

Network File System Version 4  
Internet-Draft  
Intended status: Standards Track  
Expires: July 20, 2020

C. Lever, Ed.  
Oracle  
D. Noveck  
NetApp  
January 17, 2020

**RPC-over-RDMA Version 2 Protocol**  
**draft-ietf-nfsv4-rpcrdma-version-two-01**

Abstract

This document specifies the second version of a transport protocol that conveys Remote Procedure Call (RPC) messages using Remote Direct Memory Access (RDMA). This version of the protocol is extensible.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 20, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

## Table of Contents

<a href="#">1.</a>	<a href="#">Introduction</a>	<a href="#">4</a>
<a href="#">2.</a>	<a href="#">Requirements Language</a>	<a href="#">6</a>
<a href="#">3.</a>	<a href="#">Terminology</a>	<a href="#">6</a>
<a href="#">3.1.</a>	<a href="#">Remote Procedure Calls</a>	<a href="#">6</a>
<a href="#">3.1.1.</a>	<a href="#">Upper-Layer Protocols</a>	<a href="#">6</a>
<a href="#">3.1.2.</a>	<a href="#">Requesters and Responders</a>	<a href="#">6</a>
<a href="#">3.1.3.</a>	<a href="#">RPC Transports</a>	<a href="#">7</a>
<a href="#">3.1.4.</a>	<a href="#">External Data Representation</a>	<a href="#">9</a>
<a href="#">3.2.</a>	<a href="#">Remote Direct Memory Access</a>	<a href="#">10</a>
<a href="#">3.2.1.</a>	<a href="#">Direct Data Placement</a>	<a href="#">10</a>
<a href="#">3.2.2.</a>	<a href="#">RDMA Transport Requirements</a>	<a href="#">11</a>
<a href="#">4.</a>	<a href="#">RPC-over-RDMA Framework</a>	<a href="#">12</a>
<a href="#">4.1.</a>	<a href="#">Message Framing</a>	<a href="#">13</a>
<a href="#">4.2.</a>	<a href="#">Managing Receiver Resources</a>	<a href="#">13</a>
<a href="#">4.2.1.</a>	<a href="#">Flow Control</a>	<a href="#">13</a>
<a href="#">4.2.2.</a>	<a href="#">Inline Threshold</a>	<a href="#">15</a>
<a href="#">4.2.3.</a>	<a href="#">Initial Connection State</a>	<a href="#">15</a>
<a href="#">4.3.</a>	<a href="#">XDR Encoding with Chunks</a>	<a href="#">16</a>
<a href="#">4.3.1.</a>	<a href="#">Reducing an XDR Stream</a>	<a href="#">17</a>
<a href="#">4.3.2.</a>	<a href="#">DDP-Eligibility</a>	<a href="#">17</a>
<a href="#">4.3.3.</a>	<a href="#">RDMA Segments</a>	<a href="#">17</a>
<a href="#">4.3.4.</a>	<a href="#">Chunks</a>	<a href="#">18</a>
<a href="#">4.3.5.</a>	<a href="#">Read Chunks</a>	<a href="#">19</a>
<a href="#">4.3.6.</a>	<a href="#">Write Chunks</a>	<a href="#">20</a>
<a href="#">4.4.</a>	<a href="#">Payload Format</a>	<a href="#">21</a>
<a href="#">4.4.1.</a>	<a href="#">Simple Format</a>	<a href="#">22</a>
<a href="#">4.4.2.</a>	<a href="#">Continued Format</a>	<a href="#">23</a>
<a href="#">4.4.3.</a>	<a href="#">Special Format</a>	<a href="#">25</a>
<a href="#">4.5.</a>	<a href="#">Reverse-Direction Operation</a>	<a href="#">27</a>
<a href="#">4.5.1.</a>	<a href="#">Sending a Reverse-Direction RPC Call</a>	<a href="#">27</a>
<a href="#">4.5.2.</a>	<a href="#">Sending a Reverse-Direction RPC Reply</a>	<a href="#">28</a>
<a href="#">4.5.3.</a>	<a href="#">In the Absence of Support For Reverse-Direction Operation</a>	<a href="#">28</a>
<a href="#">4.5.4.</a>	<a href="#">Using Chunks During Reverse-Direction Operation</a>	<a href="#">29</a>



4.5.5.	Reverse-Direction Retransmission . . . . .	30
5.	Transport Properties . . . . .	30
5.1.	Transport Properties Model . . . . .	31
5.2.	Current Transport Properties . . . . .	32
5.2.1.	Maximum Send Size . . . . .	33
5.2.2.	Receive Buffer Size . . . . .	33
5.2.3.	Maximum RDMA Segment Size . . . . .	34
5.2.4.	Maximum RDMA Segment Count . . . . .	34
5.2.5.	Reverse-Direction Support . . . . .	34
5.2.6.	Host Authentication Message . . . . .	35
6.	Transport Messages . . . . .	35
6.1.	Transport Header Types . . . . .	36
6.2.	Headers and Chunks . . . . .	37
6.2.1.	Common Transport Header Prefix . . . . .	37
6.2.2.	Transport Header Prefix . . . . .	38
6.2.3.	External Data Payloads . . . . .	40
6.2.4.	Remote Invalidation . . . . .	41
6.3.	Header Types . . . . .	41
6.3.1.	RDMA2_MSG: Convey RPC Message Inline . . . . .	42
6.3.2.	RDMA2_NOMSG: Convey External RPC Message . . . . .	42
6.3.3.	RDMA2_ERROR: Report Transport Error . . . . .	43
6.3.4.	RDMA2_CONNPROP: Exchange Transport Properties . . . . .	43
6.4.	Choosing a Reply Mechanism . . . . .	44
7.	Error Handling . . . . .	45
7.1.	Basic Transport Stream Parsing Errors . . . . .	45
7.1.1.	RDMA2_ERR_VERS . . . . .	45
7.1.2.	RDMA2_ERR_INVALID_HTYPE . . . . .	46
7.1.3.	RDMA2_ERR_INVALID_CONT . . . . .	46
7.2.	XDR Errors . . . . .	46
7.2.1.	RDMA2_ERR_BAD_XDR . . . . .	47
7.2.2.	RDMA2_ERR_BAD_PROPVAL . . . . .	47
7.3.	Responder RDMA Operational Errors . . . . .	47
7.3.1.	RDMA2_ERR_READ_CHUNKS . . . . .	48
7.3.2.	RDMA2_ERR_WRITE_CHUNKS . . . . .	48
7.3.3.	RDMA2_ERR_SEGMENTS . . . . .	49
7.3.4.	RDMA2_ERR_WRITE_RESOURCE . . . . .	49
7.3.5.	RDMA2_ERR_REPLY_RESOURCE . . . . .	49
7.4.	Other Operational Errors . . . . .	49
7.4.1.	RDMA2_ERR_SYSTEM . . . . .	50
7.5.	RDMA Transport Errors . . . . .	50
8.	XDR Protocol Definition . . . . .	51
8.1.	Code Component License . . . . .	51
8.2.	Extraction of the XDR Definition . . . . .	53
8.3.	XDR Definition for RPC-over-RDMA Version 2 Core Structures . . . . .	55
8.4.	XDR Definition for RPC-over-RDMA Version 2 Base Header Types . . . . .	57
8.5.	Use of the XDR Description . . . . .	59



<a href="#">9.</a>	<a href="#">RPC Bind Parameters</a>	<a href="#">60</a>
<a href="#">10.</a>	<a href="#">Implementation Status</a>	<a href="#">61</a>
<a href="#">11.</a>	<a href="#">Security Considerations</a>	<a href="#">61</a>
<a href="#">11.1.</a>	<a href="#">Memory Protection</a>	<a href="#">61</a>
<a href="#">11.1.1.</a>	<a href="#">Protection Domains</a>	<a href="#">62</a>
<a href="#">11.1.2.</a>	<a href="#">Handle (STag) Predictability</a>	<a href="#">62</a>
<a href="#">11.1.3.</a>	<a href="#">Memory Protection</a>	<a href="#">62</a>
<a href="#">11.1.4.</a>	<a href="#">Denial of Service</a>	<a href="#">62</a>
<a href="#">11.2.</a>	<a href="#">RPC Message Security</a>	<a href="#">63</a>
<a href="#">11.2.1.</a>	<a href="#">RPC-over-RDMA Protection at Other Layers</a>	<a href="#">63</a>
<a href="#">11.2.2.</a>	<a href="#">RPCSEC_GSS on RPC-over-RDMA Transports</a>	<a href="#">64</a>
<a href="#">11.3.</a>	<a href="#">Transport Properties</a>	<a href="#">66</a>
<a href="#">11.4.</a>	<a href="#">Host Authentication</a>	<a href="#">66</a>
<a href="#">12.</a>	<a href="#">IANA Considerations</a>	<a href="#">66</a>
<a href="#">13.</a>	<a href="#">References</a>	<a href="#">67</a>
<a href="#">13.1.</a>	<a href="#">Normative References</a>	<a href="#">67</a>
<a href="#">13.2.</a>	<a href="#">Informative References</a>	<a href="#">68</a>
<a href="#">Appendix A.</a>	<a href="#">ULB Specifications</a>	<a href="#">70</a>
<a href="#">A.1.</a>	<a href="#">DDP-Eligibility</a>	<a href="#">70</a>
<a href="#">A.2.</a>	<a href="#">Maximum Reply Size</a>	<a href="#">72</a>
<a href="#">A.3.</a>	<a href="#">Reverse-Direction Operation</a>	<a href="#">72</a>
<a href="#">A.4.</a>	<a href="#">Additional Considerations</a>	<a href="#">72</a>
<a href="#">A.5.</a>	<a href="#">ULP Extensions</a>	<a href="#">73</a>
<a href="#">Appendix B.</a>	<a href="#">Extending RPC-over-RDMA Version 2</a>	<a href="#">73</a>
<a href="#">B.1.</a>	<a href="#">Documentation Requirements</a>	<a href="#">74</a>
<a href="#">B.2.</a>	<a href="#">Adding New Header Types to RPC-over-RDMA Version 2</a>	<a href="#">74</a>
<a href="#">B.3.</a>	<a href="#">Adding New Header Flags to the Protocol</a>	<a href="#">75</a>
<a href="#">B.4.</a>	<a href="#">Adding New Transport properties to the Protocol</a>	<a href="#">76</a>
<a href="#">B.5.</a>	<a href="#">Adding New Error Codes to the Protocol</a>	<a href="#">76</a>
<a href="#">Appendix C.</a>	<a href="#">Differences from RPC-over-RDMA Version 1</a>	<a href="#">77</a>
<a href="#">C.1.</a>	<a href="#">Changes to the XDR Definition</a>	<a href="#">77</a>
<a href="#">C.2.</a>	<a href="#">Transport Properties</a>	<a href="#">78</a>
<a href="#">C.3.</a>	<a href="#">Credit Management Changes</a>	<a href="#">79</a>
<a href="#">C.4.</a>	<a href="#">Inline Threshold Changes</a>	<a href="#">79</a>
<a href="#">C.5.</a>	<a href="#">Message Continuation Changes</a>	<a href="#">80</a>
<a href="#">C.6.</a>	<a href="#">Host Authentication Changes</a>	<a href="#">81</a>
<a href="#">C.7.</a>	<a href="#">Support for Remote Invalidation</a>	<a href="#">81</a>
<a href="#">C.8.</a>	<a href="#">Error Reporting Changes</a>	<a href="#">82</a>
	<a href="#">Acknowledgments</a>	<a href="#">83</a>
	<a href="#">Authors' Addresses</a>	<a href="#">83</a>

## [1.](#) Introduction

Remote Direct Memory Access (RDMA) [[RFC5040](#)] [[RFC5041](#)] [[IBA](#)] is a technique for moving data efficiently between network nodes. By placing transferred data directly into destination buffers using Direct Memory Access, RDMA delivers the reciprocal benefits of faster data transfer and reduced host CPU overhead.



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 [[RFC0768](#)] or TCP [[RFC0793](#)]. On UDP, a datagram encapsulates each RPC message. Within a TCP byte stream, a record marking protocol delineates RPC messages.

An RDMA transport, too, conveys RPC messages in a fashion that must be fully defined if RPC implementations are to interoperate when using RDMA to transport RPC transactions. Although RDMA transports encapsulate messages like UDP, they deliver them reliably and in order, like TCP. Further, they implement a bulk data transfer service not provided by traditional network transports. Therefore, we treat RDMA as a novel transport type for RPC.

The RPC-over-RDMA transport introduced in the current document can transparently support any RPC application. The current document describes mechanisms that enable further optimization of data transfer when RPC applications are structured to exploit direct data placement. In this context, the Network File System (NFS) protocols, as described in [[RFC1094](#)], [[RFC1813](#)], [[RFC7530](#)], [[RFC5661](#)], and subsequent NFSv4 minor versions, are all potential beneficiaries of RPC-over-RDMA. A complete problem statement appears in [[RFC5532](#)].

Storage administrators have broadly deployed the RPC-over-RDMA version 1 protocol specified in [[RFC8166](#)]. However, there are known shortcomings to this protocol:

- o The protocol's default size of Receive buffers forces the use of RDMA Read and Write transfers for small payloads, and limits the size of reverse direction messages.
- o It is difficult to make optimizations or protocol fixes that require changes to on-the-wire behavior.
- o For some RPC procedures, the maximum reply size is difficult or impossible for an RPC client to estimate in advance.

To address these issues in a way that preserves interoperation with existing RPC-over-RDMA version 1 deployments, we present a second version of the RPC-over-RDMA transport protocol in the current document.

The version of RPC-over-RDMA presented here is extensible, enabling the introduction of OPTIONAL extensions without impacting existing implementations. See [Appendix C.1](#), for further discussion. It introduces a mechanism to exchange implementation properties to automatically provide further optimization of data transfer.





This version also contains incremental changes that relieve performance constraints and enable recovery from unusual corner cases. These changes are outlined in [Appendix C](#) and include a larger default inline threshold, the ability to convey a single RPC message using multiple RDMA Send operations, support for authentication of connection peers, richer error reporting, improved credit-based flow control, and support for Remote Invalidation.

## **2. 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.

## **3. Terminology**

### **3.1. Remote Procedure Calls**

This section highlights critical elements of the RPC protocol [[RFC5531](#)] and the External Data Representation (XDR) [[RFC4506](#)] it uses. RPC-over-RDMA version 2 enables the transmission of RPC messages built using XDR and also uses XDR internally to describe its header formats. The remainder of this document requires an understanding of RPC and its use of XDR.

#### **3.1.1. Upper-Layer Protocols**

RPCs are an abstraction used to implement the operations of an Upper-Layer Protocol (ULP). For RPC-over-RDMA, "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 the current document, the term "RPC consumer" refers to an implementation of a ULP running on an RPC client.

#### **3.1.2. 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 these consumers.

ONC RPC transactions consist of two types of messages:

- o A CALL message, or "Call", requests work. An RPC Call message is designated by the value zero (0) in the message's msg\_type field. The sender places a unique 32-bit value in the message's XID field to match this RPC Call message to a corresponding RPC Reply message.
- o A REPLY message, or "Reply", 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 sender copies the value contained in an RPC Reply message's XID field from the RPC Call message whose results the sender is reporting.

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

An RPC server endpoint acts as a "Responder", which deserializes the arguments and processes the requested operation. It then serializes the operation's results into an RPC Reply message. An RPC Reply message contains an RPC protocol header, a header describing the upper-layer reply, and all results.

The Requester deserializes the results and allows the RPC consumer 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, Requesters send RPC Call messages to Responders to initiate RPC transactions. Responders send RPC Reply messages to Requesters to complete the processing on an RPC transaction.

### **3.1.3. 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 network transport (e.g., a socket).

RPC-over-RDMA is a connection-oriented RPC transport. When a transport type is connection-oriented, clients initiate transport



connections, while servers wait passively to accept incoming connection requests.

#### **3.1.3.1. Forward Direction**

Traditionally, an RPC client acts as a Requester, while an RPC service acts as a Responder. The current document refers to this form of RPC message passing as "forward-direction" operation.

