RPC PDU Encodings

This chapter specifies the encodings of the Protocol Data Units (PDUs) used by the connectionless and connection-oriented RPC protocols. The first section provides common information for the two protocols. Subsequent sections provide connectionless and connection-oriented protocol-specific information.

RPC Protocol Data Units lists the PDUs that are specified.

Generic PDU Structure

An RPC PDU contains up to three parts:

- A PDU header that contains protocol control information. A header is present in all PDUs.
- A PDU body bgcolor="#FFFFFF" that contains data. For example, the body of a **request** or **response** PDU contains data representing the input or output parameters for an operation. A body bgcolor="#FFFFFF" is present only in some types of PDUs.
- An authentication verifier that contains data specific to an authentication protocol. For example, an authentication protocol may ensure the integrity of a packet via inclusion of an encrypted checksum in the authentication verifier. The presence of an authentication verifier depends on the PDU type and whether authentication is being used.

PDU Type	Protocol	Type Value
request	**CO/CL**	**0**
ping	CL	1
response	CO/CL	2
fault	CO/CL	3
working	CL	4
nocall	CL	5
reject	CL	6
ack	CL	7
cl_cancel	CL	8
fack	CL	9
cancel_ack	CL	10
bind	CO	11
bind_ack	CO	12
bind_nak	CO	13
alter_context	CO	14
alter_context_resp	CO	15
shutdown	CO	17
co_cancel	CO	18
orphaned	CO	19

Table: RPC Protocol Data Units

Encoding Conventions

The encodings are provided here as IDL data type declarations. The actual declarations of PDU data types are implementation-dependent.

The run-time system treats PDU headers as byte streams that are encoded according to the Network Data Representation (NDR, see Transfer Syntax NDR) encoding rules. In PDUs containing stub data, the stub data may be encoded according to any of the negotiated presentation syntaxes. PDU-specific header fields identify the stub data presentation syntax. The run-time system identifies the presentation syntax from the PDU-specific header fields and interprets the stub data accordingly.

In order to encode and decode the header fields, implementations must support the following subset of

NDR:

- integers: 1, 2 and 4 octet unsigned
- octet string
- record constructor for the preceding types
- conformant array constructor.

Implementations must follow the rule that, beginning at the first octet of the PDU (or the first octet of the optional data field), each field is encoded in the order specified without any additional information such as type, length or padding.

Alignment

For all the PDUs, the scalar header fields are aligned **[0 MOD min(8,sizeof(***field***))]**. For PDUs that contain stub data, the header is padded, if necessary, with zeros to an integral multiple of 8 octets in length. This allows the stub data to assume it will always begin with **0 MOD 8** alignment. Padding, where necessary, is included in the type declarations. The **align(***n***)** function, which appears in some connection-oriented PDU declarations, identifies a function that returns the number of padding octets to force alignment to a natural multiple of *n*. Implementations should ensure that each message in memory begins on an 8 octet alignment boundary in order to preserve the natural alignment.

Refer to Security for alignment requirements for the optional authentication verifiers.

Common Fields

Header encodings differ between connectionless and connection-oriented PDUs. However, certain fields use common sets of values with a consistent interpretation across the two protocols. These values are specified in the following sections.

PDU Types

Both connectionless and connection-oriented PDU headers contain a 1-byte field that identifies the PDU type. The values are shown in the column marked **Type Value** in RPC Protocol Data Units .

Protocol Version Numbers

Both connectionless and connection-oriented PDU headers contain version number fields that indicate the RPC protocol version. The connectionless headers contain a single version field, while the connection-oriented headers contain both a major and a minor version field. Connectionless and connection-oriented protocol version numbers vary independently, so that a given version number is not necessarily unique to one of the protocols.

The version numbers for the protocols specified in this chapter are as follows:

- The major version number for the connection-oriented protocol is 5.

- The minor version numbers for the connection-oriented protocol are either 0 (zero) or 1.

  The connection-oriented minor version number of 1 supports security credentials larger than 1400 bytes. The state machine automatically falls back to the minor version number 0 (zero) style of security credentials should the server return an RPC version mismatch error.

- The version number for the connectionless protocol is 4.

Data Representation Format Labels

Both connectionless and connection-oriented PDU headers contain an NDR data representation format label that identifies the formats used by the sender of a PDU to represent data in the PDU header, and when the transfer syntax is NDR, in the PDU body bgcolor="#FFFFFF". (The representation of data in an authentication verifier is determined by the authentication protocol.) Transfer Syntax NDR describes the NDR data representation format label.

As defined by NDR, the format label consists of 4 bytes, although the fourth byte is currently unused. Only the first 3 bytes appear in RPC connectionless PDU headers. Connection-oriented PDUs include space for all four bytes of the format label.

NDR defines only one bit layout for the format label itself, so its format is the same in all PDUs.

Reject Status Codes

Both **reject** and connection-oriented **fault** PDUs contain a 32-bit field that indicates a server's reason for rejecting an RPC call request. This field is encoded as the body bgcolor="#FFFFFF" data of the **reject** PDU and as the **status** field of the connection-oriented **fault** PDU header. Reject Status Codes lists the possible values of this field.

Connectionless RPC PDUs

The RPC run-time system uses the connectionless PDUs for the client/server communications that are required by each remote procedure call over a connectionless transport. The following sections specify the encoding of each of the connectionless PDUs.

Connectionless PDU Structure

Connectionless PDUs consist of a header, body bgcolor="#FFFFFF" data and an optional authentication verifier. The PDU header has a fixed length of 80 bytes. The encoding is described in Header Encoding . The PDU body bgcolor="#FFFFFF" can be defined as an array of the IDL **byte** type. The length of the PDU body bgcolor="#FFFFFF" is specified in the PDU header.

The authentication verifier can be defined as an array of the IDL **byte** type. The length of the authentication verifier depends on the authentication protocol specified in the PDU header. An authentication verifier is present in a PDU only if the *auth_proto* header field is not 0 (zero).

The maximum size of an RPC connectionless PDU is the fixed header length (80 bytes) plus the maximum body bgcolor="#FFFFFF" length plus the length of the authentication verifier, which is determined by the authentication protocol. The Open Group DCE defines a lower bound on the size of a single PDU that all implementations must be able to receive, **MustRecvFragSize**. A client and server may subsequently negotiate a large PDU size during the course of their conversation. This negotiation may take place explicitly (by conveying the value in the body bgcolor="#FFFFFF" of a **fack** PDU; see The fack PDU), or implicitly (by presuming that a peer can receive packets as large as those it has transmitted).

If the input data in a request or the output data in a response exceed the maximum PDU body bgcolor="#FFFFFF" size, the RPC connectionless protocols fragment the request or response into several PDUs, called fragments. The fragment number can be reused for a given call if the fragment number space is exhausted. If fragment numbers wrap around and are reused, the implementation must assure that these are unambiguous (for example, the first 50% of fragments must have been acknowledged).

Requests that are broadcast must fit in one PDU, and on some connectionless transports they may be subject to further size limitations.

