Network File System Version 4                                   C. Lever
Internet-Draft                                                    Oracle
Intended status: Informational                              May 12, 2016
Expires: November 13, 2016


            **RPC-over-RDMA Version One Implementation Experience**
            **draft-ietf-nfsv4-rfc5666-implementation-experience-03**

Abstract

   This document details experiences and challenges implementing the
   RPC-over-RDMA Version One protocol.  Specification changes are
   recommended to address avoidable interoperability failures.

Status of This Memo

Copyright Notice

Table of Contents

## 1.  Introduction

This document summarizes implementation experience with the RPC-over-RDMA Version One protocol [RFC5666], and proposes improvements to the protocol specification based on implementer experience, frequently-asked questions, and interviews with a co-author of RFC 5666.

### 1.1.  Purpose Of This Document

A key contribution of this document is to highlight areas of RFC 5666 where independent good faith readings could result in distinct implementations that do not interoperate with each other.  Correcting these specification issues is critical: fresh implementations of RPC-over-RDMA Version One continue to arise.

Recommendations are limited to the following areas:

o  Repairing specification ambiguities

o  Codifying successful implementation practices and conventions

o  Clarifying the role of Upper Layer Binding specifications

o  Exploring protocol enhancements that might be added while allowing extant implementations to interoperate with enhanced implementations

### 1.2.  Updating RFC 5666

During IETF 92, several alternatives for updating RFC 5666 were discussed with the RFC Editor and with the assembled members of the nfsv4 Working Group.  Among them were:

o  Filing individual errata for each issue

o  Introducing a new RFC that updates but does not obsolete RFC 5666, but makes no change to the protocol

o  Introducing an RFC 5666bis that replaces and thus obsoletes RFC 5666, but makes no change to the protocol

o  Introducing a new RFC that specifies RPC-over-RDMA Version Two

An additional possibility which is sometimes chosen by other Working Groups would be to update RFC 5666 as it transitions from Proposed Standard to Draft Standard.

There was general agreement during the meeting regarding the need to
update and obsolete RFC 5666 while retaining a high degree of
interoperability with current RPC-over-RDMA Version One
implementations.  This approach would avoid changes to on-the-wire
behavior without burdening implementers, who could continue to
reference a single specification of the protocol.  In addition, this
alternative extends the life of current interoperable RPC-over-RDMA
Version One implementations in the field.

Subsequent discussion within the nfsv4 Working Group has focused on
resolving specification ambiguities that make the construction of
interoperable implementations unduly difficult.  Subsequent Versions
of RPC-over-RDMA, where deeper changes can be made and new
functionality introduced, remain a possibility.

## 1.3.  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
[RFC2119].

## 2.  RPC-Over-RDMA Essentials

The following sections summarize the state of affairs defined in RFC
5666.  This is a distillation of text from RFC 5666, dialog with a
co-author of RFC 5666, and implementer experience.  The XDR
definitions are copied from RFC 5666 Section 4.3.

## 2.1.  Arguments And Results

Like a local function call, every Remote Procedure Call (RPC)
operation has a set of one or more "arguments" and a set of one or
more "results."  The calling context is not allowed to proceed until
the function's results are available.  Unlike a local function call,
the called function is executed remotely rather than in the local
application's context.

A client endpoint, or "requester", serializes an RPC call's arguments
into a byte stream using XDR [RFC4506].  This "XDR stream" is
conveyed to a server endpoint via an RPC call message (sometimes
referred to as an "RPC request").

The server endpoint, or "responder", deserializes the arguments and
processes the requested operation.  It then serializes the
operation's results into another XDR stream.  This stream is conveyed
back to the client endpoint via an RPC reply message.  The client
deserializes the results and allows the original caller to proceed.

The remainder of this document assumes a working knowledge of the RPC
protocol [RFC5531] and especially XDR [RFC4506].

## 2.2.  Remote Direct Memory Access

RPC messages may be very large.  For example, NFS READ and WRITE
operations are often 100KB or larger.

An RPC client system can be made more efficient if RPC messages are
transferred by a third party such as intelligent network interface
hardware.  Remote Direct Memory Access (RDMA) and Direct Data
Placement (DDP) enables offloading data movement to avoid the
negative performance effects of using traditional host CPU-based
network operations to move bulk data.

RFC 5666 describes how to use only the Send, Receive, RDMA Read, and
RDMA Write operations described in [RFC5040] and [RFC5041] to move
RPC calls and replies between requesters and responders.

### 2.2.1.  Direct Data Placement

RFC 5666 makes an important distinction between RDMA and Direct Data
Placement (DDP).

Very often, RPC implementations copy the contents of RPC messages
into a buffer before being sent.  A good RPC implementation may be
able to send bulk data without having to copy 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.

In RFC 5666, "RDMA" refers to the physical mechanism an RDMA
transport utilizes when moving data.  Though it may not be optimal,
before an RDMA transfer, the sender may still copy data into place.
After an RDMA transfer, the receiver may copy that data again to its
final destination.

RFC 5666 uses the term "direct data placement" to refer to an
optimization that makes it unnecessary for a host CPU to copy data to
be transferred.  RPC-over-RDMA Version One utilizes RDMA Read and
Write operations to enable DDP.  Not every RDMA-based transfer in
RPC-over-RDMA Version One is DDP, however.

2.2.2.  Channel Operation

   A Send operation initiates the transfer of a message from a local
   endpoint to a remote endpoint, similar to a datagram send operation.

   The remote endpoint pre-posts Receive operations to catch incoming
   messages.  Send operations are flow-controlled to prevent overrunning
   receive resources.  To reduce the amount of memory that must remain
   pinned awaiting incoming messages, receive buffers are limited in
   size and number.

   This transfer mode is utilized to convey size-limited RPC operations,
   and advertisements of buffer coordinates for explicit RDMA data
   transfer.  Buffers involved in Send and Receive operations are
   usually left unexposed.

2.2.3.  Explicit RDMA Operation

   A local endpoint tags memory areas to be involved in RDMA, exposes
   the areas, then advertises the coordinates of those areas to a remote
   endpoint via a Send operation.

   The remote endpoint transfers data into or out of those areas using
   RDMA Read and Write operations.  The remote registers large sink
   buffers as needed, and invalidates them when data transfer is
   complete.

   Finally the remote endpoint signals that its work is done, and the
   local endpoint ensures remote access to the memory areas is no longer
   allowed.

   This transfer mode can be utilized to convey large whole RPC
   messages, although typically only one data item within a message is
   large.  Explicit RDMA is most often used to move large argument or
   result data items directly into place.  The remaining portions of the
   message are conveyed via a channel operation.

2.3.  Transfer Models

   A "transfer model" describes which endpoint is responsible for
   performing RDMA Read and Write operations.  The opposite endpoint
   must expose part or all of its memory, and advertise the coordinates
   of that memory.

### 2.3.1.  Read-Read

   Requesters expose their memory to the responder, and the responder
   exposes its memory to requesters.  The responder employs RDMA Read
   operations to convey RPC arguments or whole RPC calls.  Requesters
   employ RDMA Read operations to convey RPC results or whole RPC
   relies.

   Although this model is specified in RFC 5666, no current RPC-over-
   RDMA Version One implementation uses the Read-Read transfer model.

### 2.3.2.  Write-Write

   Requesters expose their memory to the responder, and the responder
   exposes its memory to requesters.  Requesters employ RDMA Write
   operations to convey RPC arguments or whole RPC calls.  The responder
   employs RDMA Write operations to convey RPC results or whole RPC
   relies.

   The Write-Write transfer model is not considered in RFC 5666.

### 2.3.3.  Read-Write

   Requesters expose their memory to the responder, but the responder
   does not expose its memory.  The responder employs RDMA Read
   operations to convey RPC arguments or whole RPC calls.  The responder
   employs RDMA Write operations to convey RPC results or whole RPC
   relies.

   This model is specified in RFC 5666.  All known RPC-over-RDMA Version
   One implementations employ this model.  For clarity, the remainder of
   this document considers only the Read-Write transfer model.

### 2.4.  Upper Layer Binding Specifications

   RFC 5666 provides a framework for conveying RPC requests and replies
   on RDMA transports.  By itself this is insufficient to enable an RPC
   program, referred to as an "Upper Layer Protocol" or ULP, to operate
   over an RDMA transport.

   Arguments and results come in different sizes and have different
   serialization requirements, all depending on the Upper Layer
   Protocol.  Some arguments and results are appropriate for Direct Data
   Placement, while others are not.  Thus RFC 5666 requires additional
   separate specifications that describe how each ULP may use explicit
   RDMA operations to enable Direct Data Placement.  The set of
   requirements for a ULP to use an RDMA transport is known as an "Upper
   Layer Binding" specification, or ULB.

   An Upper Layer Binding states which specific individual RPC arguments
   and results are permitted to be transferred via RDMA Read and Write
   for the purpose of Direct Data Placement.  RFC 5666 refers to such
   arguments and results as "RDMA-eligible."  RDMA-eligibility
   restrictions do not apply when a whole RPC call or reply is
   transferred via an RDMA Read or Write (long messages).

   A ULB is required for each RPC program and version tuple that may
   operate on an RDMA transport.  A ULB may be part of another
   specification, or it may be a stand-alone document, similar to
   [RFC5667].

## 2.5.  On-The-Wire Protocol

### 2.5.1.  Inline Operation

   Each RPC call or reply message conveyed on an RDMA transport starts
   with an RPC-over-RDMA header, which is encoded into an XDR stream.  A
   requester uses a Send operation to convey the RPC-over-RDMA header to
   a responder.  A responder does likewise to convey RPC replies back to
   a requester.  All message contents sent via Send, including the RPC-
   over-RDMA header and possibly an RPC message proper, are referred to
   as "inline content."

