Internet Engineering Task Force (IETF)
Request for Comments: 8166
Obsoletes: 5666
Category: Standards Track
ISSN: 2070-1721

C. Lever, Ed.
Oracle
W. Simpson
Red Hat
T. Talpey
Microsoft

June 2017

Remote Direct Memory Access Transport for Remote Procedure Call Version 1

Abstract

This document specifies a protocol for conveying Remote Procedure Call (RPC) messages on physical transports capable of Remote Direct Memory Access (RDMA). This protocol is referred to as the RPC-over-RDMA version 1 protocol in this document. It requires no revision to application RPC protocols or the RPC protocol itself. This document obsoletes RFC 5666.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc8166.

Table of Contents

1. Introduction

This document specifies the RPC-over-RDMA version 1 protocol, based on existing implementations of RFC 5666 and experience gained through deployment. This document obsoletes RFC 5666.

This specification clarifies text that was subject to multiple interpretations and removes support for unimplemented RPC-over-RDMA version 1 protocol elements. It clarifies the role of Upper-Layer Bindings (ULBs) and describes what they are to contain.

In addition, this document describes current practice using RPCSEC_GSS [RFC7861] on RDMA transports.

The protocol version number has not been changed because the protocol specified in this document fully interoperates with implementations of the RPC-over-RDMA version 1 protocol specified in [RFC5666].

1.1. RPCs on RDMA Transports

RDMA [RFC5040] [RFC5041] [IBARCH] is a technique for moving data efficiently between end nodes. By directing data into destination buffers as it is sent on a network, and placing it via direct memory access by hardware, the benefits of faster transfers and reduced host overhead are obtained.

Open Network Computing Remote Procedure Call (ONC RPC, often shortened in NFSv4 documents to RPC) [RFC5531] is a remote procedure call protocol that runs over a variety of transports. Most RPC implementations today use UDP [RFC768] or TCP [RFC793]. On UDP, RPC messages are encapsulated inside datagrams, while on a TCP byte stream, RPC messages are delineated by a record marking protocol. An RDMA transport also conveys RPC messages in a specific fashion that must be fully described if RPC implementations are to interoperate.

RDMA transports present semantics that differ from either UDP or TCP. They retain message delineations like UDP but provide reliable and sequenced data transfer like TCP. They also provide an offloaded bulk transfer service not provided by UDP or TCP. RDMA transports are therefore appropriately viewed as a new transport type by RPC.

In this context, the Network File System (NFS) protocols, as described in [RFC1094], [RFC1813], [RFC7530], [RFC5661], and future NFSv4 minor versions, are all obvious beneficiaries of RDMA transports. A complete problem statement is presented in [RFC5532]. Many other RPC-based protocols can also benefit.

Although the RDMA transport described herein can provide relatively transparent support for any RPC application, this document also describes mechanisms that can optimize data transfer even further, when RPC applications are willing to exploit awareness of RDMA as the transport.

2. Terminology

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.2. RPCs

This section highlights key elements of the RPC [RFC5531] and External Data Representation (XDR) [RFC4506] protocols, upon which RPC-over-RDMA version 1 is constructed. Strong grounding with these protocols is recommended before reading this document.

2.2.1. Upper-Layer Protocols

RPCs are an abstraction used to implement the operations of an Upper-Layer Protocol (ULP). "ULP" refers to an RPC Program and Version tuple, which is a versioned set of procedure calls that comprise a single well-defined API. One example of a ULP is the Network File System Version 4.0 [RFC7530].

In this document, the term "RPC consumer" refers to an implementation of a ULP running on an RPC client endpoint.

2.2.2. Requesters and Responders

Like a local procedure call, every RPC procedure has a set of "arguments" and a set of "results". A calling context invokes a procedure, passing arguments to it, and the procedure subsequently returns a set of results. Unlike a local procedure call, the called procedure is executed remotely rather than in the local application’s execution context.

The RPC protocol as described in [RFC5531] is fundamentally a message-passing protocol between one or more clients (where RPC consumers are running) and a server (where a remote execution context is available to process RPC transactions on behalf of those consumers).

ONC RPC transactions are made up of two types of messages:

CALL
: An "RPC Call message" requests that work be done. This type of message is designated by the value zero (0) in the message's msg_type field. An arbitrary unique value is placed in the message's XID field in order to match this RPC Call message to a corresponding RPC Reply message.

REPLY
: An "RPC Reply message" reports the results of work requested by an RPC Call message. An RPC Reply message is designated by the value one (1) in the message's msg_type field. The value contained in an RPC Reply message's XID field is copied from the RPC Call message whose results are being reported.

The RPC client endpoint acts as a "Requester". It serializes the procedure's arguments and conveys them to a server endpoint via an RPC Call message. This message contains an RPC protocol header, a header describing the requested upper-layer operation, and all arguments.

The RPC server endpoint acts as a "Responder". It deserializes the arguments and processes the requested operation. It then serializes the operation's results into another byte stream. This byte stream is conveyed back to the Requester via an RPC Reply message. This message contains an RPC protocol header, a header describing the upper-layer reply, and all results.

The Requester deserializes the results and allows the original caller to proceed. At this point, the RPC transaction designated by the XID in the RPC Call message is complete, and the XID is retired.

In summary, RPC Call messages are sent by Requesters to Responders to initiate RPC transactions. RPC Reply messages are sent by Responders to Requesters to complete the processing on an RPC transaction.

2.2.3. RPC Transports

The role of an "RPC transport" is to mediate the exchange of RPC messages between Requesters and Responders. An RPC transport bridges the gap between the RPC message abstraction and the native operations of a particular network transport.

RPC-over-RDMA is a connection-oriented RPC transport. When a connection-oriented transport is used, clients initiate transport connections, while servers wait passively for incoming connection requests.

2.2.4. External Data Representation

One cannot assume that all Requesters and Responders represent data objects the same way internally. RPC uses External Data Representation (XDR) to translate native data types and serialize arguments and results [RFC4506].

The XDR protocol encodes data independently of the endianness or size of host-native data types, allowing unambiguous decoding of data on the receiving end. RPC Programs are specified by writing an XDR definition of their procedures, argument data types, and result data types.

XDR assumes that the number of bits in a byte (octet) and their order are the same on both endpoints and on the physical network. The smallest indivisible unit of XDR encoding is a group of four octets. XDR also flattens lists, arrays, and other complex data types so they can be conveyed as a stream of bytes.

A serialized stream of bytes that is the result of XDR encoding is referred to as an "XDR stream". A sending endpoint encodes native data into an XDR stream and then transmits that stream to a receiver. A receiving endpoint decodes incoming XDR byte streams into its native data representation format.

2.2.4.1. XDR Opaque Data

Sometimes, a data item must be transferred as is: without encoding or decoding. The contents of such a data item are referred to as "opaque data". XDR encoding places the content of opaque data items directly into an XDR stream without altering it in any way. ULPs or applications perform any needed data translation in this case. Examples of opaque data items include the content of files or generic byte strings.

2.2.4.2. XDR Roundup

The number of octets in a variable-length data item precedes that item in an XDR stream. If the size of an encoded data item is not a multiple of four octets, octets containing zero are added after the end of the item; this is the case so that the next encoded data item in the XDR stream starts on a four-octet boundary. The encoded size of the item is not changed by the addition of the extra octets. These extra octets are never exposed to ULPs.

This technique is referred to as "XDR roundup", and the extra octets are referred to as "XDR roundup padding".

2.3. RDMA

RPC Requesters and Responders can be made more efficient if large RPC messages are transferred by a third party, such as intelligent network-interface hardware (data movement offload), and placed in the receiver's memory so that no additional adjustment of data alignment has to be made (direct data placement or "DDP"). RDMA transports enable both optimizations.

2.3.1. DDP

Typically, RPC implementations copy the contents of RPC messages into a buffer before being sent. An efficient RPC implementation sends bulk data without copying it into a separate send buffer first.

However, socket-based RPC implementations are often unable to receive data directly into its final place in memory. Receivers often need to copy incoming data to finish an RPC operation: sometimes, only to adjust data alignment.

In this document, "RDMA" refers to the physical mechanism an RDMA transport utilizes when moving data. Although this may not be efficient, before an RDMA transfer, a sender may copy data into an intermediate buffer. After an RDMA transfer, a receiver may copy that data again to its final destination.

In this document, the term "DDP" refers to any optimized data transfer where it is unnecessary for a receiving host's CPU to copy transferred data to another location after it has been received.

Just as [RFC5666] did, this document focuses on the use of RDMA Read and Write operations to achieve both data movement offload and DDP. However, not all RDMA-based data transfer qualifies as DDP, and DDP can be achieved using non-RDMA mechanisms.

2.3.2. RDMA Transport Requirements

To achieve good performance during receive operations, RDMA transports require that RDMA consumers provision resources in advance to receive incoming messages.

An RDMA consumer might provide Receive buffers in advance by posting an RDMA Receive Work Request for every expected RDMA Send from a remote peer. These buffers are provided before the remote peer posts RDMA Send Work Requests; thus, this is often referred to as "pre-posting" buffers.

An RDMA Receive Work Request remains outstanding until hardware matches it to an inbound Send operation. The resources associated with that Receive must be retained in host memory, or "pinned", until the Receive completes.

Given these basic tenets of RDMA transport operation, the RPC-over-RDMA version 1 protocol assumes each transport provides the following abstract operations. A more complete discussion of these operations is found in [RFC5040].

Registered Memory
: Registered memory is a region of memory that is assigned a steering tag that temporarily permits access by the RDMA provider to perform data-transfer operations. The RPC-over-RDMA version 1 protocol assumes that each region of registered memory MUST be identified with a steering tag of no more than 32 bits and memory addresses of up to 64 bits in length.

RDMA Send
: The RDMA provider supports an RDMA Send operation, with completion signaled on the receiving peer after data has been placed in a pre-posted buffer. Sends complete at the receiver in the order they were issued at the sender. The amount of data transferred by a single RDMA Send operation is limited by the size of the remote peer’s pre-posted buffers.

RDMA Receive
: The RDMA provider supports an RDMA Receive operation to receive data conveyed by incoming RDMA Send operations. To reduce the amount of memory that must remain pinned awaiting incoming Sends, the amount of pre-posted memory is limited. Flow control to prevent overrunning receiver resources is provided by the RDMA consumer (in this case, the RPC-over-RDMA version 1 protocol).

RDMA Write
: The RDMA provider supports an RDMA Write operation to place data directly into a remote memory region. The local host initiates an RDMA Write, and completion is signaled there. No completion is signaled on the remote peer. The local host provides a steering tag, memory address, and length of the remote peer’s memory region.

  RDMA Writes are not ordered with respect to one another, but are ordered with respect to RDMA Sends. A subsequent RDMA Send completion obtained at the write initiator guarantees that prior RDMA Write data has been successfully placed in the remote peer’s memory.