Header Encoding

The connectionless header can be defined as a structure in Interface Definition Language (IDL):

```
typedef struct {
    unsigned small rpc_vers = 4; /* RPC protocol major version (4 LSB only)*/
    unsigned small ptype;      /* Packet type (5 LSB only) */
    unsigned small flags1;     /* Packet flags */
    unsigned small flags2;     /* Packet flags */
    byte           drep[3];    /* Data representation format label */
    unsigned small serial_hi;  /* High byte of serial number */
    uuid_t         object;     /* Object identifier */
    uuid_t         if_id;      /* Interface identifier */
    uuid_t         act_id;     /* Activity identifier */
    unsigned long  server_boot;/* Server boot time */
    unsigned long  if_vers;    /* Interface version */
    unsigned long  seqnum;     /* Sequence number */
    unsigned short opnum;      /* Operation number */
    unsigned short ihint;      /* Interface hint */
    unsigned short ahint;      /* Activity hint */
    unsigned short len;        /* Length of packet body bgcolor="#FFFFFF" */
    unsigned short fragnum;    /* Fragment number */
    unsigned small auth_proto; /* Authentication protocol identifier*/
    unsigned small serial_lo;  /* Low byte of serial number */
} dc_rpc_cl_pkt_hdr_t;
```

The bit layout of each field in a PDU header is determined by the following:

- The IDL type of the field, as shown in the definition for **dc_rpc_cl_pkt_hdr_t**.
- The NDR representation for that IDL type, as specified in Transfer Syntax NDR .
- The NDR character, integer and floating-point formats used by the RPC implementation that is sending the PDU.

Protocol Version Number

The protocol version number is a non-negative integer that is encoded in the 4 least significant bits of the **rpc_vers** field. (The remaining bits are reserved.) This number is incremented at each new release of the protocol. The protocol version number allows implementations of several versions of RPC to coexist in a distributed environment. The PDU formats given here are for version 4.

PDU Type

The PDU type is a non-negative integer that is encoded in the 5 least significant bits of the **ptype** field. (The remaining bits are reserved.) The values for each type are shown in the column labelled **Type Value** in RPC Protocol Data Units .

Flags Fields

Each of the flags fields is an 8-bit integer that is composed of bit flags for use in protocol control.

This document currently specifies the use of 6 bits in the first set of PDU flags. The other bits are either reserved for use by implementations or reserved for future use, as indicated in Second Set of PDU Flags and Authentication Protocol Identifiers .

Some flags are meaningful only in PDUs that are sent from the client to the server. These flags are ignored in PDUs that are sent from the server to the client.

Second Set of PDU Flags lists the bit flags in the first set of PDU flags.

PDU Flag	Hex Value	Meaning
reserved_01	**01**	***Reserved for use by implementations.***
lastfrag	02	*Meaningful in either direction. If set, the PDU is the last fragment of a multi-PDU transmission.*
frag	04	*Meaningful in either direction. If set, the PDU is a fragment of a multi-PDU transmission.*
nofack	08	*Meaningful for fragments sent in either direction. If set, the receiver is not requested to send a* ***fack*** *PDU for the fragment. Otherwise, if not set, the receiver acknowledges the received PDU with a* ***fack*** *PDU. Note that both client and server may send* ***fack*** *PDUs independent of the status of this flag.*
maybe	10	*Meaningful only from client to server. If set, the PDU is for a maybe request.*
idempotent	20	*Meaningful only from client to server. If set, the PDU is for an idempotent request.*
broadcast	40	*Meaningful only from client to server. If set, the PDU is for a broadcast request.*
reserved_80	80	*Reserved for use by implementations.*

Table: The First Set of PDU Flags

Authentication Protocol Identifiers lists the bit flags in the second set of PDU flags.

PDU Flag	Hex Value	Meaning
reserved_01	**01**	***Reserved for use by implementations.***
cancel_pending	02	*Cancel pending at the call end.*
reserved_04	04	*Reserved for future use. Must be set to 0.*
reserved_08	08	*Reserved for future use. Must be set to 0.*
reserved_10	10	*Reserved for future use. Must be set to 0.*
reserved_20	20	*Reserved for future use. Must be set to 0.*
reserved_40	40	*Reserved for future use. Must be set to 0.*
reserved_80	80	*Reserved for future use. Must be set to 0.*

Table: Second Set of PDU Flags

Data Representation Format Label

The data representation format label is described in Transfer Syntax NDR . As defined by NDR, the format label consists of 4 bytes, although the fourth byte is currently unused. Only the first 3 bytes appear in connectionless PDUs.

Serial Number

The serial number is a 16-bit non-negative integer that identifies a transmission of a fragment.

The serial number is set to 0 (zero) when a remote procedure call is initiated, and is incremented after each time a fragment is sent or resent on behalf of that call.

In a **request** or **response** PDU that is part of a multi-PDU transmission, the serial number provides information to help determine the temporal order of fragment transmissions and retransmissions. In other types of PDUs, the serial number is meaningless.

The 2 bytes of the serial number do not occupy contiguous positions in the PDU header. The most significant byte follows the data representation format label. The least significant byte follows the authentication protocol number.

Object Identifier

The object identifier is a UUID that uniquely identifies the object on which a remote procedure call is operating. If the call does not operate on an object, this field contains the nil UUID.

The server uses the object UUID, the interface UUID, the interface version number, and the operation number to select the operation to execute on the client's behalf.

Interface Identifier

The interface identifier is a UUID that uniquely identifies the interface being called.

The server uses the object UUID, the interface UUID, the interface version number, and the operation number to select the operation to execute on the client's behalf.

Activity Identifier

The activity identifier is a UUID that uniquely identifies the client activity that is making a remote procedure call. The server can use the activity UUID as a communications key between it and the client.

Server Boot Time

The server boot time is a 32-bit non-negative integer that indicates the time at which the current instantiation of a server was booted; that is, the time at which the process in which the server is running was created, not the time at which the server host was booted. Server boot time is represented as time in seconds since 1 January 1970 and must increase with each boot of the server.

A server passes its boot time in all the PDUs that it sends to a client. The client passes back this value in all the PDUs that it subsequently sends to the same server. When a client sends its first PDU to a server, it does not know the server boot time, and it passes a value of 0 (zero). The server boot time field enables the RPC connectionless protocols to prevent nonidempotent operations from being executed more than once in the face of a server crash.

The protocol also allows for PDUs to be transmitted by the endpoint mapper on behalf of a server process; for example, the endpoint mapper may return a reject PDU upon receipt of a request sent to a server that is not currently running. In such cases, any PDUs transmitted by the endpoint mapper must carry a 0 boot time, to differentiate them from PDUs that might subsequently be received by the target server.

Interface Version

The interface version is a 32-bit non-negative integer that identifies the version number of the interface being called. This field allows servers to implement several versions of one interface.

The server uses the object UUID, the interface UUID, the interface version number, and the operation number to select the operation to execute on the client’s behalf.

Sequence Number

The sequence number is a 32-bit non-negative integer that identifies the remote procedure call that an activity is making.

Each remote procedure call invoked by an activity has a unique sequence number that is assigned when the call is initiated. All RPC connectionless PDUs sent on behalf of that particular call have the same sequence number, whether the PDUs are from client to server or from server to client.

When an activity initiates a new remote procedure call, it increases the sequence number, so that each subsequent call has a larger sequence number. Together, the activity UUID and the sequence number uniquely identify a remote procedure call.

Operation Number

The operation number is a 16-bit non-negative integer that identifies a particular operation within the interface being called.

Interface Hint

The interface hint is a 16-bit non-negative integer. Although an implementation can use this field for any purpose, its intended use is to allow a server to optimise lookups of information about interfaces.

Activity Hint

The activity hint is a 16-bit non-negative integer. Although an implementation can use this field for any purpose, its intended use is to allow a server to optimise lookups of information about the state of its communications with an activity.

PDU Body Length

The PDU body bgcolor="#FFFFFF" length is a 16-bit non-negative integer that indicates the length in bytes of the PDU body bgcolor="#FFFFFF". The maximum PDU body size is 65528 bytes. The alignment requirements for the PDU header (see Alignment) also apply to the PDU body bgcolor="#FFFFFF" data.

