectionInfoLength - Specifies the maximum length of additional information that can be sent with a connection request via the **NtConnectPort** system service. The value of this parameter cannot exceed PORT_MAXIMUM_CONNECTINFO_LENGTH bytes.

MaxMessageLength - Specifies the maximum length of messages sent or received on communication ports created from this connection port. The value of this parameter cannot exceed PORT_MAXIMUM_MESSAGE_LENGTH bytes.

ReceiveAnyPort - A Boolean value that specifies if request messages queued to communication ports cloned from this connection port are to be queued to the connection port rather than the communication port.

<u>Return Value</u>: Status code that indicates whether or not the operation was successful.

A connection port is created with the name and attributes specified in the *ObjectAttributes* structure. A handle to the connection port object is returned in the location pointed to by the *PortHandle* parameter. The returned handle can then be used to listen for connection requests to that port name, using the **NtListenPort** service.

The standard object architecture defined desired access parameter is not necessary since this service can only create a new port, not access an existing port.

A named connection port cannot be used to send and receive messages. It is only valid as a parameter to the **NtListenPort** service.

An unnamed connection port can be used to send and receive messages within the process that created it.

The following errors can be returned by this function:

- STATUS_INVALID_PARAMETER
- STATUS_INVALID_PORT_ATTRIBUTES
- STATUS_OBJECT_PATH_INVALID
- STATUS_OBJECT_PATH_NOT_FOUND
- STATUS_OBJECT_PATH_SYNTAX_INVALID
- STATUS_OBJECT_NAME_INVALID
- STATUS_OBJECT_NAME_COLLISION
- STATUS_NO_MEMORY

2.2. NtConnectPort

A client process can connect to a server process by name using the **NtConnectPort** service:

NTSTATUS
NtConnectPort(
    OUT PHANDLE *PortHandle*,
    IN PSTRING *PortName*,
    IN PSECURITY_QUALITY_OF_SERVICE *SecurityQos*,
    IN ULONG *PortAttributes*,
    IN OUT PPORT_VIEW *ClientView* **OPTIONAL**,
    OUT PREMOTE_PORT_VIEW *ServerView* **OPTIONAL**,
    OUT PULONG *MaxMessageLength* **OPTIONAL**,
    IN OUT PVOID *ConnectionInformation* **OPTIONAL**,
    IN OUT PULONG *ConnectionInformationLength* **OPTIONAL**,
    IN BOOLEAN *Alertable*,
    IN PTIME *Timeout* **OPTIONAL**
)

Parameters:

PortHandle - A pointer to a variable that will receive the client communication port object handle value.

PortName - A pointer to a port name string. The form of the name is [\name...\name]\port_name.

SecurityQos - A pointer to security quality of service information to be applied to the server on the client's behalf.

PortAttributes - A set of flags that control the behavior of this port.
PortAttributes Flags: (none yet defined.)

ClientView - An optional pointer to a structure that specifies the section that all client threads will use to send messages to the server.

ClientView Structure:

ULONG *Length* - Specifies the size of this data structure in bytes.

HANDLE *SectionHandle* - Specifies an open handle to a section object.

ULONG *SectionOffset* - Specifies a field that will receive the actual offset, in bytes, from the start of the section. The initial value of this parameter specifies the byte offset within the section that the client's view is based. The value is rounded down to the next host page size boundary.

ULONG *ViewSize* - Specifies a field that will receive the actual size, in bytes, of the view. If the value of this parameter is zero, then the client's view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise, the initial value of this parameter specifies the size, in bytes, of the client's view and is rounded up to the next host page size boundary.

PVOID *ViewBase* - Specifies a field that will receive the base address of the section in the client's address space.

PVOID *ViewRemoteBase* - Specifies a field that will receive the base address of the client's section in the server's address space. Used to generate pointers that are meaningful to the server.

ServerView - An optional pointer to a structure that will receive information about the server process' view in the client's address space. The client process can use this information to validate pointers it receives from the server process.

ServerView Structure:

ULONG *Length* - Specifies the size of this data structure in bytes.

PVOID *ViewBase* - Specifies a field that will receive the base address of the server's section in the client's address space.

ULONG *ViewSize* - Specifies a field that will receive the size, in bytes, of the server's view in the client's address space. If this field is zero, then server has no view in the client's address space.