   The RPC-over-RDMA header starts with three uint32 fields:

<CODE BEGINS>

```
    struct rdma_msg {
            uint32    rdma_xid;     /* Mirrors the RPC header xid */
            uint32    rdma_vers;    /* Version of this protocol */
            uint32    rdma_credit;  /* Buffers requested/granted */
            rdma_body rdma_body;
    };
```

<CODE ENDS>

   Following these three fields is a union:

```
<CODE BEGINS>

    enum rdma_proc {
            RDMA_MSG=0,    /* An RPC call or reply msg */
            RDMA_NOMSG=1, /* An RPC call or reply msg -
                            separate body */
            . . .
            RDMA_ERROR=4  /* An RPC RDMA encoding error */
    };

    union rdma_body switch (rdma_proc proc) {
            case RDMA_MSG:
              rpc_rdma_header rdma_msg;
            case RDMA_NOMSG:
              rpc_rdma_header_nomsg rdma_nomsg;
            . . .
            case RDMA_ERROR:
              rpc_rdma_error rdma_error;
    };

    struct rpc_rdma_header {
            struct xdr_read_list    *rdma_reads;
            struct xdr_write_list   *rdma_writes;
            struct xdr_write_chunk *rdma_reply;
            /* rpc body follows */
    };

    struct rpc_rdma_header_nomsg {
            struct xdr_read_list    *rdma_reads;
            struct xdr_write_list   *rdma_writes;
            struct xdr_write_chunk *rdma_reply;
    };

<CODE ENDS>
```

In either the RDMA_MSG or RDMA_NOMSG case, the RPC-over-RDMA header
may advertise memory coordinates to be used for RDMA data transfers
associated with this RPC.

The difference between these two cases is whether or not the
traditional RPC header itself is included in this Send operation
(RDMA_MSG), or not (RDMA_NOMSG).  In the former case, the RPC header
follows immediately after the rdma_reply field.  In the latter case,
the RPC header is transfered via another mechanism (typically a
separate RDMA Read operation).

A requester may use either type of message to send an RPC call
message, depending on the requirements of the RPC call message being

conveyed.  A responder may use RDMA_NOMSG only when the requester
provides a Reply chunk (see Section 4.3).  A responder is free to use
RDMA_MSG instead in that case, depending on the requirements of the
RPC reply message.

### 2.5.2.  RDMA Segment

An "RDMA segment", or just "segment", is a part of the RPC-over-RDMA
header that contains the co-ordinates of a contiguous memory region
that is to be conveyed via an RDMA Read or RDMA Write operation.

The region defined by a segment is advertised in an RPC-over-RDMA
header to enable the receiving endpoint to drive subsequent RDMA
access of the data in that memory region.  The RPC-over-RDMA Version
One XDR represents an RDMA segment with the xdr_rdma_segment struct:

<CODE BEGINS>

```
   struct xdr_rdma_segment {
           uint32 handle;
           uint32 length;
           uint64 offset;
   };
```

<CODE ENDS>

See [RFC5040] for a discussion of what the content of these fields
means.

### 2.5.3.  Chunk

A "chunk" refers to XDR stream data that is moved via an RDMA Read or
Write operation.  Chunk data is removed from the sender's XDR stream,
is transferred by a separate RDMA operation, and is re-inserted into
the XDR stream by the receiver.

Each chunk is made up of one or more segments.  Each segment
represents a single contiguous piece of that chunk.

If a chunk is to move a whole counted array, the count of array
elements is left in the XDR stream, while the array elements appear
in the chunk.  Individual array elements appear in the chunk in their
entirety.

## 2.5.4.  Read Chunk

   One or more "read chunks" are used to advertise the coordinates of
   XDR stream data to be transferred via RDMA Read operations.

   Each read chunk is represented by the xdr_read_chunk struct:

   <CODE BEGINS>

```
   struct xdr_read_chunk {
           uint32 position;
           struct xdr_rdma_segment target;
   };
```

   <CODE ENDS>

   RFC 5666 defines a read chunk as one RDMA segment with a Position
   field.  The Position field indicates the location in the XDR stream
   where the transferred object's data would appear if it was not being
   moved in a chunk.

   The transferred data might be contained in one contiguous memory
   region.  That data can be represented by a single read chunk
   (segment).

   Alternately, the transferred data might reside in multiple
   discontiguous memory regions.  The data is represented by a list of
   read chunks (segments).  The Position field in each segment in this
   list contains the same value.

   The receiver reconstructs the transferred data by concatenating the
   contents of each segment in list order into the receiver's XDR
   stream.  The first segment begins at the XDR position in the Position
   field, and subsequent segments are concatenated afterwards until
   there are no more segments left at that XDR Position.  This enables
   gathering data from multiple buffers on the sender.

## 2.5.5.  Write Chunk

   A "Write chunk" advertises the coordinates of XDR stream data to be
   transferred via RDMA Write operations.

   A write chunk is represented by the xdr_write_chunk struct:

    <CODE BEGINS>

       struct xdr_write_chunk {
               struct xdr_rdma_segment target<>;
       };

    <CODE ENDS>

    The sender fills each segment on the receiver, in array order, with
    the transferred data.  This enables scattering data into multiple
    buffers on the receiver.

    Typically the exact size of the data cannot be predicted before the
    responder has formed its reply.  Thus the requester must provide
    enough space in the write chunk for the largest result the responder
    might generate for this RPC operation.  The responder updates the
    size field of each segment in the Write chunk when it returns the
    Write list to the requester via a matching RPC reply message.  If a
    segment is not used, the responder sets the segment size field to
    zero.

    Because the requester must pre-allocate the area in which the
    responder writes the result before the responder has formed the
    reply, giving a position and size to the data, the requester cannot
    know the XDR stream position of the result data.  Thus write chunks
    do not have a Position field.

## 2.5.6.  Read List

    Each RPC-over-RDMA Version One call has one "Read list," provided by
    the requester.  A requester provides the locations of RDMA-eligible
    argument data via read chunks.  Via a Position Zero read chunk, a
    requester may provide an entire RPC request message as a chunk in
    this list.

    A Read list is represented by the xdr_read_list struct:

    <CODE BEGINS>

       struct xdr_read_list {
               struct xdr_read_chunk entry;
               struct xdr_read_list  *next;
       };

    <CODE ENDS>

RFC 5666 does not restrict the order of read chunks in the Read list, but read chunks with the same value in their Position fields are ordered (see above).

The Read list may be empty if the RPC call has no argument data that is RDMA-eligible and the Position Zero Read chunk is not being used.

## 2.5.7.  Write List

Each RPC-over-RDMA Version One call has one "Write list," provided by the requester.  A requester provides write chunks in this list as receptacles for RDMA-eligible result data.

A Write list is represented by the xdr_write_list struct:

<CODE BEGINS>

```
   struct xdr_write_list {
           struct xdr_write_chunk entry;
           struct xdr_write_list  *next;
   };
```

<CODE ENDS>

Note that this looks similar to a Read list, but because an xdr_write_chunk is an array and not an RDMA segment, the two data structures are not the same.

The Write list may be empty if there is no RDMA-eligible result data to return.

The requester provides as many Write chunks as the Upper Layer Binding allows for the particular operation.  The responder fills in each Write chunk with an RDMA-eligible result until the Write list is exhausted or there are no more RDMA-eligible results.

## 2.5.8.  Position Zero Read Chunk

A requester may use a "Position Zero read chunk" to convey part or all of an entire RPC call, rather than including the RPC call message inline.  A Position Zero read chunk is necessary if the RPC call message is too large to fit inline.  RFC 5666 Section 5.1 defines the operation of a "Position Zero read chunk."

To support gathering a large RPC call message from multiple locations on the requester, a Position Zero read chunk may be comprised of more than one xdr_read_chunk.  Each read chunk that belongs to the Position Zero read chunk has the value zero in its Position field.

**2.5.9**.  **Reply Chunk**

   Each RPC-over-RDMA Version One call may have one "Reply chunk,"
   provided by the requester.  A responder may convey an entire RPC
   reply message in this chunk.

   A Reply chunk is a write chunk, thus it is an array of one or more
   RDMA segments.  This enables a requester to control where the
   responder scatters the parts of an RPC reply message.  In current
   implementations, there is only one RDMA segment in a Reply chunk.

   A requester provides the Reply chunk whenever it predicts the
   responder's reply might not fit inline.  It may choose to provide the
   Reply chunk even when the responder can return only a small reply.  A
   responder may use a "Reply chunk" to convey most or all of an entire
   RPC reply, rather than including the RPC reply message inline.

**3**.  **Specification Issues**

**3.1**.  **Extensibility Considerations**

   RPC-based protocols are defined solely by their XDR definitions.
   They are independent of the transport mechanism used to convey base
   RPC messages.  Protocols defined this way often have signifcant
   extensibility restrictions placed on them.

   Not all restrictions on RPC-based Upper Layer Protocols may be
   appropriate for an RPC transport protocol, however.  TCP [RFC0793],
   for example, is an RPC transport protocol that has been extended many
   times independently of the RPC and XDR standards.

   RPC-over-RDMA is partially specified by XDR, and it provides a
   version field in its headers.  However, it is distinct from other
   RPC- and XDR-based protocols in some key ways:

   o  Although it uses XDR encoding, RPC-over-RDMA is not an RPC
      program, nor is it an Upper Layer Protocol

   o  XDR objects in RPC-over-RDMA headers exist near to but outside the
      embrace of an RPC message

   o  RPC-over-RDMA relies on a more sophisticated set of base transport
      operations than traditional socket-based transports

   o  The RDMA operations generated by verbs are not part of any XDR
      definition; however interoperability depends on RPC-over-RDMA
      implementations using these verbs in a particular way

There are still reasonable restrictions, of course, that are
necessary to maintain interoperability within a single Version of
RPC-over-RDMA.  But they are left largely unstated in RFC 5666.

### 3.1.1.  Recommendations

RFC 5666bis should not alter the basic physical operations that are
in use by current implementations.  It should not alter the on-the-
wire appearance of RPC-over-RDMA Version One headers, and never by an
explicit RDMA operations.