#### **3.1.3.2. Reverse-Direction**

The RPC specification [[RFC5531](#)] does not forbid passing RPC messages in the other direction. An RPC service endpoint can act as a Requester, in which case an RPC client endpoint acts as a Responder. This form of message passing is known as "reverse-direction" operation.

During reverse-direction operation, an RPC client is responsible for establishing transport connections, even though the RPC server originates RPC Calls.

RPC clients and servers are usually optimized to perform and scale well when handling traffic in the forward direction. They might not be prepared to handle operation in the reverse direction. Not until NFS version 4.1 [[RFC5661](#)] has there been a strong need to handle reverse-direction operation.

#### **3.1.3.3. Bi-directional Operation**

A pair of connected RPC endpoints may choose to use only forward-direction or only reverse-direction operation on a particular transport connection. Or, these endpoints may send Calls in both directions concurrently on the same transport connection.

"Bi-directional operation" occurs when both transport endpoints act as a Requester and a Responder at the same time on a single connection.

Bi-directionality is an extension of RPC transport connection sharing. Two RPC endpoints wish to exchange independent RPC messages over a shared connection but in opposite directions. These messages may or may not be related to the same workloads or RPC Programs.

#### **3.1.3.4. XID Values**

[Section 9 of \[RFC5531\]](#) introduces the RPC transaction identifier, or "XID" for short. A connection peer interprets the value of an XID in the context of the message's msg\_type field.



- o The XID of a Call is arbitrary but is unique among outstanding Calls from that Requester on that connection.
- o The XID of a Reply always matches that of the initiating Call.

After receiving a Reply, a Requester matches the XID value in that Reply with a Call it previously sent.

During bi-directional operation, forward- and reverse- direction XIDs are typically generated on distinct hosts by possibly different algorithms. There is no coordination between the generation of XIDs used in forward-direction and reverse-direction operation.

Therefore, a forward-direction Requester MAY use the same XID value at the same time as a reverse-direction Requester on the same transport connection. Although such concurrent requests use the same XID value, they represent distinct RPC transactions.

#### **3.1.4. External Data Representation**

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

XDR encodes data independently of the endianness or size of host-native data types, enabling unambiguous decoding of data by a receiver.

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

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

##### **3.1.4.1. XDR Opaque Data**

Sometimes, a data item is to be transferred as-is, without encoding or decoding. We refer to the contents of such a data item 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.

#### **3.1.4.2. 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, the sender appends octets containing zero after the end of the data item. These zero octets shift the next encoded data item in the XDR stream so that it always starts on a four-octet boundary. The addition of extra octets does not change the encoded size of the data item. Receivers do not expose the extra octets to ULPs.

We refer to this technique as "XDR roundup", and the extra octets as "XDR roundup padding".

### **3.2. Remote Direct Memory Access**

When a third party transfers large RPC payloads, RPC Requesters and Responders can become more efficient. An example of such a third party might be an intelligent network interface (data movement offload), which places data in the receiver's memory so that no additional adjustment of data alignment is necessary (direct data placement or "DDP"). RDMA transports enable both of these optimizations.

In the current document, "RDMA" refers to the physical mechanism an RDMA transport utilizes when moving data.

#### **3.2.1. Direct Data Placement**

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, if only to adjust data alignment.

Although it 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 a receiving host's CPU does not move transferred data to another location after arrival.



RPC-over-RDMA version 2 enables the use of RDMA Read and Write operations to achieve both data movement offload and DDP. However, note that not all RDMA-based data transfer qualifies as DDP, and some mechanisms that do not employ explicit RDMA can place data directly.

### **3.2.2. RDMA Transport Requirements**

RDMA transports require that RDMA consumers provision resources in advance to achieve good performance during receive operations. 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 the RDMA provider 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 tenets of operation, the RPC-over-RDMA version 2 protocol assumes each transport provides the following abstract operations. A more complete discussion of these operations appears in [\[RFC5040\]](#).

#### **3.2.2.1. Memory Registration**

Memory registration assigns a steering tag to a region of memory, permitting the RDMA provider to perform data-transfer operations. The RPC-over-RDMA version 2 protocol assumes that a steering tag of no more than 32 bits and memory addresses of up to 64 bits in length identifies each registered memory region.

#### **3.2.2.2. RDMA Send**

The RDMA provider supports an RDMA Send operation, with completion signaled on the receiving peer after RDMA provider has placed data in a pre-posted buffer. Sends complete at the receiver in the order they were posted at the sender. The size of the remote peer's pre-posted buffers limits the amount of data that can be transferred by a single RDMA Send operation.

#### **3.2.2.3. 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 memory posted per Receive is limited. The RDMA consumer (in this



case, the RPC-over-RDMA version 2 protocol) provides flow control to prevent overrunning receiver resources.

#### **3.2.2.4. 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 the RDMA provider signals completion there. The remote RDMA provider does not signal completion on the remote peer. The local host provides the steering tag, the memory address, and the length of the remote peer's memory region.

RDMA Writes are not ordered relative to one another, but are ordered relative to RDMA Sends. Thus, a subsequent RDMA Send completion signaled on the local peer guarantees that prior RDMA Write data has been successfully placed in the remote peer's memory.

#### **3.2.2.5. RDMA Read**

The RDMA provider supports an RDMA Read operation to place remote source data directly into local memory. The local host initiates an RDMA Read and the RDMA provider signals completion there. The remote RDMA provider does not signal completion on the remote peer. The local host provides the steering tags, the memory addresses, and the lengths for the remote source and local destination memory regions.

The RDMA consumer (in this case, the RPC-over-RDMA version 2 protocol) signals Read completion to the remote peer as part of a subsequent RDMA Send message. The remote peer can then invalidate steering tags and subsequently free associated source memory regions.

### **4. RPC-over-RDMA Framework**

Before an RDMA data transfer can occur, an endpoint first exposes regions of its memory to a remote endpoint. The remote endpoint then initiates RDMA Read and Write operations against the exposed memory. A "transfer model" designates which endpoint exposes its memory and which is responsible for initiating the transfer of data.

In RPC-over-RDMA version 2, only Requesters expose their memory to the Responder, and only Responders initiate RDMA Read and Write operations. Read access to memory regions enables the Responder to pull RPC arguments or whole RPC Calls from each Requester. The Responder pushes RPC results or whole RPC Replies to a Requester's memory regions to which it has write access.



#### **4.1. Message Framing**

Each RPC-over-RDMA version 2 message consists of at most two XDR streams:

- o The "Transport stream" contains a header that describes and controls the transfer of the Payload stream in this RPC-over-RDMA message. Every RDMA Send on an RPC-over-RDMA version 2 connection MUST begin with a Transport stream.
- o The "Payload stream" contains part or all of a single RPC message. The sender MAY divide an RPC message at any convenient boundary but MUST send RPC message fragments in XDR stream order and MUST NOT interleave Payload streams from multiple RPC messages. The RPC-over-RDMA version 2 message carrying the final part of an RPC message is marked (see [Section 6.2.2.2](#)).

The RPC-over-RDMA framing mechanism described in this section replaces all other RPC framing mechanisms. Connection peers use RPC-over-RDMA framing even when the underlying RDMA protocol runs on a transport type with well-defined RPC framing, such as TCP. However, a ULP can negotiate the use of RDMA, dynamically enabling the use of RPC-over-RDMA on a connection established on some other transport type. 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.

#### **4.2. Managing Receiver Resources**

If any pre-posted Receive buffer on the connection is not large enough to accept an incoming RDMA Send, the RDMA provider can terminate the connection. Likewise, if a pre-posted Receive buffer is not available to accept an incoming RDMA Send, the RDMA provider can terminate the connection. Therefore, a sender needs to respect the resource limits of its peer receiver to ensure the longevity of each connection. Two operational parameters communicate these limits between connection peers: flow control, and inline threshold.

##### **4.2.1. Flow Control**

RPC-over-RDMA requires reliable and in-order delivery of data payloads. Therefore, RPC-over-RDMA transports MUST use the RDMA RC (Reliable Connected) Queue Pair (QP) type. The use of an RC QP ensures in-transit data integrity and proper recovery from packet loss or misordering.

However, RPC-over-RDMA itself provides a flow control mechanism to prevent a sender from overwhelming receiver resources. RPC-over-RDMA





transports employ end-to-end credit-based flow control for this purpose [[CBFC](#)]. Credit-based flow control is relatively simple, providing robust operation in the face of bursty traffic and automated management of receive buffer allocation.

#### **4.2.1.1. Granting Credits**

An RPC-over-RDMA version 2 credit is the capability to receive one RPC-over-RDMA version 2 message. This arrangement enables RPC-over-RDMA version 2 to support asymmetrical operation, where a message in one direction might trigger zero, one, or multiple messages in the other direction in response.

To achieve this, each posted Receive buffer on both connection peers receives one credit. Each Requester has a set of Receive credits, and each Responder has a set of Receive credits. These credit values are managed independently of one another.

[Section 7 of \[RFC8166\]](#) requires that the 32-bit field containing the credit grant is the third word in the transport header. To conform with that requirement, senders encode the two independent credit values into a single 32-bit field in the fixed portion of the transport header. At the receiver, the low-order two bytes are the number of credits that are newly granted by the sender. The granted credit value MUST NOT be zero since such a value would result in deadlock. The high-order two bytes are the maximum number of credits that can be outstanding at the sender.

A sender must avoid posting more RDMA Send messages than the receiver's granted credit limit. If the sender exceeds the granted value, the RDMA provider might signal an error, possibly terminating the connection.

The granted credit values MAY be adjusted to match the needs or policies in effect on either peer. For instance, a peer may reduce its granted credit value to accommodate the available resources in a Shared Receive Queue.

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 2 implementation.

#### **4.2.1.2. Asynchronous Credit Grants**

A special protocol convention enables one peer to refresh its credit grant to the other peer without sending a payload. Messages of this



type can also act as a keep-alive ping. See [Section 6.3.2](#) for information about this convention.

Receivers MUST always be in a position to receive one such credit grant update message, in addition to payload-bearing messages, to prevent transport deadlock. One way a receiver can do this is to post one more RDMA Receive than the credit value the receiver granted.

#### [4.2.2.](#) **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 channel operations. An inline threshold value is less than the largest number of octets the sender can post in a single RDMA Send operation. It is also less than the largest number of octets the receiver can reliably accept via a single RDMA Receive operation.

Each connection has two inline threshold values. There is 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."

Peers can advertise their inline threshold values via RPC-over-RDMA version 2 Transport Properties (see [Section 5](#)). In the absence of an exchange of Transport Properties, connection peers MUST assume both inline thresholds are 4096 octets.

#### [4.2.3.](#) **Initial Connection State**

When an RPC-over-RDMA version 2 client establishes a connection to a server, its first order of business is to determine the server's highest supported protocol version.

Upon connection establishment, a client MUST send only a single RPC-over-RDMA message until it receives a valid RPC-over-RDMA message from the server that grants client credits.

The second word of each transport header conveys the transport protocol version. In the interest of clarity, the current document refers to that word as `rdma_vers` even though in the RPC-over-RDMA version 2 XDR definition, it appears as `rdma_start.rdma_vers`.

Immediately after the client establishes a connection, it sends a single valid RPC-over-RDMA message with the value two (2) in the `rdma_vers` field. Because the server might support only RPC-over-RDMA



version 1, this initial message MUST NOT be larger than the version 1 default inline threshold of 1024 octets.

#### **4.2.3.1. Server Does Support RPC-over-RDMA Version 2**

If the server supports RPC-over-RDMA version 2, it sends RPC-over-RDMA messages back to the client with the value two (2) in the `rdma_vers` field. Both peers may assume the default inline threshold value for RPC-over-RDMA version 2 connections (4096 octets).

#### **4.2.3.2. Server Does Not Support RPC-over-RDMA Version 2**

If the server does not support RPC-over-RDMA version 2, it MUST send an RPC-over-RDMA message to the client with an `XID` that matches the client's first message, `RDMA2_ERROR` in the `rdma_start.rdma_htype` field, and with the error code `RDMA2_ERR_VERS`. This message also reports the range of RPC-over-RDMA protocol versions that the server supports. To continue operation, the client selects a protocol version in that range for subsequent messages on this connection.

If the connection is dropped immediately after an `RDMA2_ERROR/`  
`RDMA2_ERR_VERS` message is received, the client should try to avoid a version negotiation loop when re-establishing another connection. It can assume that the server does not support RPC-over-RDMA version 2. A client can assume the same situation (i.e., no server support for RPC-over-RDMA version 2) if the initial negotiation message is lost or dropped. Once the version negotiation exchange is complete, both peers may use the default inline threshold value for the negotiated transport protocol version.

#### **4.2.3.3. Client Does Not Support RPC-over-RDMA Version 2**

The server examines the RPC-over-RDMA protocol version used in the first RPC-over-RDMA message it receives. If it supports this protocol version, it MUST use it in all subsequent messages it sends on that connection. The client MUST NOT change the protocol version for the duration of the connection.

### **4.3. XDR Encoding with Chunks**

When a DDP capability is available, an RDMA provider can place the contents of one or more XDR data items directly into a receiver's memory. It can do this separately from the transfer of other parts of the containing XDR stream.



#### **4.3.1. Reducing an XDR Stream**

RPC-over-RDMA version 2 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. These items 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.

We refer to a data item that a sender removes from a Payload stream to transmit separately as a "reduced" data item. After a sender has finished removing XDR data items from a Payload stream, we refer to it as a "reduced" Payload stream. The data object in a transport header that describes memory regions containing reduced data items is known as a "chunk."

#### **4.3.2. DDP-Eligibility**

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

Determining which data items are DDP-eligible is done in additional specifications that describe how ULPs employ DDP. A "ULB specification" identifies which XDR data items a peer MAY transfer using DDP. Such data items are known as "DDP-eligible." Senders MUST NOT reduce any other XDR data items. Detailed requirements for ULB specifications appear in [Appendix A](#) of the current document.

#### **4.3.3. 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 transport 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 a transport header data object that contains the precise coordinates of a contiguous memory region. This region is conveyed separately from the Payload stream. Each RDMA segment contains the following information:





**Handle:** A 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.

#### **4.3.4. Chunks**

In RPC-over-RDMA version 2, a "chunk" refers to a portion of an RPC message that is moved independently of the Payload stream. The sender removes chunk data from the Payload stream, transfers it via separate operations, and then the receiver reinserts it into the received Payload stream to reconstruct the complete RPC message.

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

The RPC-over-RDMA version 2 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 transaction. 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.

##### **4.3.4.1. Counted Arrays**

If a chunk is to contain a counted array data type, the count of array elements MUST remain in the Payload stream. The sender MUST move the array elements into the chunk. For example, when encoding an opaque byte array as a chunk, the count of bytes stays in the Payload stream. The sender removes the bytes in the array from the Payload stream and places them in 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.



#### **4.3.4.2. Optional-Data**

If a chunk contains an optional-data data type, the "is present" field MUST remain in the Payload stream. The sender MUST move the data, if present, to the chunk.

#### **4.3.4.3. XDR Unions**

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

#### **4.3.4.4. Chunk Roundup**

Except in special cases (covered in [Section 4.4.3](#)), a chunk MUST contain only one XDR data item. This restriction makes it straightforward to reduce variable-length data items without affecting the XDR alignment of other data items in the Payload stream.

When a sender reduces a variable-length XDR data item, data items remaining in the Payload stream MUST remain on four-byte alignment. Therefore, the sender always removes XDR roundup padding for that data item from the Payload stream.

#### **4.3.5. Read Chunks**

A "Read chunk" represents an XDR data item that the Responder pulls from the Requester. A Read chunk is a list of one or more RDMA read segments. Each RDMA read segment consists of a Position field followed by an RDMA segment, as defined in [Section 4.3.3](#).

