NFSv4 C. Lever
Internet-Draft Oracle
Intended status: Informational October 9, 2015
Expires: April 11, 2016

RPC-over-RDMA Version One Implementation Experience
draft-cel-nfsv4-rfc5666-implementation-experience-01

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

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.2. Purpose Of This Document

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.

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:

- Repairing specification ambiguities

- Codifying successful implementation practices and conventions

- Clarifying the role of Upper Layer Binding specifications

- Exploring protocol enhancements that might be added without wire behavior changes

1.3. Updating RFC 5666

This section is an unofficial summary of the nfsv4 Working Group meeting held 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:

1. Filing individual errata for each issue.

2. Introducing an new RFC that updates but does not obsolete RFC 5666, but makes no change to the protocol.

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

4. 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.

The overall preference observed during IETF 92 was to update and obsolete RFC 5666, but retain full interoperability with current RPC-over-RDMA Version One implementations by avoiding changes to on-the-wire behavior (number 3 above). This eases the burden on implementers, who can then reference a single specification of the protocol. In addition, this alternative extends the life of the current implementations in the field, which utilize RPC-over-RDMA Version One effectively.

Subsequent discussion with the nfsv4 Working Group focused primarily on resolving specification ambiguities that could result in interoperability failure. A Version Two of RPC-over-RDMA, where deeper changes can be made and new functionality introduced, was left open for a later time. The priority is fixing issues with the current Proposed Standard.

Recommendations in this document accepted by the nfsv4 Working Group can be used as input when constructing an RFC 5666bis.

1.4. Scope Of This Document

This document does not specify a new Internet Protocol. It does not propose changes to an existing Internet Protocol that are visible to other implementations. It does not update a Proposed Standard, but acts simply as a place to record specific areas that need attention. Therefore the category of this document is Informational.

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]. The 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 an XDR byte 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 XDR and the RPC protocol [RFC5531].

2.2. Remote Direct Memory Access

An individual RPC argument or result may be very large. For example, NFS READ and WRITE payloads are often 100KB or larger.

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

Another benefit of RDMA data transfer is that the host CPUs on both transport endpoints are not involved. Data transfer on both the sending and receiving endpoints is zero-touch. In particular, data that is written to or read from a filesystem is opaque to the transport layer, and thus can be transferred without any serialization or other translation by the host CPU on either endpoint.

RFC 5666 describes how to use only the Send, RDMA Read, and RDMA Write operations described in [RFC5040] to move RPC calls and replies between requesters and responders. The remainder of this document assumes an understanding of RDMA and its primitives.

Because RDMA Read and Write operations work most efficiently with large payloads, RPC-over-RDMA Version One moves RPCs with large payloads differently than RPCs with small payloads.

2.2.1. Small Data Transfers

A local endpoint transfers data into small unadvertised buffers on a remote endpoint using Send operations. Each transfer behaves like a reliable datagram send operation.

This transfer mode is utilized to convey small RPC operations and advertisements of buffer coordinates for large data transfers (see below). The latency of Send operations is significantly lower than traditional network transfers, but the size of these operations is typically limited.

2.2.2. Large Data Transfers

A local endpoint tags memory areas to be involved in data transfers, then advertises the coordinates of those areas to a remote endpoint. The remote endpoint transfers data into or out of those areas using RDMA Read and Write operations.

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

This transfer mode is utilized to convey large whole RPCs or single large arguments or results.

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 used by a few other storage protocols, but 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 RDMA transfer, while others are not. Thus RFC 5666 requires additional separate specifications that describe how each ULP may use RDMA. 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's ULB states which RPC arguments and results in the RPC program are permitted to be transferred by RDMA Read and Write. These are sometimes referred to as "RDMA-eligible." These restrictions do not apply when a whole RPC call or reply is transmitted via an RDMA operation.

A ULB is required for each RPC program and version tuple that is interested in operating 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. 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. The message contents sent via Send, including an 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", contains the co-ordinates of a contiguous memory region that is to be conveyed via an RDMA Read or RDMA Write operation.