Although it is implied, RFC 5666bis should state explicitly that all
items in an RPC-over-RDMA Version One header must be conveyed via
Send and Receive operations (ie, none of these items is ever RDMA-
eligible).

RFC 5666bis should discuss when a Version bump is required.  Any
significant changes to the way RDMA operations are used should
require a Version bump, for instance.  Certain limited XDR changes
might be allowed, as long as the standards-specified set of wire
behaviors remains intact.

### 3.2.  XDR Clarifications

Even seasoned NFS/RDMA implementers have had difficulty agreeing on
precisely what a "chunk" is, and had challenges distinguishing the
structure of the Read list from structure of the Write list.

On occasion, the text of RFC 5666 uses the term "chunk" to represent
either read chunks or write chunks, even though these are different
data types and have different semantics.

For example, RFC 5666 Section 3.4 uses the term "chunk list entry"
even though the discussion is referring to an array element (a
segment).  It implies all chunk types have a Position field, even
though only read chunks have this field.

Near the end of Section 3.4, it says:

   Therefore, read chunks are encoded into a read chunk list as a
   single array, with each entry tagged by its (known) size and its
   argument's or result's position in the XDR stream.

The Read list is not an XDR array, it is always an XDR list.  A Write
chunk is an XDR array.

RFC 5666 Section 3.7, third paragraph uses the terms "chunked
element" and "chunk segment."  Neither term is defined or used

anywhere else.  The fourth paragraph refers to a "sequence of chunks" but likely means a sequence of RDMA segments.

The XDR definition for a read chunk is an RDMA segment with a position field.  RFC 5666 Section 3.4 states that multiple xdr_read_chunk structs can make up a single RPC argument if they share the same Position in the XDR stream.  Some implementations depend on using multiple RDMA segments in the same XDR Position, particularly for sending Position Zero read chunks efficiently by gathering an RPC call message from multiple discontiguous memory locations.  Other implementations do not support sending or receiving multiple Read chunks with the same Position.

An Upper Layer Binding may limit the number of Read list entries allowed for a particular operation.  In that case, the Upper Layer Binding is not restricting the total number of read chunks in the list, but rather the number of distinct Positions that appear in the list.

RFC 5666 does not restrict the boundaries of a chunk other than to imply that a chunk's starting position and its length is a multiple of an XDR data unit.  However, implementations have observed a practical restriction to facilitate straightforward integration of RDMA support into existing XDR infrastructure: A chunk containing RDMA-eligible data must be encoded or decoded as a single XDR object.

In addition, Upper Layer Bindings make RDMA-eligibility statements about specific arguments and results (or portions thereof which still are whole XDR objects).  The implication is that chunks contain only whole XDR objects, even though RFC 5666 is not explicit about this.

A Position Zero read chunk typically contains an entire RPC request message, and a Reply chunk contains an entire RPC reply message.  These are exceptions to the above restriction.

The Write list is especially confusing because it is a list of arrays of RDMA segments, rather than a simple list of xdr_read_chunk objects.  What is referred to as a Read list entry often means one xdr_read_chunk, or one segment.  That segment can be either a portion of or a whole XDR object.  A Write list entry is an array, and always represents a single XDR object in its entirety.

An Upper Layer Binding may limit the number of chunks in a Write list allowed for a particular operation.  That strictly limits the number of Write list entries.

Not having a firm one-to-one correspondence between read chunks and XDR objects is sometimes awkward.  The two chunk types should be more

symmetrical to avoid confusion, although that might be difficult to
pull off without altering the RPC-over-RDMA Version One XDR
definition.  As we will see later, the XDR roundup rules also appear
to apply asymmetrically to read chunks and write chunks.

Implementers have been aided by the ASCII art block comments in the
Linux kernel in net/sunrpc/xprtrdma/rpcrdma.c, excerpted here.  This
diagram shows exactly how the Read list and Write list are
constructed in an XDR stream.

<CODE BEGINS>

```
  /*
   * Encoding key for single-list chunks
   *         (HLOO = Handle32 Length32 Offset64):
   *
   *  Read chunklist (a linked list):
   *   N elements, position P (same P for all chunks of same arg!):
   *    1 - PHLOO - 1 - PHLOO - ... - 1 - PHLOO - 0
   *
   *  Write chunklist (a list of (one) counted array):
   *   N elements:
   *    1 - N - HLOO - HLOO - ... - HLOO - 0
   *
   *  Reply chunk (a counted array):
   *   N elements:
   *    1 - N - HLOO - HLOO - ... - HLOO
   */
```

<CODE ENDS>

### 3.2.1.  Recommendations

To aid in reader understanding, RFC 5666bis should expand the
glossary that explains and distinguishes the various elements in the
protocol.  Upper Layer Binding specifications refer to these terms.
RFC 5666bis should utilize and capitalize these glossary terms
consistently.

RFC 5666bis should introduce additional diagrams that supplement the
XDR definition in RFC 5666 Section 4.3.  RFC 5666bis should explain
the structure of the XDR and how it is used.  RFC 5666bis should
contain an explicit but brief rationalization for the structural
differences between the Read list and the Write list.

RFC 5666bis should explicitly restrict chunks containing RDMA-
eligible data so that a chunk represents exactly a single XDR object
in its entirety.

RFC 5666bis should use a consistent naming convention for all XDR definitions.  For example, all structures and union names should use an "rpcrdma1_" prefix.

To address conflation of a read chunk that is a single xdr_read_chunk and a read chunk that is a list of xdr_read_chunk elements with identical Position field values, the following specification changes should be made:

o  The XDR definition should rename the xdr_read_chunk struct as rpcrdma1_read_segment.

o  RFC 5666bis should redefine a "read chunk" as an ordered list of one or more rpcrdma1_read_segment structs that have identical Position values.

o  RFC 5666bis should redefine the "Read list" as a list of zero or more read chunks, expressed as an ordered list of rpcrdma1_read_segment structs whose Position value may vary. Segment positions in the list are non-descending.

With these changes, there would no longer be a simple XDR object that explicitly represents a read chunk, but a read chunk and a write chunk are now equivalent objects that both map to a whole XDR object. All discussion should take care to use the terms "segment" and "read segment" instead of the term "read chunk" where appropriate.

As a clean up, RFC 5666bis should remove the rpc_rdma_header_nomsg struct, and use the rpc_rdma_header struct in its place.  Since rpc_rdma_header does not comprise the entire RPC-over-RDMA header, it should be renamed rpcrdma1_chunks to avoid confusion.

XDR definitions should be enclosed in CODE BEGINS and CODE ENDS delimiters.  An appropriate copyright block should accompany the XDR definitions in RFC 5666bis.  An XDR extraction shell script should be provided in the text.

See Section 10 for a full listing of the proposed XDR definitions.

### 3.3.  Additional XDR Issues

### 3.3.1.  Mechanical Issues

There are some mechanical problems with the XDR language definition of RPC-over-RDMA Version One provided in Section 4.3 of [RFC5666]:

o  No copyright boilerplate is provided

o  An extraction script is not provided, and there is no escape
   sequence around the code

o  There is at least one XDR definition error that prevents the
   extracted XDR from compiling

### 3.3.2.  XDR Definition Recursiveness

The usual practice when defining an XDR-based protocol is that there
is one encompassing data type that represents one message in the
protocol.

This is not true for RPC-over-RDMA.  The header is defined by one
data type (struct rdma_msg) but the RPC message payload is not
formally represented in the XDR definition in Section 4.3.  The
presence or absence of the RPC message payload is indicated by the
message type, and the body of that payload is noted only with a code
comment.

### 3.3.3.  Recommendations

The XDR presented in RFC5666bis should correct the deficiencies
described above.

To correct the lack of formal recursiveness issue without forcing an
on-the-wire behavior change, RFC5666bis should place the RPC-over-
RDMA header and the RPC message payload in separate XDR streams.

### 3.4.  The Position Zero Read Chunk

RFC 5666 Section 5.1 defines the operation of the Position Zero read
chunk.  A requester uses the Position Zero read chunk in place of
inline content.  A requester is required to use the Position Zero
read chunk when the total size of an RPC call message exceeds the
size of the responder's receive buffers, and RDMA-eligible data has
already been removed from the message.

RFC 5666 Section 3.4 says:

   Semantically speaking, the protocol has no restriction regarding
   data types that may or may not be represented by a read or write
   chunk.  In practice however, efficiency considerations lead to the
   conclusion that certain data types are not generally "chunkable".
   Typically, only those opaque and aggregate data types that may
   attain substantial size are considered to be eligible.  With
   today's hardware, this size may be a kilobyte or more.  However,
   any object MAY be chosen for chunking in any given message.

    The eligibility of XDR data items to be candidates for being moved
    as data chunks (as opposed to being marshaled inline) is not
    specified by the RPC-over-RDMA protocol.  Chunk eligibility
    criteria MUST be determined by each upper-layer in order to
    provide for an interoperable specification.

The intention of this text is to spell out that RDMA-eligibility
applies only to individual XDR data objects in the Upper Layer
Protocol.  RDMA-eligibility criteria are specified within a separate
specification, rather than in RFC 5666.

The Position Zero read chunk is an exception to both of these
guidelines.  The Position Zero read chunk, by virtue of the fact that
it typically conveys an entire RPC call message, may contain multiple
arguments, independent of whether any particular argument in the RPC
call is RDMA-eligible.

Unlike the read chunks described in the RFC 5666 excerpt above, the
content of a Position Zero read chunk is typically marshaled and
copied on both ends of the transport, so it cannot benefit from
Direct Data Placement.  In particular, the Position Zero read chunk
is not for conveying performance critical Upper Layer operations.