Position: The byte offset in the unreduced Payload stream where the receiver reinserts the data item conveyed in the chunk. The Requester MUST compute the Position value 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 containing data items intended for RDMA Read operations. It advertises the coordinates of these regions by adding Read chunks to the 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 inserts the first RDMA segment in a Read chunk into the Payload stream at the byte offset indicated by



its Position field. The Responder concatenates RDMA segments whose Position field value matches this offset until there are no more RDMA segments at that Position value.

The Position field in an RDMA 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.

#### **4.3.5.1. Decoding Read Chunks**

The Responder initiates an RDMA Read to pull a Read chunk's data content into registered local memory whenever the XDR offset in the Payload stream matches that of a Read chunk. The Responder acknowledges that it is finished with Read chunk source buffers when it sends the corresponding RPC Reply message to the Requester. The Requester may then release Read chunks advertised in the RPC-over-RDMA Call.

#### **4.3.5.2. Read Chunk Roundup**

When reducing a variable-length argument data item, the Requester **MUST NOT** include the data item's XDR roundup padding in the chunk itself. The chunk's total length **MUST** be the same as the encoded length of the data item.

#### **4.3.6. 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 a Responder is to push to a Requester. It consists of an array of zero or more plain segments.

A Requester provisions Write chunks long before the Responder has prepared the reply message. 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** provision Write chunks large enough to accommodate the maximum possible size of each returned data item.

Note that 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 DDP-eligible one data item to the Requester. It fills the chunk



contiguously and in segment array order until the Responder has written that data item to the Requester in its entirety. 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 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.

#### **4.3.6.1. 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.

#### **4.3.6.2. Write Chunk Roundup**

When provisioning a Write chunk for a variable-length result data item, the Requester MUST NOT include additional space for XDR roundup padding. A Responder MUST NOT write XDR roundup padding into a Write chunk, even if the result is shorter than the available space in the chunk. 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.

### **4.4. Payload Format**

Unlike RPC-over-TCP and RPC-over-UDP transports, RPC-over-RDMA transports are aware of the XDR encoding of each RPC message payload. For efficiency, the transport can convey DDP-eligible XDR data items separately from the RPC message itself. Also, receivers are required to post adequate receive resources in advance of each RPC message.

RPC-over-RDMA version 2 provides several ways to arrange conveyance of an RPC-over-RDMA message. A sender chooses the specific format for a message among several factors:

- o The existence of DDP-eligible data items in the RPC message
- o The size of the RPC message
- o The direction of the RPC message (i.e., Call or Reply)





- o The available hardware resources
- o The arrangement of source and sink memory buffers

The following subsections describe in detail how Requesters and Responders format RPC-over-RDMA message payloads.

#### **4.4.1. Simple Format**

All RPC messages conveyed via RPC-over-RDMA version 2 require at least one RDMA Send operation to convey. Thus, 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. When no chunks are present, senders construct Calls and Replies the same way, and no other operations are needed.

##### **4.4.1.1. Simple Format with Chunks**

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

When chunks are present, senders construct Calls differently than Replies.

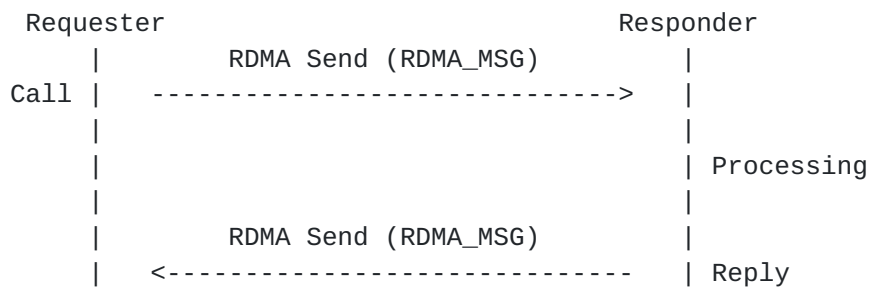
Simple Call: After receiving the Transport and Payload streams of an RPC Call message with Read chunks, the Responder uses RDMA Read operations to move the reduced data items contained in Read chunks. RPC-over-RDMA Calls can carry Write chunks for the Responder to use when sending the matching Reply.

Simple Reply

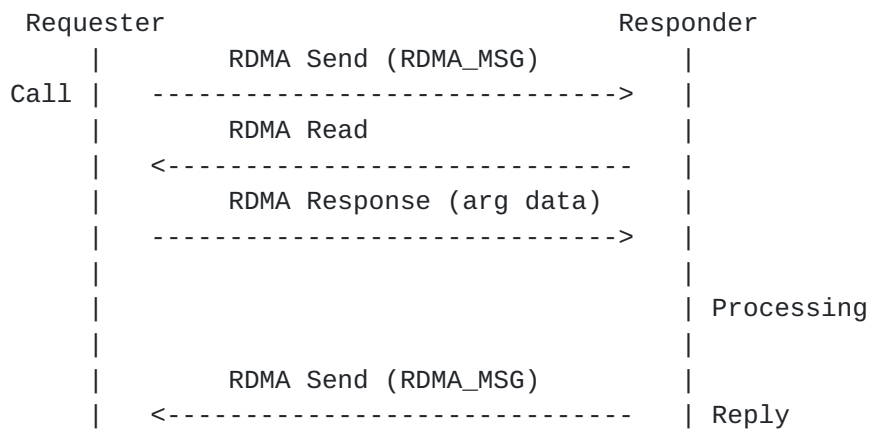
The Responder uses RDMA Write operations to move reduced data items contained in Write chunks. Afterward, it sends the Transport and Payload streams of the RPC Reply message using one RDMA Send. RPC-over-RDMA Replies always carry an empty Read chunk list.

##### **4.4.1.2. Simple Format Examples**

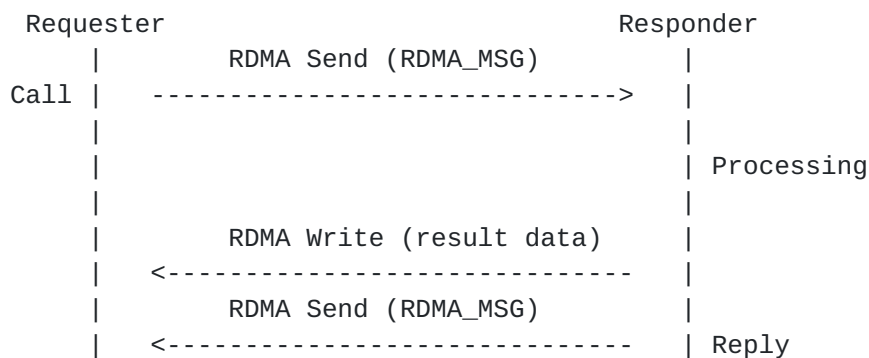




A Simple Call without chunks and a Simple Reply without chunks



A Simple Call with a Read chunk and a Simple Reply without chunks



A Simple Call without chunks and a Simple Reply with a Write chunk

#### [4.4.2.](#) Continued Format

For various reasons, a sender can choose to split a message payload over multiple RPC-over-RDMA messages. The Payload stream of each RPC-over-RDMA message contains a part of the RPC message. The receiver reconstructs the original RPC message by concatenating in sequence the Payload stream of each RPC-over-RDMA message. A sender MAY split an RPC message payload on any convenient boundary.



#### 4.4.2.1. Continued Format with Chunks

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

As with Simple Format messages, when chunks are present, senders construct Calls differently than Replies.

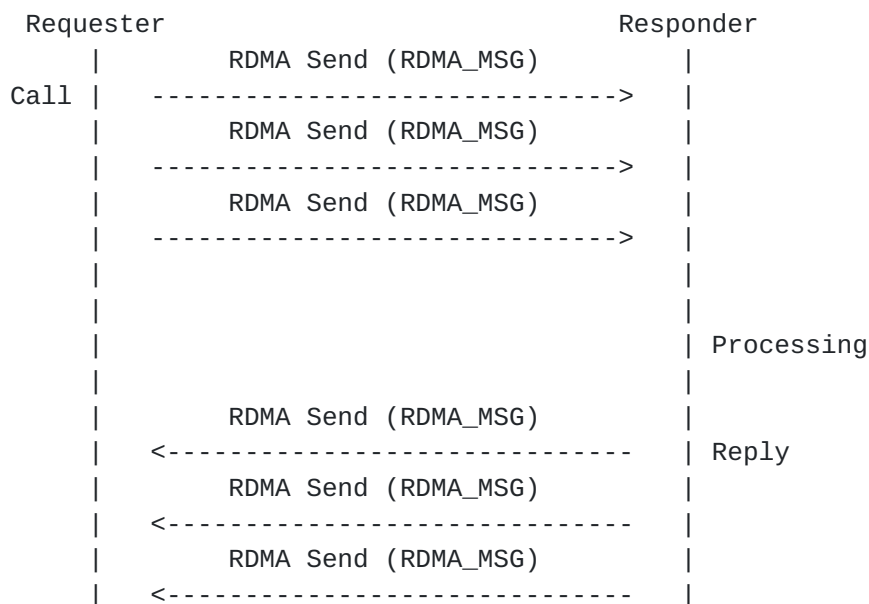
##### Continued Call

After receiving the Transport and Payload streams of an RPC Call message with Read chunks, the Responder uses RDMA Read operations to move the reduced data items contained in Read chunks. RPC-over-RDMA Calls can carry Write chunks for the Responder to use when sending the matching Reply.

##### Continued Reply

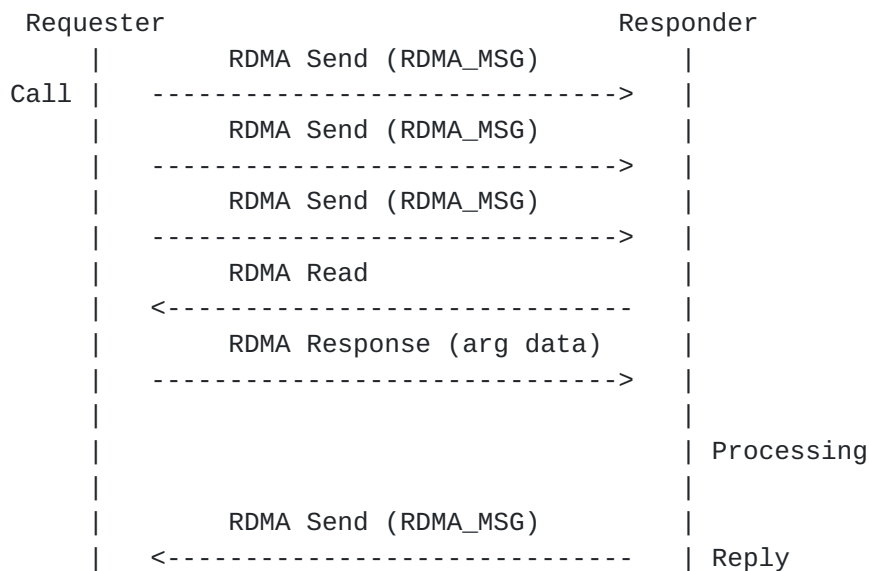
The Responder uses RDMA Write operations to move reduced data items contained in Write chunks. Afterward, it sends the Transport and Payload streams of the RPC Reply message using multiple RDMA Sends. RPC-over-RDMA Replies always carry an empty Read chunk list.

#### 4.4.2.2. Continued Format Examples

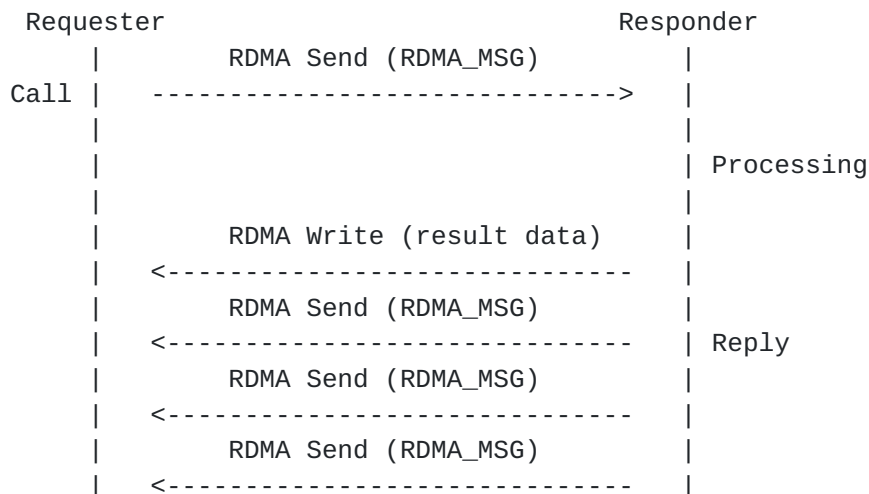


A Continued Call without chunks and a Continued Reply without chunks





A Continued Call with a Read chunk and a Simple Reply without chunks



A Simple Call without chunks and a Continued Reply with a Write chunk

#### **4.4.3. Special Format**

Sometimes, after DDP-eligible data items have been removed, a Payload stream is still too large to send using only RDMA Send operations. In those cases, the sender can use RDMA Read or Write operations to convey the entire RPC message. We refer to this as a "Special Format" message.

To transmit a Special Format message, the sender transmits only the Transport stream with an RDMA Send operation. The sender does not include the Payload stream in the send buffer. Instead, the





Requester provides chunks that the Responder uses to move the Payload stream.

Because chunks are always present in Special Format messages, the sender always handles Calls and Replies differently.

#### Special Call

The Requester provides a Read chunk that contains the RPC Call message's Payload stream. Every read segment in this chunk MUST contain zero (0) in its Position field. This type of Read chunk is known as a "Position Zero Read chunk."

#### Special Reply

The Requester provisions a single Write chunk in advance, known as a "Reply chunk", in which the Responder places 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.

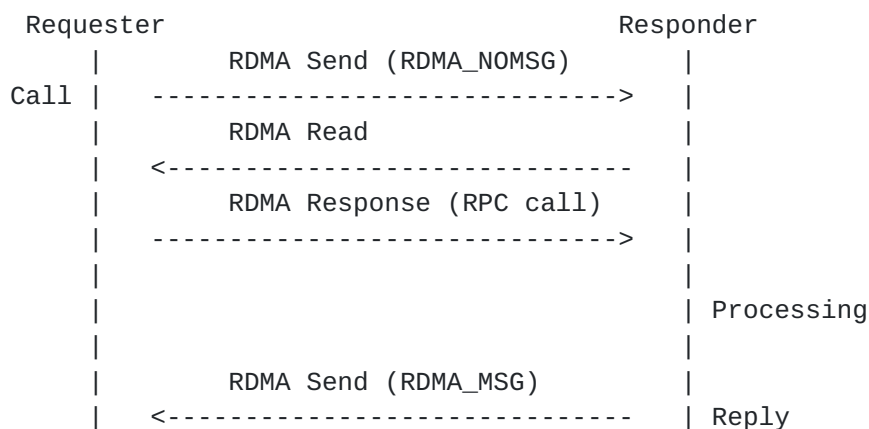
One purpose of a Special Format message is to handle large RPC messages. However, Requesters MAY use a Special Format message at any time to convey an RPC Call message.

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

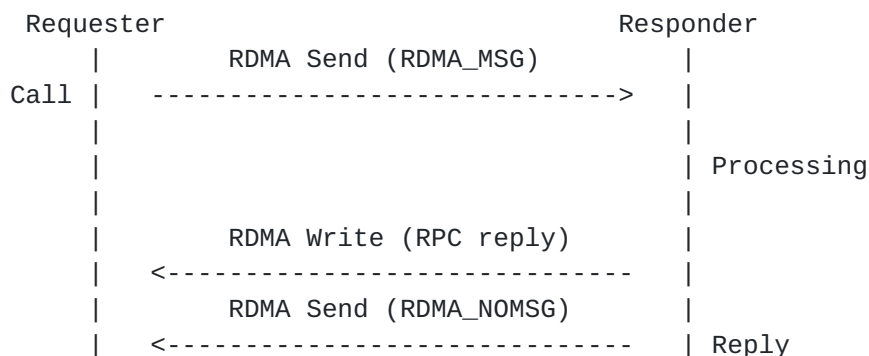
XDR data items may appear in these chunks without regard to their DDP-eligibility. As these chunks contain a Payload stream, they MUST include appropriate XDR roundup padding to maintain proper XDR alignment of their contents.