MaxMessageLength - An optional pointer to a variable that will receive the maximum length of messages that can be sent to the server. The value of this parameter will be equal to or greater than the value specified for the *MaxMessageLength* parameter to **NtCreatePort**. It might be greater to allow for optimal use of the memory associated with a port object.

ConnectionInformation - An optional pointer to uninterpreted data. This data is intended for clients to pass package, version and protocol identification information to the server to allow the server to determine if it can satisfy the client before accepting the connection. Upon return to the client, the *ConnectionInformation* data block contains any information passed back from the server by its call to the **NtAcceptConnectPort** service. The output data overwrites the input data.

ConnectionInformationLength - Pointer to the length of the *ConnectionInformation* data block. The output value is the length of the data stored in the *ConnectionInformation* data block by the server's call to the **NtAcceptConnectPort** service. This parameter is **OPTIONAL** only if the *ConnectionInformation* parameter is null, otherwise it is required.

Alertable - A Boolean value that specifies if the wait is user mode alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which any wait is to be completed. If not specified then the service will wait indefinitely.

Return Value: Status code that indicates whether or not the operation was successful.

The *PortName* parameter specifies the name of the server port to connect to. It must correspond to an object name specified on a call to **NtCreatePort**. The service sends a connection request to the server thread that is listening for them with the **NtListenPort** service. The client thread then blocks until a server thread receives the connection request and responds with a call to the **NtAcceptConnectPort** service. The server thread receives the ID of the client thread, along with any information passed via the *ConnectionInformation* parameter. The server thread then decides to either accept or reject the connection request.

The server communicates the acceptance or rejection with the **NtAcceptConnectPort** service. The server can pass back data to the client about the acceptance or rejection via the *ConnectionInformation* data block.

If the server accepts the connection request, then the client receives a communication port object in the location pointed to by the *PortHandle* parameter. This object handle has no name associated with it and is private to the client process (i.e. it cannot be inherited by a child process). The client uses the handle to send and receive messages to/from the server process using the **NtRequestWaitReplyPort** service.

If the *ClientView* parameter was specified, then the section handle is examined. If it is a valid section handle, then the portion of the section described by the *SectionOffset* and *ViewSize* fields will be mapped into both the client and server process' address spaces. The address in client address space will be returned in the *ViewBase* field. The address in the server address space will be returned in the *ViewRemoteBase* field. The actual offset and size used to map the section will be returned in the *SectionOffset* and *ViewSize* fields. Since the LPC system services do not explicitly manage the memory described by the view section, it is up to the caller to insure that the memory is committed prior to being referenced.

A client can control how the server gets to use its security attributes (IDs, privileges, et cetera) at port connection time. This is done by specifying security quality of service information using the *SecurityQos* parameter.

If the server rejects the connection request, then no communication port object handle is returned, and the return status indicates an error occurred. The server may optionally return information in the *ConnectionInformation* data block giving the reason the connection requests was rejected.

If the *PortName* does not exist, or the client process does not have sufficient access rights then the returned status will indicate that the port was not found.

The following errors can be returned by this function:

- STATUS_INVALID_PARAMETER
- STATUS_INVALID_PORT_ATTRIBUTES
- STATUS_OBJECT_PATH_INVALID

- STATUS_OBJECT_PATH_NOT_FOUND
- STATUS_OBJECT_PATH_SYNTAX_INVALID
- STATUS_OBJECT_NAME_INVALID
- STATUS_OBJECT_NAME_NOT_FOUND
- STATUS_ACCESS_DENIED
- STATUS_PORT_CONNECTION_REFUSED
- STATUS_INVALID_PORT_HANDLE
- STATUS_NO_MEMORY

2.3. NtListenPort

A server thread can listen for connection requests from client threads using the **NtListenPort** service:

NTSTATUS
NtListenPort(
  IN HANDLE *PortHandle*,
  OUT PCONNECTION_REQUEST *ConnectionRequest*,
  OUT PVOID *ConnectionInformation* **OPTIONAL**,
  IN OUT PULONG *ConnectionInformationLength* **OPTIONAL**,
  IN BOOLEAN *Alertable*,
  IN PTIME *Timeout* **OPTIONAL**
)

Parameters:

PortHandle - Specifies the connection port to listen for connection requests to.