Thus the requirements for what may or may not appear in the Position
Zero read chunk are indeed specified by RFC 5666, in contradiction to
the second paragraph quoted above.  Upper Layer Binding
specifications may have something to say about what may appear in the
Position Zero read chunk, but the basic definition of Position Zero
should be made clear in RFC 5666bis as distinct from a read chunk
whose Position field is non-zero.

Because a read chunk is defined as one RDMA segment with a Position
field, at least one implementation allows only a single chunk segment
in Position zero read chunks.  This is a problem for two reasons:

o  Some RPCs are constructed in multiple non-contiguous buffers.
   Allowing only one read segment in Position Zero would mean a
   single large contiguous buffer would be have to be allocated and
   registered, and then the components of the XDR stream would have
   to be copied into that buffer.

o  Some requesters might not be able to register memory regions
   larger than the platform's physical page size.  Allowing only one
   read segment in Position Zero would limit the maximum size of RPC-
   over-RDMA messages to a single page.  Allowing multiple read
   segments means the message size can be as large as the maximum
   number of read chunks that can be sent in an RPC-over-RDMA header.

RFC 5666 does not limit the number of read segments in a read chunk, nor does it limit the number of chunks that can appear in the Read list.  The Position Zero read chunk, despite its name, is not limited to a single xdr_read_chunk.

### 3.4.1.  Recommendations

RFC 5666bis should state that the guidelines in RFC 5666 Section 3.4 apply only to RDMA_MSG type calls.  When the Position Zero read chunk is introduced in RFC 5666 Section 5.1, enumerate the differences between it and the read chunks previously described in RFC 5666 Section 3.4.

RFC 5666bis should describe what restrictions an Upper Layer Binding may make on Position Zero read chunks.

### 3.5.  RDMA_NOMSG Call Messages

The second paragraph of RFC 5667 Section 4 says, in reference to NFSv2 and NFSv3 WRITE and SYMLINK operations:

> . . . a single RDMA Read list entry MAY be posted by the client to supply the opaque file data for a WRITE request or the pathname for a SYMLINK request.  The server MUST ignore any Read list for other NFS procedures, as well as additional Read list entries beyond the first in the list.

However, large non-write NFS operations are conveyed via a Read list containing at least a Position Zero read chunk.  Strictly speaking, the above requirement means large non-write NFS operations may never be conveyed because the responder MUST ignore the read chunk in such requests.

It is likely the authors of RFC 5667 intended this limit to apply only to RDMA_MSG type calls.  If that is true, however, an NFS implementation could legally skirt the stated restriction simply by using an RDMA_NOMSG type call that conveys both a Position Zero and a non-zero position read chunk to send a non-write NFS operation.

Unless either RFC 5666 or the protocol's Upper Layer Binding explicitly prohibits it, allowing a read chunk in a non-zero Position in an RDMA_NOMSG type call means an Upper Layer Protocol may ignore Binding requirements like the above.

Typically there is no benefit to allowing multiple read chunks for RDMA_NOMSG type calls.  Any non-zero Position read segments can always be conveyed as part of the Position Zero read chunk.

However, there is a class of RPC operations where RDMA_NOMSG with
multiple read chunks is useful: when the body of an RPC call message
is larger than the inline buffer size, even after RDMA-eligible
argument data has been moved to read chunks.

A similar discussion applies to RDMA_NOMSG replies with large reply
bodies and RDMA-eligible result data.  Such replies would use both
the Write list and the Reply chunk simultaneously.  However, write
chunks do not have Position fields.

### 3.5.1.  Recommendations

RFC 5666bis should continue to allow RDMA_NOMSG type calls with
additional read chunks.  The rules about RDMA-eligibility in RFC
5666bis should discuss when the use of this construction is
beneficial, and when it should be avoided.

Authors of Upper Layer Bindings should be warned about ignoring these
cases.  RPC 5666bis should provide a default behavior that applies
when Upper Layer Bindings omit this discussion.

### 3.6.  RDMA_MSG Call with Position Zero Read Chunk

The first item in the header of both RPC calls and RPC replies is the
XID field [RFC5531].  RFC 5666 Section 4.1 says:

   A header of message type RDMA_MSG or RDMA_MSGP MUST be followed by
   the RPC call or RPC reply message body, beginning with the XID.

This is a strong implication that the RPC header in an RDMA_MSG type
message starts at XDR position zero.  Assume for a moment that, by
definition, the RPC header in an RPC-over-RDMA XDR stream starts at
XDR position zero.

An RDMA_MSG type call message includes the RPC header and zero or
more read chunks.  Recall the definition of a read chunk as a list of
read segments whose Position field contains the same value.  The
value of the Position field determines where the read chunk appears
in the XDR stream that comprises an RPC call message.

A Position Zero read chunk, therefore, starts at XDR position zero,
just like RPC header does.  In an RDMA_NOMSG type call message, which
does not include an RPC header, a Position Zero read chunk conveys
the RPC header.

There is no prohibition in RFC 5666 against an RDMA_MSG type call
messsage with a Position Zero read chunk.  However, it's not clear
how a responder should interpret such a message.  RFC 5666 requires

the RPC header to start at XDR position zero, but there is a Position
Zero read chunk, which also starts at XDR position zero.

### 3.6.1.  Recommendations

RPC 5666bis should clearly define what is meant by an XDR stream.
RFC 5666bis should state that the value in the xdr_read_chunk
"position" field is measured relative to the start of the RPC header,
which is the first byte of the header's XID field.

RFC 5666bis should prohibit requesters from providing a Position Zero
read chunk in RDMA_MSG type calls.  Likewise, RFC 5666bis should
prohibit responders from utilizing a Reply chunk in RDMA_MSG type
replies.

The diagrams in RFC 5666 Section 3.8 which number chunks starting
with 1 should be revised.  Readers confuse this number with an XDR
position.

### 3.7.  Padding Inline Content After A Chunk

To help clarify the discussion in this section, the term "read chunk"
here always means the new definition where one or more read segments
that have identical values in their Position fields represents
exactly one RDMA-eligible XDR object.

A read chunk conveys a large argument payload via one or more RDMA
transfers.  For instance, the data payload of an NFS WRITE operation
may be be transferred using a read chunk [RFC5667].

NFSv3 WRITE operations place the data payload at the end of an RPC
call message [RFC1813].  The RPC call's XDR stream starts in an
inline buffer, continues in a read chunk, then ends there.

An NFSv4 WRITE operation may occur as a middle operation in an NFSv4
COMPOUND [RFC5661].  The read chunk containing the data payload
argument of the WRITE operation might finish before the RPC call's
XDR stream does.  In this case, the RPC call's XDR stream starts in
an inline buffer, continues in the Read list, then finishes back in
the inline buffer.

The length of a chunk is the sum of the lengths of the segments that
make up that chunk.  The data payload in a chunk may have a length
that is not evenly divisible by four.  One or more of the segments
may have an unaligned length.

RFC 5666 Section 3.7 describes how to manage XDR roundup in a read
chunk when its length is not XDR-aligned.  The sender is not required

to send the extra pad bytes at the end of a chunk because a) the
receiver never references their content, therefore it is wasteful to
transmit them, and b) each read chunk has a Position field and length
that determines exactly where that chunk starts and ends in the XDR
stream.

A question arises, however, when considering where the next XDR
object after a read chunk should appear.  XDR requires each object to
begin on 4-byte alignment [RFC4506].  But a read chunk's XDR padding
is optional (see above) and thus may not appear in the chunk as
actual zero bytes.

The next read chunk's position field determines where it is placed in
the XDR stream, so in that case there is no ambiguity.  Inline
content following a read chunk does not have a Position field to
guide the receiver in the reassembly of the XDR stream, however.

Paragraph 4 of RFC 5666 Section 3.7 says:

   When roundup is present at the end of a sequence of chunks, the
   length of the sequence will terminate it at a non-4-byte XDR
   position.  When the receiver proceeds to decode the remaining part
   of the XDR stream, it inspects the XDR position indicated by the
   next chunk.  Because this position will not match (else roundup
   would not have occurred), the receiver decoding will fall back to
   inspecting the remaining inline portion.  If in turn, no data
   remains to be decoded from the inline portion, then the receiver
   MUST conclude that roundup is present, and therefore it advances
   the XDR decode position to that indicated by the next chunk (if
   any).  In this way, roundup is passed without ever actually
   transferring additional XDR bytes.

This paragraph adequately describes XDR padding requirements when a
read chunk is followed by another read chunk.  But it leaves unspoken
any requirements for XDR padding and alignment when a read chunk is
followed in the XDR stream by more inline content.

Applying the rules of XDR, the XDR pad for the read chunk must not
appear in the inline content, even if it was also not included in the
chunk itself.  This is because the inline content that preceded the
read chunk will have been padded to 4-byte alignment.  The next
position in the inline buffer is already on a 4-byte boundary, thus
no padding is necessary.

### 3.7.1. Recommendations

State the above requirement in RFC 5666bis in its equivalent of RFC 5666 Section 3.7.  When a responder forms a reply, the same restriction applies to inline content interleaved with write chunks.

Because all XDR objects must start on an XDR alignment boundary, all read and write chunks and all inline XDR objects in any XDR stream must start on an XDR alignment boundary.  This has implications for the values allowed in read chunk Position fields, for how XDR roundup works for chunks, and for how XDR objects are placed in inline buffers.  XDR alignment in inline buffers is always relative to Position Zero (or, where the RPC header starts).

### 3.8. Write Chunk XDR Roundup

The final paragraph of RFC 5666 Section 3.7 says:

> For RDMA Write Chunks, a simpler encoding method applies.  Again, roundup bytes are not transferred, instead the chunk length sent to the receiver in the reply is simply increased to include any roundup.

A responder should avoid writing XDR pad bytes, as the requester's upper layer does not reference them, though the language does not fully prohibit writing these bytes.  A requester always provides the extra space for XDR padding anyway.

A problem arises if the data item written into a Write chunk is shorter than the chunk and requires an XDR pad.  A responder may write the XDR pad past the end of the data content.  For a short directly-placed write, the pad bytes are then exposed in the RPC consumer's data buffer.