#### [4.4.3.1](#). Special Format Examples





A Special Call and a Simple Reply without chunks



A Simple Call without chunks and a Special Reply

## 4.5. Reverse-Direction Operation

### 4.5.1. Sending a Reverse-Direction RPC Call

An RPC-over-RDMA server endpoint constructs the transport header for a reverse-direction RPC Call as follows:

- o The server generates a new XID value (see [Section 3.1.3.4](#) for full requirements) and places it in the `rdma_xid` field of the transport header and the `xid` field of the RPC Call message. The RPC Call header MUST start with the same XID value that is present in the transport header.
- o The `rdma_vers` field of each reverse-direction Call MUST contain the same value as forward-direction Calls on the same connection.
- o The server fills in the `rdma_credits` with the credit values for the connection, as described in [Section 4.2.1.1](#).



- o The server determines the Payload format for the RPC message and fills in the `rdma_hype` field as appropriate (see Sections [4.4](#) and [4.5.4](#)). [Section 4.5.4](#) also covers the disposition of the chunk lists.
- o The server MUST clear the `RDMA2_F_RESPONSE` flag in the `rdma_flags` field. It sets the `RDMA2_F_MORE` flag in the `rdma_flags` field as described in [Section 6.2.2.2](#).

#### **[4.5.2](#). Sending a Reverse-Direction RPC Reply**

An RPC-over-RDMA server endpoint constructs the transport header for a reverse-direction RPC Reply as follows:

- o The server copies the `XID` value from the matching RPC Call and places it in the `rdma_xid` field of the transport header and the `xid` field of the RPC Reply message. The RPC Reply header MUST start with the same `XID` value that is present in the transport header.
- o The `rdma_vers` field of each reverse-direction Call MUST contain the same value as forward-direction Replies on the same connection.
- o The server fills in the `rdma_credits` with the credit values for the connection, as described in [Section 4.2.1.1](#).
- o The server determines the Payload format for the RPC message and fills in the `rdma_hype` field as appropriate (see Sections [4.4](#) and [4.5.4](#)). [Section 4.5.4](#) also covers the disposition of the chunk lists.
- o The server MUST set the `RDMA2_F_RESPONSE` flag in the `rdma_flags` field. It sets the `RDMA2_F_MORE` flag in the `rdma_flags` field as described in [Section 6.2.2.2](#).

#### **[4.5.3](#). In the Absence of Support For Reverse-Direction Operation**

An RPC-over-RDMA transport endpoint does not have to support reverse-direction operation. There might be no mechanism in the transport implementation to do so. Or, the transport implementation might support operation in the reverse direction, but the Upper-Layer Protocol might not yet have configured the transport to handle reverse-direction traffic.

If an endpoint is unprepared to receive a reverse-direction message, loss of the RDMA connection might result. Thus a denial of service can occur if an RPC server continues to send reverse-direction



messages after a client that is not prepared to receive them reconnects to an endpoint.

Connection peers indicate their support for reverse-direction operation as part of the exchange of Transport Properties just after a connection is established (see [Section 5.2.5](#)).

When dealing with the possibility that the remote peer has no transport level support for reverse-direction operation, the Upper-Layer Protocol is responsible for informing peers when reverse direction operation is supported. Otherwise, even a simple reverse direction RPC NULL procedure from a peer could result in a lost connection. Therefore, an Upper-Layer Protocol MUST NOT perform reverse-direction RPC operations until the RPC server indicates support for them.

#### **[4.5.4](#). Using Chunks During Reverse-Direction Operation**

Reverse-direction operations can use chunks, as defined in [Section 4.3.4](#), for DDP-eligible data items or in Special payload formats. Reverse-direction chunks operate the same way as in forward-direction operation. Connection peers indicate their support for reverse-direction chunks as part of the exchange of Transport Properties just after a connection is established (see [Section 5.2.5](#)).

However, an implementation might support only Upper-Layer Protocols that have no DDP-eligible data items. Such Upper-Layer Protocols can use only small messages, or they might have a native mechanism for restricting the size of reverse-direction RPC messages, obviating the need to handle chunks in the reverse direction.

When there is no Upper-Layer Protocol need for chunks in the reverse direction, implementers MAY choose not to provide support for chunks in the reverse direction, thus avoiding the complexity of implementing support for RDMA Reads and Writes in the reverse direction.

When an RPC-over-RDMA transport implementation does not support chunks in the reverse direction, RPC endpoints use only the Simple Payload format without chunks or the Continued Payload format without chunks to send RPC messages in the reverse direction.

If a reverse-direction Requester provides a non-empty chunk list to a Responder that does not support chunks, the Responder MUST report its lack of support using one of the error values defined in [Section 7.3](#).





#### **4.5.5. Reverse-Direction Retransmission**

In rare cases, an RPC server cannot complete an RPC transaction and cannot send a Reply. In these cases, the Requester may send the RPC transaction again using the same RPC XID. We refer to this as an "RPC retransmission" or a "replay."

In the forward direction, the Requester is the RPC client. The client is always responsible for ensuring a transport connection is in place before sending a dropped Call again.

With reverse-direction operation, the Requester is an RPC server. Because an RPC server is not responsible for establishing transport connections with clients, the Requester is unable to retransmit a reverse-direction Call whenever there is no transport connection. In this case, the RPC server must wait for the RPC client to re-establish a transport connection before it can retransmit reverse-direction RPC Calls.

If the forward-direction Requester has no work to do, it can be some time before the RPC client re-establishes a transport connection. An RPC server may need to abandon a waiting reverse-direction RPC Call to avoid waiting indefinitely for the client to re-establish a transport connection.

Therefore forward-direction Requesters SHOULD maintain a transport connection as long as the RPC server might send reverse-direction Calls. For example, while an NFS version 4.1 client has open delegated files or active pNFS layouts, it maintains one or more transport connections to enable the NFS server to perform callback operations.

### **5. Transport Properties**

RPC-over-RDMA version 2 enables connection endpoints to exchange information about implementation properties. Compatible endpoints use this information to optimize data transfer. Initially, only a small set of transport properties are defined. The protocol provides a single message type to exchange transport properties (see [Section 6.3.4](#)).

Both the set of transport properties and the operations used to communicate them may be extended. Within RPC-over-RDMA version 2, such extensions are OPTIONAL. A discussion of extending the set of transport properties appears in [Appendix B.4](#).



### **5.1. Transport Properties Model**

The current document specifies a basic set of receiver and sender properties. Such properties are specified using a code point that identifies the particular transport property and a nominally opaque array containing the XDR encoding of the property.

The following XDR types handle transport properties:

<CODE BEGINS>

```
typedef rpcrdma2_propid uint32;

struct rpcrdma2_propval {
    rpcrdma2_propid rdma_which;
    opaque          rdma_data<>;
};

typedef rpcrdma2_propval rpcrdma2_propset<>;

typedef uint32 rpcrdma2_propsubset<>;

<CODE ENDS>
```

The `rpcrdma2_propid` type specifies a distinct transport property. The property code points are defined as `const` values rather than elements in an enum type to enable the extension by concatenating XDR definition files.

The `rpcrdma2_propval` type carries the value of a transport property. The `rdma_which` field identifies the particular property, and the `rdma_data` field contains the associated value of that property. A zero-length `rdma_data` field represents the default value of the property specified by `rdma_which`.

Although the `rdma_data` field is opaque, receivers interpret its contents using the XDR type associated with the property specified by `rdma_which`. When the contents of the `rdma_data` field do not conform to that XDR type, the receiver **MUST** return the error `RDMA2_ERR_BAD_PROPVAL` using the header type `RDMA2_ERROR`, as described in Section [Section 6.3.3](#).

For example, the receiver of a message containing a valid `rpcrdma2_propval` returns this error if the length of `rdma_data` is greater than the length of the transferred message. Also, when the receiver recognizes the `rpcrdma2_propid` contained in `rdma_which`, it **MUST** report the error `RDMA2_ERR_BAD_PROPVAL` if either of the following occurs:



- o The nominally opaque data within `rdma_data` is not valid when interpreted using the property-associated typedef.
- o The length of `rdma_data` is insufficient to contain the data represented by the property-associated typedef.

A receiver does not report an error if it does not recognize the value contained in `rdma_which`. In that case, the receiver does not process that `rpcrdma2_propval`. Processing continues with the next `rpcrdma2_propval`, if any.

The `rpcrdma2_propset` type specifies a set of transport properties. The protocol does not impose a particular ordering of the `rpcrdma2_propval` items within it.

The `rpcrdma2_propsubset` type identifies a subset of the properties in a `rpcrdma2_propset`. Each bit in the mask denotes a particular element in a previously specified `rpcrdma2_propset`. If a particular `rpcrdma2_propval` is at position `N` in the array, then bit number `N mod 32` in word `N div 32` specifies whether the defined subset includes that particular `rpcrdma2_propval`. Words beyond the last one specified are assumed to contain zero.

## 5.2. Current Transport Properties

Table 1 specifies a basic set of transport properties. The columns contain the following information:

- o The column labeled "Property" contains a name of the transport property described by the current row.
- o The column labeled "Code" specifies the code point that identifies this property.
- o The column labeled "XDR type" gives the XDR type of the data used to communicate the value of this property. This data type overlays the data portion of the nominally opaque `rdma_data` field.
- o The column labeled "Default" gives the default value for the property.
- o The column labeled "Section" indicates the section within the current document that explains the use of this property.



Property	Code	XDR type	Default	Section
Maximum Send Size	1	uint32	4096	5.2.1
Receive Buffer Size	2	uint32	4096	5.2.2
Maximum RDMA Segment Size	3	uint32	1048576	5.2.3
Maximum RDMA Segment Count	4	uint32	16	5.2.4
Reverse-Direction Support	5	uint32	0	5.2.5
Host Auth Message	6	opaque<>	N/A	5.2.6

Table 1

### 5.2.1. Maximum Send Size

The value of this property specifies the maximum size, in octets, of Send payloads. The endpoint receiving this value can size its Receive buffers based on the value of this property.

<CODE BEGINS>

```
const uint32 RDMA2_PROPID_SBSIZ = 1;
typedef uint32 rpcrdma2_prop_sbsiz;
```

<CODE ENDS>

### 5.2.2. Receive Buffer Size

The value of this property specifies the minimum size, in octets, of pre-posted receive buffers.

<CODE BEGINS>

```
const uint32 RDMA2_PROPID_RBSIZ = 2;
typedef uint32 rpcrdma2_prop_rbsiz;
```

<CODE ENDS>

A sender can subsequently use this value to determine when a message to be sent fits in pre-posted receive buffers that the receiver has set up. In particular:

- o Requesters may use the value to determine when to provide a Position Zero Read chunk or use Message Continuation when sending a Call.





- o Requesters may use the value to determine when to provide a Reply chunk when sending a Call, based on the maximum possible size of the Reply.
- o Responders may use the value to determine when to use a Reply chunk provided by the Requester, given the actual size of a Reply.

#### **5.2.3. Maximum RDMA Segment Size**

The value of this property specifies the maximum size, in octets, of an RDMA segment this endpoint is prepared to send or receive.

<CODE BEGINS>

```
const uint32 RDMA2_PROPID_RSSIZ = 3;
typedef uint32 rpcrdma2_prop_rssiz;
```

<CODE ENDS>

#### **5.2.4. Maximum RDMA Segment Count**

The value of this property specifies the maximum number of RDMA segments that can appear in a Requester's transport header.

<CODE BEGINS>

```
const uint32 RDMA2_PROPID_RCSIZ = 4;
typedef uint32 rpcrdma2_prop_rcsiz;
```

<CODE ENDS>

#### **5.2.5. Reverse-Direction Support**

The value of this property specifies a client implementation's readiness to process messages that are part of reverse direction RPC requests.

<CODE BEGINS>

```
const uint32 RDMA_RVRSDIR_NONE = 0;
const uint32 RDMA_RVRSDIR_SIMPLE = 1;
const uint32 RDMA_RVRSDIR_CONT = 2;
const uint32 RDMA_RVRSDIR_GENL = 3;
```

```
const uint32 RDMA2_PROPID_BRS = 5;
typedef uint32 rpcrdma2_prop_brs;
```

<CODE ENDS>



Multiple levels of support are distinguished:

- o The value `RDMA2_RVRSDIR_NONE` indicates that the sender does not support reverse-direction operation.
- o The value `RDMA2_RVRSDIR_SIMPLE` indicates that the sender supports using only Simple Format messages without chunks for reverse-direction messages.
- o The value `RDMA2_RVRSDIR_CONT` indicates that the sender supports using either Simple Format without chunks or Continued Format messages without chunks for reverse-direction messages.
- o The value `RDMA2_RVRSDIR_GENL` indicates that the sender supports reverse-direction messages in the same way as forward-direction messages.

When a peer does not provide this property, the default is the peer does not support reverse-direction operation.

#### **[5.2.6.](#) Host Authentication Message**

The value of this transport property enables the exchange of host authentication material. This property can accommodate authentication handshakes that require multiple challenge-response interactions and potentially large amounts of material.

<CODE BEGINS>

```
const uint32 RDMA2_PROPID_HOSTAUTH = 6;
typedef opaque rpcrdma2_prop_hostauth<>;
```

<CODE ENDS>

When this property is not present, the peer(s) remain unauthenticated. Local security policy on each peer determines whether the connection is permitted to continue.

## **[6.](#) Transport Messages**

Each transport message consists of multiple sections.

- o A transport header prefix, as defined in [Section 6.2.2](#). Among other things, this structure indicates the header type.
- o The transport header proper, as defined by one of the sub-sections below. See [Section 6.1](#) for the mapping between header types and the corresponding header structure.



- o Potentially, all or part of an RPC message payload.

This organization differs from that presented in the definition of RPC-over-RDMA version 1 [[RFC8166](#)], which defined the first and second of the items above as a single XDR data structure. The new organization is in keeping with RPC-over-RDMA version 2's extensibility model, which enables the definition of new header types without modifying the XDR definition of existing header types.

### 6.1. Transport Header Types

Table 2 lists the RPC-over-RDMA version 2 header types. The columns contain the following information:

- o The column labeled "Operation" names the particular operation.
- o The column labeled "Code" specifies the value of the header type for this operation.
- o The column labeled "XDR type" gives the XDR type of the data structure used to organize the information in this new message type. This data immediately follows the universal portion on the transport header present in every RPC-over-RDMA transport header.
- o The column labeled "Msg" indicates whether this operation is followed (or not) by an RPC message payload.
- o The column labeled "Section" refers to the section within the current document that explains the use of this header type.

Operation	Code	XDR type	Msg	Section
Convey Appended RPC Message	0	rpcrdma2_msg	Yes	6.3.1
Convey External RPC Message	1	rpcrdma2_nomsg	No	6.3.2
Report Transport Error	4	rpcrdma2_err	No	6.3.3
Specify Properties at Connection	5	rpcrdma2_connprop	No	6.3.4

Table 2

Support for the operations in Table 2 is REQUIRED. RPC-over-RDMA version 2 implementations that receive an unrecognized header type MUST respond with an RDMA2\_ERROR message with an rdma\_err field



containing RDMA2\_ERR\_INVALID\_HTYPE and drop the incoming message without processing it further.

## 6.2. Headers and Chunks

Most RPC-over-RDMA version 2 data structures have antecedents in corresponding structures in RPC-over-RDMA version 1. As is typical for new versions of an existing protocol, the XDR data structures have new names, and there are a few small changes in content. In some cases, there have been structural re-organizations to enable protocol extensibility.

### 6.2.1. Common Transport Header Prefix

The `rpcrdma_common` structure defines the initial part of each RPC-over-RDMA transport header for RPC-over-RDMA version 2 and subsequent versions.