RDMA Read
: The RDMA provider supports an RDMA Read operation to place peer source data directly into the read initiator's memory. The local host initiates an RDMA Read, and completion is signaled there. No completion is signaled on the remote peer. The local host provides steering tags, memory addresses, and a length for the remote source and local destination memory region.

  The local host signals Read completion to the remote peer as part of a subsequent RDMA Send message. The remote peer can then release steering tags and subsequently free associated source memory regions.

The RPC-over-RDMA version 1 protocol is designed to be carried over RDMA transports that support the above abstract operations. This protocol conveys information sufficient for an RPC peer to direct an RDMA provider to perform transfers containing RPC data and to communicate their result(s).

3. RPC-over-RDMA Protocol Framework

3.1. Transfer Models

A "transfer model" designates which endpoint exposes its memory and which is responsible for initiating the transfer of data. To enable RDMA Read and Write operations, for example, an endpoint first exposes regions of its memory to a remote endpoint, which initiates these operations against the exposed memory.

Read-Read
: Requesters expose their memory to the Responder, and the Responder exposes its memory to Requesters. The Responder reads, or pulls, RPC arguments or whole RPC calls from each Requester. Requesters pull RPC results or whole RPC relies from the Responder.

Write-Write
: Requesters expose their memory to the Responder, and the Responder exposes its memory to Requesters. Requesters write, or push, RPC arguments or whole RPC calls to the Responder. The Responder pushes RPC results or whole RPC relies to each Requester.

Read-Write
: Requesters expose their memory to the Responder, but the Responder does not expose its memory. The Responder pulls RPC arguments or whole RPC calls from each Requester. The Responder pushes RPC results or whole RPC relies to each Requester.

Write-Read
: The Responder exposes its memory to Requesters, but Requesters do not expose their memory. Requesters push RPC arguments or whole RPC calls to the Responder. Requesters pull RPC results or whole RPC relies from the Responder.

3.2. Message Framing

On an RPC-over-RDMA transport, each RPC message is encapsulated by an RPC-over-RDMA message. An RPC-over-RDMA message consists of two XDR streams.

RPC Payload Stream
: The "Payload stream" contains the encapsulated RPC message being transferred by this RPC-over-RDMA message. This stream always begins with the Transaction ID (XID) field of the encapsulated RPC message.

Transport Stream
: The "Transport stream" contains a header that describes and controls the transfer of the Payload stream in this RPC-over-RDMA message. This header is analogous to the record marking used for RPC on TCP sockets but is more extensive, since RDMA transports support several modes of data transfer.

In its simplest form, an RPC-over-RDMA message consists of a Transport stream followed immediately by a Payload stream conveyed together in a single RDMA Send. To transmit large RPC messages, a combination of one RDMA Send operation and one or more other RDMA operations is employed.

RPC-over-RDMA framing replaces all other RPC framing (such as TCP record marking) when used atop an RPC-over-RDMA association, even when the underlying RDMA protocol may itself be layered atop a transport with a defined RPC framing (such as TCP).

However, it is possible for RPC-over-RDMA to be dynamically enabled in the course of negotiating the use of RDMA via a ULP exchange. Because RPC framing delimits an entire RPC request or reply, the resulting shift in framing must occur between distinct RPC messages, and in concert with the underlying transport.

3.3. Managing Receiver Resources

It is critical to provide RDMA Send flow control for an RDMA connection. If any pre-posted Receive buffer on the connection is not large enough to accept an incoming RDMA Send, or if a pre-posted Receive buffer is not available to accept an incoming RDMA Send, the

RDMA connection can be terminated. This is different than conventional TCP/IP networking, in which buffers are allocated dynamically as messages are received.

The longevity of an RDMA connection mandates that sending endpoints respect the resource limits of peer receivers. To ensure messages can be sent and received reliably, there are two operational parameters for each connection.

3.3.1. RPC-over-RDMA Credits

Flow control for RDMA Send operations directed to the Responder is implemented as a simple request/grant protocol in the RPC-over-RDMA header associated with each RPC message.

An RPC-over-RDMA version 1 credit is the capability to handle one RPC-over-RDMA transaction. Each RPC-over-RDMA message sent from Requester to Responder requests a number of credits from the Responder. Each RPC-over-RDMA message sent from Responder to Requester informs the Requester how many credits the Responder has granted. The requested and granted values are carried in each RPC-over-RDMA message's rdma_credit field (see Section 4.2.3).

Practically speaking, the critical value is the granted value. A Requester MUST NOT send unacknowledged requests in excess of the Responder's granted credit limit. If the granted value is exceeded, the RDMA layer may signal an error, possibly terminating the connection. The granted value MUST NOT be zero, since such a value would result in deadlock.

RPC calls complete in any order, but the current granted credit limit at the Responder is known to the Requester from RDMA Send ordering properties. The number of allowed new requests the Requester may send is then the lower of the current requested and granted credit values, minus the number of requests in flight. Advertised credit values are not altered when individual RPCs are started or completed.

The requested and granted credit values MAY be adjusted to match the needs or policies in effect on either peer. For instance, a Responder may reduce the granted credit value to accommodate the available resources in a Shared Receive Queue. The Responder MUST ensure that an increase in receive resources is effected before the next RPC Reply message is sent.

A Requester MUST maintain enough receive resources to accommodate expected replies. Responders have to be prepared for there to be no receive resources available on Requesters with no pending RPC transactions.

Certain RDMA implementations may impose additional flow-control restrictions, such as limits on RDMA Read operations in progress at the Responder. Accommodation of such restrictions is considered the responsibility of each RPC-over-RDMA version 1 implementation.

3.3.2. Inline Threshold

An "inline threshold" value is the largest message size (in octets) that can be conveyed in one direction between peer implementations using RDMA Send and Receive. The inline threshold value is the smaller of the largest number of bytes the sender can post via a single RDMA Send operation and the largest number of bytes the receiver can accept via a single RDMA Receive operation. Each connection has two inline threshold values: one for messages flowing from Requester-to-Responder (referred to as the "call inline threshold") and one for messages flowing from Responder-to-Requester (referred to as the "reply inline threshold").

Unlike credit limits, inline threshold values are not advertised to peers via the RPC-over-RDMA version 1 protocol, and there is no provision for inline threshold values to change during the lifetime of an RPC-over-RDMA version 1 connection.

3.3.3. Initial Connection State

When a connection is first established, peers might not know how many receive resources the other has, nor how large the other peer’s inline thresholds are.

As a basis for an initial exchange of RPC requests, each RPC-over-RDMA version 1 connection provides the ability to exchange at least one RPC message at a time, whose RPC Call and Reply messages are no more than 1024 bytes in size. A Responder MAY exceed this basic level of configuration, but a Requester MUST NOT assume more than one credit is available and MUST receive a valid reply from the Responder carrying the actual number of available credits, prior to sending its next request.

Receiver implementations MUST support inline thresholds of 1024 bytes but MAY support larger inline thresholds values. An independent mechanism for discovering a peer’s inline thresholds before a connection is established may be used to optimize the use of RDMA Send and Receive operations. In the absence of such a mechanism, senders and receives MUST assume the inline thresholds are 1024 bytes.

3.4. XDR Encoding with Chunks

When a DDP capability is available, the transport places the contents of one or more XDR data items directly into the receiver's memory, separately from the transfer of other parts of the containing XDR stream.

3.4.1. Reducing an XDR Stream

RPC-over-RDMA version 1 provides a mechanism for moving part of an RPC message via a data transfer distinct from an RDMA Send/Receive pair. The sender removes one or more XDR data items from the Payload stream. They are conveyed via other mechanisms, such as one or more RDMA Read or Write operations. As the receiver decodes an incoming message, it skips over directly placed data items.

The portion of an XDR stream that is split out and moved separately is referred to as a "chunk". In some contexts, data in an RPC-over-RDMA header that describes these split out regions of memory may also be referred to as a "chunk".

A Payload stream after chunks have been removed is referred to as a "reduced" Payload stream. Likewise, a data item that has been removed from a Payload stream to be transferred separately is referred to as a "reduced" data item.

3.4.2. DDP-Eligibility

Not all XDR data items benefit from DDP. For example, small data items or data items that require XDR unmarshaling by the receiver do not benefit from DDP. In addition, it is impractical for receivers to prepare for every possible XDR data item in a protocol to be transferred in a chunk.

To maintain interoperability on an RPC-over-RDMA transport, a determination must be made of which few XDR data items in each ULP are allowed to use DDP.

This is done by additional specifications that describe how ULPs employ DDP. A "ULB specification" identifies which specific individual XDR data items in a ULP MAY be transferred via DDP. Such data items are referred to as "DDP-eligible". All other XDR data items MUST NOT be reduced.

Detailed requirements for ULBs are provided in Section 6.

3.4.3. RDMA Segments

When encoding a Payload stream that contains a DDP-eligible data item, a sender may choose to reduce that data item. When it chooses to do so, the sender does not place the item into the Payload stream. Instead, the sender records in the RPC-over-RDMA header the location and size of the memory region containing that data item.

The Requester provides location information for DDP-eligible data items in both RPC Call and Reply messages. The Responder uses this information to retrieve arguments contained in the specified region of the Requester's memory or place results in that memory region.

An "RDMA segment", or "plain segment", is an RPC-over-RDMA Transport header data object that contains the precise coordinates of a contiguous memory region that is to be conveyed separately from the Payload stream. Plain segments contain the following information:

Handle
: Steering tag (STag) or R_key generated by registering this memory with the RDMA provider.

Length
: The length of the RDMA segment's memory region, in octets. An "empty segment" is an RDMA segment with the value zero (0) in its length field.

Offset
: The offset or beginning memory address of the RDMA segment's memory region.

See [RFC5040] for further discussion.

3.4.4. Chunks

In RPC-over-RDMA version 1, a "chunk" refers to a portion of the Payload stream that is moved independently of the RPC-over-RDMA Transport header and Payload stream. Chunk data is removed from the sender's Payload stream, transferred via separate operations, and then reinserted into the receiver's Payload stream to form a complete RPC message.

Each chunk is comprised of RDMA segments. Each RDMA segment represents a single contiguous piece of that chunk. A Requester MAY divide a chunk into RDMA segments using any boundaries that are convenient. The length of a chunk is the sum of the lengths of the RDMA segments that comprise it.

The RPC-over-RDMA version 1 transport protocol does not place a limit on chunk size. However, each ULP may cap the amount of data that can be transferred by a single RPC (for example, NFS has "rsize" and "wsize", which restrict the payload size of NFS READ and WRITE operations). The Responder can use such limits to sanity check chunk sizes before using them in RDMA operations.

3.4.4.1. Counted Arrays

If a chunk contains a counted array data type, the count of array elements MUST remain in the Payload stream, while the array elements MUST be moved to the chunk. For example, when encoding an opaque byte array as a chunk, the count of bytes stays in the Payload stream, while the bytes in the array are removed from the Payload stream and transferred within the chunk.