ConnectionRequest - Pointer to a structure that describes the connection request the client is making:
  ConnectionRequest Structure:
  ULONG *Length* - Specifies the size of this data structure in bytes.
  CLIENT_ID *ClientId* - A structure that contains the client identifier of the thread that sent the request.
    ClientId Structure:
    ULONG *UniqueProcessId* - A unique value for each process in the system.

ULONG *UniqueThreadId* - A unique value for each thread in the system.
ULONG *RequestId* - A unique value that identifies this connection request.
ULONG *PortAttributes* - Specifies the value of the *PortAttributes* parameter that the client specified on the **NtConnectPort** call.
ULONG *ClientViewSize* - Specifies the value of the *ViewSize* field of the *ClientView* parameter that the client specified on the **NtConnectPort** call. Allows the server to prevent clients from consuming an unreasonable amount of the server's address space.

ConnectionInformation - An optional pointer to uninterpreted data from the corresponding call to **NtConnectPort**. This data is intended for clients to pass package, version and protocol identification to the server to allow it to determine if it can satisfy the client before accepting the connection.

ConnectionInformationLength - A pointer to the maximum length of the *ConnectionInformation* data block. The output value is the actual length of data stored in the *ConnectionInformation* data block. This parameter is **OPTIONAL** only if the *ConnectionInformation* parameter is NULL, otherwise it is required.

Alertable - A Boolean value that specifies if the wait is user mode alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which any wait is to be completed. If not specified then the service will wait indefinitely.

Return Value: Status code that indicates whether or not the operation was successful.

This call will return each time a client thread makes a call to the **NtConnectPort** service with a name that matches the name of the connection port object specified by the *PortHandle* parameter. Upon return, the location pointed to by the *ConnectionRequest* parameter will contain information about the connection request. The contents of this data structure must be passed to the **NtAcceptConnectPort** service to either accept or reject the connection request.

If the *ConnectionInformation* parameter is specified, then it will receive the corresponding data the client specified with the *ConnectionInformation* parameter of the **NtConnectPort** service. The *ConnectionInformationLength* parameter specifies the amount of data that can be received and its output value is the actual amount of data received.

The server process can examine the *ConnectionRequest* and *ConnectionInformation* data structures to determine whether or not to accept the connection request. In either case the server process must respond with a call to the **NtAcceptConnectPort** service to communicate the acceptance or rejection to the client thread.

If the server accepts a connection request, then it must also call the **NtCompleteConnectPort** service to actually release the client process from its wait inside **NtConnectPort**. The reason for a two stage acceptance is to allow the server process to save away in its internal data structure that describes a connection to a client, the OUT parameters returned by the **NtAcceptConnectPort** service.

A server process can maintain multiple threads waiting for connection requests in the **NtListenPort** service, although it is anticipated that only one thread will be used in most cases.

2.4. NtAcceptConnectPort

A server process can accept or reject a client connection request using the **NtAcceptConnectPort** service:

```
NTSTATUS
NtAcceptConnectPort(
   OUT PHANDLE PortHandle,
   IN PVOID PortContext,
   IN PCONNECTION_REQUEST ConnectionRequest,
   IN BOOLEAN AcceptConnection,
   IN BOOLEAN ReceiveThisPort,
   IN OUT PPORT_VIEW ServerView OPTIONAL,
   OUT PREMOTE_PORT_VIEW ClientView OPTIONAL,
   IN PVOID ConnectionInformation OPTIONAL,
   IN ULONG ConnectionInformationLength OPTIONAL
   )
```

Parameters:

PortHandle - A pointer to a variable that will receive the server communication port object handle value.

PortContext - A pointer value that is saved in the server communication port structure created by this service. This value is not interpreted by the system, but is returned by the **NtReplyWaitReceive** service whenever a message is received from the server communication port created by this call.

ConnectionRequest - A pointer to a structure that describes the connection request being accepted or rejected:

> *ConnectionRequest* Structure:
>
> **ULONG** *Length* - Specifies the size of this data structure in bytes.
>
> **CLIENT_ID** *ClientId* - Specifies a structure that contains the client identifier of the thread that sent the request.
>
> > *ClientId* Structure:
> >
> > **ULONG** *UniqueProcessId* - A unique value for each process in the system.
> >
> > **ULONG** *UniqueThreadId* - A unique value for each thread in the system.
>
> **ULONG** *RequestId* - A unique value that identifies the connection request being completed.
>
> **ULONG** *PortAttributes* - This field has no meaning for this service.
>
> **ULONG** *ClientViewSize* - This field has no meaning for this service.