<CODE BEGINS>

```
struct rpcrdma_common {
    uint32      rdma_xid;
    uint32      rdma_vers;
    uint32      rdma_credit;
    uint32      rdma_htype;
};
```

<CODE ENDS>

RPC-over-RDMA version 2's use of these first four words matches that of version 1 as required by [RFC8166]. However, there are crucial structural differences in the XDR definition of RPC-over-RDMA version 2: in the way that these words are described by the respective XDR descriptions:

- o The header type is represented as a `uint32` rather than as an enum type. An enum would need to be modified to reflect additions to the set of header types made by later extensions.
- o The header type field is part of an XDR structure devoted to representing the transport header prefix, rather than being part of a discriminated union, that includes the body of each transport header type.
- o There is now a prefix structure (see [Section 6.2.2](#)) of which the `rpcrdma_common` structure is the initial segment. This prefix is a newly defined XDR object within the protocol description, which





constrains the universal portion of all header types to the four words in `rpcrdma_common`.

These changes are part of a more considerable structural change in the XDR definition of RPC-over-RDMA version 2 that facilitates a cleaner treatment of protocol extension. The XDR appearing in [Section 8](#) reflects these changes, which [Appendix C.1](#) discusses in further detail.

### **6.2.2. Transport Header Prefix**

The following prefix structure appears at the start of each RPC-over-RDMA version 2 transport header.

<CODE BEGINS>

```
const RDMA2_F_RESPONSE      0x00000001;
const RDMA2_F_MORE          0x00000002;
const RDMA2_F_TPMORE        0x00000004;

struct rpcrdma2_hdr_prefix
    struct rpcrdma_common    rdma_start;
    uint32                   rdma_flags;
};
```

<CODE ENDS>

The `rdma_flags` field is new to RPC-over-RDMA version 2. Currently, the only flags defined within this word are the `RDMA2_F_RESPONSE` flag and the `RDMA2_F_MORE` flag. The other flags are reserved for future use as described in [Appendix B.3](#). The sender MUST set reserved flags to zero, and the receiver MUST ignore reserved flags.

#### **6.2.2.1. RDMA2\_F\_RESPONSE Flag**

The `RDMA2_F_RESPONSE` flag qualifies the value contained in the transport header's `rdma_xid` field. The `RDMA2_F_RESPONSE` flag enables a receiver to avoid performing an XID lookup on incoming reverse direction Call messages. Therefore:

- o When the `rdma_htype` field has the value `RDMA2_MSG` or `RDMA2_NOMSG`, the value of the `RDMA2_F_RESPONSE` flag MUST be the same as the value of the associated RPC message's `msg_type` field.
- o When the header type is anything else and a whole or partial RPC message payload is present, the value of the `RDMA2_F_RESPONSE` flag MUST be the same as the value of the associated RPC message's `msg_type` field.



- o When no RPC message payload is present, a Requester MUST set the value of RDMA2\_F\_RESPONSE to reflect how the receiver is to interpret the rdma\_xid field.
- o When the rdma\_htype field has the value RDMA2\_ERROR, the RDMA2\_F\_RESPONSE flag MUST be set.

#### **6.2.2.2. RDMA2\_F\_MORE Flag**

The RDMA2\_F\_MORE flag signifies that the RPC-over-RDMA message payload continues in the next message.

When the sender asserts the RDMA2\_F\_MORE flag, the receiver is to concatenate the data payload of the next received message to the end of the data payload of the received message. The sender clears the RDMA2\_F\_MORE flag in the final message in the sequence.

All RPC-over-RDMA messages in such a sequence MUST have the same values in the rdma\_xid and rdma\_htype fields. Otherwise, the receiver MUST drop the message without processing it further. If the receiver is a Responder, it MUST also respond with an RDMA2\_ERROR message with the rdma\_err field set to RDMA2\_ERR\_INVALID\_CONT.

If a peer receives an RPC-over-RDMA message with the RDMA2\_F\_MORE flag set, and the rdma\_htype field does not contain the values RDMA2\_MSG or RDMA2\_CONNPORP, the receiver MUST drop the message without processing it further. If the receiver is a Responder, it MUST also respond with an RDMA2\_ERROR message with the rdma\_err field set to RDMA2\_ERR\_INVALID\_CONT.

The sender includes chunks only in the final message in a sequence, in which the RDMA2\_F\_MORE flag is clear. If a peer receives an RPC-over-RDMA message with the RDMA2\_F\_MORE flag set, and its chunk lists are not empty, the receiver MUST drop the message without processing it further. If the receiver is a Responder, it MUST also respond with an RDMA2\_ERROR message with the rdma\_err field set to RDMA2\_ERR\_INVALID\_CONT.

There is no protocol-defined limit on the number of concatenated messages in a sequence. If the sender exhausts the receiver's credit grant before sending the final message in the sequence, the sender waits for a further credit grant from the receiver before continuing to send messages.

Credit exhaustion can occur at the receiver in the middle of a sequence of continued messages. The receiver can grant more credits by sending an RPC message payload or an out-of-band credit grant (see



[Section 4.2.1.2](#)) to enable the sender to send the remaining messages in the sequence.

#### **6.2.2.3. RDMA2\_F\_TPMORE Flag**

The RDMA2\_F\_TPMORE flag indicates that the sender has additional Transport Properties to send in a subsequent RPC-over-RDMA message. If a peer receives any message type other than RDMA2\_CONNPROP with the RDMA2\_F\_TPMORE flag set, it MUST respond with an RDMA2\_ERROR message type whose rdma\_err field contains RDMA2\_ERR\_INVALID\_HTYPE, and then silently discard the ingress message without processing it.

The RDMA2\_F\_TPDONE flag is clear in the final RDMA2\_CONNPROP message type from this peer on this connection. If a peer receives an RDMA2\_CONNPROP message type after it has received an RDMA2\_CONNPROP message type with a clear RDMA2\_F\_TPDONE flag, it MUST respond with an RDMA2\_ERROR message type whose rdma\_err field contains RDMA2\_ERR\_INVALID\_HTYPE, and then silently discard the ingress message without processing it.

After both connection peers have indicated they have finished sending their Transport Properties, they may begin passing RPC traffic.

#### **6.2.3. External Data Payloads**

The rpcrdma2\_chunk\_lists structure specifies how explicit RDMA operations convey the message payload.

<CODE BEGINS>

```
struct rpcrdma2_chunk_lists {
    uint32                rdma_inv_handle;
    struct rpcrdma2_read_list *rdma_reads;
    struct rpcrdma2_write_list *rdma_writes;
    struct rpcrdma2_write_chunk *rdma_reply;
};
```

<CODE ENDS>

The rdma\_reads, rdma\_writes, rdma\_reply fields provide, respectively, the chunks used to read a Special Format Call or pull directly placed data from the Requester; the chunks used to push directly placed response data into the Requester's memory; and the chunk used to push a long Reply into the Requester's memory. See [Section 4.3](#) for further details on how a sender constructs chunks.



#### **6.2.4. Remote Invalidation**

A central addition relative to the corresponding RPC-over-RDMA version 1 `rdma_header` structures is the `rdma_inv_handle` field. This field enables remote invalidation of one Requester memory registration by using the RDMA Send With Invalidate operation.

To solicit the use of Remote Invalidation, a Requester sets the value of the `rdma_inv_handle` field in an RPC Call's transport header to a non-zero value that matches one of the `rdma_handle` fields in that header. If the Responder may invalidate none of the `rdma_handle` values in the header conveying the Call, the Requester sets the RPC Call's `rdma_inv_handle` field to the value zero.

If the Responder chooses not to use remote invalidation for this particular RPC Reply, or the RPC Call's `rdma_inv_handle` field contains the value zero, the Responder simply uses RDMA Send to transmit the matching RPC reply. However, if the Responder chooses to use Remote Invalidation, it uses RDMA Send With Invalidate to transmit the RPC Reply. It MUST use the value in the corresponding Call's `rdma_inv_handle` field to construct the Send With Invalidate Work Request.

#### **6.3. Header Types**

The header types defined and used in RPC-over-RDMA version 1 are carried over into RPC-over-RDMA version 2, although there are some limited changes in the definitions of existing header types:

- o To simplify interoperability with RPC-over-RDMA version 1, only the `RDMA2_ERROR` header (defined in [Section 6.3.3](#)) has an XDR definition that differs from that in RPC-over-RDMA version 1, and its modifications are all compatible extensions.
- o `RDMA2_MSG` and `RDMA2_NOMSG` (defined in [Sections 6.3.1](#) and [6.3.2](#)) have XDR definitions that match the corresponding RPC-over-RDMA version 1 header types. However, because of the changes to the header prefix, the version 1 and version 2 header types differ in on-the-wire format.
- o `RDMA2_CONNPTR` (defined in [Section 6.3.4](#)) is an entirely new header type devoted to enabling connection peers to exchange information about their transport properties.





### **6.3.1. RDMA2\_MSG: Convey RPC Message Inline**

RDMA2\_MSG conveys all or part of an RPC message immediately following the transport header in the send buffer.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_MSG = 0;

struct rpcrdma2_msg {
    struct rpcrdma2_chunk_lists rdma_chunks;

    /* The rpc message starts here and continues
     * through the end of the transmission. */
    uint32 rdma_rpc_first_word;
};
```

<CODE ENDS>

### **6.3.2. RDMA2\_NOMSG: Convey External RPC Message**

RDMA2\_NOMSG conveys an entire RPC message payload using explicit RDMA operations. In particular, it is a Special Format Call when the Responder reads the RPC payload from a memory area specified by a Position Zero Read chunk. It is a Special Format Reply when the Responder writes the RPC payload into a memory area specified by a Reply chunk. In both cases, the sender sets the `rdma_xid` field to the same value as the `xid` of the RPC message payload.

If all the chunk lists are empty the message conveys a credit grant refresh. The header prefix of this message contains a credit grant refresh in the `rdma_credit` field. In this case, the sender **MUST** set the `rdma_xid` field to zero.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_NOMSG = 1;

struct rpcrdma2_nomsg {
    struct rpcrdma2_chunk_lists rdma_chunks;
};
```

<CODE ENDS>

In RPC-over-RDMA version 2, a sender should use Message Continuation as an alternative to using a Special Format message.



### **6.3.3. RDMA2\_ERROR: Report Transport Error**

RDMA2\_ERROR reports a transport layer error on a previous transmission.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_ERROR = 4;

struct rpcrdma2_err_vers {
    uint32 rdma_vers_low;
    uint32 rdma_vers_high;
};

struct rpcrdma2_err_write {
    uint32 rdma_chunk_index;
    uint32 rdma_length_needed;
};

union rpcrdma2_error switch (rpcrdma2_errcode rdma_err) {
    case RDMA2_ERR_VERS:
        rpcrdma2_err_vers rdma_vrange;
    case RDMA2_ERR_READ_CHUNKS:
        uint32 rdma_max_chunks;
    case RDMA2_ERR_WRITE_CHUNKS:
        uint32 rdma_max_chunks;
    case RDMA2_ERR_SEGMENTS:
        uint32 rdma_max_segments;
    case RDMA2_ERR_WRITE_RESOURCE:
        rpcrdma2_err_write rdma_writeres;
    case RDMA2_ERR_REPLY_RESOURCE:
        uint32 rdma_length_needed;
    default:
        void;
};
```

<CODE ENDS>

See [Section 7](#) for details on the use of this header type.

### **6.3.4. RDMA2\_CONNPROP: Exchange Transport Properties**

The RDMA2\_CONNPROP message type enables a connection peer to publish the properties of its implementation to its remote peer.



<CODE BEGINS>