In addition, for the chunk length to be rounded up as described, the requester must provide adequate extra space in the chunk for the XDR pad.  A requester can provide space for the XDR pad using one of two approaches:

1.  It can extend the last segment in the chunk.

2.  It can provide another segment after the segments that receive RDMA Write payloads.

Case 1 is adequate when there is no danger that the responder's RDMA Write operations will overwrite existing data on the requester in memory following the advertised receive buffers.

In Direct Data Placement scenarios, an extra segment must be provided separately to avoid overwriting existing data that follows the sink buffer (case 2).  Thus, an extra registration is needed for just a handful of bytes that may not be written by the responder, and are ignored by the requester.  Even so, this does not force the responder to direct the XDR pad bytes into this extra segment, should the data item in that chunk be shorter than the chunk itself.

Registering the extra buffer is a needless cost.  It would be more efficient if the XDR pad at the end of a write chunk were treated the same as it is for Read chunks.  Because RPC result data must begin on an XDR alignment boundary, the result following the write chunk in the reply's XDR stream must begin on an XDR alignment boundary.  There is no need for a XDR pad to be present for the receiver to re-assemble the RPC reply's XDR stream properly.

One responder implementation requires the requester to provide the extra buffer space in the Write chunk, but does not write to it.  This follows the letter of the last paragraph of Section 3.7 of [RFC5666].

Another responder implementation does not rely on having the extra space (operation proceeds if it is missing) but when the extra space is present, this responder does write zeroes to it.  While the intention of Section 3.7 is that the responder does not write the pad, it is not strictly forbidden.

Client implementations all appear to provide the extra buffer space needed to accommodate the XDR pad.  However, one implementation does not register this extra buffer, since the responder is not expected to write into it, while another implementation does.

These implementations may not be 100% interoperable.  The language of Section 3.7 of [RFC5666] appears to allow all of this behavior (in particular, it does not prohibit a responder from writing the XDR pad using RFC2119-style keywords, and does not require that requesters register the extra space to accommodate the XDR pad).

Note that because the Reply chunk is a write chunk, these roundup rules also apply to it.

## 3.8.1.  Recommendations

The current specification allows XDR pad bytes to leak into user buffers, and none of the current implementations prevent this leak.  There may be room to adjust the protocol specification independently of current implementation behavior.

RFC 5666bis should explicitly discuss the requirements around write
chunk roundup separately from the discussion of read chunk roundup.

Explicit RFC2119-style interoperability requirements should be
provided for write chunks.  Responders MUST NOT write XDR pad bytes
at the end of a Write chunk.

Allocating and registering extra space for XDR pad bytes that are
never written is wasteful.  RFC 5666bis should forbid it.  Responders
should not expect requesters to provide space for XDR pad bytes.

## 3.9.  Write List Error Cases

RFC 5666 Section 3.6 says:

    When a write chunk list is provided for the results of the RPC
    call, the RPC server MUST provide any corresponding data via RDMA
    Write to the memory referenced in the chunk list entries.

This requires the responder to use the Write list when it is
provided.  Another way to say it is a responder is not permitted to
return bulk data inline or in the Reply chunk when the requester has
provided a Write list.

This requirement is less clear when it comes to situations where a
particular RPC reply is allowed to use a provided Write list, but
does not have a bulk data payload to return.  For example, RFC 5667
Section 4 permits requester to provide a Write list for NFS READ
operations.  However, NFSv3 READ operations have a union reply
[RFC1813]:

```
<CODE BEGINS>

    struct READ3resok {
            post_op_attr file_attributes;
            count3      count;
            bool        eof;
            opaque      data<>;
    };

    struct READ3resfail {
            post_op_attr file_attributes;
    };

    union READ3res switch (nfsstat3 status) {
    case NFS3_OK:
            READ3resok resok;
    default:
            READ3resfail resfail;
    };

<CODE ENDS>
```

When an NFS READ operation fails, no data is returned.  The arm of
the READ3res union which is used when a read error occurs does not
have a bulk data argument.

RFC 5666 does not prescribe how a responder should behave when RDMA-
eligible result data for which the Write list is provided does not
appear in the reply.  RFC 5666 Section 3.4 says:

   Individual write chunk list elements MAY thereby result in being
   partially or fully filled, or in fact not being filled at all.
   Unused write chunks, or unused bytes in write chunk buffer lists,
   are not returned as results, and their memory is returned to the
   upper layer as part of RPC completion.

It also says:

   The RPC reply conveys this by returning the write chunk list to
   the client with the lengths rewritten to match the actual
   transfer.

The disposition of the advertised write buffers is therefore clear.
The requirements for how the Write list must appear in an RPC reply
are somewhat less than clear.

Here we are concerned with two cases:

o  When a result consumes fewer RDMA segments than the requester
   provided in the Write chunk for that result, what values are
   provided for the chunk's segment count and the lengths of the
   unused segments

o  When a result is not used (say, the reply uses the arm of an XDR
   union that does not contain the result corresponding to a Write
   chunk provided for that result), what values are provided for the
   chunk's segment count and the lengths of the unused segments

The language above suggests the proper value for the Write chunk's
segment count is always the same value that the requester sent, even
when the chunk is not used in the reply.  The proper value for the
length of an unused segment in a Write chunk is always zero.

Inspection of one existing server implementation shows that when an
NFS READ operation fails, the returned Write list contains one entry:
a chunk array containing zero elements.  Another server
implementation returns the original Write list chunk in this case.

In either case, requesters appear to ignore the Write list when no
bulk data payload is expected.  Thus it appears that, currently,
responders may put whatever they like in the Write list.

Current NFSv4 client implementations behave like legacy NFS
implementations in the sense that each READ COMPOUND requests only
one contiguous data payload that is never larger than the rsize
setting of the mount.  However it is legal for an NFSv4 COMPOUND to
contain more than one READ operation.  Each READ request in a
COMPOUND may have an RDMA-eligible result in the COMPOUND reply.

In general, a complex Upper Layer Binding may wish to return more
than one RDMA-eligible result in a single RPC reply.  Depending on
the RPC program, there may be nested or sequential switched unions in
the reply.  There is no Position field in the segments making up a
Write chunk, so both sender and receiver must be careful about how
the reply message is re-assembled.

It should always be unambiguous which Write chunk matches with which
result.  To ensure interoperability, the responder associates the
first RDMA-eligible result with the first chunk in the Write list,
and so on, until either results or Write chunks are exhausted.  The
receiver makes the same associations while parsing the XDR stream of
the reply.  It should be the responsibility of the Upper Layer
Binding to avoid ambiguous situations by appropriately restricting
RDMA-eligible data items.

Remember that a responder MUST use the Write list if the requester
provided it and the responder has RDMA-eligible result data.  If the
requester has not provided enough Write chunks in the Write list, the
responder may have to use a long message as well, depending on the
remaining size of the RPC reply.

### 3.9.1.  Recommendations

RFC 5666bis should explicitly discuss responder behavior when an RPC
reply does not need to use a Write list entry provided by a
requester.  This is generic behavior, independent of any Upper Layer
Binding.  The explanation can be partially or wholly copied from RFC
5667 Section 5's discussion of NFSv4 COMPOUND.

A number of places in RFC 5666 Section 3.6 hint at how a responder
behaves when it is to return data that does not use every byte of
every provided Write chunk segment.  RFC 5666bis should state
specific requirements about how a responder should form the Write
list in RPC replies, and/or it should explicitly require requesters
to ignore the Write list in these cases.  RFC 5666bis should require
that the responder not alter the count of segments in the Write
chunk.  One or more explicit examples should be provided in RFC
5666bis.

RFC 5666bis should provide clear instructions on how Upper Layer
Bindings are to be written to take care of switched unions.

### 4.  Operational Considerations

### 4.1.  Computing Request Buffer Requirements

The size maximum of a single Send operation includes both the RPC-
over-RDMA header and the RPC header.  Combined, those two headers
must not exceed the size of one receive buffer.

Senders often construct the RPC-over-RDMA header and the RPC call or
reply message in separate buffers, then combine them via an iovec
into a single Send.  This does not mean each element of that iovec
can be as large as the inline threshold.

An HCA or RNIC may have a small limit on the size of a registered
memory region.  In that case, RDMA-eligible data may be comprised of
many chunk segments.

This has implications for the size of the Read and Write lists, which
take up a variable amount of space in the RPC-over-RDMA header.  The
sum of the size of the RPC-over-RDMA header, including the Read and

   Write lists, and the size of the RPC header must not exceed the
   inline threshold.  This limits the maximum Upper Layer payload size.

## 4.1.1.  Recommendations

   RFC 5666bis should provide implementation guidance on how the inline
   threshold (the maximum send size) is computed.

## 4.2.  Default Inline Buffer Size

   Section 6 of RFC 5666 specifies an out-of-band protocol that allows
   an endpoint to discover a peer endpoint's receive buffer size, to
   avoid overrunning the receiving buffer, causing a connection loss.

   Not all RPC-over-RDMA Version One implementations also implement CCP,
   as it is optional.  Given the importance of knowing the receiving
   end's receive buffer size, there should be some way that a sender can
   choose a size that is guaranteed to work with no CCP interaction.

   RFC 5666 Section 6.1 describes a 1KB receive buffer limit for the
   first operation on a connection with an unfamiliar responder.  In the
   absence of CCP, the client cannot discover that responder's true
   limit without risking the loss of the transport connection.

## 4.2.1.  Recommendations

   RFC 5666bis should specify a fixed send/receive buffer size as part
   of the RPC-over-RDMA Version One protocol, to use when CCP is not
   available.  For example, the following could be added to the RFC
   5666bis equivalent of RFC 5666 Section 6.1: "In the absence of CCP,
   requesters and responders MUST assume 1KB receive buffers for all
   Send operations."