AcceptConnection - Specifies a Boolean value which indicates where the connection request is being accepted or rejected. A value of TRUE means that the connection request is accepted and a server communication port handle will be created and connected to the client's communication port handle. A value of FALSE means that the connection request is not accepted.

ReceiveThisPort - Specifies a Boolean value which indicates if this server communication port should have its own receive queue or if all messages sent to this port should be queued in the receive queue of the communication port.

ServerView - A pointer to a structure that specifies the section that the server process will use to send messages back to the client process connected to this port.

> *ServerView* Structure:

ULONG *Length* - Specifies the size of this data structure in bytes.
HANDLE *SectionHandle* - Specifies an open handle to a section object.
ULONG *SectionOffset* - Specifies a field that will receive the actual offset, in bytes, from the start of the section. The initial value of this parameter specifies the byte offset within the section that the server's view is based. The value is rounded down to the next host page size boundary.
ULONG *ViewSize* - Specifies a field that will receive the actual size, in bytes, of the view. If the value of this parameter is zero, then the server's view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise, the initial value of this parameter specifies the size, in bytes, of the server's view and is rounded up to the next host page size boundary.
PVOID *ViewBase* - Specifies a field that will receive the base address of the section in the server's address space.
PVOID *ViewRemoteBase* - Specifies a field that will receive the base address of the server's section in the client's address space. Used to generate pointers that are meaningful to the client.

ClientView - An optional pointer to a structure that will receive information about the client process' view in the server's address space. The server process can use this information to validate pointers it receives from the client process.
ClientView Structure:
ULONG *Length* - Specifies the size of this data structure in bytes.
PVOID *ViewBase* - Specifies a field that will receive the base address of the client's section in the server's address space.
ULONG *ViewSize* - Specifies a field that will receive the size, in bytes, of the client's view in the server's address space. If this field is zero, then client has no view in the server's address space.

ConnectionInformation - An optional pointer to uninterpreted data that is to be returned to the caller of **NtConnectPort**.

ConnectionInformationLength - Specifies the length of the *ConnectionInformation* data block. This parameter is **OPTIONAL** only if the *ConnectionInformation* parameter is null, otherwise it is required. The length cannot be greater than the length received from the client.

<u>Return Value</u>: Status code that indicates whether or not the operation was successful.

The *ConnectionRequest* parameter must specify a connection request returned by a previous call to the **NtListenPort** service. This service will either complete the connection if the *AcceptConnection* parameter is TRUE, or reject the connection request if the *AcceptConnection* parameter is FALSE.

In either case, if the *ConnectionInformation* parameter is specified, then any data it points to is returned to the client process via the *ConnectionInformation* parameter that was specified on the **NtConnectPort** service call. The amount of data returned to the client is specified by the *ConnectionInformationLength* parameter.

If the connection request is accepted, then two communication port objects will be created and connected together. One will be inserted in the client process' handle table and returned to the client via the *PortHandle* parameter it specified on the **NtConnectPort** service. The other will be inserted in the server process' handle table and returned via the *PortHandle* parameter specified on the **NtAcceptConnectPort** service. In addition all of the server's communication ports will be linked together with the head of the queue in the connection port object. This allows a server thread to wait for any message to any of its connected communication ports.

If the connection request is accepted, and the *ServerView* parameter was specified, then the section handle is examined. If it is valid, then the portion of the section described by the *SectionOffset* and *ViewSize* fields will be mapped into both the client and server process address spaces. The address in server's address space will be returned in the *ViewBase* field. The address in the client's address space will be returned in the *ViewRemoteBase* field. The actual offset and size used to map the section will be returned in the *SectionOffset* and *ViewSize* fields. Since the LPC system services do not explicitly manage the memory described by the view section, it is up to the caller to insure that the memory is committed prior to being referenced.

If the server accepts a connection request, then it must also call the **NtCompleteConnectPort** service to actually release the client process from its wait inside **NtConnectPort**. The reason for a two stage acceptance is to allow the server process to save away in its internal data structure that describes a connection to a client, the OUT parameters returned by the **NtAcceptConnectPort** service, *PortHandle*, *ClientView* and *ServerView*.