```
struct rpcrdma2_connprop {  
    rpcrdma2_propset rdma_props;  
};
```

<CODE ENDS>

Each peer sends an RDMA2\_CONNPROP message type as the first message after the client has established a connection. The size of this initial message is limited to the default inline threshold for the RPC-over-RDMA version that is in effect. If a peer has more or larger Transport Properties than can fit in the initial RDMA2\_CONNPROP message type, it sets the RDMA2\_F\_TPMORE flag. The final RDMA2\_CONNPROP message type the peer sends on that connection must have a clear RDMA2\_F\_TPMORE flag.

A peer may encounter properties that it does not recognize or support. In such cases, the receiver ignores unsupported properties without generating an error response.

#### **6.4. Choosing a Reply Mechanism**

A Requester provisions all necessary registered memory resources for both an RPC Call and its matching RPC Reply. A Requester constructs each RPC Call, thus it can compute the exact memory resources needed to send every Call. However, the Requester allocates memory resources to receive the corresponding Reply before the Responder has constructed it. Occasionally, it is challenging for the Requester to know in advance precisely what resources are needed to receive the Reply.

In RPC-over-RDMA version 2, a Requester can provide a Reply chunk for any transaction. The Responder can use the provided Reply chunk or it can decide to use another means to convey the RPC Reply. If the combination of the provided Write chunk list and Reply chunk is not adequate to convey a Reply, the Responder SHOULD use Message Continuation (see [Section 6.2.2.2](#)) to send that Reply. If even that is not possible, the Responder sends an RDMA2\_ERROR message to the Requester, as described in [Section 6.3.3](#):

- o If the Write chunk list cannot accommodate the ULP's DDP-eligible data payload, the Responder sends an RDMA2\_ERR\_WRITE\_RESOURCE error.
- o If the Reply chunk cannot accommodate the parts of the Reply that are not DDP-eligible, the Responder sends an RDMA2\_ERR\_REPLY\_RESOURCE error.



When receiving such errors, the Requester can retry the ULP call using more substantial reply resources. In cases where retrying the ULP request is not possible (e.g., the request is non-idempotent), the Requester terminates the RPC transaction and presents an error to the RPC consumer.

## **7. Error Handling**

A receiver performs validity checks on ingress RPC-over-RDMA messages before it passes the conveyed RPC message to the RPC layer. For example, if an ingress message is not as long as the size of struct `rpcrdma2_hdr_prefix` (20 octets), the receiver cannot trust the value of the `rdma_xid` field. In this case, the receiver **MUST** silently discard the ingress message without processing it further, and without a response to the sender.

However, in many other cases, the receiver needs to actively report a problem with the RPC-over-RDMA message to its sender. The `RDMA2_ERROR` message type is used for this purpose. To form an `RDMA2_ERROR` message type:

- o The `rdma_xid` field **MUST** contain the same XID that was in the `rdma_xid` field in the ingress request.
- o The `rdma_vers` field **MUST** contain the same version that was in the `rdma_vers` field in the ingress request.
- o The sender sets the `rdma_credit` field to the credit values in effect for this connection.
- o The `rdma_htype` field **MUST** contain the value `RDMA2_ERROR`.
- o The `rdma_err` field contains a value that reflects the type of error that occurred, as described in the subsections below.

When a peer receives an `RDMA2_ERROR` message type with an unrecognized or unsupported value in its `rdma_err` field, it **MUST** silently discard the message without processing it further.

### **7.1. Basic Transport Stream Parsing Errors**

#### **7.1.1. RDMA2\_ERR\_VERS**

When a Responder detects an RPC-over-RDMA header version that it does not support (the current document defines version 2), it **MUST** respond with an `RDMA2_ERROR` message type and set its `rdma_err` field to `RDMA2_ERR_VERS`. The Responder then fills in the `rpcrdma2_err_vers` structure with the RPC-over-RDMA versions it supports. The Responder





MUST silently discard the ingress message without passing it to the RPC layer

When a Requester receives this error, it uses the information in the `rpcrdma2_err_vers` structure to select an RPC-over-RDMA version that both peers support. A Requester MUST NOT subsequently send a message that uses a version that the Responder has indicated it does not support. `RDMA2_ERR_VERS` indicates a permanent error. Receipt of this error completes the RPC transaction associated with XID in the `rdma_xid` field.

#### **7.1.2. RDMA2\_ERR\_INVALID\_HTYPE**

If a Responder recognizes the value in an ingress `rdma_vers` field, but it does not recognize the value in the `rdma_htype` field or does not support that header type, it MUST set the `rdma_err` field to `RDMA2_ERR_INVALID_HTYPE`. The Responder MUST silently discard the incoming message without passing it to the RPC layer.

A Requester MUST NOT subsequently send a message that uses an `htype` that the Responder has indicated it does not support. `RDMA2_ERR_INVALID_HTYPE` indicates a permanent error. Receipt of this error completes the RPC transaction associated with XID in the `rdma_xid` field.

#### **7.1.3. RDMA2\_ERR\_INVALID\_CONT**

If a Responder detects a problem with an ingress RPC-over-RDMA message that is part of a Message Continuation sequence, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_INVALID_CONT`. [Section 6.2.2.2](#) discusses the types of problems to watch for. The Responder MUST silently discard all ingress messages with an `rdma_xid` field that matches the failing message without reassembling the payload.

`RDMA2_ERR_INVALID_CONT` indicates a permanent error. Receipt of this error completes the RPC transaction associated with XID in the `rdma_xid` field.

### **7.2. XDR Errors**

A receiver might encounter an XDR parsing error that prevents it from processing an ingress Transport stream. Examples of such errors include:

- o An invalid value in the `rdma_proc` field.



- o An RDMA\_NOMSG message where the Read list, Write list, and Reply chunk are marked not present.
- o The value of the rdma\_xid field does not match the value of the XID field in the accompanying RPC message.

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, the Responder returns an RPC Reply with status GARBAGE\_ARGS, using an RDMA2\_MSG message type. This type of parsing failure might be due to mismatches between chunk sizes or offsets and the contents of the Payload stream, for example. In this case, the error is permanent, but the Requester has no way to know how much processing the Responder has completed for this RPC transaction.

#### **7.2.1. RDMA2\_ERR\_BAD\_XDR**

If a Responder recognizes the values in the rdma\_vers field, but it cannot otherwise parse the ingress Transport stream, it MUST set the rdma\_err field to RDMA2\_ERR\_BAD\_XDR. The Responder MUST silently discard the ingress message without passing it to the RPC layer.

RDMA2\_ERR\_BAD\_XDR indicates a permanent error. Receipt of this error completes the RPC transaction associated with XID in the rdma\_xid field.

#### **7.2.2. RDMA2\_ERR\_BAD\_PROPVAL**

If a receiver recognizes the value in an ingress rdma\_which field, but it cannot parse the accompanying propval, it MUST set the rdma\_err field to RDMA2\_ERR\_BAD\_PROPVAL (see [Section 5.1](#)). The receiver MUST silently discard the ingress message without applying any of its property settings.

### **7.3. Responder RDMA Operational Errors**

In RPC-over-RDMA version 2, 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:

- o 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.



- o If a Requester's Receive buffer is too small, the Responder's Send operation completes with a Local Length Error.
- o 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 can be 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 RDMA2\_ERROR response to indicate that no RPC-level reply is possible for that transaction.

#### **7.3.1. RDMA2\_ERR\_READ\_CHUNKS**

If a Requester presents more DDP-eligible arguments than a Responder is prepared to Read, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_READ_CHUNKS` and set the `rdma_max_chunks` field to the maximum number of Read chunks the Responder can process. If the Responder implementation cannot handle any Read chunks for a request, it MUST set the `rdma_max_chunks` to zero in this response. The Responder MUST silently discard the ingress message without processing it further.

The Requester can reconstruct the Call using Message Continuation or a Special Format payload and resend it. If the Requester does not resend the Call, it MUST terminate this RPC transaction with an error.

#### **7.3.2. RDMA2\_ERR\_WRITE\_CHUNKS**

If a Requester has constructed an RPC Call with more DDP-eligible results than the Responder is prepared to Write, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_WRITE_CHUNKS` and set the `rdma_max_chunks` field to the maximum number of Write chunks the Responder can return. The Requester can reconstruct the Call with no Write chunks and a Reply chunk of appropriate size. If the Requester does not resend the Call, it MUST terminate this RPC transaction with an error.

If the Responder implementation cannot handle any Write chunks for a request and cannot send the Reply using Message Continuation, it MUST return a response of `RDMA2_ERR_REPLY_RESOURCE` instead (see below).



### **7.3.3. RDMA2\_ERR\_SEGMENTS**

If a Requester has constructed an RPC Call with a chunk that contains more segments than the Responder supports, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_SEGMENTS` and set the `rdma_max_segments` field to the maximum number of segments the Responder can process. The Requester can reconstruct the Call and resend it. If the Requester does not resend the Call, it MUST terminate this RPC transaction with an error.

### **7.3.4. RDMA2\_ERR\_WRITE\_RESOURCE**

If a Requester has provided a Write chunk that is not large enough to contain a DDP-eligible result, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_WRITE_RESOURCE`.

The Responder MUST set the `rdma_chunk_index` field to point to the first Write chunk in the transport header that is too short, or to zero to indicate that it was not possible to determine which chunk is too small. Indexing starts at one (1), which represents the first Write chunk. The Responder MUST set the `rdma_length_needed` to the number of bytes needed in that chunk to convey the result data item.

The Requester can reconstruct the Call with more reply resources and resend it. If the Requester does not resend the Call (for instance, if the Responder set the index and length fields to zero), it MUST terminate this RPC transaction with an error.

### **7.3.5. RDMA2\_ERR\_REPLY\_RESOURCE**

If a Responder cannot send an RPC Reply using Message Continuation and the Reply does not fit in the Reply chunk, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_REPLY_RESOURCE`. The Responder MUST set the `rdma_length_needed` to the number of Reply chunk bytes needed to convey the reply.

The Requester can reconstruct the Call with more reply resources and resend it. If the Requester does not resend the Call (for instance, if the Responder set the length field to zero), it MUST terminate this RPC transaction with an error.

## **7.4. 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, 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.

#### **7.4.1. RDMA2\_ERR\_SYSTEM**

If some problem occurs on a Responder that does not fit into the above categories, the Responder MAY report it to the Requester by setting the `rdma_err` field to `RDMA2_ERR_SYSTEM`. The Responder MUST silently discard the message(s) associated with the failing transaction without further processing.

`RDMA2_ERR_SYSTEM` is a permanent error. This error does not indicate how much of the transaction the Responder has processed, nor does it indicate a particular recovery action for the Requester. A Requester that receives this error MUST terminate the RPC transaction associated with the `XID` value in the `RDMA2_ERROR` message's `rdma_xid` field.

#### **7.5. 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 link layer [[IBA](#)] all provide Cyclic Redundancy Check (CRC) protection of RDMA payloads. 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 typically 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 11](#) for further discussion of RPC-level integrity schemes.



## **8. XDR Protocol Definition**

This section contains a description of the core features of the RPC-over-RDMA version 2 protocol expressed in the XDR language [[RFC4506](#)]. It organizes the description to make it simple to extract into a form that is ready to compile or combine with similar descriptions published later as extensions to RPC-over-RDMA version 2.

### **8.1. Code Component License**

Code Components extracted from the current document must include the following license text. When combining the extracted XDR code with other XDR code which has an identical license, only a single copy of the license text needs to be retained.



<CODE BEGINS>

```
/// /*
///  * Copyright (c) 2010-2018 IETF Trust and the persons
///  * identified as authors of the code.  All rights reserved.
///  *
///  * The authors of the code are:
///  * B. Callaghan, T. Talpey, C. Lever, and D. Noveck.
///  *
///  * 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>



## **8.2. Extraction of the XDR Definition**

Implementers can apply the following sed script to the current document to produce a machine-readable XDR description of the base RPC-over-RDMA version 2 protocol.

<CODE BEGINS>

```
sed -n -e 's:^ */// ::p' -e 's:^ *///$::p'
```

<CODE ENDS>

That is, if this document is in a file called "spec.txt", then implementers can do the following to extract an XDR description file and store it in the file rpcrdma-v2.x.

<CODE BEGINS>

```
sed -n -e 's:^ */// ::p' -e 's:^ *///$::p' \  
    < spec.txt > rpcrdma-v2.x
```

<CODE ENDS>

Although this file is a usable description of the base protocol, when extensions are to be supported, it may be desirable to divide the description into multiple files. The following script achieves that purpose:





<CODE BEGINS>

```
#!/usr/local/bin/perl
open(IN, "rpcrdma-v2.x");
open(OUT, ">temp.x");
while(<IN>)
{
    if (m/FILE ENDS: (.*)$/)
    {
        close(OUT);
        rename("temp.x", $1);
        open(OUT, ">temp.x");
    }
    else
    {
        print OUT $_;
    }
}
close(IN);
close(OUT);
```

<CODE ENDS>

Running the above script results in two files:

- o The file `common.x`, containing the license plus the shared XDR definitions that need to be made available to both the base protocol and any subsequent extensions.
- o The file `baseops.x` containing the XDR definitions for the base protocol defined in this document.

Extensions to RPC-over-RDMA version 2, published as Standards Track documents, should have similarly structured XDR definitions. Once an implementer has extracted the XDR for all desired extensions and the base XDR definition contained in the current document, she can concatenate them to produce a consolidated XDR definition that reflects the set of extensions selected for her RPC-over-RDMA version 2 implementation.

Alternatively, the XDR descriptions can be compiled separately. In that case, the combination of `common.x` and `baseops.x` defines the base transport. The combination of `common.x` and the XDR description of each extension produces a full XDR definition of that extension.



**8.3. XDR Definition for RPC-over-RDMA Version 2 Core Structures**

```

<CODE BEGINS>
/// /*****
///  *      Transport Header Prefixes
///  *****/
///
/// struct rpcrdma_common {
///     uint32      rdma_xid;
///     uint32      rdma_vers;
///     uint32      rdma_credit;
///     uint32      rdma_htype;
/// };
///
/// const RDMA2_F_RESPONSE      0x00000001;
/// const RDMA2_F_MORE          0x00000002;
/// const RDMA2_F_TPMORE        0x00000004;
///
/// struct rpcrdma2_hdr_prefix
///     struct rpcrdma_common      rdma_start;
///     uint32                      rdma_flags;
/// };
///
/// /*****
///  *      Chunks and Chunk Lists
///  *****/
///
/// struct rpcrdma2_segment {
///     uint32 rdma_handle;
///     uint32 rdma_length;
///     uint64 rdma_offset;
/// };
///
/// struct rpcrdma2_read_segment {
///     uint32      rdma_position;
///     struct rpcrdma2_segment rdma_target;
/// };
///
/// struct rpcrdma2_read_list {
///     struct rpcrdma2_read_segment rdma_entry;
///     struct rpcrdma2_read_list    *rdma_next;
/// };
///
/// struct rpcrdma2_write_chunk {
///     struct rpcrdma2_segment rdma_target<>;
/// };
///
/// struct rpcrdma2_write_list {

```



```
///      struct rpcrdma2_write_chunk rdma_entry;
///      struct rpcrdma2_write_list  *rdma_next;
/// };
///
/// struct rpcrdma2_chunk_lists {
///      uint32          rdma_inv_handle;
///      struct rpcrdma2_read_list  *rdma_reads;
///      struct rpcrdma2_write_list *rdma_writes;
///      struct rpcrdma2_write_chunk *rdma_reply;
/// };
///
/// /*****
///  *   Transport Properties
///  *****/
///
/// /*
///  * Types for transport properties model
///  */
/// typedef rpcrdma2_propid uint32;
///
/// struct rpcrdma2_propval {
///      rpcrdma2_propid rdma_which;
///      opaque          rdma_data<>;
/// };
///
/// typedef rpcrdma2_propval rpcrdma2_propset<>;
/// typedef uint32 rpcrdma2_propsubset<>;
///
/// /*
///  * Transport propid values for basic properties
///  */
/// const uint32 RDMA2_PROPID_SBSIZ = 1;
/// const uint32 RDMA2_PROPID_RBSIZ = 2;
/// const uint32 RDMA2_PROPID_RSSIZ = 3;
/// const uint32 RDMA2_PROPID_RCSIZ = 4;
/// const uint32 RDMA2_PROPID_BRS = 5;
/// const uint32 RDMA2_PROPID_HOSTAUTH = 6;
///
/// /*
///  * Types specific to particular properties
///  */
/// typedef uint32 rpcrdma2_prop_sbsiz;
/// typedef uint32 rpcrdma2_prop_rbsiz;
/// typedef uint32 rpcrdma2_prop_rssiz;
/// typedef uint32 rpcrdma2_prop_rcsiz;
/// typedef uint32 rpcrdma2_prop_brs;
/// typedef opaque rpcrdma2_prop_hostauth<>;
///
```



```

/// const uint32 RDMA2_RVRSDIR_NONE = 0;
/// const uint32 RDMA2_RVRSDIR_SIMPLE = 1;
/// const uint32 RDMA2_RVRSDIR_CONT = 1;
/// const uint32 RDMA2_RVRSDIR_GENL = 3;
///
/// /* FILE ENDS: common.x; */

```

<CODE ENDS>

#### 8.4. XDR Definition for RPC-over-RDMA Version 2 Base Header Types

```

<CODE BEGINS>
/// /*****
///  *   Descriptions of RPC-over-RDMA Header Types
///  *****/
///
/// /*
///  * Header Type Codes.
///  */
/// const rpcrdma2_proc RDMA2_MSG = 0;
/// const rpcrdma2_proc RDMA2_NOMSG = 1;
/// const rpcrdma2_proc RDMA2_ERROR = 4;
/// const rpcrdma2_proc RDMA2_CONNPROP = 5;
///
/// /*
///  * Header Types to Convey RPC Messages.
///  */
/// struct rpcrdma2_msg {
///     struct rpcrdma2_chunk_lists rdma_chunks;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32 rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_nomsg {
///     struct rpcrdma2_chunk_lists rdma_chunks;
/// };
///
/// /*
///  * Header Type to Report Errors.
///  */
/// const uint32 RDMA2_ERR_VERS = 1;
/// const uint32 RDMA2_ERR_BAD_XDR = 2;
/// const uint32 RDMA2_ERR_BAD_PROPVAL = 3;
/// const uint32 RDMA2_ERR_INVALID_HTYPE = 4;
/// const uint32 RDMA2_ERR_INVALID_CONT = 5;
/// const uint32 RDMA2_ERR_READ_CHUNKS = 6;

```





```
/// const uint32 RDMA2_ERR_WRITE_CHUNKS = 7;
/// const uint32 RDMA2_ERR_SEGMENTS = 8;
/// const uint32 RDMA2_ERR_WRITE_RESOURCE = 9;
/// const uint32 RDMA2_ERR_REPLY_RESOURCE = 10;
/// const uint32 RDMA2_ERR_SYSTEM = 100;
///
/// struct rpcrdma2_err_vers {
///     uint32 rdma_vers_low;
///     uint32 rdma_vers_high;
/// };
///
/// struct rpcrdma2_err_write {
///     uint32 rdma_chunk_index;
///     uint32 rdma_length_needed;
/// };
///
/// union rpcrdma2_error switch (rpcrdma2_errcode rdma_err) {
///     case RDMA2_ERR_VERS:
///         rpcrdma2_err_vers rdma_vrange;
///     case RDMA2_ERR_READ_CHUNKS:
///         uint32 rdma_max_chunks;
///     case RDMA2_ERR_WRITE_CHUNKS:
///         uint32 rdma_max_chunks;
///     case RDMA2_ERR_SEGMENTS:
///         uint32 rdma_max_segments;
///     case RDMA2_ERR_WRITE_RESOURCE:
///         rpcrdma2_err_write rdma_writeres;
///     case RDMA2_ERR_REPLY_RESOURCE:
///         uint32 rdma_length_needed;
///     default:
///         void;
/// };
///
/// /*
///  * Header Type to Exchange Transport Properties.
///  */
/// struct rpcrdma2_connprop {
///     rpcrdma2_propset rdma_props;
/// };
///
/// /* FILE ENDS: baseops.x; */
```

<CODE ENDS>



### **8.5. Use of the XDR Description**

The files `common.x` and `baseops.x`, when combined with the XDR descriptions for extension defined later, produce a human-readable and compilable description of the RPC-over-RDMA version 2 protocol with the included extensions.

Although this XDR description can generate encoders and decoders for the Transport and Payload streams, there are elements of the operation of RPC-over-RDMA version 2 that cannot be expressed within the XDR language. Implementations that use the output of an automated XDR processor need to provide additional code to bridge these gaps.