Fragment Number

The fragment number is a 16-bit non-negative integer that identifies a PDU within a multi-PDU transmission.

In a **request** or **response** PDU that is part of a multi-PDU transmission, the fragment number indicates the fragment that is being sent. The fragment number is incremented for each fragment. The first fragment is fragment 0, the second is fragment 1, and so on.

In a **fack** PDU and a **nocall** PDU with a body bgcolor="#FFFFFF", the fragment number indicates the fragments that have been received in order, as follows:

- If fragment 0 through fragment *n* have been received, and fragment *n*+1 has not been received, the fragment number field should be *n*.
- If fragment 0 has not been received, the fragment number field should contain the hexadecimal value FFFF.

In effect, fragments received out of order are not acknowledged.

Authentication Protocol Identifier

The authentication protocol identifier is an 8-bit non-negative integer that identifies an authentication protocol.

Authentication Protocol Identifiers lists the possible values for the authentication protocol identifier field and briefly describes the authentication protocol identified by each value. All other values are reserved for future use.

Identifier	Protocol Used
0	**None**
1	OSF DCE Private Key Authentication

Table: Authentication Protocol Identifiers

Connectionless PDU Definitions

The following sections describe the contents and use of each of the connectionless PDUs.

The ack PDU

A client sends an **ack** PDU after it has received a response to an **at-most-once** request. An **ack** PDU explicitly acknowledges that the client has received the response; it tells the server to cease resending the response and discard the response PDU. (A client can also implicitly acknowledge receipt of a response by sending a new request to the server.)

An **ack** PDU contains no body bgcolor="#FFFFFF" data.

The cancel_ack PDU

A server sends a **cancel_ack** PDU after it has received a **cancel** PDU. A **cancel_ack** PDU acknowledges that the server has cancelled or orphaned a remote call or indicates that the server is not accepting cancels.

A **cancel_ack** PDUs can optionally have a body bgcolor="#FFFFFF". A **cancel_ack** PDU without a body bgcolor="#FFFFFF" acknowledges orphaning of a call, whereas a **cancel_ack** PDU with a body bgcolor="#FFFFFF" acknowledges cancellation of a call. Orphaned calls do not perform any further processing. Canceled calls transparently deliver a notification to the server manager routine without altering the run-time system state of the call. The run-time system's processing of a cancelled call continues uninterrupted.

When a **cancel_ack** PDU has a body bgcolor="#FFFFFF", its format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned32  vers = 0; /* cancel-request body bgcolor="#FFFFFF" format versi
    unsigned32  cancel_id;/* id of a cancel-request event being ack'd */
    boolean     server_is_accepting; /* server accepting cancels ? */
} rpc_dg_cancel_ack_body bgcolor="#FFFFFF"_t;
```

- An NDR **unsigned long** that indicates the version number of the **cancel_ack** PDU body bgcolor="#FFFFFF". This is independent of the protocol version number contained in the PDU header. This specification is for version 0.

- An NDR **unsigned long** that identifies the cancel request event that is being acknowledged.

- An NDR **boolean** that indicates whether the server is accepting the cancel request. TRUE means that it is accepting.

The version number for the format of the **cancel_ack** body bgcolor="#FFFFFF" is the first byte of the body bgcolor="#FFFFFF". This version number changes independently of the protocol version number in the PDU header.

The cancel PDU

A client sends a **cancel** PDU when it has incurred a cancel fault. A **cancel** PDU tells the server to cancel or orphan a remote operation. Canceled calls transparently deliver a notification to the server manager routine without altering the run-time system state of the call. The run-time system's processing of a cancelled call continues uninterrupted.

The **cancel** PDU body bgcolor="#FFFFFF" format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned32  vers = 0; /* cancel body bgcolor="#FFFFFF" format version */
    unsigned32  cancel_id;/* id of a cancel-request event */
} rpc_dg_cancel_body bgcolor="#FFFFFF"_t;
```

- An NDR **unsigned long** that indicates the version number of the **cancel** PDU body bgcolor="#FFFFFF". This is independent of the protocol version number contained in the PDU header. This specification is for version 0.

- An NDR **unsigned long** that identifies a cancel request event.

The version number for the format of the **cancel** body bgcolor="#FFFFFF" is the first byte of the body bgcolor="#FFFFFF". This version number changes independently of the protocol version number in the PDU header.

The fack PDU

Both clients and servers send **fack** PDUs.

A *client* sends a **fack** PDU after it has received a fragment of a multi-PDU response. A **fack** PDU explicitly acknowledges that the client has received the fragment; it may tell the sender to stop sending for a while.

A *server* sends a **fack** PDU after it has received a fragment of a multi-PDU request. A **fack** PDU explicitly acknowledges that the server has received the fragment; it may tell the sender to stop sending for a while.

The **nofack** bit flag in a request or response fragment header can be used to control the sending of **fack** PDUs by the receiver as follows:

- The receiver of a fragment must send an **fack** if the fragment's **nofack** flag is not set.

- The receiver of a fragment may choose not to send a **fack** if the fragment's **nofack** flag is set.

A **fack** PDU may contain a body bgcolor="#FFFFFF" with data that can be used by the sender of a request or response to increase the efficiency of a multi-PDU transmission. The contents of a **fack** PDU body bgcolor="#FFFFFF" are specified, but use of the data is left to implementations. Implementations must be able to receive **fack** PDUs without bodies.

The first byte of a **fack** PDU body bgcolor="#FFFFFF" is a version number for the format of the **fack** body bgcolor="#FFFFFF". This version number can change independently of the protocol version number in the PDU header.

At version 0, the **fack** PDU body bgcolor="#FFFFFF" format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned8 vers = 0;       /* Fack packet body bgcolor="#FFFFFF" version */
    u_char pad1;
    unsigned16 window_size;   /* Sender's receive window size (in pkts) */
    unsigned32 max_tsdu;      /* largest local TPDU size */
    unsigned32 max_frag_size; /* largest TPDU not fragmented */
    unsigned16 serial_num;    /* serial # of packet that induced this fack */
    unsigned16 selack_len;    /* number of elements in the selack array    */
    unsigned32 selack[1];     /* variable number of 32 bit selective ack   */
                              /* bit masks.                               */
} rpc_dg_fack_body bgcolor="#FFFFFF"_t;
```

- An NDR **unsigned short** integer indicates the version number of the **fack** PDU body bgcolor="#FFFFFF".

- An uninitialised pad byte.

- An NDR **unsigned short** integer indicates the size in KB of the receive window at the sender of the **fack**. The window size indicates how much additional data the sender of the **fack** is prepared to receive. The window size field can be used by senders and receivers to coordinate their rates of transmission and reception.

- An NDR **unsigned long** integer indicates the size in bytes of the largest Transport Protocol Data Unit (TPDU) that can be passed through the local transport service interface at the sender of the **fack**.

- An NDR **unsigned long** integer indicating the **fack** sender's suggested maximum PDU size for this conversation. The actual PDU size used may be further limited by the **fack** receiver.

- An NDR **unsigned short** integer indicates the serial number of the fragment transmission that the sender of the **fack** is acknowledging. See Serial Numbers for the semantics of serial numbers.

- An NDR **unsigned short** integer that indicates the number of elements in the array of selective acknowledgement bit masks. Selective acknowledgement bit masks identify any fragments that the sender of the **fack** has received out of order; that is, any fragments whose fragment numbers are greater than the fragment number in the header of the **fack**.

- An array of NDR **unsigned long** integers that function as bit masks, indicating any fragments that the sender of the **fack** has received out of order.

Suppose that the sender of a **fack** has received fragments 0 through n, but not fragment n+1 of a multi-PDU transmission; n is therefore the fragment number in the header of the **fack**. Let m be an index for a bit mask in the bit mask array, where a value of m=0 indicates the first bit mask in the array. Let b be an index for a bit in a bit mask, where a value of b=0 indicates the least significant bit and a value of b=31 indicates the most significant bit. The value of bit b in mask m indicates whether the sender of the **fack** has received the fragment with fragment number $n+32m+b+1$. A value of 1 indicates that the fragment has been received, and a value of 0 indicates that the fragment has not been received.

The first bit in the first mask (*m*=0 and *b*=0) must always have a value of 0, since fragment *n*+1 is the first missing fragment.

A **fack** body bgcolor="#FFFFFF" contains only as many selective acknowledgement bit masks as are necessary to acknowledge fragments received out of order. The last bit mask in the **fack** body bgcolor="#FFFFFF" must always have at least one non-zero bit.

The information in selective acknowledgement bit masks is intended to tell the sender of a multi-PDU transmission which fragments it may need to resend.

The fault PDU

A server sends a **fault** PDU if an operation incurs a fault while executing on the server side.

The **fault** PDU body bgcolor="#FFFFFF" format is specified by the following IDL declaration:

```
typedef struct
{
    unsigned32 st; /* status code */
} rpc_dg_fault_body bgcolor="#FFFFFF"_t;
```

The body bgcolor="#FFFFFF" of a **fault** PDU contains a status code that indicates the fault that a requested operation incurred. The status code is represented by an NDR **unsigned long**. The fault status values and the corresponding application level **fault_status** parameter values are listed in Reject Status Codes and Parameters .

The nocall PDU

A server sends a **nocall** PDU in reply to a **ping** PDU. This reply indicates that the server is not currently processing the client’s call. The server may have never received the request, or some of the request fragments may have been lost and need to be retransmitted.

A **nocall** PDU can optionally carry a body bgcolor="#FFFFFF" whose format is the same as the optional **fack** PDU body bgcolor="#FFFFFF". (See The fack PDU .) If the server wants to indicate that the call in question is queued but not yet processed, it sets the receive window size to zero to indicate to the client that it need not resend the request.

The ping PDU

A client sends a **ping** PDU when it wants to inquire about an outstanding request.

A **ping** PDU contains no body bgcolor="#FFFFFF" data.

The reject PDU

A server sends a **reject** PDU if an RPC request is rejected. The body bgcolor="#FFFFFF" of a **reject**

PDU contains a status code indicating why a callee is rejecting a **request** PDU from a caller. The body bgcolor="#FFFFFF" format is the same as that of the **fault PDU**. (See The fault PDU .) The status code is represented by an NDR **unsigned long**. Reject status codes are listed Reject Status Codes and Parameters .

Note that reject status codes map to application level **comm_status** parameter values. This mapping is given in Reject Status Codes and Parameters .

The request PDU

A client sends a **request** PDU when it wants to execute a remote operation. In a multi-PDU request, the request consists of a series of **request** PDUs with the same sequence number and monotonically increasing fragment numbers. The body bgcolor="#FFFFFF" of a **request** PDU contains data that represents input parameters for the operation.

A **request** PDU may have one of the following types:

idempotent
: The request is for an idempotent operation. An idempotent request has the **idempotent** bit flag set.

broadcast
: The request is to be broadcast to all hosts on the local network. A broadcast request has the **idempotent** and **broadcast** bit flags set. Broadcast calls are never authenticated. The request must be sent in one PDU.

maybe
: The client does not expect a response. The protocols do not guarantee that the server will receive the request. A maybe request has the **idempotent** and **maybe** bit flags set.

broadcast/maybe
: The request is to be broadcast to all hosts on the local network and the client does not expect a response. A broadcast/maybe request has the **idempotent**, **broadcast** and **maybe** bit flags set. The request must be sent in one PDU.

at-most-once
: The request is for an operation that cannot be executed more than once. An at-most-once request is the default; none of the **idempotent**, **broadcast** or **maybe** bit flags is set.

The body bgcolor="#FFFFFF" of a **request** PDU consists of an NDR representation of the input parameters for the request.

The response PDU

A server sends a **response** PDU if an operation invoked by an idempotent, broadcast or at-most-once request executes successfully. Servers do not send responses for maybe or broadcast/maybe requests. A multi-PDU response consists of a series of **response** PDUs with the same sequence number and monotonically increasing fragment numbers.

The body bgcolor="#FFFFFF" of a **response** PDU consists of the NDR representation of the output parameters for the response.

The working PDU

A server sends a **working** PDU in reply to a **ping** PDU. This reply indicates that the server is processing the client's call.

A **working** PDU contains no body bgcolor="#FFFFFF" data.

Connection-oriented RPC PDUs

The RPC run-time system uses the connection-oriented PDUs for the client/server communications required by each remote procedure call over a connection-oriented transport. This section specifies the encoding of each of the connection-oriented PDUs listed in RPC Protocol Data Units .

The client and server **CALL** protocol machines communicate using a set of call PDUs, and the client and server **ASSOCIATION** protocol machines communicate using a set of association PDUs. (The **ASSOCIATION GROUP** state machines are purely local to the client and server and exchange no PDUs.) The PDUs in each group are:

Association
- **bind**
- **bind_ack**
- **bind_nak**
- **alter_context**
- **alter_context_response**

Call
- **request**
- **response**
- **fault**
- **shutdown**
- **cancel**
- **orphaned**

The association and call PDUs are encoded by the RPC run-time system and delivered to the underlying transport for transmission. This document does not specify encoding of any connection PDUs. In the RPC connection management model, connections are established by the underlying transport. The RPC run-time system assumes that the underlying transport provides certain services (see RPC Protocol Definitions), but it does not specify the concrete implementation of these services or any protocol encodings.

Connection-oriented PDU Structure

The connection-oriented PDUs follow the general structure described in Generic PDU Structure ; that is, a header followed by body bgcolor="#FFFFFF" data and an optional authentication verifier. Connection-oriented PDU headers vary in size; every header includes a set of common header fields, but in some connection-oriented PDUs, this is followed by PDU specific header fields. The authentication verifier may be present in **bind**, **bind_ack**, **alter_context** and **alter_context_response** PDUs; it is never present in **bind_nak** and **shutdown** PDUs; and it is optionally present, depending on security protocol, in the other PDUs. In minor version 1 the connection-oriented authentication verifier data may be larger than can fit into a single Transport Service Data Unit (TSDU). These PDUs must be fragmented by the RPC run-time system.

Fragmentation and Reassembly

The connection-oriented PDUs for minor version 1 may contain arbitrary amounts of authentication verification data. If the PDU has a minor version number of 0 (zero), the run-time assumes no fragmentation. If the minor version number is 1 and the **PFC_LAST_FRAG** flag is not set, the PDU is fragmented. The authentication data is reassembled by concatenating all authentication data until a PDU arrives with the **PFC_LAST_FRAG** flag set, this is the last PDU containing authentication data. No other flags or fields will change while reassembling the authentication data. Only **bind**, **bind_ack**, **alter_context** and **alter_context_response** PDUs will be fragmented. Since the connection-oriented transport guarantees sequentiality, the receiver will always receive the fragments in order.

RPC request and response service requests may contain arbitrary amounts of stub data. Either of these requests may be fragmented by the RPC run-time system into multiple Transport Service Data Units (TSDUs) and reassembled by the receiving RPC run-time system before or during unmarshalling.

Each fragment is labelled as such using the **PFC_FIRST_FRAG** and **PFC_LAST_FRAG** flags in the header **pfc_flags** field. If a service request needs only a single fragment, that fragment will have both the **PFC_FIRST_FRAG** and **PFC_LAST_FRAG** flags set to TRUE. Since the connection-oriented transport guarantees sequentiality, the receiver will always receive the fragments in order.

Note:

The flags encoding for the connectionless and connection-oriented protocols is different. The connectionless **PF_FRAG** flag is not required in the connection-based encoding, which explicitly labels both the first and last fragment.

The Open Group DCE defines a lower bound on the size of a single fragment (TSDU) that all implementations must be able to receive, **MustRecvFragSize**. A client and server may also negotiate larger fragment sizes as part of the binding operation.

The client determines, and then sends in the **bind** PDU, its desired maximum size for transmitting fragments, and its desired maximum receive fragment size. Similarly, the server determines its desired maximum sizes for transmitting and receiving fragments. Transmit and receive sizes may be different to help preserve buffering. When the server receives the client’s values, it sets its operational transmit size to the minimum of the client’s receive size (from the **bind** PDU) and its own desired transmit size. Then it sets its actual receive size to the minimum of the client’s transmit size (from the **bind**) and its own desired receive size. The server then returns its operational values in the **bind_ack** PDU. The client then sets its operational values from the received **bind_ack** PDU. The received transmit size becomes the client’s receive size, and the received receive size becomes the client’s transmit size.

Either party may use receive buffers larger than negotiated- although this will not provide any advantage-but may not transmit larger fragments than negotiated.

Note:

> An implementation may ignore the negotiation by always specifying the default encodings of 0 (zero) in the PDUs. This is consistent with the negotiation algorithm described here.
>
> An implementation may wish to adjust its desired fragment sizes to tune them to the most common data links expected.

When receiving a **request** PDU, the **PFC_PENDING_CANCEL** and **PFC_MAYBE** flag values on the **first_frag** are authoritative. When receiving a **response** or **fault** PDU, the **PFC_DID_NOT_EXECUTE** and **PFC_PENDING_CANCEL** flag values and the **cancel_count** and **status** fields of the **last_frag** are authoritative.

Connection-oriented PDU Data Types

The following sections provide IDL declarations for header data types. These include both the common header fields and other header fields that appear in various PDUs. These sections also discuss the use of several fields and data types by the connection-oriented RPC protocol.

Declarations

The following synonyms appear in many of the declarations:

```
typedef     unsigned hyper  u_int64;
typedef     unsigned long   u_int32;
typedef     unsigned short  u_int16;
typedef     unsigned small  u_int8;   /* single octet unsigned int */
```

The common header fields, which appear in all PDU types, are as follows. The comment fields show the exact octet alignment and octet length of each element.

```
/* start 8-octet aligned */
u_int8  rpc_vers = 5;        /* 00:01 RPC version */
u_int8  rpc_vers_minor;      /* 01:01 minor version */
u_int8  PTYPE;               /* 02:01 packet type */
u_int8  pfc_flags;           /* 03:01 flags (see PFC_... ) */
byte    packed_drep[4];   /* 04:04 NDR data representation format label */
u_int16 frag_length;         /* 08:02 total length of fragment */
u_int16 auth_length;         /* 10:02 length of auth_value */
u_int32 call_id;             /* 12:04 call identifier */
```

The NDR data representation format label is discussed in Transfer Syntax NDR .

This document defines the following values for the common header **pfc_flags** field:

```
#define PFC_FIRST_FRAG          0x01/* First fragment */
#define PFC_LAST_FRAG           0x02/* Last fragment */
#define PFC_PENDING_CANCEL      0x04/* Cancel was pending at sender */
#define PFC_RESERVED_1          0x08
#define PFC_CONC_MPX            0x10/* supports concurrent multiplexing
```

```
                                            * of a single connection. */
#define PFC_DID_NOT_EXECUTE      0x20/* only meaningful on 'fault' packet;
                                            * if true, guaranteed call did not
                                            * execute. */
#define PFC_MAYBE                0x40/* 'maybe' call semantics requested */
#define PFC_OBJECT_UUID          0x80/* if true, a non-nil object UUID
                                            * was specified in the handle, and
                                            * is present in the optional object
                                            * field. If false, the object field
                                            * is omitted. */
```

Several elements are used for presentation context identification and negotiation. Local context identifiers are defined as:

```
typedef   u_int16   p_context_id_t;
```

Presentation syntax is identified by:

```
typedef   struct   {
             uuid_t   if_uuid;
             u_int32   if_version;
          } p_syntax_id_t;
```

For abstract syntax, **if_uuid** is set to the interface UUID, and **if_version** is set to the interface version. For transfer syntax, these are set to the UUID and version created for the data representation. The major version is encoded in the 16 least significant bits of **if_version** and the minor version in the 16 most significant bits.

One element in a presentation context list is defined as:

```
typedef   struct {
          p_context_id_t   p_cont_id;
          u_int8           n_transfer_syn;     /* number of items */
          u_int8           reserved;           /* alignment pad, m.b.z. */
          p_syntax_id_t    abstract_syntax;    /* transfer syntax list */
          p_syntax_id_t [size_is(n_transfer_syn)] transfer_syntaxes[];
          } p_cont_elem_t;
```

The whole list is defined as:

```
typedef   struct {
          u_int8          n_context_elem;      /* number of items */
          u_int8          reserved;            /* alignment pad, m.b.z. */
          u_short         reserved2;           /* alignment pad, m.b.z. */
          p_cont_elem_t [size_is(n_cont_elem)] p_cont_elem[];
          } p_cont_list_t;
```

The following declarations are for the results of a presentation context negotiation. Result types are defined as:

```
typedef  short enum {
         acceptance, user_rejection, provider_rejection
         } p_cont_def_result_t;
```

Reasons for rejection of a context element are defined as:

```
typedef   short enum {
          reason_not_specified,
          abstract_syntax_not_supported,
          proposed_transfer_syntaxes_not_supported,
          local_limit_exceeded
          } p_provider_reason_t;
```

The meanings of these rejection reasons are defined in Section 2 of the ISO 8823 standard.

A result list returns the results of the context negotiation. A list element is declared as:

```
typedef struct {
        p_cont_def_result_t   result;
        p_provider_reason_t   reason;  /* only relevant if result !=
                                              * acceptance */
        p_syntax_id_t         transfer_syntax;/* tr syntax selected
                                                  * 0 if result not
                                                  * accepted */
        } p_result_t;
```

The entire list is defined as:

```
/* Same order and number of elements as in bind request */

typedef struct {
        u_int8   n_results;      /* count */
        u_int8   reserved;       /* alignment pad, m.b.z. */
        u_int16 reserved2;       /* alignment pad, m.b.z. */
        p_result_t [size_is(n_results)] p_results[];
        } p_result_list_t;
```

The protocol version data type is defined as:

```
typedef struct {
        u_int8   major;
        u_int8   minor;
        } version_t;
```

The run-time version data type is synonymous:

```
typedef   version_t   p_rt_version_t;
```

When the protocol negotiation fails, the list of supported protocols is returned as:

```
typedef struct {
        u_int8   n_protocols;   /* count */
        p_rt_version_t  [size_is(n_protocols)] p_protocols[];
        } p_rt_versions_supported_t;
```

The following data structure is used when a bind request returns a secondary address. It holds a string representation of the local port part of the address only. The length includes the C NULL string termination.