Communication port objects are temporary objects that have no names and cannot be inherited. When either the client or server process calls the **NtClose** service for a communication port, the port will be deleted since there can never be more than one outstanding handle for each communication port. The port object type specific delete procedure will then be invoked. This delete procedure will examine the communication port, and if it is connected to a server communication port, it will queue an LPC_PORT_CLOSED datagram to the server's message queue. This will allow server process to notice when a port becomes disconnected, either because of an explicit call to **NtClose** or an implicit call due to process termination.

2.5. NtCompleteConnectPort

After accepting a connection with the **NtAcceptConnectPort** service, a server process can release a client process from its wait inside of the **NtConnectPort** service with the **NtCompleteConnectPort** service:

NTSTATUS
NtCompleteConnectPort(
   IN HANDLE *PortHandle*
)

Parameters:

PortHandle - Specifies a server communication port returned by the **NtAcceptConnectPort** Service.

Return Value: Status code that indicates whether or not the operation was successful.

A server process must call this service whenever it accepts a connection request with the **NtAcceptConnectPort** service. This service will then satisfy the wait of the client process inside of the **NtConnectPort** service.

The reason for a two stage acceptance is to allow the server process to save away in its internal data structure that describes a connection to a client, the OUT parameters returned by the **NtAcceptConnectPort** service, *PortHandle*, *ClientView* and *ServerView*.

2.6. Port Message Structure

A port message is a variable size structure that can be placed in the message queue of a port. It contains information filled in by the system that uniquely identifies the thread that sent the message, along with a serial number that uniquely identifies the message so that reply messages can be matched up with the message being replied to. The type declaration for a port message only specifies the fixed size header that is associated with all messages. The application specific data associated with a message should immediately follow the fixed size header.

```
typedef struct _PORT_MESSAGE {
   CSHORT DataLength;
   CSHORT TotalLength;
   CSHORT Type;
   CSHORT MapInfoOffset;
   CLIENT_ID ClientId;
   ULONG MessageId;
} PORT_MESSAGE, *PPORT_MESSAGE;
```

PORT_MESSAGE Structure:

DataLength - Specifies the size, in bytes, of the data portion of this message. Must be less than the *TotalLength* field.

TotalLength - Specifies the total size of this data structure in bytes. The maximum size of the message is limited by the value of the *MaxMessageLength* parameter to the **NtCreatePort** service.

Type - For messages being sent, this field is filled in by the system service that sends the message. For messages being received, this field identifies the source of the message.

Type Values:

LPC_DATAGRAM - Indicates that this is a request message generated by a call to **NtRequestPort**. No reply is expected.
LPC_REQUEST - Indicates that this is a request message generated by a call to **NtRequestWaitReplyPort**. The sender is expecting a reply.
LPC_REPLY - Indicates that this is a reply to a previous request message that was generated by a call to **NtReplyPort** or **NtReplyWaitReplyPort** or **NtReplyWaitReceivePort**.
LPC_LOST_REPLY - Indicates that the recipient of a message was unable to reply to a message it received. The message data buffer contains the reply that could not be delivered. This message is queue to a communication port whenever a reply was attempted and the sending thread was not waiting for a reply.
LPC_PORT_CLOSED - Indicates that the client thread has closed its port and therefore become disconnected from the server. This message is sent from the client process to the server when the port is closed.
LPC_CLIENT_DIED - Indicates that the client thread has died. The *ClientId* field contains the ID of the client thread. The message data buffer contains the thread termination code. This datagram is queued to each port associated with the dying thread via the **NtRegisterThreadTerminationPort** system service.
LPC_EXCEPTION - Indicates that an unhandled exception occurred in the client thread. The message data buffer contains the number of the exception. The client thread is blocked waiting for a reply to this message. The reply states whether or not the server process handled the exception. This message is sent to the *ExceptionPort* associated with a process as specified via the call to the **NtCreateProcess** system service.
LPC_DEBUG_EVENT - Indicates that this is a debugger event. This message is sent to the *DebugPort* associated with a process as specified via the call to the **NtCreateProcess** system service.

MapInfoOffset - Offset within the message structure of a PORT_MAP_INFORMATION data structure. If zero, then there is no PORT_MAP_INFORMATION data structure associated with this message. Must be a valid offset within the message buffer.