   It should be safe for Upper Layer Binding specifications to provide a
   different default inline threshold.  Care must be taken when an
   endpoint is associated with multiple RPC programs that have different
   default inline thresholds.

## 4.3.  When To Use Reply Chunks

   RFC 5666 Section 3.6 says:

      When a write chunk list is provided for the results of the RPC
      call, the RPC server MUST provide any corresponding data via RDMA
      Write to the memory referenced in the chunk list entries.

It is not clear whether the authors of RFC 5666 intended the above
requirement to apply only to the Write list, or to both the Write
list and to the Reply chunk, which is not a list.

Implementation experience has shown that setting up an explicit RDMA
operation to move a few hundred bytes of data is inefficient,
especially if there is no DDP opportunity.  Channel operations are
nearly always the best choice when handling a small RPC reply.

o  To reduce memory registration and invalidation costs, a requester
   might prefer to provide a Reply chunk only when a reply could be
   larger than the inline threshold.  To make that judgement,
   however, a requester must know the size of the responder's send
   buffers, which might be smaller than its own receive buffers.

o  Even when a requester has provided a Reply chunk, to reduce round
   trip costs, a responder might prefer to RDMA Write a Reply chunk
   only when a reply is actually larger than the inline threshold.
   To make that judgement, however, the responder must know the size
   of the requester's receive buffers, which might be smaller than
   its send buffers.

If a requester does not provide a Reply chunk when one is needed, the
responder must reply with ERR_CHUNK (see RFC 5666, Section 4.2).  The
requester simply has to send the request again, this time with a
Reply chunk.  However ERR_CHUNK a generic failure mode.  The
requester may have some difficulty identifying the problem as a
missing Reply chunk.

To maintain 100% interoperability, a requester should always provide
a Reply chunk, and the responder should always use it.  However, as
noted, this is likely to be inefficient.

### 4.3.1.  Recommendations

To provide a stronger guarantee of interoperation while ensuring
efficient operation, RFC 5666bis should explicitly specify when a
requester must offer a Reply chunk, and when the responder must use
an offered Reply chunk.

Mandating a default buffer size would allow both sides to choose
correctly with an in-advance CCP exchange.

### 4.4.  Computing Credit Values

The third paragraph of Section 3.3 of RFC 5666 leaves open the exact
mechanism of how often the requested and granted credit limits are
supposed to be adjusted.  A reader might believe that these values

are adjusted whenever an RPC call or reply is received, to reflect
the number of posted receive buffers on each side.

Although adjustments are allowed by RFC 5666 due to changing
availability of resources on either endpoint, current implementations
use a fixed value.  Advertised credit values are always the sum of
the in-process receive buffers and the ready-to-use receive buffers.

### 4.4.1.  Recommendations

RFC 5666bis should clarify the method used to calculate these values.
RFC 5666bis might also discuss how flow control is impacted when a
server endpoint utilizes a shared receive queue.

### 4.5.  Race Windows

The second paragraph of RFC 5666 Section 3.3 says:

> Additionally, for protocol correctness, the RPC server must always
> be able to reply to client requests, whether or not new buffers
> have been posted to accept future receives.

It is true that the RPC server must always be able to reply, and that
therefore the client must provide an adequate number of receive
buffers.  The dependent clause "whether or not new buffers have been
posted to accept future receives" is problematic, however.

It's not clear whether this clause refers to a server leaving even a
small window where the sum of posted and in-process receive buffers
is less than the credit limit; or refers to a client leaving a window
where the sum of posted and in-process receive buffers is less than
its advertised credit limit.  In either case, such a window could
result in lost messages or be catastrophic for the transport
connection.

### 4.5.1.  Recommendations

Clarify or remove the dependent clause in the section in RFC 5666bis
that is equivalent to RFC 5666 Section 3.3.

### 4.6.  Detection Of Unsupported Protocol Versions

Section 4.2 of [RFC5666] is explicit about how a responder must
handle RPC-over-RDMA messages that carry an unrecognized RPC-over-
RDMA protocol version:

> When a peer receives an RPC RDMA message, it MUST perform the
> following basic validity checks on the header and chunk contents.

If such errors are detected in the request, an RDMA_ERROR reply
MUST be generated.

When the peer detects an RPC-over-RDMA header version that it does
not support (currently this document defines only version 1), it
replies with an error code of ERR_VERS, and provides the low and
high inclusive version numbers it does, in fact, support.  The
version number in this reply MUST be any value otherwise valid at
the receiver.

However, one widely deployed RPC-over-RDMA Version One server
implementation is known to discard requests that do not contain the
value one (1) in their rdma_vers field.  This server implementation
does not reply with RDMA_ERROR / RDMA_ERR_VERS in this case.

Without a proper protocol version detection mechanism, it is not
possible for RPC-over-RDMA Version One implementations to
interoperate with implementations that support newer protocol
versions.

### 4.6.1.  Recommendations

RPC-over-RDMA Version One implementations that discard non-Version
One requests without an error response are considered non-compliant
with [RFC5666].  No changes to the specification are needed.

### 5.  Pre-requisites For NFSv4

### 5.1.  Bi-directional Operation

NFSv4.1 moves the backchannel onto the same transport as forward
requests [RFC5661].  Typically RPC client endpoints do not expect to
receive RPC call messages.  To support NFSv4.1 callback operations,
client and server implementations must be updated to support bi-
directional operation.

Because of RDMA's requirement to pre-post unadvertised receive
buffers, special considerations are needed for bi-directional
operation.  Conventions have been provided to allow bi-direction,
with a limit on backchannel message size, such that no changes to the
RPC-over-RDMA Version One protocol are needed
[I-D.ietf-nfsv4-rpcrdma-bidirection].

### 5.1.1.  Recommendations

RFC 5666bis should cite or include the bulk of
[I-D.ietf-nfsv4-rpcrdma-bidirection].

6.  Considerations For Upper Layer Binding Specifications

   RFC 5666 requires a Binding specification for any RPC program wanting
   to use RPC-over-RDMA.  The requirement appears in two separate
   places: The fourth paragraph of Section 3.4, and the final paragraph
   of Section 3.6.  As critical as it is to have a Binding
   specification, RFC 5666's text regarding these specifications is
   sparse and not easy to find.

6.1.  Organization Of Binding Specification Requirements

   Throughout RPC 5666, various Binding requirements appear, such as:

      The mapping of write chunk list entries to procedure arguments
      MUST be determined for each protocol.

   A similar specific requirement for read list entries is missing.

   Usually these statements are followed by a reference to the NFS
   Binding specification [RFC5667].  There is no summary of these
   requirements, however.

   Additional advice appears in the middle of Section 3.4:

      It is NOT RECOMMENDED that upper-layer RPC client protocol
      specifications omit write chunk lists for eligible replies,

   This requirement, being in the middle of a dense paragraph about how
   write lists are formed, is easy for an author of Upper Layer Binding
   specifications to miss.

6.1.1.  Recommendations

   RFC 5666bis should summarize explicit generic requirements for the
   contents of an Upper Layer Binding specification in one separate
   section, perhaps in an Appendix.  In particular, move the third,
   fourth and fifth paragraph of RFC 5666 Section 3.4 to this new
   section discussing Binding specification requirements.

6.2.  RDMA-Eligibility

   Any RPC message that fits in an inline buffer is conveyed via a Send
   operation.  Any RPC message that is too large to fit in an inline
   buffer is conveyed by transferring the whole RPC message via an RDMA
   Read (i.e., a Position Zero Read chunk) or an RDMA Write (i.e., a
   Reply chunk).

RPC-over-RDMA also allows a mixture of these two mechanisms, where argument or result data is removed from the XDR stream and conveyed via a separate RDMA transfer.  The receiving end assembles the disparate buffers into a single XDR stream that represents the whole RPC message.

RFC 5666 uses the term "RDMA eligibility" to mean that an particular argument or result object is allowed to be moved as a separate chunk for the purpose of Direct Data Placement.  The RPC program's Upper Layer Binding makes eligibility statements permitting particular RPC argument or result objects to be directly placed.

The third paragraph of Section 3.4 states that any XDR object MAY be RDMA-eligible in any given message, but that:

   Typically, only those opaque and aggregate data types that may
   attain substantial size are considered to be eligible.

Any large XDR object that can benefit from Direct Data Placement is a good candidate for being moved in a chunk.  When data alignment matters, or when the NFS stack on either end of the connection does not need to manipulate the transferred data, the Upper Layer Binding should make that object eligible for Direct Data Placement.

Section 3.4 is specifically not discussing long messages, where a whole RPC message is moved via RDMA.  When an RPC message is too large to fit inline, even after RDMA-eligible arguments or results are removed, the message is always moved via a long message.  All arguments or results in the message are moved via RDMA in this case.

For instance, an NFSv3 READDIR result can be large.  However, an NFS server assembles this result in place, encoding each section individually.  The NFS client must perform the converse actions.  Though there is potentially a large amount of data, the benefit of direct data placement is lost because of the need for both host CPUs to be involved in marshaling and decoding.

Thus the NFSv3 Upper Layer Binding [RFC5667] does not make any part of an NFSv3 READDIR reply RDMA-eligible.  However, any NFS READDIR reply that is larger than an inline buffer is still moved via RDMA (a Reply chunk, in this case).

## 6.2.1.  Recommendations

RFC 5666bis should define the term "Upper Layer Binding", and explain what it specifies.  RFC 5666bis should explicitly require an Upper Layer Binding for every RPC program that may operate on RDMA

transports.  Separate bindings may be required for different versions
of that program.

The term "RDMA eligibility" should be retired.  It is easy to confuse
the use of RDMA for Direct Data Placement with the use of RDMA in
long messages.  Instead, RFC 5666bis should use a more precise term
such as DDP-eligibility, which should be clearly defined before it is
used.

RFC 5666bis should provide generic guidance about what makes an XDR
object or data type eligible for Direct Data Placement.  RFC 5666bis
should state that the DDP-eligibility of any XDR object not mentioned
explicitly in an Upper Layer Binding is "not eligible."