- o The Transport stream is not a single XDR object. Instead, the header prefix is one XDR data item, and the rest of the header is a separate XDR data item. Table 2 expresses the mapping between the header type in the header prefix and the XDR object representing the header type.
- o The relationship between the Transport stream and the Payload stream is not specified using XDR. Comments within the XDR text make clear where transported messages, described by their own XDR definitions, need to appear. Such data is opaque to the transport.
- o Continuation of RPC messages across transport message boundaries requires that message assembly facilities not specifiable within XDR are part of transport implementations.
- o Transport properties are constant integer values. Table 1 expresses the mapping between each property's code point and the XDR typedef that represents the structure of the property's value. XDR does not possess the facility to express that mapping in an extensible way.

The role of XDR in RPC-over-RDMA specifications is more limited than for protocols where the totality of the protocol is expressible within XDR. XDR lacks the facility to represent the embedding of XDR-encoded payload material. Also, the need to cleanly accommodate extensions has meant that those using `rpcgen` in their applications need to take an active role to provide the facilities that cannot be expressed within XDR.



## 9. RPC Bind Parameters

Before establishing a new connection, an RPC client obtains a transport address for the RPC server. 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 typically 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 distinct service address (port number) is sometimes required for operation on RPC-over-RDMA.

When mapped atop the iWARP transport [[RFC5040](#)] [[RFC5041](#)], which uses IP port addressing due to its layering on TCP 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 [[RFC8267](#)].

When mapped atop InfiniBand [[IBA](#)], 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 [[IBA](#)], 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:

- o One possibility is to have the server register its mapped IP port with the rpcbind service under the netid (or netids) defined in [[RFC8166](#)]. An RPC-over-RDMA-aware RPC client can then resolve its desired service to a mappable port and proceed to connect. This method is the most flexible and compatible approach for those upper layers that are defined to use the rpcbind service.
- o A second possibility is to have the RPC server's portmapper register itself on the RDMA interconnect at a "well-known" service address (on UDP or TCP, this corresponds to port 111). An RPC client can 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).



- o Alternately, an RPC client can 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 assignments. The current document leaves the specific method for each RPC-over-RDMA-enabled ULB.

[RFC8166] 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 [[IBA](#)]. Additional RDMA-capable networks MAY define their own netids, or if they provide a port translation, they MAY share the one defined in [[RFC8166](#)].

## **[10.](#) Implementation Status**

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [[RFC7942](#)]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs.

Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

At this time, no known implementations of the protocol described in the current document exist.

## **[11.](#) Security Considerations**

### **[11.1.](#) Memory Protection**

A primary consideration is the protection of the integrity and confidentiality of host 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 memory owned by user processes. Any RDMA provider used for RPC transport MUST conform to the requirements of [[RFC5042](#)] to satisfy these protections.





#### **11.1.1. Protection Domains**

The use of a Protection Domain 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 Connected behavior, implementations SHOULD cryptographically authenticate connection endpoints.

#### **11.1.2. Handle (STag) Predictability**

Implementations should use unpredictable memory handles for any operation requiring exposed memory regions. Exposing 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 underway. Therefore, implementations should avoid the use of persistently registered memory.

#### **11.1.3. Memory Protection**

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

Requesters should invalidate memory regions as soon as related RPC operations are complete. Invalidation and DMA unmapping of memory regions should complete before the receiver checks message integrity, and before the RPC consumer can use or alter the contents of the exposed memory region.

An RPC transaction on a Requester can terminate before a Reply arrives, for example, if the RPC consumer is signaled, or a segmentation fault occurs. When an RPC terminates abnormally, memory regions associated with that RPC should be invalidated before the Requester reuses those regions for other purposes.

#### **11.1.4. Denial of Service**

A detailed discussion of denial-of-service exposures that can result from the use of an RDMA transport appears in [Section 6.4 of \[RFC5042\]](#).

A Responder is not obliged to pull unreasonably large Read chunks. A Responder can use an RDMA2\_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 provider typically terminates the connection. For further



discussion, see [Section 6.3.3](#). Such repeated connection termination 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 (see [Section 4.2.1](#)). A sender can trigger a self-denial of service by exceeding the credit grant repeatedly.

When an RPC transaction terminates due to a signal or premature exit of an application process, a Requester should 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 Requester has released that memory.

On the Requester, a malfunctioning application or a malicious user can create a situation where RPCs initiate and abort continuously, 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.

## **[11.2.](#) 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 a layer above the RPC-over-RDMA transport. The integrity and privacy services require significant computation and movement of data on each endpoint host. Some performance benefits enabled by RDMA transports can be lost.

### **[11.2.1.](#) RPC-over-RDMA Protection at Other Layers**

For any RPC transport, utilizing RPCSEC\_GSS integrity or privacy services has performance implications. Protection below the RPC implementation is often a better choice in performance-sensitive deployments, especially if it, too, can be offloaded. Certain implementations of IPsec can be co-located in RDMA hardware, for example, without change to RDMA consumers and with 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.

Implementations MAY negotiate the use of protection in another layer 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](#)].

#### **11.2.2. RPCSEC\_GSS on RPC-over-RDMA Transports**

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

RPCSEC\_GSS extends the ONC RPC protocol 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.

Senders MUST NOT reduce protocol elements of RPCSEC\_GSS that appear in the Payload stream of an RPC-over-RDMA message. Such elements include control messages exchanged as part of establishing or destroying a security context, or data items that are part of RPCSEC\_GSS authentication material.

##### **11.2.2.1. RPCSEC\_GSS Context Negotiation**

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

##### **11.2.2.2. 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 Calls 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 larger on average as a result. ULP operations that fit in a Simple Format message when a simpler form of authentication is in use might need to be reduced or conveyed via a Special Format message when RPCSEC\_GSS authentication is in use. It is therefore more likely that a Requester provides



both a Read list and a Reply chunk in the same RPC-over-RDMA Transport header to convey a Special Format Call and provision a receptacle for a Special Format Reply.

In addition to this cost, the XDR encoding and decoding of each RPC message using RPCSEC\_GSS authentication requires per-message host compute resources to construct the GSS verifier.

#### **11.2.2.3. RPC-over-RDMA with RPCSEC\_GSS Integrity or Privacy**

The RPCSEC\_GSS integrity service enables endpoints to detect the 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, and before they have been XDR decoded after receipt. Connection endpoints use intermediate buffers to prevent exposure of encrypted or unverified cleartext data to RPC consumers. After a sender has verified, encrypted, and wrapped a message, 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 an encoded Payload stream. In a non-protected RPC-over-RDMA message, a reduced data item does not include XDR padding. 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 and 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. Senders can use only Simple and Continued Formats without chunks, or Special Format. In this mode, an RPC-over-RDMA transport operates in the same manner as a transport that does not support DDP.

#### **11.2.2.4. Protecting RPC-over-RDMA Transport Headers**

Like the header fields in an RPC message (e.g., the xid and mtype fields), RPCSEC\_GSS does not protect the RPC-over-RDMA Transport stream. XIDs, connection credit limits, and chunk lists (though not the content of the data items they refer to) are exposed to malicious behavior, which can 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 Transport 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 gain a similar effect. If such alterations occur, the use of RPCSEC\_GSS integrity or privacy services enables a Requester to detect unexpected material in a received RPC message.

Encryption at other layers, as described in [Section 11.2.1](#), protects the content of the Transport stream. RDMA transport implementations should conform to [\[RFC5042\]](#) to address attacks on RDMA protocols themselves.

### **[11.3.](#) Transport Properties**

Like other fields that appear in the Transport stream, transport properties are sent in the clear with no integrity protection, making them vulnerable to man-in-the-middle attacks.

For example, if a man-in-the-middle were to change the value of the Receive buffer size, it could reduce connection performance or trigger loss of connection. Repeated connection loss can impact performance or even prevent a new connection from being established. The recourse is to deploy on a private network or use transport layer encryption.

### **[11.4.](#) Host Authentication**

[ cel: This subsection is unfinished. ]

Wherein we use the relevant sections of [\[RFC3552\]](#) to analyze the addition of host authentication to this RPC-over-RDMA transport.

The authors refer readers to [Appendix C of \[RFC8446\]](#) for information on how to design and test a secure authentication handshake implementation.

## **[12.](#) IANA Considerations**

The RPC-over-RDMA family of transports have been assigned RPC netids by [\[RFC8166\]](#). A netid is an rpcbind [\[RFC1833\]](#) string used to identify the underlying protocol in order for RPC to select appropriate transport framing and the format of the service addresses and ports.



The following netid registry strings are already 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" when IPv6 addressing is employed. The netid assignment policy and registry are defined in [RFC5665]. The current document does not alter these netid assignments.

These netids MAY be used for any RDMA network that satisfies the requirements of [Section 3.2.2](#) and that is able to identify service endpoints using IP port addressing, possibly through use of a translation service as described in [Section 9](#).

## **[13. References](#)**

### **[13.1. Normative References](#)**

- [RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", [RFC 1833](#), DOI 10.17487/RFC1833, August 1995, <<https://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, <<https://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, <<https://www.rfc-editor.org/info/rfc4506>>.
- [RFC5042] Pinkerton, J. and E. Deleganes, "Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMA) Security", [RFC 5042](#), DOI 10.17487/RFC5042, October 2007, <<https://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, <<https://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, <<https://www.rfc-editor.org/info/rfc5531>>.



- [RFC5660] Williams, N., "IPsec Channels: Connection Latching", [RFC 5660](#), DOI 10.17487/RFC5660, October 2009, <<https://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, <<https://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, <<https://www.rfc-editor.org/info/rfc7861>>.
- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", [BCP 205](#), [RFC 7942](#), DOI 10.17487/RFC7942, July 2016, <<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC8166] Lever, C., Ed., Simpson, W., and T. Talpey, "Remote Direct Memory Access Transport for Remote Procedure Call Version 1", [RFC 8166](#), DOI 10.17487/RFC8166, June 2017, <<https://www.rfc-editor.org/info/rfc8166>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in [RFC 2119](#) Key Words", [BCP 14](#), [RFC 8174](#), DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8267] Lever, C., "Network File System (NFS) Upper-Layer Binding to RPC-over-RDMA Version 1", [RFC 8267](#), DOI 10.17487/RFC8267, October 2017, <<https://www.rfc-editor.org/info/rfc8267>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", [RFC 8446](#), DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

### **13.2. Informative References**

- [CBFC] Kung, H., Blackwell, T., and A. Chapman, "Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing", Proc. ACM SIGCOMM '94 Symposium on Communications Architectures, Protocols and Applications, pp. 101-114., August 1994.



[I-D.ietf-nfsv4-rpc-tls]

Lever, C. and T. Myklebust, "Towards Remote Procedure Call Encryption By Default", [draft-ietf-nfsv4-rpc-tls-05](#) (work in progress), January 2020.

[IBA]

InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1", Release 1.3, March 2015.

Available from <https://www.infinibandta.org/>

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

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

[RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", [RFC 1094](#), DOI 10.17487/RFC1094, March 1989, <<https://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, <<https://www.rfc-editor.org/info/rfc1813>>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", [BCP 72](#), [RFC 3552](#), DOI 10.17487/RFC3552, July 2003, <<https://www.rfc-editor.org/info/rfc3552>>.

[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, <<https://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, <<https://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, <<https://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, <<https://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, <<https://www.rfc-editor.org/info/rfc5662>>.
- [RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", [RFC 7530](#), DOI 10.17487/RFC7530, March 2015, <<https://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, <<https://www.rfc-editor.org/info/rfc8167>>.

## **[Appendix A.](#) ULB Specifications**

Typically, an Upper-Layer Protocol (ULP) is defined without regard to a particular RPC transport. An Upper-Layer Binding (ULB) specification provides guidance that helps a ULP interoperate correctly and efficiently over a particular transport. For RPC-over-RDMA version 2, a ULB may provide:

- o A taxonomy of XDR data items that are eligible for DDP
- o Constraints on which upper-layer procedures a sender may reduce, and on how many chunks may appear in a single RPC message
- o A method enabling a Requester to determine the maximum size of the reply Payload stream for all procedures in the ULP
- o An rpcbind port assignment for the RPC Program and Version when operating on the particular transport

Each RPC Program and Version tuple that operates on RPC-over-RDMA version 2 needs to have a ULB specification.

### **[A.1.](#) DDP-Eligibility**

A ULB designates specific XDR data items as eligible for DDP. As a sender constructs an RPC-over-RDMA message, it can remove DDP-eligible data items from the Payload stream so that the RDMA provider can place them 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:

- o The XDR data item is frequently sent or received, and its size is often much larger than typical inline thresholds.
- o If the XDR data item is a result, its maximum size must be predictable in advance by the Requester.
- o 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.
- o 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.
- o The XDR data item itself does not contain DDP-eligible data items.

In addition to defining the set of data items that are DDP-eligible, a ULB may 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 limit the number of chunks that a Requester can provide for a particular upper-layer procedure.

Senders never reduce data items that are not DDP-eligible. Such data items can, however, be part of a Special Format payload.

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:

- o A Requester reduces a non-DDP-eligible argument data item. The Responder reports the violation as described in [Section 6.3.3](#).



- o A Responder reduces a non-DDP-eligible result data item. The Requester terminates the pending RPC transaction and reports an appropriate permanent error to the RPC consumer.
- o A Responder does not reduce a DDP-eligible result data item into an available Write chunk. The Requester terminates the pending RPC transaction and reports an appropriate permanent error to the RPC consumer.

#### **A.2. Maximum Reply Size**

When expecting small and moderately-sized Replies, a Requester should rely on Message Continuation rather than provision a Reply chunk. For each ULP procedure where there is no clear Reply size maximum and the maximum can be substantial, the ULB should specify a dependable means for determining the maximum Reply size.

#### **A.3. Reverse-Direction Operation**

The direction of operation does not preclude the need for DDP-eligibility statements.

Reverse-direction operation occurs on an already-established connection. Specification of RPC binding parameters is usually not necessary in this case.

Other considerations may apply when distinct RPC Programs share an RPC-over-RDMA transport connection concurrently.

#### **A.4. Additional Considerations**

There may be other details provided in a ULB.

- o A ULB may recommend inline threshold values or other transport-related parameters for RPC-over-RDMA version 2 connections bearing that ULP.
- o A ULP may provide a means to communicate transport-related parameters between peers.
- o Multiple ULPs may share a single RPC-over-RDMA version 2 connection when their ULBs allow the use of RPC-over-RDMA version 2 and the rpcbind port assignments for those protocols permit connection sharing. In this case, the same transport parameters (such as inline threshold) apply to all ULPs 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.

#### [A.5.](#) ULP Extensions

An RPC Program and Version tuple may be extensible. For instance, the RPC version number may not reflect a ULP minor versioning scheme, or the ULP may allow the specification of additional features after the publication of the original RPC Program specification. ULPs 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.

[ cel: The final sentence is unclear, and may be inaccurate. I believe I copied this section directly from [RFC 8166](#). Is there more to be said, now that we have some experience? ]

#### [Appendix B.](#) Extending RPC-over-RDMA Version 2

This Appendix is not addressed to protocol implementers, but rather to authors of documents that intend to extend the protocol specified in the current document.

RPC-over-RDMA version 2 extensibility facilitates limited extensions to the base protocol presented in the current document so that new optional capabilities can be introduced without a protocol version change, while maintaining robust interoperability with existing RPC-over-RDMA version 2 implementations. It allows extensions to be defined, including the definition of new protocol elements, without requiring modification or recompilation of the XDR for the base protocol.

Standards Track documents may introduce extensions to the base RPC-over-RDMA version 2 protocol in two ways:

- o They may introduce new OPTIONAL transport header types. [Appendix B.2](#) covers such transport header types.
- o They may define new OPTIONAL transport properties. [Appendix B.4](#) describes such transport properties.

These documents may also add the following sorts of ancillary protocol elements to the protocol to support the addition of new transport properties and header types:

- o They may create new error codes, as described in [Appendix B.5](#).





- o They may define new flags to use within the `rdma_flags` field, as discussed in [Appendix B.3](#).