```
typedef struct {
        u_int16 length;
        char   [size_is(length)] port_spec; /* port string spec */
        } port_any_t;
```

Reasons for rejection of an association are returned in the **bind_nak** PDU. These are defined as:

```
#define REASON_NOT_SPECIFIED             0
#define TEMPORARY_CONGESTION             1
#define LOCAL_LIMIT_EXCEEDED             2
#define CALLED_PADDR_UNKNOWN             3 /* not used */
#define PROTOCOL_VERSION_NOT_SUPPORTED   4
#define DEFAULT_CONTEXT_NOT_SUPPORTED    5 /* not used */
#define USER_DATA_NOT_READABLE           6 /* not used */
#define NO_PSAP_AVAILABLE                7 /* not used */
```

Connection-Oriented Protocol Versions

Each PDU contains the sender's major and minor RPC run-time protocol version numbers. The client's and server's major versions must be equal. Backward compatible changes in the protocol are indicated by higher minor version numbers. Therefore, a server's minor version must be greater than or equal to the client's. However, for connection-oriented minor version numbers of 0 (zero) and 1, both the server and the client are backwards compatible. That is, if a client with minor version 1 contacts a lower versioned server, the association open will fail. If this condition occurs the client will try to open the association again using the lower minor version number and the old style of authorisation. However, if the server's minor version exceeds the client's minor version, it must return the client's minor version and restrict its use of the protocol to the minor version specified by the client. A protocol version mismatch causes the **nca_s_rpc_version_mismatch** error status to be returned.

The PDU formats given here are for major version **DC_PROTO_VERS_MAJOR_1**, as defined in The First Set of PDU Flags , minor version 0.

The protocol version is negotiated using the **bind**, **bind_ack** and **bind_nak** messages. For all other messages, the protocol version in the header only serves as a sanity check; if it is incorrect, it indicates a massive error and the connection should be terminated with the error **nca_s_rpc_version_mismatch**.

The frag_length Field

The **frag_length** field represents the length of the entire PDU, including all of the header, optional header fields, stub body bgcolor="#FFFFFF" and optional authentication verifier, if applicable.

Context Identifiers

Presentation context identifiers are transmitted on each request or response. The client defines the values of the context identifiers, and both the client and server must be able to map between the specific client's context identifier and the selected abstract and transfer syntax, which indicate the interface and data representation.

The client must assign context identifiers that are unique within at least a single association. Context identifiers may also be unique within an association group or across the entire client instantiation. A server must interpret context identifiers relative to each particular association; that is, different associations within the same association group from the same client to the same server may legally use the same context identifier with a different meaning.

The call_id Field

Each run-time protocol message contains a **call_id** field. This field is used by the caller to guarantee that it is matching the proper response and request. Otherwise, responses for the wrong call, or orphaned responses from calls that were cancelled, and the cancel timed out, could be confused with current responses. The caller must guarantee that at any time, all **call_id**s for all outstanding messages for the same association group are distinct. The server just returns the **call_id** on the corresponding message.

The **call_id** field is also used to guarantee proper matching of **bind_ack**, **bind_nak** or **alter_context_response** messages in order to guarantee proper behaviour under certain conditions; for example, cancel timeout causes an orphan.

Note:

> The most straightforward implementation is for each client process to maintain a single **u_int32** sequence number counter to use for the **call_id**. Alternatively, the client may assign a value representing a call data structure, and use that, provided it does sufficient bookkeeping to insure that it cannot be reused if a call is cancelled and times out, at least until the the entire orphaned response is received.

The assoc_group_id Field

The client should set the **assoc_group_id** field either to 0 (zero), to indicate a new association group, or to the known value. When the server receives a value of 0, this indicates that the client has requested a new association group, and it assigns a server unique value to the group. This value is returned in the

rpc_bind_ack PDU.

The alloc_hint Field

The **alloc_hint** field may be used by the transmitter as a hint to the receiver, informing it how much buffer space, in units of octets, to allocate contiguously for fragmented requests. This is only a potential optimisation; a receiver is required to work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

Authentication Data

If the **auth_length** header field value is non-zero, then the message contains optional authentication and/or authorisation data in an authentication verifier. Each message format specifies the location of the verifier, which always follows any stub data, if applicable.

The contents of the authentication verifier are specified in Security .

Optional Connect Reject and Disconnect Data

If the transport supports optional connect reject or disconnect data, the RPC run-time system may transmit additional error information as the optional data. The following data types are used to declare this optional data:

```
typedef  u_int16 rpcrt_reason_code_t;/* 0..65535 */

typedef  struct {
         u_int8    rpc_vers;
         u_int8    rpc_vers_minor;
         u_int8    reserved[2];/* must be zero */
         byte      packed_drep[4];
         u_int32   reject_status;
         u_int8    reserved[4];
         } rpcrt_optional_data_t;
```

The format for optional reject data is as follows:

```
typedef struct {
        rpcrt_reason_code_t     reason_code;  /* 0..65535   */
        rpcrt_optional_data_t   rpc_info;     /* may be RPC specific */
        }  rpcconn_reject_optional_data_t;
```

The format for optional disconnect data is as follows:

```
typedef struct {
        rpcrt_reason_code_t     reason_code;  /* 0..65535 */
        rpcrt_optional_data_t   rpc_info;     /* may be RPC-specific */
        } rpcconn_disc_optional_data_t;
```

Connection-oriented PDU Definitions

The following sections give IDL declarations and descriptions for each of the the connection-oriented PDUs.

The alter_context PDU

The IDL declaration of the **alter_context** PDU is as follows:

```
typedef struct {
      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;          /* 00:01 RPC version */
        u_int8  rpc_vers_minor ;       /* 01:01 minor version */
        u_int8  PTYPE = alter_context; /* 02:01 alter context PDU */
        u_int8  pfc_flags;             /* 03:01 flags */
        byte    packed_drep[4];        /* 04:04 NDR data rep format label*/
        u_int16 frag_length;           /* 08:02 total length of fragment */
        u_int16 auth_length;           /* 10:02 length of auth_value */
        u_int32  call_id;              /* 12:04 call identifier */

      /* end common fields */

        u_int16 max_xmit_frag;         /* ignored */
        u_int16 max_recv_frag;         /* ignored */
        u_int32 assoc_group_id;        /* ignored */

      /* presentation context list */

        p_cont_list_t  p_context_elem; /* variable size */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_verifier_co_t   auth_verifier;

        } rpcconn_alter_context_hdr_t;
```

The **alter_context** PDU is used to request additional presentation negotiation for another interface and/or version, or to negotiate a new security context, or both. The format is identical to the **bind** PDU, except that the value of the **PTYPE** field is set to **alter_context**. The **max_xmit_frag**, **max_recv_frag** and **assoc_group_id** fields are be ignored.

The alter_context_resp PDU

The IDL declaration of the **alter_context_resp** PDU is as follows:

```
typedef struct {
      /* start 8-octet aligned */
```

```
    /* common fields */
      u_int8  rpc_vers = 5;        /* 00:01 RPC version */
      u_int8  rpc_vers_minor       /* 01:01 minor version */
      u_int8  PTYPE = alter_context_response; /* 02:01 alter
                                         context response PDU */
      u_int8  pfc_flags;           /* 03:01 flags */
      byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
      u_int16 frag_length;         /* 08:02 total length of fragment */
      u_int16 auth_length;         /* 10:02 length of auth_value */
      u_int32  call_id;            /* 12:04 call identifier */

    /* end common fields */

      u_int16 max_xmit_frag;       /* ignored */
      u_int16 max_recv_frag;       /* ignored */
      u_int32 assoc_group_id;      /* ignored */
      port_any_t sec_addr;         /* ignored */