ClientId - a structure that contains the client identifier of the thread that sent the message.

ClientId Structure:
ULONG *UniqueProcessId* - A unique value for each process in the system.
ULONG *UniqueThreadId* - A unique value for each thread in the system.

MessageId - unique value that identifies this message.

The *MessageType*, *ClientId* and *MessageId* fields are filled in by the system service that sends the messages. For messages being received they identify the source of the message. For messages being replied to the *ClientId* and *MessageId* fields are used to determine who is waiting for the reply.

Note that the actual message structure contains two unions to allow for the code generated by the stub compiler to efficiently initialize the first four fields of a message using just two store instructions.

2.7. Port Map Information Structure

In order to support passing large pieces of data efficiently, the LPC mechanism supports the ability to pass objects that lie on a page boundary and whose size is a multiple of the page size. The *MapInfoOffset* field above enables this feature and causes one or more page aligned regions in the sender's address space to be doubly mapped into the receiver's address space for the duration of the message.

If the *MapInfoOffset* field is not zero, then it is an offset within the message data buffer of a **PORT_MAP_INFORMATION** data structure.

```
typedef struct _PORT_MAP_INFORMATION {
   ULONG CountMapEntries;
   PORT_MAP_ENTRY MapEntries[];
} PORT_MAP_INFORMATION, *PPORT_MAP_INFORMATION;
```

PORT_MAP_INFORMATION Structure:

CountMapEntries - The number of entries in the *MapEntries* array.

MapEntries - Specifies an array of **PORT_MAP_ENTRY** structures.
    PORT_MAP_ENTRY Structure:

PVOID *Base* - Specifies the address of first page in a region of the sender's address space that is to be doubly mapped into the receiver's address space. This address must be aligned on a page boundary.
ULONG *Size* - Specifies the number of bytes to map. This value must be a multiple of the page size; if zero, then no pages are mapped.

The address of the Port Map Information structure can be computed as follows:

PPORT_MAP_INFORMATION *MapInfo* =
    (**PPORT_MAP_INFORMATION**)((**PCH**)*PortMsg* + *PortMsg*->*MapInfoOffset*);

The mapping occurs when the message is received by the target thread. The mapping is destroyed when the target thread replies to the message.

2.8. NtRequestPort

A client and server process can send datagram messages using the **NtRequestPort** service:

NTSTATUS
NtRequestPort(
   IN HANDLE *PortHandle*,
   IN PPORT_MESSAGE *RequestMessage*
)

Parameters:

PortHandle - Specifies the handle of the communication port to send the request message to.

RequestMessage - Specifies a pointer to the request message.

Return Value: Status code that indicates whether or not the operation was successful.

The *Type* field of the message is set to LPC_DATAGRAM by the service.

The message pointed to by the *RequestMessage* parameter is placed in the message queue of the port connected to the communication port specified by the *PortHandle* parameter. This service returns an error if *PortHandle* is invalid or if the *MessageId* field of the *PortMessage* structure is non-zero.

2.9. NtRequestWaitReplyPort

A client and server process can send a request and wait for a reply using the **NtRequestWaitReplyPort** service:

NTSTATUS
NtRequestWaitReplyPort(
   IN HANDLE *PortHandle*,
   IN PPORT_MESSAGE *RequestMessage*,
   OUT PPORT_MESSAGE *ReplyMessage*,
   IN BOOLEAN *Alertable*,
   IN PTIME *Timeout* **OPTIONAL**
)

Parameters:

PortHandle - Specifies the handle of the communication port to send the request message to.

RequestMessage - Specifies a pointer to a request message to send.

ReplyMessage - Specifies the address of a variable that will receive the reply message. This parameter may point to the same buffer as the *RequestMessage* parameter.

Alertable - A Boolean value that specifies if the wait is user mode alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which any wait is to be completed. If not specified then the service will wait indefinitely.

Return Value: Status code that indicates whether or not the operation was successful.

If the *Type* field of the *RequestMessage* structure is equal to LPC_REQUEST, then this is identified as a callback request. The *ClientId* and *MessageId* fields are used to identify the thread that is waiting for a reply. This thread is unblocked and the current thread that called this service then blocks waiting for a reply.