Individual array elements appear in a chunk in their entirety. For example, when encoding an array of arrays as a chunk, the count of items in the enclosing array stays in the Payload stream, but each enclosed array, including its item count, is transferred as part of the chunk.

3.4.4.2. Optional-Data

If a chunk contains an optional-data data type, the "is present" field MUST remain in the Payload stream, while the data, if present, MUST be moved to the chunk.

3.4.4.3. XDR Unions

A union data type MUST NOT be made DDP-eligible, but one or more of its arms MAY be DDP-eligible, subject to the other requirements in this section.

3.4.4.4. Chunk Roundup

Except in special cases (covered in Section 3.5.3), a chunk MUST contain exactly one XDR data item. This makes it straightforward to reduce variable-length data items without affecting the XDR alignment of data items in the Payload stream.

When a variable-length XDR data item is reduced, the sender MUST remove XDR roundup padding for that data item from the Payload stream so that data items remaining in the Payload stream begin on four-byte alignment.

3.4.5. Read Chunks

A "Read chunk" represents an XDR data item that is to be pulled from the Requester to the Responder.

A Read chunk is a list of one or more RDMA read segments. An RDMA read segment consists of a Position field followed by a plain segment. See Section 4.1.2 for details.

Position
: The byte offset in the unreduced Payload stream where the receiver reinserts the data item conveyed in a chunk. The Position value MUST be computed from the beginning of the unreduced Payload stream, which begins at Position zero. All RDMA read segments belonging to the same Read chunk have the same value in their Position field.

While constructing an RPC Call message, a Requester registers memory regions that contain data to be transferred via RDMA Read operations. It advertises the coordinates of these regions in the RPC-over-RDMA Transport header of the RPC Call message.

After receiving an RPC Call message sent via an RDMA Send operation, a Responder transfers the chunk data from the Requester using RDMA Read operations. The Responder reconstructs the transferred chunk data by concatenating the contents of each RDMA segment, in list order, into the received Payload stream at the Position value recorded in that RDMA segment.

Put another way, the Responder inserts the first RDMA segment in a Read chunk into the Payload stream at the byte offset indicated by its Position field. RDMA segments whose Position field value match this offset are concatenated afterwards, until there are no more RDMA segments at that Position value.

The Position field in a read segment indicates where the containing Read chunk starts in the Payload stream. The value in this field MUST be a multiple of four. All segments in the same Read chunk share the same Position value, even if one or more of the RDMA segments have a non-four-byte-aligned length.

3.4.5.1. Decoding Read Chunks

While decoding a received Payload stream, whenever the XDR offset in the Payload stream matches that of a Read chunk, the Responder initiates an RDMA Read to pull the chunk's data content into registered local memory.

The Responder acknowledges its completion of use of Read chunk source buffers when it sends an RPC Reply message to the Requester. The Requester may then release Read chunks advertised in the request.

3.4.5.2. Read Chunk Roundup

When reducing a variable-length argument data item, the Requester SHOULD NOT include the data item's XDR roundup padding in the chunk. The length of a Read chunk is determined as follows:

- If the Requester chooses to include roundup padding in a Read chunk, the chunk's total length MUST be the sum of the encoded length of the data item and the length of the roundup padding. The length of the data item that was encoded into the Payload stream remains unchanged.

  The sender can increase the length of the chunk by adding another RDMA segment containing only the roundup padding, or it can do so by extending the final RDMA segment in the chunk.

- If the sender chooses not to include roundup padding in the chunk, the chunk's total length MUST be the same as the encoded length of the data item.

3.4.6. Write Chunks

While constructing an RPC Call message, a Requester prepares memory regions in which to receive DDP-eligible result data items. A "Write chunk" represents an XDR data item that is to be pushed from a Responder to a Requester. It is made up of an array of zero or more plain segments.

Write chunks are provisioned by a Requester long before the Responder has prepared the reply Payload stream. A Requester often does not know the actual length of the result data items to be returned, since the result does not yet exist. Thus, it MUST register Write chunks long enough to accommodate the maximum possible size of each returned data item.

In addition, the XDR position of DDP-eligible data items in the reply's Payload stream is not predictable when a Requester constructs an RPC Call message. Therefore, RDMA segments in a Write chunk do not have a Position field.

For each Write chunk provided by a Requester, the Responder pushes one data item to the Requester, filling the chunk contiguously and in segment array order until that data item has been completely written to the Requester. The Responder MUST copy the segment count and all segments from the Requester-provided Write chunk into the RPC Reply message's Transport header. As it does so, the Responder updates each segment length field to reflect the actual amount of data that is being returned in that segment. The Responder then sends the RPC Reply message via an RDMA Send operation.

An "empty Write chunk" is a Write chunk with a zero segment count. By definition, the length of an empty Write chunk is zero. An "unused Write chunk" has a non-zero segment count, but all of its segments are empty segments.

3.4.6.1. Decoding Write Chunks

After receiving the RPC Reply message, the Requester reconstructs the transferred data by concatenating the contents of each segment, in array order, into the RPC Reply message's XDR stream at the known XDR position of the associated DDP-eligible result data item.

3.4.6.2. Write Chunk Roundup

When provisioning a Write chunk for a variable-length result data item, the Requester SHOULD NOT include additional space for XDR roundup padding. A Responder MUST NOT write XDR roundup padding into a Write chunk, even if the Requester made space available for it. Therefore, when returning a single variable-length result data item, a returned Write chunk's total length MUST be the same as the encoded length of the result data item.

3.5. Message Size

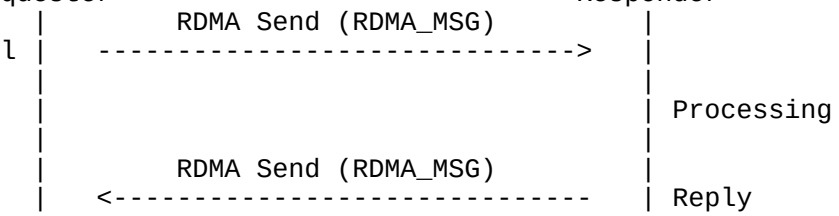
A receiver of RDMA Send operations is required by RDMA to have previously posted one or more adequately sized buffers. Memory savings are achieved on both Requesters and Responders by posting small Receive buffers. However, not all RPC messages are small. RPC-over-RDMA version 1 provides several mechanisms that allow messages of any size to be conveyed efficiently.

3.5.1. Short Messages

RPC messages are frequently smaller than typical inline thresholds. For example, the NFS version 3 GETATTR operation is only 56 bytes: 20 bytes of RPC header, a 32-byte file handle argument, and 4 bytes for its length. The reply to this common request is about 100 bytes.

Since all RPC messages conveyed via RPC-over-RDMA require an RDMA Send operation, the most efficient way to send an RPC message that is smaller than the inline threshold is to append the Payload stream directly to the Transport stream. An RPC-over-RDMA header with a small RPC Call or Reply message immediately following is transferred using a single RDMA Send operation. No other operations are needed.

An RPC-over-RDMA transaction using Short Messages:

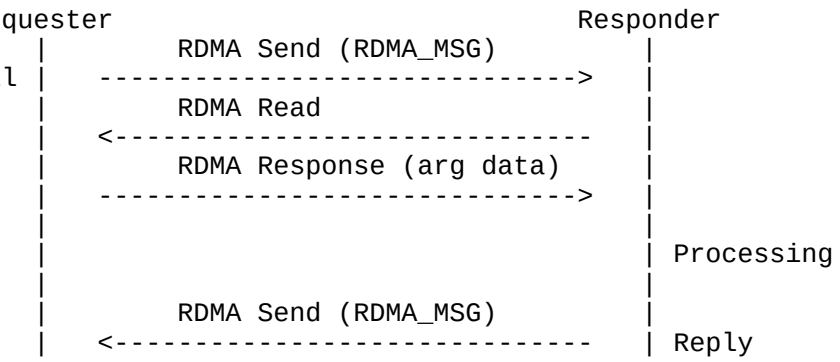


3.5.2. Chunked Messages

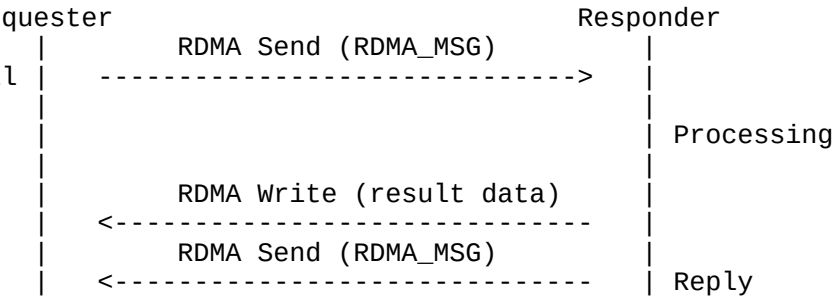
If DDP-eligible data items are present in a Payload stream, a sender MAY reduce some or all of these items by removing them from the Payload stream. The sender uses a separate mechanism to transfer the reduced data items. The Transport stream with the reduced Payload stream immediately following is then transferred using a single RDMA Send operation.

After receiving the Transport and Payload streams of an RPC Call message accompanied by Read chunks, the Responder uses RDMA Read operations to move reduced data items in Read chunks. Before sending the Transport and Payload streams of an RPC Reply message containing Write chunks, the Responder uses RDMA Write operations to move reduced data items in Write and Reply chunks.

An RPC-over-RDMA transaction with a Read chunk:



An RPC-over-RDMA transaction with a Write chunk:



3.5.3. Long Messages

When a Payload stream is larger than the receiver's inline threshold, the Payload stream is reduced by removing DDP-eligible data items and placing them in chunks to be moved separately. If there are no DDP-eligible data items in the Payload stream, or the Payload stream is still too large after it has been reduced, the RDMA transport MUST use RDMA Read or Write operations to convey the Payload stream itself. This mechanism is referred to as a "Long Message".

To transmit a Long Message, the sender conveys only the Transport stream with an RDMA Send operation. The Payload stream is not included in the Send buffer in this instance. Instead, the Requester provides chunks that the Responder uses to move the Payload stream.

Long Call
: To send a Long Call message, the Requester provides a special Read chunk that contains the RPC Call message's Payload stream. Every RDMA read segment in this chunk MUST contain zero in its Position field. Thus, this chunk is known as a "Position Zero Read chunk".