    /* restore 4-octet alignment */

      u_int8 [size_is(align(4))] pad2;

    /* presentation context result list, including hints */

      p_result_list_t     p_result_list;    /* variable size */

    /* optional authentication verifier */
    /* following fields present iff auth_length != 0 */

      auth_verifier_co_t   auth_verifier; /* xx:yy */
  } rpcconn_alter_context_response_hdr_t;
```

The **alter_context_response** PDU is used to indicate the server's response to an **alter_context** request. The PDU format is identical to **bind_ack**, except that the value of the **PTYPE** field is set to **alter_context_response**. The **max_xmit_frag**, **max_recv_frag**, **assoc_group_id** and **sec_addr** fields are ignored.

The bind PDU

The IDL declaration of the **bind** PDU is as follows:

```
/*  bind header */
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor;      /* 01:01 minor version */
        u_int8  PTYPE = bind;        /* 02:01 bind PDU */
        u_int8  pfc_flags;           /* 03:01 flags */
        byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
        u_int16 frag_length;         /* 08:02 total length of fragment */
        u_int16 auth_length;         /* 10:02 length of auth_value */
        u_int32  call_id;            /* 12:04 call identifier */
```

```
/* end common fields */

  u_int16 max_xmit_frag;      /* 16:02 max transmit frag size, bytes */
  u_int16 max_recv_frag;      /* 18:02 max receive  frag size, bytes */
  u_int32 assoc_group_id;     /* 20:04 incarnation of client-server
                                               * assoc group */
/* presentation context list */

  p_cont_list_t  p_context_elem; /* variable size */

/* optional authentication verifier */
/* following fields present iff auth_length != 0 */

  auth_verifier_co_t   auth_verifier;

  } rpcconn_bind_hdr_t;
```

The **bind** PDU is used to initiate the presentation negotiation for the body bgcolor="#FFFFFF" data, and optionally, authentication. The presentation negotiation follows the model of the OSI presentation layer.

The PDU contains a priority-ordered list of supported presentation syntaxes, both abstract and transfer, and context identifiers (local handles). (This differs from OSI, which does not specify any order for the list.) The abstract and transfer syntaxes are represented as a record of interface UUID and interface version. (These may map one-to-one into OSI object identifiers by providing suitable prefixes and changing the encoding.) Each supported data representation, such as NDR, will be assigned an interface UUID, and will use that UUID as part of its transfer syntax value. Each stub computes its abstract syntax value given its interface UUID and interface version. The transfer syntax value for NDR is defined in Protocol Identifiers .

If **pfc_flags** does not have **PFC_LAST_FRAG** set and **rpc_vers_minor** is 1, then the PDU has fragmented **auth_verifier** data. The server will assemble the data concatenating sequentially each **auth_verifier** field until a PDU is sent with **PFC_LAST_FRAG** flag set. This completed buffer is then used as **auth_verifier** data.

The fields **max_xmit_frag** and **max_recv_frag** are used for fragment size negotiation as described in Connection-oriented PDU Data Types .

The **assoc_group_id** field contains either an association group identifier that was created during a previous bind negotiation or 0 (zero) to indicate a request for a new group.

This PDU shall not exceed the **MustRecvFragSize**, since no size negotiation has yet occurred. If the **p_context_elem** is too long, the leading subset should be transmitted, and additional presentation context negotiation can occur in subsequent **alter_context** PDUs, as needed, after a successful **bind_ack**.

The bind_ack PDU

The IDL declaration of the **bind_ack** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;       /* 00:01 RPC version */
        u_int8  rpc_vers_minor ;    /* 01:01 minor version */
        u_int8  PTYPE = bind_ack;   /* 02:01 bind ack PDU */
        u_int8  pfc_flags;          /* 03:01 flags */
        byte    packed_drep[4];     /* 04:04 NDR data rep format label*/
        u_int16 frag_length;        /* 08:02 total length of fragment */
        u_int16 auth_length;        /* 10:02 length of auth_value */
        u_int32  call_id;           /* 12:04 call identifier */

      /* end common fields */

        u_int16 max_xmit_frag;      /* 16:02 max transmit frag size */
        u_int16 max_recv_frag;      /* 18:02 max receive  frag size */
        u_int32 assoc_group_id;     /* 20:04 returned assoc_group_id */
        port_any_t sec_addr;        /* 24:yy optional secondary address
                                     * for process incarnation; local port
                                     * part of address only */
      /* restore 4-octet alignment */

        u_int8 [size_is(align(4))] pad2;

      /* presentation context result list, including hints */

        p_result_list_t     p_result_list;    /* variable size */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_verifier_co_t   auth_verifier; /* xx:yy */
    } rpcconn_bind_ack_hdr_t;
```

The **bind_ack** PDU is returned by the server when it accepts a bind request initiated by the client's **bind** PDU. It contains the results of presentation context and fragment size negotiations. It may also contain a new association group identifier if one was requested by the client.

The **max_xmit_frag** and **max_recv_frag** fields contain the maximum transmit and receive fragment sizes as determined by the server in response to the client's desired sizes.

The **p_result_list** contains the results of the presentation context negotiation initiated by the client. It is possible for a **bind_ack** not to contain any mutually supported syntaxes.

If the client requested a new association group, **assoc_group_id** contains the identifier of the new association group created by the server. Otherwise, it contains the identifier of the previously created association group requested by the client.

The bind_nak PDU

The IDL declaration of the **bind_nak** PDU is as follows:

```
typedef struct {

  /* start 8-octet aligned */

  /* common fields */
    u_int8  rpc_vers = 5;       /* 00:01 RPC version */
    u_int8  rpc_vers_minor ;    /* 01:01 minor version */
    u_int8  PTYPE = bind_nak;   /* 02:01 bind nak PDU */
    u_int8  pfc_flags;          /* 03:01 flags */
    byte    packed_drep[4];     /* 04:04 NDR data rep format label*/
    u_int16 frag_length;        /* 08:02 total length of fragment */
    u_int16 auth_length;        /* 10:02 length of auth_value */
    u_int32  call_id;           /* 12:04 call identifier */

  /* end common fields */

    p_reject_reason_t provider_reject_reason; /* 16:02 presentation
                                                  context reject */

    p_rt_versions_supported_t versions; /* 18:yy array of protocol
                                         * versions supported */
    } rpcconn_bind_nak_hdr_t;
```

The **bind_nak** PDU is returned by the server when it rejects an association request initiated by the client's **bind** PDU. The **provider_reject_reason** field holds the rejection reason code. When the reject reason is **protocol_version_not_supported**, the **versions** field contains a list of run-time protocol versions supported by the server.

The **bind_nak** PDU never contains an authentication verifier.