The Type field of the message is set to LPC_REQUEST by the service. Otherwise the Type field of the message must be zero and it will be set to LPC_REQUEST by the service. The message pointed to by the *RequestMessage* parameter is placed in the message queue of the port connected to the communication port specified by the *PortHandle* parameter. This service returns an error if *PortHandle* is invalid. The calling thread then blocks waiting for a reply.

The reply message is stored in the location pointed to by the *ReplyMessage* parameter. The *ClientId*, *MessageId* and message type fields will be filled in by the service.

The *Timeout* parameter is used as the timeout value when waiting for a reply. If the wait times out, then an error code is returned.

2.10. NtReplyPort

A client and server process can send a reply to a previous request message with the **NtReplyPort** service:

```
NTSTATUS
NtReplyPort(
   IN HANDLE PortHandle,
   IN PPORT_MESSAGE ReplyMessage,
   IN LPC_REPLY_BOOST ReplyBoost OPTIONAL
   )
```

<u>Parameters</u>:

PortHandle - Specifies the handle of the communication port that the original message was received from.

ReplyMessage - Specifies a pointer to the reply message to be sent. The *ClientId* and *MessageId* fields determine which thread will get the reply.

ReplyBoost - This optional parameter specifies the amount of priority boost the thread waiting for the reply message is to receive. This parameter may only be specified if the calling thread has the SE_LPC_REPLY_BOOST_PRIVILEGE privilege. This parameter and the privilege test are ignored if the *ReplyMessage* parameter is not specified.

ReplyBoost Values:

NoLpcReplyBoost - no priority boost will be given to the thread waiting for the reply. No privilege is required to specify this value.

LowLpcReplyBoost - a small priority boost will be given to the thread waiting for the reply.

MediumLpcReplyBoost - a medium priority boost will be given to the thread waiting for the reply.

HighLpcReplyBoost - a large priority boost will be given to the thread waiting for the reply.

<u>Return Value</u>: Status code that indicates whether or not the operation was successful.

The Type field of the message is set to LPC_REPLY by the service. If the *MapInfoOffset* field of the reply message is non-zero, then the PORT_MAP_INFORMATION structure it points to will be processed and the relevant pages in the caller's address space will be unmapped.

The *ClientId* and *MessageId* fields of the *ReplyMessage* structure are used to identify the thread waiting for this reply. If the target thread is in fact waiting for this reply message, then the reply message is copied into the thread's message buffer and the thread's wait is satisfied.

If the thread is not waiting for a reply or is waiting for a reply to some other *MessageId*, then the message is placed in the message queue of the port that is connected to the communication port specified by the *PortHandle* parameter and the *Type* field of the message is set to LPC_LOST_REPLY.

2.11. NtReplyWaitReplyPort

A client and server process can send a reply to a previous message and block waiting for a reply using the **NtReplyWaitReplyPort** service:

NTSTATUS
NtReplyWaitReplyPort(
  IN HANDLE *PortHandle,*
  IN OUT PPORT_MESSAGE *ReplyMessage,*
  IN LPC_REPLY_BOOST *ReplyBoost* **OPTIONAL,**
  IN BOOLEAN *Alertable,*
  IN PTIME *Timeout* **OPTIONAL**
)

<u>Parameters</u>:

PortHandle - Specifies the handle of the communication port that the original message was received from.

ReplyMessage - Specifies a pointer to the reply message to be sent. The *ClientId* and *MessageId* fields determine which thread will get the reply. This buffer also receives any reply that comes back from the wait.

ReplyBoost - This optional parameter specifies the amount of priority boost the thread waiting for the reply message is to receive. This parameter may only be specified if the calling thread has the SE_LPC_REPLY_BOOST_PRIVILEGE privilege. This parameter and the privilege test are ignored if the *ReplyMessage* parameter is not specified.

ReplyBoost Values:

NoLpcReplyBoost - no priority boost will be given to the thread waiting for the reply. No privilege is required to specify this value.

LowLpcReplyBoost - a small priority boost will be given to the thread waiting for the reply.

MediumLpcReplyBoost - a medium priority boost will be given to the thread waiting for the reply.

HighLpcReplyBoost - a large priority boost will be given to the thread waiting for the reply.

Alertable - A Boolean value that specifies if the wait is user mode alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which any wait is to be completed. If not specified then the service will wait indefinitely.