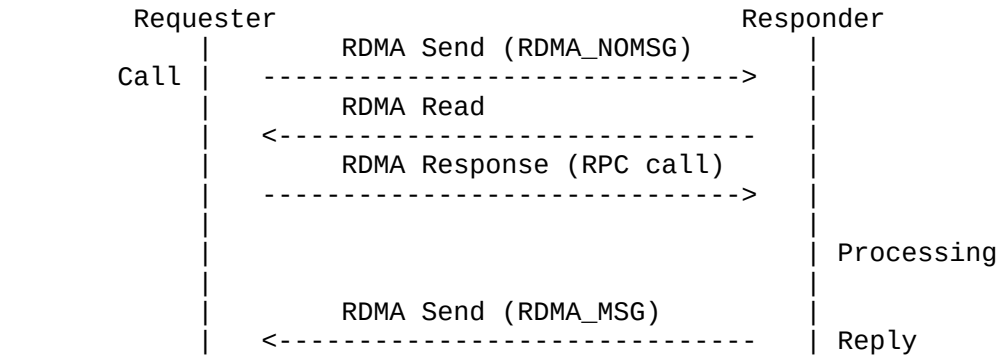
Long Reply
: To send a Long Reply, the Requester provides a single special Write chunk in advance, known as the "Reply chunk", that will contain the RPC Reply message's Payload stream. The Requester sizes the Reply chunk to accommodate the maximum expected reply size for that upper-layer operation.

Though the purpose of a Long Message is to handle large RPC messages, Requesters MAY use a Long Message at any time to convey an RPC Call message.

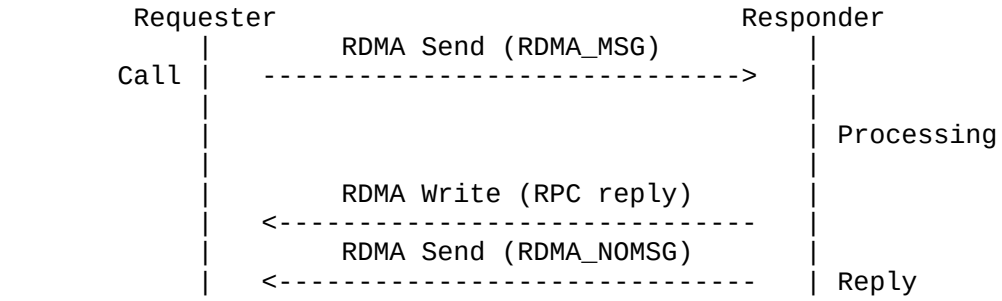
A Responder chooses which form of reply to use based on the chunks provided by the Requester. If Write chunks were provided and the Responder has a DDP-eligible result, it first reduces the reply Payload stream. If a Reply chunk was provided and the reduced Payload stream is larger than the reply inline threshold, the Responder MUST use the Requester-provided Reply chunk for the reply.

XDR data items may appear in these special chunks without regard to their DDP-eligibility. As these chunks contain a Payload stream, such chunks MUST include appropriate XDR roundup padding to maintain proper XDR alignment of their contents.

An RPC-over-RDMA transaction using a Long Call:



An RPC-over-RDMA transaction using a Long Reply:



4. RPC-over-RDMA in Operation

Every RPC-over-RDMA version 1 message has a header that includes a copy of the message's transaction ID, data for managing RDMA flow-control credits, and lists of RDMA segments describing chunks. All RPC-over-RDMA header content is contained in the Transport stream; thus, it MUST be XDR encoded.

RPC message layout is unchanged from that described in [RFC5531] except for the possible reduction of data items that are moved by separate operations.

The RPC-over-RDMA protocol passes RPC messages without regard to their type (CALL or REPLY). Apart from restrictions imposed by ULBs, each endpoint of a connection MAY send RDMA_MSG or RDMA_NOMSG message header types at any time (subject to credit limits).

4.1. XDR Protocol Definition

This section contains a description of the core features of the RPC-over-RDMA version 1 protocol, expressed in the XDR language [RFC4506].

This description is provided in a way that makes it simple to extract into ready-to-compile form. The reader can apply the following shell script to this document to produce a machine-readable XDR description of the RPC-over-RDMA version 1 protocol.

<CODE BEGINS>

```
#!/bin/sh
grep '^ *///' | sed 's?^ /// ??' | sed 's?^ *///$??'
```

<CODE ENDS>

That is, if the above script is stored in a file called "extract.sh" and this document is in a file called "spec.txt", then the reader can do the following to extract an XDR description file:

<CODE BEGINS>

```
sh extract.sh < spec.txt > rpcrdma_corev1.x
```

<CODE ENDS>

4.1.1. Code Component License

Code components extracted from this document must include the following license text. When the extracted XDR code is combined with other complementary XDR code, which itself has an identical license, only a single copy of the license text need be preserved.

<CODE BEGINS>

```
/// /*
///  * Copyright (c) 2010-2017 IETF Trust and the persons
///  * identified as authors of the code.  All rights reserved.
///  *
///  * The authors of the code are:
///  * B. Callaghan, T. Talpey, and C. Lever
///  *
///  * Redistribution and use in source and binary forms, with
///  * or without modification, are permitted provided that the
///  * following conditions are met:
///  *
///  * - Redistributions of source code must retain the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer.
///  *
///  * - Redistributions in binary form must reproduce the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer in the documentation and/or other
///  *   materials provided with the distribution.
///  *
///  * - Neither the name of Internet Society, IETF or IETF
///  *   Trust, nor the names of specific contributors, may be
///  *   used to endorse or promote products derived from this
///  *   software without specific prior written permission.
///  *
///  *   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
///  *   AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
///  *   WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
///  *   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
///  *   FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
///  *   EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
///  *   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
///  *   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
///  *   NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
///  *   SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
///  *   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
///  *   LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
///  *   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
///  *   IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
///  *   ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
///  */
///
```

<CODE ENDS>

4.1.2. RPC-over-RDMA Version 1 XDR

XDR data items defined in this section encodes the Transport Header Stream in each RPC-over-RDMA version 1 message. Comments identify items that cannot be changed in subsequent versions.

<CODE BEGINS>

```
/// /*
///  * Plain RDMA segment (Section 3.4.3)
///  */
/// struct xdr_rdma_segment {
///    uint32 handle;           /* Registered memory handle */
///    uint32 length;           /* Length of the chunk in bytes */
///    uint64 offset;           /* Chunk virtual address or offset */
/// };
///
/// /*
///  * RDMA read segment (Section 3.4.5)
///  */
/// struct xdr_read_chunk {
///    uint32 position;        /* Position in XDR stream */
///    struct xdr_rdma_segment target;
/// };
///
/// /*
///  * Read list (Section 4.3.1)
///  */
/// struct xdr_read_list {
///         struct xdr_read_chunk entry;
///         struct xdr_read_list  *next;
/// };
///
/// /*
///  * Write chunk (Section 3.4.6)
///  */
/// struct xdr_write_chunk {
///         struct xdr_rdma_segment target<>;
/// };
///
/// /*
///  * Write list (Section 4.3.2)
///  */
/// struct xdr_write_list {
///         struct xdr_write_chunk entry;
///         struct xdr_write_list  *next;
/// };
///
```

```
/// /*
///  * Chunk lists (Section 4.3)
///  */
/// struct rpc_rdma_header {
///    struct xdr_read_list   *rdma_reads;
///    struct xdr_write_list  *rdma_writes;
///    struct xdr_write_chunk *rdma_reply;
///    /* rpc body follows */
/// };
///
/// struct rpc_rdma_header_nomsg {
///    struct xdr_read_list   *rdma_reads;
///    struct xdr_write_list  *rdma_writes;
///    struct xdr_write_chunk *rdma_reply;
/// };
///
/// /* Not to be used */
/// struct rpc_rdma_header_padded {
///    uint32                 rdma_align;
///    uint32                 rdma_thresh;
///    struct xdr_read_list   *rdma_reads;
///    struct xdr_write_list  *rdma_writes;
///    struct xdr_write_chunk *rdma_reply;
///    /* rpc body follows */
/// };
///
/// /*
///  * Error handling (Section 4.5)
///  */
/// enum rpc_rdma_errcode {
///    ERR_VERS = 1,       /* Value fixed for all versions */
///    ERR_CHUNK = 2
/// };
///
/// /* Structure fixed for all versions */
/// struct rpc_rdma_errvers {
///    uint32 rdma_vers_low;
///    uint32 rdma_vers_high;
/// };
///
/// union rpc_rdma_error switch (rpc_rdma_errcode err) {
///    case ERR_VERS:
///      rpc_rdma_errvers range;
///    case ERR_CHUNK:
///      void;
/// };
///
/// /*
```

```
///  * Procedures (Section 4.2.4)
///  */
/// enum rdma_proc {
///    RDMA_MSG = 0,     /* Value fixed for all versions */
///    RDMA_NOMSG = 1,   /* Value fixed for all versions */
///    RDMA_MSGP = 2,    /* Not to be used */
///    RDMA_DONE = 3,    /* Not to be used */
///    RDMA_ERROR = 4    /* Value fixed for all versions */
/// };
///
/// /* The position of the proc discriminator field is
///  * fixed for all versions */
/// union rdma_body switch (rdma_proc proc) {
///    case RDMA_MSG:
///      rpc_rdma_header rdma_msg;
///    case RDMA_NOMSG:
///      rpc_rdma_header_nomsg rdma_nomsg;
///    case RDMA_MSGP:   /* Not to be used */
///      rpc_rdma_header_padded rdma_msgp;
///    case RDMA_DONE:   /* Not to be used */
///      void;
///    case RDMA_ERROR:
///      rpc_rdma_error rdma_error;
/// };
///
/// /*
///  * Fixed header fields (Section 4.2)
///  */
/// struct rdma_msg {
///    uint32    rdma_xid;      /* Position fixed for all versions */
///    uint32    rdma_vers;     /* Position fixed for all versions */
///    uint32    rdma_credit;   /* Position fixed for all versions */
///    rdma_body rdma_body;
/// };
```

<CODE ENDS>

4.2. Fixed Header Fields

The RPC-over-RDMA header begins with four fixed 32-bit fields that control the RDMA interaction.

The first three words are individual fields in the rdma_msg structure. The fourth word is the first word of the rdma_body union, which acts as the discriminator for the switched union. The contents of this field are described in Section 4.2.4.

These four fields must remain with the same meanings and in the same positions in all subsequent versions of the RPC-over-RDMA protocol.

4.2.1. Transaction ID (XID)

The XID generated for the RPC Call and Reply messages. Having the XID at a fixed location in the header makes it easy for the receiver to establish context as soon as each RPC-over-RDMA message arrives. This XID MUST be the same as the XID in the RPC message. The receiver MAY perform its processing based solely on the XID in the RPC-over-RDMA header, and thereby ignore the XID in the RPC message, if it so chooses.

4.2.2. Version Number

For RPC-over-RDMA version 1, this field MUST contain the value one (1). Rules regarding changes to this transport protocol version number can be found in Section 7.

4.2.3. Credit Value

When sent with an RPC Call message, the requested credit value is provided. When sent with an RPC Reply message, the granted credit value is returned. Further discussion of how the credit value is determined can be found in Section 3.3.