RFC 5666bis should note that Position Zero read chunks and Reply
chunks may contain any and all argument and results regardless of
their DDP-eligibility.  RFC 5666bis should remind authors of Upper
Layer Bindings that the Reply chunk and Position Zero read chunks are
expressly not for performance-critical Upper Layer operations.

It is the responsibility of the Upper Layer Binding to specify RDMA-
eligibity rules so that if an RDMA-eligible XDR object is embedded
within another, only one of these two objects is to be represented by
a chunk.  This ensures that the mapping from XDR position to the XDR
object represented is unambiguous.

## 6.3.  Inline Threshold Requirements

An RPC-over-RDMA connection has two connection parameters that affect
the operation of Upper Layer Protocols: The credit limit, which is
how many outstanding RPCs are allowed on that connection; and the
inline threshold, which is the maximum payload size of an RDMA Send
on that connection.  All ULPs sharing a connection also share the
same credits and inline threshold values.

The inline threshold is set when a connection is established.  The
base RPC-over-RDMA protocol does not provide a mechanism for altering
the inline threshold of a connection once it has been established.

[RFC5667] places normative requirements on the inline threshold value
for a connection.  There is no guidance provided on how
implementations should behave when two ULPs that have different
inline threshold requirements share the same connection.

Further, current NFS implementations ignore the inline threshold
requirements stated in [RFC5667].  It is unlikely that they would
interoperate successfully with any new implementation that followed
the letter of [RFC5667].

### 6.3.1.  Recommendations

Upper Layer Protocols should be able to operate no matter what inline
threshold is in use.

An Upper Layer Binding might provide informative guidance about
optimal values of an inline threshold, but normative requirements are
difficult to enforce unless connection sharing is explicitly not
permitted.

### 6.4.  Violations Of Binding Rules

Section 3.4 of RFC 5666 introduces the idea of an Upper Layer Binding
specification to state which Upper Layer operations are allowed to
use explicit RDMA to transfer a bulk payload item.

The fifth paragraph of this section states:

    The interface by which an upper-layer implementation communicates
    the eligibility of a data item locally to RPC for chunking is out
    of scope for this specification.  In many implementations, it is
    possible to implement a transparent RPC chunking facility.

If the Upper Layer on a receiver is not aware of the presence and
operation of an RPC-over-RDMA transport under it, it could be
challenging to discover when a sender has violated an Upper Layer
Binding rule.

If a violation does occur, RFC 5666 does not define an unambiguous
mechanism for reporting the violation.  The violation of Binding
rules is an Upper Layer Protocol issue, but it is likely that there
is nothing the Upper Layer can do but reply with the equivalent of
BAD XDR.

When an erroneously-constructed reply reaches a requester, there is
no recourse but to drop the reply, and perhaps the transport
connection as well.

### 6.4.1.  Recommendations

Policing DDP-eligibility must be done in co-operation with the Upper
Layer Protocol by its receive endpoint implementation.

It is the Upper Layer Binding's responsibility to specify how a
responder must reply if a requester violates a DDP-eligibilty rule.
The Binding specification should provide similar guidance for
requesters about handling invalid RPC-over-RDMA replies.

6.5.  Binding Specification Completion Assessment

   RFC 5666 Section 3.4 states:

      Typically, only those opaque and aggregate data types that may
      attain substantial size are considered to be eligible.  However,
      any object MAY be chosen for chunking in any given message.

      Chunk eligibility criteria MUST be determined by each upper-layer
      in order to provide for an interoperable specification.

   Authors of Upper Layer Binding specifications should consider each
   data type in the Upper Layer's XDR definition, in particular compound
   types such as arrays and lists, when restricting what XDR objects are
   eligible for Direct Data Placement.

   In addition, there are requirements related to using NFS with RPC-
   over-RDMA in [RFC5667], and there are some in [RFC5661].  It could be
   helpful to have guidance about what kind of requirements belong in an
   Upper Layer Binding specification versus what belong in the Upper
   Layer Protocol specification.

6.5.1.  Recommendations

   RFC 5666bis should describe what makes a Binding specification
   complete (i.e. ready for publication).

7.  Unimplemented Protocol Features

   There are features of RPC-over-RDMA Version One that remain
   unimplemented in current implementations.  Some are candidates to be
   removed from the protocol because they have proven unnecessary or
   were not properly specified.

   Other features are unimplemented, unspecified, or have only one
   implementation (thus interoperability remains unproven).  These are
   candidates to be retained and properly specified.

7.1.  Unimplemented Features To Be Removed

7.1.1.  Connection Configuration Protocol

   No implementation has seen fit to support the Connection
   Configuration Protocol.  While a need to exchange pertinent
   connection information remains, the preference is to exchange that
   information as part of the set up of each connection, rather than as
   settings that apply to all connections (and thus all ULPs) between
   two peers.

#### 7.1.1.1.  Recommendations

CCP should be removed from RFC 5666bis.

### 7.1.2.  Read-Read Transfer Model

All existing RPC-over-RDMA Version One implementations use a Read-Write data transfer model.  The server endpoint is responsible for initiating all RDMA data transfers.  The Read-Read transfer model has been deprecated, but because it appears in RFC 5666, implementations are still responsible for supporting it.  By removing the specification and discussion of Read-Read, the protocol and specification can be made simpler and more clear.

Once the Read-Read transfer model is no longer supported, a responder would no longer be allowed to send a Read list to a requester. Sending a Read list would be needed if a requester has not provided enough memory space in the form of a Reply chunk or Write list to receive a large RPC Reply.

There is currently no mechanism in the RPC-over-RDMA Version One protocol for a responder to indicate that inadequate reply buffer resources were provided by a requester.  Therefore, requesters should be fully responsible for providing all necessary memory resources to receive each RPC reply, including a properly populated Write list and/or a Reply chunk.

#### 7.1.2.1.  Recommendations

Remove Read-Read from RFC 5666bis, in particular from its equivalent of RFC 5666 Section 3.8.  RFC 5666bis should require implementations not to send RDMA_DONE; an implementation receiving it should ignore it.  The XDR definition should reserve RDMA_DONE.  RFC 5666bis should explicitly state requirements for requesters to allocate and prepare reply buffer resources for each RPC-over-RDMA message.

### 7.1.3.  RDMA_MSGP

It has been observed that the current specification of RDMA_MSGP is not clear enough to result in interoperable implementations. Possibly as a result, current receive endpoints do recognize and process RDMA_MSGP messages, though they do not take advantage of the passed alignment parameters.  Receivers treat RDMA_MSGP messages like RDMA_MSG messages.

Currently senders do not use RDMA_MSGP messages.  RDMA_MSGP depends on bulk payload occurring at the end of RPC messages, which is often

not true of NFSv4 COMPOUND requests.  Most NFSv3 requests are small
enough not to need RDMA_MSGP.

To be effective, RDMA_MSGP depends on getting alignment preferences
in advance via CCP.  There are no CCP implementations to date.
Without CCP, there is no way for peers to discover a receiver
endpoint's preferred alignment parameters, unless the implementation
provides an administrative interface for specifying a remote's
alignment parameters.  RDMA_MSGP is useless without that knowledge.

### 7.1.3.1.  Recommendations

To maintain backward-compatibility, RDMA_MSGP must remain in the
protocol.  RFC 5666bis should require implementations to not send
RDMA_MSGP messages.  If an RDMA_MSGP message is seen by a receiver,
it should ignore the alignment parameters and treat RDMA_MSGP
messages as RDMA_MSG messages.  The XDR definition should reserve
RDMA_MSGP.

### 7.2.  Unimplemented Features To Be Retained

### 7.2.1.  RDMA_ERROR Type Messages

Server implementations the author is familiar with can send
RDMA_ERROR type messages, but only when an RPC-over-RDMA version
mismatch occurs.  There is no facility to return the ERR_CHUNK error.
These implementations treat unrecognized message types and other
parsing errors as an RDMA_MSG type message.  Obviously this behavior
does not comply with RFC 5666, but it is also recognized that this
behavior is not an improvement over the specification.

### 7.2.1.1.  Recommendations

RFC 5666bis should provide stronger guidance for error checking, and
in particular, when a connection must be broken.

Implementations that do not adequately check incoming RPC-over-RDMA
headers must be updated.

### 7.2.2.  RPCSEC_GSS On RPC-over-RDMA

The second paragraph of RFC 5666 Section 11 says:

   For efficiency, a more appropriate security mechanism for RDMA
   links may be link-level protection, such as certain configurations
   of IPsec, which may be co-located in the RDMA hardware.  The use
   of link-level protection MAY be negotiated through the use of the
   new RPCSEC_GSS mechanism defined in [RFC5403] in conjunction with

the Channel Binding mechanism [RFC5056] and IPsec Channel
Connection Latching [RFC5660].  Use of such mechanisms is REQUIRED
where integrity and/or privacy is desired, and where efficiency is
required.

However, consider:

o  As of this writing, no implementation of RPCSEC_GSS v2 Channel
   Binding or Connection Latching exist.  Thus, though it is
   sensible, this part of RFC 5666 has never been implemented.

o  Not all fabrics and RNICs support a link-layer protection
   mechanism that includes a privacy service.

o  When multiple users access a storage service from the same client,
   it is appropriate to deploy a message authentication service
   concurrently with link-layer protection.

Therefore, despite its performance impact, RPCSEC_GSS can add
important function to RPC-over-RDMA deployments.

Currently there is an InfiniBand-only client and server
implementation of RPCSEC_GSS on RPC-over-RDMA that supports the
authentication, integrity, and privacy services.  This pair of
implementations was created without the benefit of normative guidance
from RFC 5666.  This client and server pair interoperates with each
other, but there are no independent implementations to test with.