Return Value: Status code that indicates whether or not the operation was successful.

This service works the same as **NtReplyPort**, except that after delivering the reply message, it blocks waiting for a reply to a previous message. When the reply is received, it will be placed in the location specified by the *ReplyMessage* parameter.

2.12. NtReplyWaitReceivePort

A client and server process can receive messages using the **NtReplyWaitReceivePort** service:

```
NTSTATUS
NtReplyWaitReceivePort(
   IN HANDLE PortHandle,
   OUT PVOID *PortContext OPTIONAL,
   IN PPORT_MESSAGE ReplyMessage OPTIONAL,
   IN LPC_REPLY_BOOST ReplyBoost OPTIONAL,
   OUT PPORT_MESSAGE ReceiveMessage,
   IN BOOLEAN Alertable,
   IN PTIME Timeout OPTIONAL
   )
```

Parameters:

PortHandle - Specifies the handle of the connection or communication port to do the receive from.

PortContext - Specifies an optional pointer to a variable that is to receive the context value associated with the communication port that the message is being received from. This context variable was specified on the call to the **NtAcceptConnectPort** service.

ReplyMessage - This optional parameter specifies the address of a reply message to be sent. The *ClientId* and *MessageId* fields determine which thread will get the reply. See description of **NtReplyPort** for how the reply is sent. The reply is sent before blocking for the receive.

ReplyBoost - This optional parameter specifies the amount of priority boost the thread waiting for the reply message is to receive. This parameter may only be specified if the calling thread has the SE_LPC_REPLY_BOOST_PRIVILEGE privilege. This parameter and the privilege test are ignored if the *ReplyMessage* parameter is not specified.

ReplyBoost Values:

NoLpcReplyBoost - no priority boost will be given to the thread waiting for the reply. No privilege is required to specify this value.

LowLpcReplyBoost - a small priority boost will be given to the thread waiting for the reply.

MediumLpcReplyBoost - a medium priority boost will be given to the thread waiting for the reply.

HighLpcReplyBoost - a large priority boost will be given to the thread waiting for the reply.

ReceiveMessage - Specifies the address of a variable to receive the message.

Alertable - A Boolean value that specifies if the wait is user mode alertable.

Timeout - An optional pointer to timeout value that specifies the absolute or relative time over which any wait is to be completed. If not specified then the service will wait indefinitely.

Return Value: Status code that indicates whether or not the operation was successful.

If the *ReplyMessage* parameter is specified, then the reply will be sent using **NtReplyPort**.

If the *PortHandle* parameter specifies a connection port, then the receive will return whenever a message is sent to a server communication port that does not have its own receive queue and the message is therefore queued to the receive queue of the connection port.

If the *PortHandle* parameter specifies a server communication port that does not have a receive queue, then behaves as if the associated connection port handle was specified. Otherwise the receive will return whenever message is placed in the receive queue associated with the server communication port.

The received message will be returned in the variable specified by the *ReceiveMessage* parameter. If the *MapInfoOffset* field of the reply message is non-zero, then the PORT_MAP_INFORMATION structure it points to will be processed and the relevant pages will be mapped into the caller's address space. The service returns an error if there is not enough room in the caller's address space to accommodate the mappings.

2.13. NtImpersonateClientOfPort

A server process can utilize the security context of a client process with the **NtImpersonateClientOfPort** service:

NTSTATUS
NtImpersonateClientOfPort(
   IN HANDLE *PortHandle*,
   IN PPORT_MESSAGE *Message*
)

Parameters:

PortHandle - Specifies the handle of the communication port that the message was received from.

Message - Specifies an address of a message that was received from the client that is to be impersonated. The *ClientId* field of the message identifies the client thread that is to be impersonated. The client thread must be waiting for a reply to the message in order to impersonate the client.

Return Value: Status code that indicates whether or not the operation was successful.

This service establishes an impersonation token for the calling thread. The impersonation token corresponds to the context provided by the port client. The client must currently be waiting for a reply to the specified message.

This service returns an error status code if the client thread is not waiting for a reply to the message. The security quality of service parameters specified by the client upon connection dictate what use the server will have of the client's security context.

For complicated or extended impersonation needs, the server may open a copy of the client's token (using **NtOpenThreadToken()**). This must be done while impersonating the client.