4.2.4. Procedure Number

RDMA_MSG = 0	indicates that chunk lists and a Payload stream follow. The format of the chunk lists is discussed below.
RDMA_NOMSG = 1	indicates that after the chunk lists there is no Payload stream. In this case, the chunk lists provide information to allow the Responder to transfer the Payload stream using explicit RDMA operations.
RDMA_MSGP = 2	is reserved.
RDMA_DONE = 3	is reserved.
RDMA_ERROR = 4	is used to signal an encoding error in the RPC-over-RDMA header.

An RDMA_MSG procedure conveys the Transport stream and the Payload stream via an RDMA Send operation. The Transport stream contains the four fixed fields followed by the Read and Write lists and the Reply

chunk, though any or all three MAY be marked as not present. The Payload stream then follows, beginning with its XID field. If a Read or Write chunk list is present, a portion of the Payload stream has been reduced and is conveyed via separate operations.

An RDMA_NOMSG procedure conveys the Transport stream via an RDMA Send operation. The Transport stream contains the four fixed fields followed by the Read and Write chunk lists and the Reply chunk. Though any of these MAY be marked as not present, one MUST be present and MUST hold the Payload stream for this RPC-over-RDMA message. If a Read or Write chunk list is present, a portion of the Payload stream has been excised and is conveyed via separate operations.

An RDMA_ERROR procedure conveys the Transport stream via an RDMA Send operation. The Transport stream contains the four fixed fields followed by formatted error information. No Payload stream is conveyed in this type of RPC-over-RDMA message.

A Requester MUST NOT send an RPC-over-RDMA header with the RDMA_ERROR procedure. A Responder MUST silently discard RDMA_ERROR procedures.

The Transport stream and Payload stream can be constructed in separate buffers. However, the total length of the gathered buffers cannot exceed the inline threshold.

4.3. Chunk Lists

The chunk lists in an RPC-over-RDMA version 1 header are three XDR optional-data fields that follow the fixed header fields in RDMA_MSG and RDMA_NOMSG procedures. Read Section 4.19 of [RFC4506] carefully to understand how optional-data fields work. Examples of XDR-encoded chunk lists are provided in Section 4.7 as an aid to understanding.

Often, an RPC-over-RDMA message has no associated chunks. In this case, the Read list, Write list, and Reply chunk are all marked "not present".

4.3.1. Read List

Each RDMA_MSG or RDMA_NOMSG procedure has one "Read list". The Read list is a list of zero or more RDMA read segments, provided by the Requester, that are grouped by their Position fields into Read chunks. Each Read chunk advertises the location of argument data the Responder is to pull from the Requester. The Requester has reduced the data items in these chunks from the call’s Payload stream.

A Requester may transmit the Payload stream of an RPC Call message using a Position Zero Read chunk. If the RPC Call message has no argument data that is DDP-eligible and the Position Zero Read chunk is not being used, the Requester leaves the Read list empty.

Responders MUST leave the Read list empty in all replies.

4.3.1.1. Matching Read Chunks to Arguments

When reducing a DDP-eligible argument data item, a Requester records the XDR stream offset of that data item in the Read chunk's Position field. The Responder can then tell unambiguously where that chunk is to be reinserted into the received Payload stream to form a complete RPC Call message.

4.3.2. Write List

Each RDMA_MSG or RDMA_NOMSG procedure has one "Write list". The Write list is a list of zero or more Write chunks, provided by the Requester. Each Write chunk is an array of plain segments; thus, the Write list is a list of counted arrays.

If an RPC Reply message has no possible DDP-eligible result data items, the Requester leaves the Write list empty. When a Requester provides a Write list, the Responder MUST push data corresponding to DDP-eligible result data items to Requester memory referenced in the Write list. The Responder removes these data items from the reply's Payload stream.

4.3.2.1. Matching Write Chunks to Results

A Requester constructs the Write list for an RPC transaction before the Responder has formulated its reply. When there is only one DDP-eligible result data item, the Requester inserts only a single Write chunk in the Write list. If the returned Write chunk is not an unused Write chunk, the Requester knows with certainty which result data item is contained in it.

When a Requester has provided multiple Write chunks, the Responder fills in each Write chunk with one DDP-eligible result until there are either no more DDP-eligible results or no more Write chunks.

The Requester might not be able to predict in advance which DDP-eligible data item goes in which chunk. Thus, the Requester is responsible for allocating and registering Write chunks large enough to accommodate the largest result data item that might be associated with each chunk in the Write list.

As a Requester decodes a reply Payload stream, it is clear from the contents of the RPC Reply message which Write chunk contains which result data item.

4.3.2.2. Unused Write Chunks

There are occasions when a Requester provides a non-empty Write chunk but the Responder is not able to use it. For example, a ULP may define a union result where some arms of the union contain a DDP-eligible data item while other arms do not. The Responder is required to use Requester-provided Write chunks in this case, but if the Responder returns a result that uses an arm of the union that has no DDP-eligible data item, that Write chunk remains unconsumed.

If there is a subsequent DDP-eligible result data item in the RPC Reply message, it MUST be placed in that unconsumed Write chunk. Therefore, the Requester MUST provision each Write chunk so it can be filled with the largest DDP-eligible data item that can be placed in it.

If this is the last or only Write chunk available and it remains unconsumed, the Responder MUST return this Write chunk as an unused Write chunk (see Section 3.4.6). The Responder sets the segment count to a value matching the Requester-provided Write chunk, but returns only empty segments in that Write chunk.

Unused Write chunks, or unused bytes in Write chunk segments, are returned to the RPC consumer as part of RPC completion. Even if a Responder indicates that a Write chunk is not consumed, the Responder may have written data into one or more segments before choosing not to return that data item. The Requester MUST NOT assume that the memory regions backing a Write chunk have not been modified.

4.3.2.3. Empty Write Chunks

To force a Responder to return a DDP-eligible result inline, a Requester employs the following mechanism:

- When there is only one DDP-eligible result item in an RPC Reply message, the Requester provides an empty Write list.

- When there are multiple DDP-eligible result data items and a Requester prefers that a data item is returned inline, the Requester provides an empty Write chunk for that item (see Section 3.4.6). The Responder MUST return the corresponding result data item inline and MUST return an empty Write chunk in that Write list position in the RPC Reply message.

As always, a Requester and Responder must prepare for a Long Reply to be used if the resulting RPC Reply might be too large to be conveyed in an RDMA Send.

4.3.3. Reply Chunk

Each RDMA_MSG or RDMA_NOMSG procedure has one "Reply chunk" slot. A Requester MUST provide a Reply chunk whenever the maximum possible size of the RPC Reply message's Transport and Payload streams is larger than the inline threshold for messages from Responder to Requester. Otherwise, the Requester marks the Reply chunk as not present.

If the Transport stream and Payload stream together are smaller than the reply inline threshold, the Responder MAY return the RPC Reply message as a Short message rather than using the Requester-provided Reply chunk.

When a Requester provides a Reply chunk in an RPC Call message, the Responder MUST copy that chunk into the Transport header of the RPC Reply message. As with Write chunks, the Responder modifies the copied Reply chunk in the RPC Reply message to reflect the actual amount of data that is being returned in the Reply chunk.

4.4. Memory Registration

The cost of registering and invalidating memory can be a significant proportion of the cost of an RPC-over-RDMA transaction. Thus, an important implementation consideration is how to minimize registration activity without exposing system memory needlessly.

4.4.1. Registration Longevity

Data transferred via RDMA Read and Write can reside in a memory allocation not in the control of the RPC-over-RDMA transport. These memory allocations can persist outside the bounds of an RPC transaction. They are registered and invalidated as needed, as part of each RPC transaction.

The Requester endpoint must ensure that memory regions associated with each RPC transaction are protected from Responder access before allowing upper-layer access to the data contained in them. Moreover, the Requester must not access these memory regions while the Responder has access to them.

This includes memory regions that are associated with canceled RPCs. A Responder cannot know that the Requester is no longer waiting for a reply, and it might proceed to read or even update memory that the Requester might have released for other use.

4.4.2. Communicating DDP-Eligibility

The interface by which a ULP implementation communicates the eligibility of a data item locally to its local RPC-over-RDMA endpoint is not described by this specification.

Depending on the implementation and constraints imposed by ULBs, it is possible to implement reduction transparently to upper layers. Such implementations may lead to inefficiencies, either because they require the RPC layer to perform expensive registration and invalidation of memory "on the fly", or they may require using RDMA chunks in RPC Reply messages, along with the resulting additional handshaking with the RPC-over-RDMA peer.

However, these issues are internal and generally confined to the local interface between RPC and its upper layers, one in which implementations are free to innovate. The only requirement, beyond constraints imposed by the ULB, is that the resulting RPC-over-RDMA protocol sent to the peer be valid for the upper layer.

4.4.3. Registration Strategies

The choice of which memory registration strategies to employ is left to Requester and Responder implementers. To support the widest array of RDMA implementations, as well as the most general steering tag scheme, an Offset field is included in each RDMA segment.

While zero-based offset schemes are available in many RDMA implementations, their use by RPC requires individual registration of each memory region. For such implementations, this can be a significant overhead. By providing an offset in each chunk, many pre-registration or region-based registrations can be readily supported.

4.5. Error Handling

A receiver performs basic validity checks on the RPC-over-RDMA header and chunk contents before it passes the RPC message to the RPC layer. If an incoming RPC-over-RDMA message is not as long as a minimal size RPC-over-RDMA header (28 bytes), the receiver cannot trust the value of the XID field; therefore, it MUST silently discard the message before performing any parsing. If other errors are detected in the RPC-over-RDMA header of an RPC Call message, a Responder MUST send an

RDMA_ERROR message back to the Requester. If errors are detected in the RPC-over-RDMA header of an RPC Reply message, a Requester MUST silently discard the message.

To form an RDMA_ERROR procedure:

- The rdma_xid field MUST contain the same XID that was in the rdma_xid field in the failing request;
- The rdma_vers field MUST contain the same version that was in the rdma_vers field in the failing request;
- The rdma_proc field MUST contain the value RDMA_ERROR; and
- The rdma_err field contains a value that reflects the type of error that occurred, as described below.

An RDMA_ERROR procedure indicates a permanent error. Receipt of this procedure completes the RPC transaction associated with XID in the rdma_xid field. A receiver MUST silently discard an RDMA_ERROR procedure that it cannot decode.

4.5.1. Header Version Mismatch

When a Responder detects an RPC-over-RDMA header version that it does not support (currently this document defines only version 1), it MUST reply with an RDMA_ERROR procedure and set the rdma_err value to ERR_VERS, also providing the low and high inclusive version numbers it does, in fact, support.

4.5.2. XDR Errors

A receiver might encounter an XDR parsing error that prevents it from processing the incoming Transport stream. Examples of such errors include an invalid value in the rdma_proc field; an RDMA_NOMSG message where the Read list, Write list, and Reply chunk are marked not present; or the value of the rdma_xid field does not match the value of the XID field in the accompanying RPC message. If the rdma_vers field contains a recognized value, but an XDR parsing error occurs, the Responder MUST reply with an RDMA_ERROR procedure and set the rdma_err value to ERR_CHUNK.