RPC-over-RDMA requesters are responsible for providing adequate reply
resources to responders.  These resources require special treatment
when an integrity or privacy service is in use.  Direct data
placement cannot be used with software integrity checking or
encryption.  Thus standards guidance is imperative to ensure that
independent RPCSEC_GSS implementations can interoperate on RPC-over-
RDMA transports.

## 7.2.2.1.  Recommendations

RFC 5666bis should continue to require the use of link layer
protection when facilities are available to support it.

At the least, RPCSEC_GSS per-message authentiction is valuable, even
if link layer protection is in use.  Integrity and privacy should
also be made available even if they do not perform well, because
there is no link layer protection for some fabrics.

Therefore, RFC 5666bis should provide a specification for RPCSEC_GSS
on RPC-over-RDMA, codifying the one existing implementation so that
others may interoperate with it.

## 8.  Security Considerations

To enable RDMA Read and Write operations, an RPC-over-RDMA Version
One requester exposes some or all of its memory to other hosts.  RFC
5666bis should suggest best implementation practices to minimize
exposure to careless or potentially malicious implementations that
share the same fabric.  Important considerations include:

o  The use of Protection Domains to limit the exposure of memory
   regions to a single connection is critical.  Any attempt by a host
   not participating in that connection to re-use R_keys will result
   in a connection failure.  Because ULP security relies on this
   behavior of Reliable Connections, strong authentication of the
   remote is recommended.

o  Unpredictable R_keys should be used for any operation requiring
   advertised memory regions.  Advertising a continuously registered
   memory region allows a remote host to read or write its contents
   even when an RPC involving that memory is not under way.
   Therefore this practice should be avoided.

o  Advertised memory regions should be invalidated as soon as related
   RPC operations are complete.  Invalidation and DMA unmapping of
   regions should be complete before an RPC application is allowed to
   continue execution and use the contents of a memory region.

## 9.  IANA Considerations

This document does not require actions by IANA.

## 10.  Appendix A: XDR Language Description

Revised XDR definition of RPC-over-RDMA Version One.  The original
definition is in Section 4.3 of RFC 5666.

The XDR stream position of the fields and their use are not altered
by this revision.  The significant changes are:

1.  Copyright boilerplate has been provided

2.  The structure, field, and enum names have been made consistent
    with other standard XDR definitions

3.  The xdr_read_chunk structure is now called an
    rpcrdma1_read_segment because that structure functions the same
    way that an rpcrdma1_segment element in a Write chunk array does

4.  Duplicate definitions of the chunk list fields have been removed

5.  As the Read-Read transfer model is deprecated, RDMA_DONE is now a
    reserved value

6.  As RDMA_MSGP messages are deprecated, RDMA_MSGP is now a reserved
    value

Code components extracted from this document must include the
following license:

<CODE BEGINS>

```
    /*
     * Copyright (c) 2010, 2015 IETF Trust and the persons
     * identified as authors of the code.  All rights reserved.
     *
     * The authors of the code are:
     * B. Callaghan, T. Talpey, and C. Lever.
     *
     * Redistribution and use in source and binary forms, with
     * or without modification, are permitted provided that the
     * following conditions are met:
     *
     * - Redistributions of source code must retain the above
     *   copyright notice, this list of conditions and the
     *   following disclaimer.
     *
     * - Redistributions in binary form must reproduce the above
     *   copyright notice, this list of conditions and the
     *   following disclaimer in the documentation and/or other
     *   materials provided with the distribution.
     *
     * - Neither the name of Internet Society, IETF or IETF
     *   Trust, nor the names of specific contributors, may be
     *   used to endorse or promote products derived from this
     *   software without specific prior written permission.
     *
     *   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
     *   AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
     *   WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
     *   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
     *   FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
     *   EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
```

```
 *    LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 *    EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 *    NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
 *    SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 *    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 *    LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 *    OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
 *    IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 *    ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

struct rpcrdma1_segment {
        uint32 rdma_handle;
        uint32 rdma_length;
        uint64 rdma_offset;
};

struct rpcrdma1_read_segment {
        uint32                  rdma_position;
        struct rpcrdma1_segment rdma_target;
};

struct rpcrdma1_read_list {
        struct rpcrdma1_read_segment rdma_entry;
        struct rpcrdma1_read_list    *rdma_next;
};

struct rpcrdma1_write_chunk {
        struct rpcrdma1_segment rdma_target<>;
};

struct rpcrdma1_write_list {
        struct rpcrdma1_write_chunk rdma_entry;
        struct rpcrdma1_write_list  *rdma_next;
};

struct rpcrdma1_msg {
        uint32        rdma_xid;
        uint32        rdma_vers;
        uint32        rdma_credit;
        rpcrdma1_body rdma_body;
};

enum rpcrdma1_proc {
        RDMA_MSG = 0,
        RDMA_NOMSG = 1,
        RDMA_MSGP = 2,  /* Reserved */
        RDMA_DONE = 3,  /* Reserved */
```

```
            RDMA_ERROR = 4
      };

      struct rpcrdma1_chunks {
            struct rpcrdma1_read_list    *rdma_reads;
            struct rpcrdma1_write_list   *rdma_writes;
            struct rpcrdma1_write_chunk *rdma_reply;
      };

      enum rpcrdma1_errcode {
            RDMA_ERR_VERS = 1,
            RDMA_ERR_CHUNK = 2
      };

      union rpcrdma1_error switch (rpcrdma1_errcode err) {
            case RDMA_ERR_VERS:
              uint32 rdma_vers_low;
              uint32 rdma_vers_high;
            case RDMA_ERR_CHUNK:
              void;
      };

      union rdma_body switch (rpcrdma1_proc proc) {
            case RDMA_MSG:
            case RDMA_NOMSG:
              rpcrdma1_chunks rdma_chunks;
            case RDMA_MSGP:
              uint32          rdma_align;
              uint32          rdma_thresh;
              rpcrdma1_chunks rdma_achunks;
            case RDMA_DONE:
              void;
            case RDMA_ERROR:
              rpcrdma1_error rdma_error;
      };

   <CODE ENDS>
```

## 11.  Appendix B: Binding Requirement Summary

This appendix collects the known generic Binding Requirements from
RFC 5666 and this document.  This might not be an exhaustive list.
Note that RFC 5666 uses RFC 2119-style terms to specify binding
requirements, even though the requirement statements apply to
protocol specifications rather than to a particular protocol.

1.   "Chunk eligibility criteria MUST be determined by each upper-
     layer in order to provide for an interoperable specification."
     (RFC 5666 Section 3.4)

2.   More specifically, an Upper Layer Binding is required for every
     RPC program interested in using RPC-over-RDMA.  Separate
     bindings may be required for different versions of that program.

3.   Upper Layer Bindings make DDP-eligibility statements about
     specific arguments and results (or portions thereof which still
     are whole XDR objects).  A chunk must contain only one whole XDR
     object.

4.   DDP-eligibility of any XDR object not mentioned explicitly in an
     Upper Layer Binding is "not eligible."

5.   Any XDR object may appear in a Position Zero read chunk or a
     Reply chunk regardless of its DDP-eligibility.

6.   An Upper Layer Binding may limit the number of unique read chunk
     Positions allowed for a particular operation.  An Upper Layer
     Binding may limit the number of chunks in a Write list allowed
     for a particular operation.

7.   An Upper Layer Binding must take care not to allow abuses of the
     Position Zero read chunk to avoid DDP-eligibility restrictions.

8.   "It is NOT RECOMMENDED that upper-layer RPC client protocol
     specifications omit write chunk lists for eligible replies, due
     to the lower performance of the additional handshaking to
     perform data transfer, and the requirement that the RPC server
     must expose (and preserve) the reply data for a period of time."
     (RFC 5666 Section 3.4)

9.   "The mapping of write chunk list entries to procedure arguments
     MUST be determined for each protocol."  (RFC 5666 Section 3.6)

10.  More specifically: by default, the requester provides as many
     Write chunks as the Upper Layer Binding allows for the
     particular operation.  The responder fills in each Write chunk
     with an RDMA-eligible result until the Write list is exhausted
     or there are no more RDMA-eligible results.  If this default
     behavior leads to ambiguity when the requester re-assembles the
     XDR stream, the Binding must explain how to resolve the
     ambiguity, or restrict DDP-eligibility to ensure confusion
     cannot occur.

11.  It is the responsibility of the Upper Layer Binding to specify
     DDP-eligibity rules so that if an DDP-eligible XDR object is
     embedded within another, only one of these two objects is to be
     represented by a chunk.

12.  The Upper Layer Binding must specify how a responder should
     reply if a requester violates a DDP-eligibilty rule.  The
     Binding specification should provide guidance for requesters
     about handling invalid RPC-over-RDMA replies.

## 12.  Acknowledgements

The author gratefully acknowledges the contributions of Dai Ngo,
Karen Deitke, Chunli Zhang, Mahesh Siddheshwar, Dominique Martinet,
and William Simpson.

The author also wishes to thank Dave Noveck and Bill Baker for their
support of this work.  Special thanks go to nfsv4 Working Group Chair
Spencer Shepler and nfsv4 Working Group Secretary Tom Haynes for
their support.

## 13.  References

### 13.1.  Normative References

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

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

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

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

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

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

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

   [RFC5403]  Eisler, M., "RPCSEC_GSS Version 2", RFC 5403, DOI
              10.17487/RFC5403, February 2009,
              <http://www.rfc-editor.org/info/rfc5403>.

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

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

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

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

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

## 13.2.  Informative References

   [I-D.ietf-nfsv4-rpcrdma-bidirection]
              Lever, C., "Size-Limited Bi-directional Remote Procedure
              Call On Remote Direct Memory Access Transports", draft-
              ietf-nfsv4-rpcrdma-bidirection-01 (work in progress),
              September 2015.

Author's Address

Charles Lever
Oracle Corporation
1015 Granger Avenue
Ann Arbor, MI  48104
US

Phone: +1 734 274 2396
Email: chuck.lever@oracle.com