The cancel PDU

The IDL declaration of the **cancel** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5 ;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor;       /* 01:01 minor version */
        u_int8  PTYPE = co_cancel;    /* 02:01 CO cancel PDU */
        u_int8  pfc_flags;            /* 03:01 flags */
        byte    packed_drep[4];       /* 04:04 NDR data rep format label*/
        u_int16 frag_length;          /* 08:02 total length of fragment */
        u_int16 auth_length;          /* 10:02 length of auth_value */
        u_int32  call_id;             /* 12:04 call identifier */

      /* end common fields */

      /* optional authentication verifier
```

```
          * following fields present iff auth_length != 0 */

        auth_verifier_co_t   auth_verifier; /* xx:yy */

    } rpcconn_cancel_hdr_t;
```

The **cancel** PDU is used to forward a cancel.

The fault PDU

The IDL declaration of the **fault** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor;      /* 01:01 minor version */
        u_int8  PTYPE = fault;       /* 02:01 fault PDU */
        u_int8  pfc_flags;           /* 03:01 flags */
        byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
        u_int16 frag_length;         /* 08:02 total length of fragment */
        u_int16 auth_length;         /* 10:02 length of auth_value */
        u_int32  call_id;            /* 12:04 call identifier */

      /* end common fields */

      /* needed for request, response, fault */

        u_int32  alloc_hint;      /* 16:04 allocation hint */
        p_context_id_t p_cont_id; /* 20:02 pres context, i.e. data rep */

      /* needed for response or fault */

        u_int8  cancel_count      /* 22:01 received cancel count */
        u_int8  reserved;         /* 23:01 reserved, m.b.z. */

      /* fault code  */

        u_int32  status           /* 24:04 run-time fault code or zero */

      /* always pad to next 8-octet boundary */

        u_int8  reserved2[4];     /* 28:04 reserved padding, m.b.z. */

      /* stub data here, 8-octet aligned
               .
               .
               .                           */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

       auth_verifier_co_t   auth_verifier; /* xx:yy */

} rpcconn_fault_hdr_t;
```

The **fault** PDU is used to indicate either an RPC run-time, RPC stub, or RPC-specific exception to the client. The **p_cont_id** field holds a context identifier that identifies the data representation.

The **alloc_hint** field is optionally used by the client to provide a hint to the receiver of the amount of buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation. The server must work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

The **status** field indicates run-time status. The value may either be an architected non-zero value, indicating a run-time error, such as an interface version mismatch, or 0 (zero), indicating a stub defined exception that is specified with the stub data. If a non-zero value is present, no stub data is allowed. Possible values are given in Reject Status Codes .

Certain status values imply that the call did not execute. To keep such status values consistent with the flag, an implementation should model all fault messages as being initialised with the PFC_DID_NOT_EXECUTE flag set to TRUE, then cleared when the run-time system (or stub, if the implementation allows) passes control to the server stub routine.

The orphaned PDU

The IDL declaration of the **orphaned** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor;      /* 01:01 minor version */
        u_int8  PTYPE = orphaned;    /* 02:01 orphaned PDU */
        u_int8  pfc_flags;           /* 03:01 flags */
        byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
        u_int16 frag_length;         /* 08:02 total length of fragment */
        u_int16 auth_length;         /* 10:02 length of auth_value */
        u_int32  call_id;            /* 12:04 call identifier */

      /* end common fields */

      /* optional authentication verifier
         * following fields present iff auth_length != 0 */

        auth_verifier_co_t   auth_verifier; /* xx:yy */

} rpcconn_orphaned_hdr_t;
```

The **orphaned** PDU is used by a client to notify a server that it is aborting a request in progress that has not been entirely transmitted yet, or that it is aborting a (possibly lengthy) response in progress.

The request PDU

The IDL declaration of the **request** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;       /* 00:01 RPC version */
        u_int8  rpc_vers_minor;     /* 01:01 minor version */
        u_int8  PTYPE = request ;   /* 02:01 request PDU */
        u_int8  pfc_flags;          /* 03:01 flags */
        byte    packed_drep[4];     /* 04:04 NDR data rep format label*/
        u_int16 frag_length;        /* 08:02 total length of fragment */
        u_int16 auth_length;        /* 10:02 length of auth_value */
        u_int32  call_id;           /* 12:04 call identifier */

      /* end common fields */

      /* needed on request, response, fault */

        u_int32  alloc_hint;        /* 16:04 allocation hint */
        p_context_id_t p_cont_id    /* 20:02 pres context, i.e. data rep */
        u_int16 opnum;              /* 22:02 operation #
                                     * within the interface */

      /* optional field for request, only present if the PFC_OBJECT_UUID
         * field is non-zero */

        uuid_t  object;              /* 24:16 object UID */

      /* stub data, 8-octet aligned
                    .
                    .
                    .                /*

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_verifier_co_t   auth_verifier; /* xx:yy */

} rpcconn_request_hdr_t;
```

The **request** PDU is used for an initial call request. The **p_cont_id** field holds a presentation context identifier that identifies the data representation. The **opnum** field identifies the operation being invoked within the interface.

The PDU may also contain an object UUID. In this case the PFC_OBJECT_UUID flag is set in **pfc_flags**, and the PDU includes the **object** field. If the PFC_OBJECT_UUID flag is not set, the PDU does not include the **object** field.

The **alloc_hint** field is optionally used by the client to provide a hint to the receiver of the amount of buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation. The server must work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

The minimum size of an **rpcconn_request_hdr_t** is 24 octets. If a non-nil object UUID or authentication and/or integrity or privacy services are used, the size will be larger.

The size of the stub data is calculated as follows:

```
stub_data_length = frag_length - fixed_header_length - auth_length;
if pfc_flags & PFC_OBJECT_UUID {
    stub_data_length = stub_data_length - sizeof(uuid_t);
    }
```

where the current value of **fixed_header_length** is 24 octets.

The response PDU

The IDL declaration of the **response** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor;      /* 01:01 minor version */
        u_int8  PTYPE = response;    /* 02:01 response PDU */
        u_int8  pfc_flags;           /* 03:01 flags */
        byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
        u_int16 frag_length;         /* 08:02 total length of fragment */
        u_int16 auth_length;         /* 10:02 length of auth_value */
        u_int32 call_id;             /* 12:04 call identifier */

      /* end common fields */

      /* needed for request, response, fault */

        u_int32  alloc_hint;         /* 16:04 allocation hint */
        p_context_id_t p_cont_id;    /* 20:02 pres context, i.e.
                                      * data rep */

      /* needed for response or fault */

        u_int8  cancel_count         /* 22:01 cancel count */
        u_int8  reserved;            /* 23:01 reserved, m.b.z. */

      /* stub data here, 8-octet aligned
                .
                .
                .                          */

      /* optional authentication verifier */
      /* following fields present iff auth_length != 0 */

        auth_verifier_co_t   auth_verifier; /* xx:yy */
```

```
} rpcconn_response_hdr_t;
```

The **response** PDU is used to respond to an active call. The **p_cont_id** field holds a context identifier that identifies the data representation. The **cancel_count** field holds a count of cancels received.

The **alloc_hint** field is optionally used by the transmitter to provide a hint to the receiver of the amount of buffer space to allocate contiguously for fragmented requests. This is only a potential optimisation. The receiver must work correctly regardless of the value passed. The value 0 (zero) is reserved to indicate that the transmitter is not supplying any information.

The shutdown PDU

The IDL declaration of the **shutdown** PDU is as follows:

```
typedef struct {

      /* start 8-octet aligned */

      /* common fields */
        u_int8  rpc_vers = 5;        /* 00:01 RPC version */
        u_int8  rpc_vers_minor;      /* 01:01 minor version */
        u_int8  PTYPE = shutdown ;   /* 02:01 shutdown PDU */
        u_int8  pfc_flags;           /* 03:01 flags */
        byte    packed_drep[4];      /* 04:04 NDR data rep format label*/
        u_int16 frag_length;         /* 08:02 total length of fragment */
        u_int16 auth_length;         /* 10:02 */
        u_int32  call_id;            /* 12:04 call identifier */

      /* end common fields */

} rpcconn_shutdown_hdr_t;
```

The **shutdown** PDU is sent by the server to request that a client terminate the connection, freeing the related resources.

The **shutdown** PDU never contains an authentication verifier even if authentication services are in use.