When a Responder receives a valid RPC-over-RDMA header but the Responder’s ULP implementation cannot parse the RPC arguments in the RPC Call message, the Responder SHOULD return an RPC Reply message with status GARBAGE_ARGS, using an RDMA_MSG procedure. This type of parsing failure might be due to mismatches between chunk sizes or offsets and the contents of the Payload stream, for example.

4.5.3. Responder RDMA Operational Errors

In RPC-over-RDMA version 1, the Responder initiates RDMA Read and Write operations that target the Requester's memory. Problems might arise as the Responder attempts to use Requester-provided resources for RDMA operations. For example:

- Usually, chunks can be validated only by using their contents to perform data transfers. If chunk contents are invalid (e.g., a memory region is no longer registered or a chunk length exceeds the end of the registered memory region), a Remote Access Error occurs.

- If a Requester's Receive buffer is too small, the Responder's Send operation completes with a Local Length Error.

- If the Requester-provided Reply chunk is too small to accommodate a large RPC Reply message, a Remote Access Error occurs. A Responder might detect this problem before attempting to write past the end of the Reply chunk.

RDMA operational errors are typically fatal to the connection. To avoid a retransmission loop and repeated connection loss that deadlocks the connection, once the Requester has re-established a connection, the Responder should send an RDMA_ERROR reply with an rdma_err value of ERR_CHUNK to indicate that no RPC-level reply is possible for that XID.

4.5.4. Other Operational Errors

While a Requester is constructing an RPC Call message, an unrecoverable problem might occur that prevents the Requester from posting further RDMA Work Requests on behalf of that message. As with other transports, if a Requester is unable to construct and transmit an RPC Call message, the associated RPC transaction fails immediately.

After a Requester has received a reply, if it is unable to invalidate a memory region due to an unrecoverable problem, the Requester MUST close the connection to protect that memory from Responder access before the associated RPC transaction is complete.

While a Responder is constructing an RPC Reply message or error message, an unrecoverable problem might occur that prevents the Responder from posting further RDMA Work Requests on behalf of that message. If a Responder is unable to construct and transmit an RPC Reply or RPC-over-RDMA error message, the Responder MUST close the connection to signal to the Requester that a reply was lost.

4.5.5. RDMA Transport Errors

The RDMA connection and physical link provide some degree of error detection and retransmission. iWARP's Marker PDU Aligned (MPA) layer (when used over TCP), the Stream Control Transmission Protocol (SCTP), as well as the InfiniBand [IBARCH] link layer all provide Cyclic Redundancy Check (CRC) protection of the RDMA payload, and CRC-class protection is a general attribute of such transports.

Additionally, the RPC layer itself can accept errors from the transport and recover via retransmission. RPC recovery can handle complete loss and re-establishment of a transport connection.

The details of reporting and recovery from RDMA link-layer errors are described in specific link-layer APIs and operational specifications and are outside the scope of this protocol specification. See Section 8 for further discussion of the use of RPC-level integrity schemes to detect errors.

4.6. Protocol Elements No Longer Supported

The following protocol elements are no longer supported in RPC-over-RDMA version 1. Related enum values and structure definitions remain in the RPC-over-RDMA version 1 protocol for backwards compatibility.

4.6.1. RDMA_MSGP

The specification of RDMA_MSGP in Section 3.9 of [RFC5666] is incomplete. To fully specify RDMA_MSGP would require:

- Updating the definition of DDP-eligibility to include data items that may be transferred, with padding, via RDMA_MSGP procedures
- Adding full operational descriptions of the alignment and threshold fields
- Discussing how alignment preferences are communicated between two peers without using CCP
- Describing the treatment of RDMA_MSGP procedures that convey Read or Write chunks

The RDMA_MSGP message type is beneficial only when the padded data payload is at the end of an RPC message's argument or result list. This is not typical for NFSv4 COMPOUND RPCs, which often include a GETATTR operation as the final element of the compound operation array.

Without a full specification of RDMA_MSGP, there has been no fully implemented prototype of it. Without a complete prototype of RDMA_MSGP support, it is difficult to assess whether this protocol element has benefit or can even be made to work interoperably.

Therefore, senders MUST NOT send RDMA_MSGP procedures. When receiving an RDMA_MSGP procedure, Responders SHOULD reply with an RDMA_ERROR procedure, setting the rdma_err field to ERR_CHUNK; Requesters MUST silently discard the message.

4.6.2. RDMA_DONE

Because no implementation of RPC-over-RDMA version 1 uses the Read-Read transfer model, there is never a need to send an RDMA_DONE procedure.

Therefore, senders MUST NOT send RDMA_DONE messages. Receivers MUST silently discard RDMA_DONE messages.

4.7. XDR Examples

RPC-over-RDMA chunk lists are complex data types. In this section, illustrations are provided to help readers grasp how chunk lists are represented inside an RPC-over-RDMA header.

A plain segment is the simplest component, being made up of a 32-bit handle (H), a 32-bit length (L), and 64 bits of offset (OO). Once flattened into an XDR stream, plain segments appear as

```
HLOO
```

An RDMA read segment has an additional 32-bit position field (P). RDMA read segments appear as

```
PHLOO
```

A Read chunk is a list of RDMA read segments. Each RDMA read segment is preceded by a 32-bit word containing a one if a segment follows or a zero if there are no more segments in the list. In XDR form, this would look like

```
1 PHLOO 1 PHLOO 1 PHLOO 0
```

where P would hold the same value for each RDMA read segment belonging to the same Read chunk.

The Read list is also a list of RDMA read segments. In XDR form, this would look like a Read chunk, except that the P values could vary across the list. An empty Read list is encoded as a single 32-bit zero.

One Write chunk is a counted array of plain segments. In XDR form, the count would appear as the first 32-bit word, followed by an HLOO for each element of the array. For instance, a Write chunk with three elements would look like

```
3 HLOO HLOO HLOO
```

The Write list is a list of counted arrays. In XDR form, this is a combination of optional-data and counted arrays. To represent a Write list containing a Write chunk with three segments and a Write chunk with two segments, XDR would encode

```
1 3 HLOO HLOO HLOO 1 2 HLOO HLOO 0
```

An empty Write list is encoded as a single 32-bit zero.

The Reply chunk is a Write chunk. However, since it is an optional-data field, there is a 32-bit field in front of it that contains a one if the Reply chunk is present or a zero if it is not. After encoding, a Reply chunk with two segments would look like

```
1 2 HLOO HLOO
```

Frequently, a Requester does not provide any chunks. In that case, after the four fixed fields in the RPC-over-RDMA header, there are simply three 32-bit fields that contain zero.

5. RPC Bind Parameters

In setting up a new RDMA connection, the first action by a Requester is to obtain a transport address for the Responder. The means used to obtain this address, and to open an RDMA connection, is dependent on the type of RDMA transport and is the responsibility of each RPC protocol binding and its local implementation.

RPC services normally register with a portmap or rpcbind service [RFC1833], which associates an RPC Program number with a service address. This policy is no different with RDMA transports. However, a different and distinct service address (port number) might sometimes be required for ULP operation with RPC-over-RDMA.

When mapped atop the iWARP transport [RFC5040] [RFC5041], which uses IP port addressing due to its layering on TCP and/or SCTP, port mapping is trivial and consists merely of issuing the port in the connection process. The NFS/RDMA protocol service address has been assigned port 20049 by IANA, for both iWARP/TCP and iWARP/SCTP [RFC5667].

When mapped atop InfiniBand [IBARCH], which uses a service endpoint naming scheme based on a Group Identifier (GID), a translation MUST be employed. One such translation is described in Annexes A3 (Application Specific Identifiers), A4 (Sockets Direct Protocol (SDP)), and A11 (RDMA IP CM Service) of [IBARCH], which is appropriate for translating IP port addressing to the InfiniBand network. Therefore, in this case, IP port addressing may be readily employed by the upper layer.

When a mapping standard or convention exists for IP ports on an RDMA interconnect, there are several possibilities for each upper layer to consider:

- One possibility is to have the Responder register its mapped IP port with the rpcbind service under the netid (or netids) defined here. An RPC-over-RDMA-aware Requester can then resolve its desired service to a mappable port and proceed to connect. This is the most flexible and compatible approach, for those upper layers that are defined to use the rpcbind service.

- A second possibility is to have the Responder's portmapper register itself on the RDMA interconnect at a "well-known" service address (on UDP or TCP, this corresponds to port 111). A Requester could connect to this service address and use the portmap protocol to obtain a service address in response to a program number, e.g., an iWARP port number or an InfiniBand GID.

- Alternately, the Requester could simply connect to the mapped well-known port for the service itself, if it is appropriately defined. By convention, the NFS/RDMA service, when operating atop such an InfiniBand fabric, uses the same 20049 assignment as for iWARP.

Historically, different RPC protocols have taken different approaches to their port assignment. Therefore, the specific method is left to each RPC-over-RDMA-enabled ULB and is not addressed in this document.

In Section 9, this specification defines two new netid values, to be used for registration of upper layers atop iWARP [RFC5040] [RFC5041] and (when a suitable port translation service is available) InfiniBand [IBARCH]. Additional RDMA-capable networks MAY define their own netids, or if they provide a port translation, they MAY share the one defined in this document.

6. ULB Specifications

An ULP is typically defined independently of any particular RPC transport. An ULB (ULB) specification provides guidance that helps the ULP interoperate correctly and efficiently over a particular transport. For RPC-over-RDMA version 1, a ULB may provide:

- A taxonomy of XDR data items that are eligible for DDP
- Constraints on which upper-layer procedures may be reduced and on how many chunks may appear in a single RPC request
- A method for determining the maximum size of the reply Payload stream for all procedures in the ULP
- An rpcbind port assignment for operation of the RPC Program and Version on an RPC-over-RDMA transport

Each RPC Program and Version tuple that utilizes RPC-over-RDMA version 1 needs to have a ULB specification.

6.1. DDP-Eligibility

An ULB designates some XDR data items as eligible for DDP. As an RPC-over-RDMA message is formed, DDP-eligible data items can be removed from the Payload stream and placed directly in the receiver’s memory.

An XDR data item should be considered for DDP-eligibility if there is a clear benefit to moving the contents of the item directly from the sender’s memory to the receiver’s memory. Criteria for DDP-eligibility include:

- The XDR data item is frequently sent or received, and its size is often much larger than typical inline thresholds.
- If the XDR data item is a result, its maximum size must be predictable in advance by the Requester.

- Transport-level processing of the XDR data item is not needed. For example, the data item is an opaque byte array, which requires no XDR encoding and decoding of its content.

- The content of the XDR data item is sensitive to address alignment. For example, a data copy operation would be required on the receiver to enable the message to be parsed correctly, or to enable the data item to be accessed.