New capabilities can be proposed and developed independently of each other. Implementers can choose among them, making it straightforward to create and document experimental features and then bring them through the standards process.

### **[B.1.](#) Documentation Requirements**

As described earlier, a Standards Track document introduces a set of new protocol elements. Together these elements are considered an OPTIONAL feature. Each implementation is either aware of all the protocol elements introduced by that feature or is aware of none of them.

Documents specifying extensions to RPC-over-RDMA version 2 should contain:

- o An explanation of the purpose and use of each new protocol element.
- o An XDR description including all of the new protocol elements, and a script to extract it.
- o A discussion of interactions with other extensions. This discussion includes requirements for other OPTIONAL features to be present, or that a particular level of support for an OPTIONAL facility is required.

Implementers combine the XDR descriptions of the new features they intend to use with the XDR description of the base protocol in the current document. This combination is necessary to create a valid XDR input file because extensions are free to use XDR types defined in the base protocol, and later extensions may use types defined by earlier extensions.

The XDR description for the RPC-over-RDMA version 2 base protocol combined with that for any selected extensions should provide a human-readable and compilable definition of the extended protocol.

### **[B.2.](#) Adding New Header Types to RPC-over-RDMA Version 2**

New transport header types are defined similar to Sections [6.3.1](#) through 6.3.4. In particular, what is needed is:

- o A description of the function and use of the new header type.



- o A complete XDR description of the new header type.
- o A description of how receivers report errors, including mechanisms for reporting errors outside the available choices already available in the base protocol or other extensions.
- o An indication of whether a Payload stream must be present, and a description of its contents and how receivers use such Payload streams to reconstruct RPC messages.

There needs to be additional documentation that is made necessary due to the OPTIONAL status of new transport header types:

- o The document should discuss constraints on support for the new header types. For example, if support for one header type is implied or foreclosed by another one, this needs to be documented.
- o The document should describe the preferred method by which a sender determines whether its peer supports a particular header type. It is always possible to send a test invocation of a particular header type to see if support is available. However, when more efficient means are available (e.g., the value of a transport property), this should be noted.

### **B.3. Adding New Header Flags to the Protocol**

New flag bits are to be defined similarly to Sections [6.2.2.1](#) and [6.2.2.2](#). Each new flag definition should include:

- o An XDR description of the new flag.
- o A description of the function and use of the new flag.
- o An indication for which header types the flag value is meaningful, and for which header types it is an error to set the flag or to leave it unset.
- o A means to determine whether peers are prepared to receive transport headers with the new flag set.

There needs to be additional documentation that is made necessary due to the OPTIONAL status of new flags:

- o The document should discuss constraints on support for the new flags. For example, if support for one flag is implied or foreclosed by another one, this needs to be documented.



#### **B.4. Adding New Transport properties to the Protocol**

A Standards Track document defining a new transport property should include the following information paralleling that provided in this document for the transport properties defined herein:

- o The `rpcrdma2_propid` value identifying the new property.
- o The XDR typedef specifying the structure of its property value.
- o A description of the new property.
- o An explanation of how the receiver can use this information.
- o The default value if a peer never receives the new property.

There is no requirement that `propid` assignments occur in a continuous range of values. Implementations should not rely on all such values being small integers.

Before the defining Standards Track document is published, the `nfsv4` Working Group should select a unique `propid` value, and ensure that:

- o `rpcrdma2_propid` values specified in the document do not conflict with those currently assigned or in use by other pending working group documents defining transport properties.
- o `rpcrdma2_propid` values specified in the document do not conflict with the range reserved for experimental use, as defined in [Section 8.2](#).

[ cel: There is no [section 8.2](#). ]

[ cel: Should we request the creation of an IANA registry for `propid` values? ].

When a Standards Track document proposes additional transport properties, reviewers should deal with possible security issues exposed by those new transport properties.

#### **B.5. Adding New Error Codes to the Protocol**

The same Standards Track document that defines a new header type may introduce new error codes used to support it. A Standards Track document may similarly define new error codes that an existing header type can return.



For error codes that do not require the return of additional information, a peer can use the existing RDMA\_ERR2 header type to report the new error. The sender sets the new error code as the value of `rdma_err` with the result that the default switch arm of the `rpcrdma2_error` (i.e., `void`) is selected.

For error codes that do require the return of related information together with the error, a new header type should be defined that returns the error together with the related information. The sender of a new header type needs to be prepared to accept header types necessary to report associated errors.

### **Appendix C. Differences from RPC-over-RDMA Version 1**

The goal of RPC-over-RDMA version 2 is to relieve certain constraints that have become evident in RPC-over-RDMA version 1 with deployment experience:

- o RPC-over-RDMA version 1 has been challenging to update to address shortcomings or improve data transfer efficiency.
- o The average size of NFSv4 COMPOUNDS is significantly greater than NFSv3 requests, requiring the use of Long messages for frequent operations.
- o Reply size estimation is awkward more often than expected.

This section details specific changes in RPC-over-RDMA version 2 that address these constraints directly.

#### **C.1. Changes to the XDR Definition**

Several XDR structural changes enable within-version protocol extensibility.

[RFC8166] defines the RPC-over-RDMA version 1 transport header as a single XDR object, with an RPC message potentially following it. In RPC-over-RDMA version 2, there are separate XDR definitions of the transport header prefix (see [Section 6.2.2](#)), which specifies the transport header type to be used, and the transport header itself (defined within one of the subsections of [Section 6.3](#)). This construction is similar to an RPC message, which consists of an RPC header (defined in [\[RFC5531\]](#)) followed by a message defined by an Upper-Layer Protocol.

As a new version of the RPC-over-RDMA transport protocol, RPC-over-RDMA version 2 exists within the versioning rules defined in [\[RFC8166\]](#). In particular, it maintains the first four words of the





protocol header, as specified in [Section 4.2 of \[RFC8166\]](#), even though, as explained in [Section 6.2.1](#) of the current document, the XDR definition of those words is structured differently.

Although each of the first four fields retains its semantic function, there are differences in interpretation:

- o The first word of the header, the `rdma_xid` field, retains the format and function that it had in RPC-over-RDMA version 1. Because RPC-over-RDMA version 2 messages can convey non-RPC messages, a receiver should not use the contents of this field without consideration of the protocol version and header type.
- o The second word of the header, the `rdma_vers` field, retains the format and function that it had in RPC-over-RDMA version 1. To clearly distinguish version 1 and version 2 messages, senders need to fill in the correct version (fixed after version negotiation). Receivers should check that the content of the `rdma_vers` is correct before using the content of any other header field.
- o The third word of the header, the `rdma_credit` field, retains the size and general purpose that it had in RPC-over-RDMA version 1. However, RPC-over-RDMA version 2 divides this field into two 16-bit subfields. See [Section 4.2.1](#) for further details.
- o The fourth word of the header, previously the union discriminator field `rdma_proc`, retains its format and general function even though the set of valid values has changed. Within RPC-over-RDMA version 2, this word is the `rdma_htype` field of the structure `rdma_start`. The value of this field is now an unsigned 32-bit integer rather than an enum type, to facilitate header type extension.

Beyond conforming to the restrictions specified in [\[RFC8166\]](#), RPC-over-RDMA version 2 tightly limits the scope of the changes made to ensure interoperability. Version 2 retains all existing transport header types used in version 1, as defined in [\[RFC8166\]](#). And, it expresses chunks in the same format and uses them the same way.

## **[C.2.](#) Transport Properties**

RPC-over-RDMA version 2 provides a mechanism for exchanging an implementation's operational properties. The purpose of this exchange is to help endpoints improve the efficiency of data transfer by exploiting the characteristics of both peers rather than falling back on the lowest common denominator default settings. A full discussion of transport properties appears in [Section 5](#).



### **C.3. Credit Management Changes**

RPC-over-RDMA transports employ credit-based flow control to ensure that a Requester does not emit more RDMA Sends than the Responder is prepared to receive.

[Section 3.3.1 of \[RFC8166\]](#) explains the operation of RPC-over-RDMA version 1 credit management in detail. In that design, each RDMA Send from a Requester contains an RPC Call with a credit request, and each RDMA Send from a Responder contains an RPC Reply with a credit grant. The credit grant implies that enough Receives have been posted on the Responder to handle the credit grant minus the number of pending RPC transactions (the number of remaining Receive buffers might be zero).

Each RPC Reply acts as an implicit ACK for a previous RPC Call from the Requester. Without an RPC Reply message, the Requester has no way to know that the Responder is ready for subsequent RPC Calls.

Because version 1 embeds credit management in each message, there is a strict one-to-one ratio between RDMA Send and RPC message. There are interesting use cases that might be enabled if this relationship were more flexible:

- o RPC-over-RDMA operations that do not carry an RPC message, e.g., control plane operations.
- o A single RDMA Send that conveys more than one RPC message, e.g., for interrupt mitigation.
- o An RPC message that requires several sequential RDMA Sends, e.g., to reduce the use of explicit RDMA operations for moderate-sized RPC messages.
- o An RPC transaction that requires multiple exchanges or an odd number of RPC-over-RDMA operations to complete.

RPC-over-RDMA version 2 provides a more sophisticated credit accounting mechanism to address these shortcomings. [Section 4.2.1](#) explains the new mechanism in detail.

### **C.4. Inline Threshold Changes**

An "inline threshold" value is the largest message size (in octets) that can be conveyed on an RDMA connection using only RDMA Send and Receive. Each connection has two inline threshold values: one for messages flowing from client-to-server (referred to as the "client-to-server inline threshold") and one for messages flowing from



server-to-client (referred to as the "server-to-client inline threshold").

A connection's inline thresholds determine, among other things, when RDMA Read or Write operations are required because an RPC message cannot be conveyed via a single RDMA Send and Receive pair. When an RPC message does not contain DDP-eligible data items, a Requester can prepare a Special Format Call or Reply to convey the whole RPC message using RDMA Read or Write operations.

RDMA Read and Write operations require that data payloads reside in memory registered with the local RNIC. When an RPC completes, that memory is invalidated to fence it from the Responder. Memory registration and invalidation typically have a latency cost that is insignificant compared to data handling costs.

When a data payload is small, however, the cost of registering and invalidating memory where the payload resides becomes a significant part of total RPC latency. Therefore the most efficient operation of an RPC-over-RDMA transport occurs when the peers use explicit RDMA Read and Write operations for large payloads but avoid those operations for small payloads.

When the authors of [\[RFC8166\]](#) first conceived RPC-over-RDMA version 1, the average size of RPC messages that did not involve a significant data payload was under 500 bytes. A 1024-byte inline threshold adequately minimized the frequency of inefficient Long messages.

With NFS version 4 [\[RFC7530\]](#), the increased size of NFS COMPOUND operations resulted in RPC messages that are, on average, larger than previous versions of NFS. With a 1024-byte inline threshold, frequent operations such as GETATTR and LOOKUP require RDMA Read or Write operations, reducing the efficiency of data transport.

To reduce the frequency of Special Format messages, RPC-over-RDMA version 2 increases the default size of inline thresholds. This change also increases the maximum size of reverse-direction RPC messages.

### **[C.5.](#) Message Continuation Changes**

In addition to a larger default inline threshold, RPC-over-RDMA version 2 introduces Message Continuation. Message Continuation is a mechanism that enables the transmission of a data payload using more than one RDMA Send. The purpose of Message Continuation is to provide relief in several essential cases:



- o If a Requester finds that it is inefficient to convey a moderately-sized data payload using Read chunks, the Requester can use Message Continuation to send the RPC Call.
- o If a Requester has provided insufficient Reply chunk space for a Responder to send an RPC Reply, the Responder can use Message Continuation to send the RPC Reply.
- o If a sender has to convey a sizeable non-RPC data payload (e.g., a large transport property), the sender can use Message Continuation to avoid having to register memory.

### **C.6. Host Authentication Changes**

For the general operation of NFS on open networks, we eventually intend to rely on RPC-on-TLS [[I-D.ietf-nfsv4-rpc-tls](#)] to provide cryptographic authentication of the two ends of each connection. In turn, this can improve the trustworthiness of AUTH\_SYS-style user identities that flow on TCP, which are not cryptographically protected. We do not have a similar solution for RPC-over-RDMA, however.

Here, the RDMA transport layer already provides a strong guarantee of message integrity. On some network fabrics, IPsec or TLS can protect the privacy of in-transit data. However, this is not the case for all fabrics (e.g., InfiniBand [[IBA](#)]).

Thus, RPC-over-RDMA version 2 introduces a mechanism for authenticating connection peers (see [Section 5.2.6](#)). And like GSS channel binding, there is also a way to determine when the use of host authentication is unnecessary.

### **C.7. Support for Remote Invalidation**

When an RDMA consumer uses FRWR or Memory Windows to register memory, that memory may be invalidated remotely [[RFC5040](#)]. These mechanisms are available when a Requester's RNIC supports MEM\_MGT\_EXTENSIONS.

For this discussion, there are two classes of STags. Dynamically-registered STags appear in a single RPC, then are invalidated. Persistently-registered STags survive longer than one RPC. They may persist for the life of an RPC-over-RDMA connection or even longer.

An RPC-over-RDMA Requester can provide more than one STag in a transport header. It may provide a combination of dynamically- and persistently-registered STags in one RPC message, or any combination of these in a series of RPCs on the same connection. Only





dynamically-registered STags using Memory Windows or FRWR may be invalidated remotely.

There is no transport-level mechanism by which a Responder can determine how a Requester-provided STag was registered, nor whether it is eligible to be invalidated remotely. A Requester that mixes persistently- and dynamically-registered STags in one RPC, or mixes them across RPCs on the same connection, must, therefore, indicate which STag the Responder may invalidate remotely via a mechanism provided in the Upper-Layer Protocol. RPC-over-RDMA version 2 provides such a mechanism.

A sender uses the RDMA Send With Invalidate operation to invalidate an STag on the remote peer. It is available only when both peers support MEM\_MGT\_EXTENSIONS (can send and process an IETH).

Existing RPC-over-RDMA transport protocol specifications [[RFC8166](#)] [[RFC8167](#)] do not forbid direct data placement in the reverse direction. Moreover, there is currently no Upper-Layer Protocol that makes data items in reverse direction operations eligible for direct data placement.

When chunks are present in a reverse direction RPC request, Remote Invalidation enables the Responder to trigger invalidation of a Requester's STags as part of sending an RPC Reply, the same way as is done in the forward direction.

However, in the reverse direction, the server acts as the Requester, and the client is the Responder. The server's RNIC, therefore, must support receiving an IETH, and the server must have registered its STags with an appropriate registration mechanism.

### **[C.8.](#) Error Reporting Changes**

RPC-over-RDMA version 2 expands the repertoire of errors that connection peers may report to each other. The goals of this expansion are:

- o To fill in details of peer recovery actions.
- o To enable retrying certain conditions caused by mis-estimation of the maximum reply size.
- o To minimize the likelihood of a Requester waiting forever for a Reply when there are communications problems that prevent the Responder from sending it.



## Acknowledgments

The authors gratefully acknowledge the work of Brent Callaghan and Tom Talpey on the original RPC-over-RDMA version 1 specification ([RFC 5666](#)). The authors also wish to thank Bill Baker, Greg Marsden, and Matt Benjamin for their support of this work.

The XDR extraction conventions were first described by the authors of the NFS version 4.1 XDR specification [[RFC5662](#)]. Herbert van den Bergh suggested the replacement sed script used in this document.

Special thanks go to Transport Area Director Magnus Westerlund, NFSV4 Working Group Chairs Spencer Shepler and Brian Pawlowski, and NFSV4 Working Group Secretary Thomas Haynes for their support.

## Authors' Addresses

Charles Lever (editor)  
Oracle Corporation  
United States of America

Email: [chuck.lever@oracle.com](mailto:chuck.lever@oracle.com)

David Noveck  
NetApp  
1601 Trapelo Road  
Waltham, MA 02451  
United States of America

Phone: +1 781 572 8038  
Email: [davenoveck@gmail.com](mailto:davenoveck@gmail.com)