- The XDR data item does not contain DDP-eligible data items.

In addition to defining the set of data items that are DDP-eligible, a ULB may also limit the use of chunks to particular upper-layer procedures. If more than one data item in a procedure is DDP-eligible, the ULB may also limit the number of chunks that a Requester can provide for a particular upper-layer procedure.

Senders MUST NOT reduce data items that are not DDP-eligible. Such data items MAY, however, be moved as part of a Position Zero Read chunk or a Reply chunk.

The programming interface by which an upper-layer implementation indicates the DDP-eligibility of a data item to the RPC transport is not described by this specification. The only requirements are that the receiver can re-assemble the transmitted RPC-over-RDMA message into a valid XDR stream, and that DDP-eligibility rules specified by the ULB are respected.

There is no provision to express DDP-eligibility within the XDR language. The only definitive specification of DDP-eligibility is a ULB.

In general, a DDP-eligibility violation occurs when:

- A Requester reduces a non-DDP-eligible argument data item. The Responder MUST NOT process this RPC Call message and MUST report the violation as described in Section 4.5.2.

- A Responder reduces a non-DDP-eligible result data item. The Requester MUST terminate the pending RPC transaction and report an appropriate permanent error to the RPC consumer.

- A Responder does not reduce a DDP-eligible result data item into an available Write chunk. The Requester MUST terminate the pending RPC transaction and report an appropriate permanent error to the RPC consumer.

6.2. Maximum Reply Size

A Requester provides resources for both an RPC Call message and its matching RPC Reply message. A Requester forms the RPC Call message itself; thus, the Requester can compute the exact resources needed.

A Requester must allocate resources for the RPC Reply message (an RPC-over-RDMA credit, a Receive buffer, and possibly a Write list and Reply chunk) before the Responder has formed the actual reply. To accommodate all possible replies for the procedure in the RPC Call message, a Requester must allocate reply resources based on the maximum possible size of the expected RPC Reply message.

If there are procedures in the ULP for which there is no clear reply size maximum, the ULB needs to specify a dependable means for determining the maximum.

6.3. Additional Considerations

There may be other details provided in a ULB.

- An ULB may recommend inline threshold values or other transport-related parameters for RPC-over-RDMA version 1 connections bearing that ULP.

- An ULP may provide a means to communicate these transport-related parameters between peers. Note that RPC-over-RDMA version 1 does not specify any mechanism for changing any transport-related parameter after a connection has been established.

- Multiple ULPs may share a single RPC-over-RDMA version 1 connection when their ULBs allow the use of RPC-over-RDMA version 1 and the rpcbind port assignments for the Protocols allow connection sharing. In this case, the same transport parameters (such as inline threshold) apply to all Protocols using that connection.

Each ULB needs to be designed to allow correct interoperation without regard to the transport parameters actually in use. Furthermore, implementations of ULPs must be designed to interoperate correctly regardless of the connection parameters in effect on a connection.

6.4. ULP Extensions

An RPC Program and Version tuple may be extensible. For instance, there may be a minor versioning scheme that is not reflected in the RPC version number, or the ULP may allow additional features to be specified after the original RPC Program specification was ratified.

ULBs are provided for interoperable RPC Programs and Versions by extending existing ULBs to reflect the changes made necessary by each addition to the existing XDR.

7. Protocol Extensibility

The RPC-over-RDMA header format is specified using XDR, unlike the message header used with RPC-over-TCP. To maintain a high degree of interoperability among implementations of RPC-over-RDMA, any change to this XDR requires a protocol version number change. New versions of RPC-over-RDMA may be published as separate protocol specifications without updating this document.

The first four fields in every RPC-over-RDMA header must remain aligned at the same fixed offsets for all versions of the RPC-over-RDMA protocol. The version number must be in a fixed place to enable implementations to detect protocol version mismatches.

For version mismatches to be reported in a fashion that all future version implementations can reliably decode, the rdma_proc field must remain in a fixed place, the value of ERR_VERS must always remain the same, and the field placement in struct rpc_rdma_errvers must always remain the same.

7.1. Conventional Extensions

Introducing new capabilities to RPC-over-RDMA version 1 is limited to the adoption of conventions that make use of existing XDR (defined in this document) and allowed abstract RDMA operations. Because no mechanism for detecting optional features exists in RPC-over-RDMA version 1, implementations must rely on ULPs to communicate the existence of such extensions.

Such extensions must be specified in a Standards Track RFC with appropriate review by the NFSv4 Working Group and the IESG. An example of a conventional extension to RPC-over-RDMA version 1 is the specification of backward direction message support to enable NFSv4.1 callback operations, described in [RFC8167].

8. Security Considerations

8.1. Memory Protection

A primary consideration is the protection of the integrity and confidentiality of local memory by an RPC-over-RDMA transport. The use of an RPC-over-RDMA transport protocol MUST NOT introduce vulnerabilities to system memory contents nor to memory owned by user processes.

It is REQUIRED that any RDMA provider used for RPC transport be conformant to the requirements of [RFC5042] in order to satisfy these protections. These protections are provided by the RDMA layer specifications, and in particular, their security models.

8.1.1. Protection Domains

The use of Protection Domains to limit the exposure of memory regions to a single connection is critical. Any attempt by an endpoint not participating in that connection to reuse memory handles needs to result in immediate failure of that connection. Because ULP security mechanisms rely on this aspect of Reliable Connection behavior, strong authentication of remote endpoints is recommended.

8.1.2. Handle Predictability

Unpredictable memory handles should be used for any operation requiring advertised memory regions. Advertising a continuously registered memory region allows a remote host to read or write to that region even when an RPC involving that memory is not under way. Therefore, implementations should avoid advertising persistently registered memory.

8.1.3. Memory Protection

Requesters should register memory regions for remote access only when they are about to be the target of an RPC operation that involves an RDMA Read or Write.

Registered memory regions should be invalidated as soon as related RPC operations are complete. Invalidation and DMA unmapping of memory regions should be complete before message integrity checking is done and before the RPC consumer is allowed to continue execution and use or alter the contents of a memory region.

An RPC transaction on a Requester might be terminated before a reply arrives if the RPC consumer exits unexpectedly (for example, it is signaled or a segmentation fault occurs). When an RPC terminates abnormally, memory regions associated with that RPC should be invalidated appropriately before the regions are released to be reused for other purposes on the Requester.

8.1.4. Denial of Service

A detailed discussion of denial-of-service exposures that can result from the use of an RDMA transport is found in Section 6.4 of [RFC5042].

A Responder is not obliged to pull Read chunks that are unreasonably large. The Responder can use an RDMA_ERROR response to terminate RPCs with unreadable Read chunks. If a Responder transmits more data than a Requester is prepared to receive in a Write or Reply chunk, the RDMA Network Interface Cards (RNICs) typically terminate the connection. For further discussion, see Section 4.5. Such repeated chunk errors can deny service to other users sharing the connection from the errant Requester.

An RPC-over-RDMA transport implementation is not responsible for throttling the RPC request rate, other than to keep the number of concurrent RPC transactions at or under the number of credits granted per connection. This is explained in Section 3.3.1. A sender can trigger a self denial of service by exceeding the credit grant repeatedly.

When an RPC has been canceled due to a signal or premature exit of an application process, a Requester may invalidate the RPC's Write and Reply chunks. Invalidation prevents the subsequent arrival of the Responder's reply from altering the memory regions associated with those chunks after the memory has been reused.

On the Requester, a malfunctioning application or a malicious user can create a situation where RPCs are continuously initiated and then aborted, resulting in Responder replies that terminate the underlying RPC-over-RDMA connection repeatedly. Such situations can deny service to other users sharing the connection from that Requester.

8.2. RPC Message Security

ONC RPC provides cryptographic security via the RPCSEC_GSS framework [RFC7861]. RPCSEC_GSS implements message authentication (rpc_gss_svc_none), per-message integrity checking (rpc_gss_svc_integrity), and per-message confidentiality (rpc_gss_svc_privacy) in the layer above RPC-over-RDMA. The latter two services require significant computation and movement of data on each endpoint host. Some performance benefits enabled by RDMA transports can be lost.

8.2.1. RPC-over-RDMA Protection at Lower Layers

For any RPC transport, utilizing RPCSEC_GSS integrity or privacy services has performance implications. Protection below the RPC transport is often more appropriate in performance-sensitive deployments, especially if it, too, can be offloaded. Certain configurations of IPsec can be co-located in RDMA hardware, for example, without change to RDMA consumers and little loss of data

movement efficiency. Such arrangements can also provide a higher degree of privacy by hiding endpoint identity or altering the frequency at which messages are exchanged, at a performance cost.

The use of protection in a lower layer MAY be negotiated through the use of an RPCSEC_GSS security flavor defined in [RFC7861] in conjunction with the Channel Binding mechanism [RFC5056] and IPsec Channel Connection Latching [RFC5660]. Use of such mechanisms is REQUIRED where integrity or confidentiality is desired and where efficiency is required.

8.2.2. RPCSEC_GSS on RPC-over-RDMA Transports

Not all RDMA devices and fabrics support the above protection mechanisms. Also, per-message authentication is still required on NFS clients where multiple users access NFS files. In these cases, RPCSEC_GSS can protect NFS traffic conveyed on RPC-over-RDMA connections.

RPCSEC_GSS extends the ONC RPC protocol [RFC5531] without changing the format of RPC messages. By observing the conventions described in this section, an RPC-over-RDMA transport can convey RPCSEC_GSS-protected RPC messages interoperably.

As part of the ONC RPC protocol, protocol elements of RPCSEC_GSS that appear in the Payload stream of an RPC-over-RDMA message (such as control messages exchanged as part of establishing or destroying a security context or data items that are part of RPCSEC_GSS authentication material) MUST NOT be reduced.

8.2.2.1. RPCSEC_GSS Context Negotiation

Some NFS client implementations use a separate connection to establish a Generic Security Service (GSS) context for NFS operation. These clients use TCP and the standard NFS port (2049) for context establishment. To enable the use of RPCSEC_GSS with NFS/RDMA, an NFS server MUST also provide a TCP-based NFS service on port 2049.

8.2.2.2. RPC-over-RDMA with RPCSEC_GSS Authentication

The RPCSEC_GSS authentication service has no impact on the DDP-eligibility of data items in a ULP.

However, RPCSEC_GSS authentication material appearing in an RPC message header can be larger than, say, an AUTH_SYS authenticator. In particular, when an RPCSEC_GSS pseudoflavor is in use, a Requester

needs to accommodate a larger RPC credential when marshaling RPC Call messages and needs to provide for a maximum size RPCSEC_GSS verifier when allocating reply buffers and Reply chunks.

RPC messages, and thus Payload streams, are made larger as a result. ULP operations that fit in a Short Message when a simpler form of authentication is in use might need to be reduced, or conveyed via a Long Message, when RPCSEC_GSS authentication is in use. It is more likely that a Requester provides both a Read list and a Reply chunk in the same RPC-over-RDMA header to convey a Long Call and provision a receptacle for a Long Reply. More frequent use of Long Messages can impact transport efficiency.

8.2.2.3. RPC-over-RDMA with RPCSEC_GSS Integrity or Privacy

The RPCSEC_GSS integrity service enables endpoints to detect modification of RPC messages in flight. The RPCSEC_GSS privacy service prevents all but the intended recipient from viewing the cleartext content of RPC arguments and results. RPCSEC_GSS integrity and privacy services are end-to-end. They protect RPC arguments and results from application to server endpoint, and back.

The RPCSEC_GSS integrity and encryption services operate on whole RPC messages after they have been XDR encoded for transmit, and before they have been XDR decoded after receipt. Both sender and receiver endpoints use intermediate buffers to prevent exposure of encrypted data or unverified cleartext data to RPC consumers. After verification, encryption, and message wrapping has been performed, the transport layer MAY use RDMA data transfer between these intermediate buffers.

The process of reducing a DDP-eligible data item removes the data item and its XDR padding from the encoded XDR stream. XDR padding of a reduced data item is not transferred in an RPC-over-RDMA message. After reduction, the Payload stream contains fewer octets than the whole XDR stream did beforehand. XDR padding octets are often zero bytes, but they don’t have to be. Thus, reducing DDP-eligible items affects the result of message integrity verification or encryption.

Therefore, a sender MUST NOT reduce a Payload stream when RPCSEC_GSS integrity or encryption services are in use. Effectively, no data item is DDP-eligible in this situation, and Chunked Messages cannot be used. In this mode, an RPC-over-RDMA transport operates in the same manner as a transport that does not support DDP.

When an RPCSEC_GSS integrity or privacy service is in use, a Requester provides both a Read list and a Reply chunk in the same RPC-over-RDMA header to convey a Long Call and provision a receptacle for a Long Reply.

8.2.2.4. Protecting RPC-over-RDMA Transport Headers

Like the base fields in an ONC RPC message (XID, call direction, and so on), the contents of an RPC-over-RDMA message's Transport stream are not protected by RPCSEC_GSS. This exposes XIDs, connection credit limits, and chunk lists (but not the content of the data items they refer to) to malicious behavior, which could redirect data that is transferred by the RPC-over-RDMA message, result in spurious retransmits, or trigger connection loss.

In particular, if an attacker alters the information contained in the chunk lists of an RPC-over-RDMA header, data contained in those chunks can be redirected to other registered memory regions on Requesters. An attacker might alter the arguments of RDMA Read and RDMA Write operations on the wire to similar effect. If such alterations occur, the use of RPCSEC_GSS integrity or privacy services enable a Requester to detect unexpected material in a received RPC message.

Encryption at lower layers, as described in Section 8.2.1, protects the content of the Transport stream. To address attacks on RDMA protocols themselves, RDMA transport implementations should conform to [RFC5042].

9. IANA Considerations

A set of RPC netids for resolving RPC-over-RDMA services is specified by this document. This is unchanged from [RFC5666].

The RPC-over-RDMA transport has been assigned an RPC netid, which is an rpcbind [RFC1833] string used to describe the underlying protocol in order for RPC to select the appropriate transport framing, as well as the format of the service addresses and ports.

The following netid registry strings are defined for this purpose:

```
NC_RDMA "rdma"
NC_RDMA6 "rdma6"
```

The "rdma" netid is to be used when IPv4 addressing is employed by the underlying transport, and "rdma6" for IPv6 addressing. The netid assignment policy and registry are defined in [RFC5665].

These netids MAY be used for any RDMA network that satisfies the requirements of Section 2.3.2 and that is able to identify service endpoints using IP port addressing, possibly through use of a translation service as described in Section 5.

The use of the RPC-over-RDMA protocol has no effect on RPC Program numbers or existing registered port numbers. However, new port numbers MAY be registered for use by RPC-over-RDMA-enabled services, as appropriate to the new networks over which the services will operate.

For example, the NFS/RDMA service defined in [RFC5667] has been assigned the port 20049 in the "Service Name and Transport Protocol Port Number Registry". This is distinct from the port number defined for NFS on TCP, which is assigned the port 2049 in the same registry. NFS clients use the same RPC Program number for NFS (100003) when using either transport [RFC5531] (see the "Remote Procedure Call (RPC) Program Numbers" registry).

10. References

10.1. Normative References

[RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995, <http://www.rfc-editor.org/info/rfc1833>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <http://www.rfc-editor.org/info/rfc4506>.

[RFC5042] Pinkerton, J. and E. Deleganes, "Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security", RFC 5042, DOI 10.17487/RFC5042, October 2007, <http://www.rfc-editor.org/info/rfc5042>.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <http://www.rfc-editor.org/info/rfc5056>.

[RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <http://www.rfc-editor.org/info/rfc5531>.

[RFC5660] Williams, N., "IPsec Channels: Connection Latching", RFC 5660, DOI 10.17487/RFC5660, October 2009, <http://www.rfc-editor.org/info/rfc5660>.

[RFC5665] Eisler, M., "IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats", RFC 5665, DOI 10.17487/RFC5665, January 2010, <http://www.rfc-editor.org/info/rfc5665>.

[RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <http://www.rfc-editor.org/info/rfc7861>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <http://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[IBARCH] InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1", Release 1.3, March 2015, <http://www.infinibandta.org/content/pages.php?pg=technology_download>.

[RFC768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <http://www.rfc-editor.org/info/rfc768>.

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <http://www.rfc-editor.org/info/rfc1094>.

[RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <http://www.rfc-editor.org/info/rfc1813>.

[RFC5040] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia, "A Remote Direct Memory Access Protocol Specification", RFC 5040, DOI 10.17487/RFC5040, October 2007, <http://www.rfc-editor.org/info/rfc5040>.

[RFC5041] Shah, H., Pinkerton, J., Recio, R., and P. Culley, "Direct Data Placement over Reliable Transports", RFC 5041, DOI 10.17487/RFC5041, October 2007, <http://www.rfc-editor.org/info/rfc5041>.

[RFC5532] Talpey, T. and C. Juszczak, "Network File System (NFS) Remote Direct Memory Access (RDMA) Problem Statement", RFC 5532, DOI 10.17487/RFC5532, May 2009, <http://www.rfc-editor.org/info/rfc5532>.

[RFC5661] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 5661, DOI 10.17487/RFC5661, January 2010, <http://www.rfc-editor.org/info/rfc5661>.

[RFC5662] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, DOI 10.17487/RFC5662, January 2010, <http://www.rfc-editor.org/info/rfc5662>.

[RFC5666] Talpey, T. and B. Callaghan, "Remote Direct Memory Access Transport for Remote Procedure Call", RFC 5666, DOI 10.17487/RFC5666, January 2010, <http://www.rfc-editor.org/info/rfc5666>.

[RFC5667] Talpey, T. and B. Callaghan, "Network File System (NFS) Direct Data Placement", RFC 5667, DOI 10.17487/RFC5667, January 2010, <http://www.rfc-editor.org/info/rfc5667>.

[RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <http://www.rfc-editor.org/info/rfc7530>.

[RFC8167] Lever, C., "Bidirectional Remote Procedure Call on RPC-over-RDMA Transports", RFC 8167, DOI 10.17487/RFC8167, June 2017, <http://www.rfc-editor.org/info/rfc8167>.

Appendix A. Changes from RFC 5666

A.1. Changes to the Specification

The following alterations have been made to the RPC-over-RDMA version 1 specification. The section numbers below refer to [RFC5666].

- Section 2 has been expanded to introduce and explain key RPC [RFC5531], XDR [RFC4506], and RDMA [RFC5040] terminology. These terms are now used consistently throughout the specification.
- Section 3 has been reorganized and split into subsections to help readers locate specific requirements and definitions.
- Sections 4 and 5 have been combined to improve the organization of this information.
- The optional Connection Configuration Protocol has never been implemented. The specification of CCP has been deleted from this specification.
- A section consolidating requirements for ULBs has been added.
- An XDR extraction mechanism is provided, along with full copyright, matching the approach used in [RFC5662].
- The "Security Considerations" section has been expanded to include a discussion of how RPC-over-RDMA security depends on features of the underlying RDMA transport.
- A subsection describing the use of RPCSEC_GSS [RFC7861] with RPC-over-RDMA version 1 has been added.

A.2. Changes to the Protocol

Although the protocol described herein interoperates with existing implementations of [RFC5666], the following changes have been made relative to the protocol described in that document:

- Support for the Read-Read transfer model has been removed. Read-Read is a slower transfer model than Read-Write. As a result, implementers have chosen not to support it. Removal of Read-Read simplifies explanatory text, and the RDMA_DONE procedure is no longer part of the protocol.

- The specification of RDMA_MSGP in [RFC5666] is not adequate, although some incomplete implementations exist. Even if an adequate specification were provided and an implementation were produced, benefit for protocols such as NFSv4.0 [RFC7530] is doubtful. Therefore, the RDMA_MSGP message type is no longer supported.

- Technical issues with regard to handling RPC-over-RDMA header errors have been corrected.

- Specific requirements related to implicit XDR roundup and complex XDR data types have been added.

- Explicit guidance is provided related to sizing Write chunks, managing multiple chunks in the Write list, and handling unused Write chunks.

- Clear guidance about Send and Receive buffer sizes has been introduced. This enables better decisions about when a Reply chunk must be provided.

Acknowledgments

The editor gratefully acknowledges the work of Brent Callaghan and Tom Talpey on the original RPC-over-RDMA Version 1 specification [RFC5666].

Dave Noveck provided excellent review, constructive suggestions, and consistent navigational guidance throughout the process of drafting this document. Dave also contributed much of the organization and content of Section 7 and helped the authors understand the complexities of XDR extensibility.

The comments and contributions of Karen Deitke, Dai Ngo, Chunli Zhang, Dominique Martinet, and Mahesh Siddheshwar are accepted with great thanks. The editor also wishes to thank Bill Baker, Greg Marsden, and Matt Benjamin for their support of this work.

The extract.sh shell script and formatting conventions were first described by the authors of the NFSv4.1 XDR specification [RFC5662].

Special thanks go to Transport Area Director Spencer Dawkins, NFSV4 Working Group Chair and Document Shepherd Spencer Shepler, and NFSV4 Working Group Secretary Thomas Haynes for their support.

Authors' Addresses

Charles Lever (editor)
Oracle Corporation
1015 Granger Avenue
Ann Arbor, MI 48104
United States of America

Phone: +1 248 816 6463
Email: chuck.lever@oracle.com

William Allen Simpson
Red Hat
1384 Fontaine
Madison Heights, MI 48071
United States of America

Email: william.allen.simpson@gmail.com

Tom Talpey
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
United States of America

Phone: +1 425 704-9945
Email: ttalpey@microsoft.com