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. Read Chunk

One or more "read chunks" are used to advertise the coordinates of an RPC argument to be transferred via an RDMA Read operation. 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>
```

A read chunk is one RDMA segment with a Position field. The Position field indicates the location in an XDR stream where the argument’s data would appear if it were being transferred inline.

A single RPC argument might be contained in one contiguous memory region. That RPC argument can be represented by a single read chunk.

Alternately, a single RPC argument might reside in multiple discontiguous memory regions. Since the memory regions are not contiguous, each region is represented by a single read chunk in a list of chunks. The definition of Position in RFC 5666 Section 3.4 implies this by saying "all chunks belonging to a single RPC argument... will have the same position."

Thus all read chunks that belong to the same RPC argument have the same value in their Position field, and are read in list order into memory regions on the responder. This enables gathering RPC argument data from multiple buffers on the requester.

2.5.4. Write Chunk

A "Write chunk" conveys an RPC result object using one or more RDMA Write operations.

Each write chunk is an array of RDMA segments. One RDMA-eligible RPC result is always conveyed in a single write chunk. This is unlike an RDMA-eligible RPC argument, which may be conveyed in more than one read chunk.

A write chunk is represented by the xdr_write_chunk struct:

<CODE BEGINS>

```
struct xdr_write_chunk {
        struct xdr_rdma_segment target<>;
};
```

<CODE ENDS>

These segments are written in array order into memory regions on the requester This enables scattering an RPC result's data into multiple buffers on the requester.

A requester provides a write chunk as a receptacle for an RPC result. Typically the exact size of the result 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 of each segment in the Write chunk when it returns the Write list to the requester via a matching RPC reply message.

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 position of the reply object. Thus write chunks do not have a Position field.

2.5.5. Read List

Each RPC-over-RDMA Version One call has one "Read list," provided by the requester. This is a list of zero or more RDMA segments with Position values that make up all the RPC arguments in this RPC request to be conveyed via RDMA Read operations.

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>

The Read list may be empty if the RPC call has no RPC arguments that are RDMA-eligible.

2.5.6. Write List

Each RPC-over-RDMA Version One call has one "Write list," provided by the requester. This is a list of zero or more RDMA segment arrays that will catch the RPC results in this RPC request to be conveyed via RDMA Write operations.

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 are no RPC results which are RDMA-eligible.

2.5.7. Position Zero Read Chunk

A requester may use a "Position Zero read chunk" to convey most 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.

2.5.8. Reply Chunk

Each RPC-over-RDMA Version One call may have one "Reply chunk," provided by the requester. 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 the RPC reply message. Typically there is only one segment in a Reply chunk.

A requester provides the Reply chunk whenever it predicts the responder’s reply cannot 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. 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. 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 Read list is typically used for Upper Layer WRITE operations such as NFS WRITE, while the Write list is typically used for Upper Layer READ operations such as NFS READ. If the Read-Read transfer model is removed from RFC 5666bis, it would be less confusing to readers of Upper Layer Binding specifications to call the Read list the Argument list, and call the Write list the Result list.

The XDR definition for a read chunk is an RDMA segment with a position field. It is implied in RFC 5666 Section 3.4 that multiple xdr_read_chunk objects can make up a single RPC argument object 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.

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

The XDR definition for a write chunk is an array of segments. One xdr_write_chunk represents one RPC result object. An RPC argument is represented by one or more read chunks, but an RPC result is always represented by a single write chunk.

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 RPC argument. A Write list entry is an array, and is always a whole RPC result.

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 RPC arguments 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>

3.1.1. Recommendations

To aid in understanding, RFC 5666bis should include a glossary that explains and distinguishes the various elements in the protocol. Upper Layer Binding specifications may also 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 use a consistent naming convention for all XDR definitions. For example, all structures and union names should use an "rpc_rdma1_" 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:

- Rename the xdr_read_chunk XDR object as rpc_rdma1_read_segment.

- Define a "read chunk" as an ordered list of rpc_rdma1_read_segment objects that have identical Position values.

- Define the "Read list" as a list of zero or more read chunks, expressed as an ordered list of rpc_rdma1_read_segment objects whose Position value may vary.

With this change, there would no longer be a simple XDR object that explicitly represents a read chunk. A read chunk and a write chunk are now equivalent objects: One read chunk will always map to a single RPC argument, just like a write chunk always maps to a single RPC result. All discussion should take care to use the term "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 rpc_rdma1_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.

3.2. 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 exceeds the size of the responder's receive buffers and the ULB prohibits the use of RDMA for large RPC arguments. The requester conveys the co-ordinates of the

Position Zero read chunk with a Send operation, then the responder uses an RDMA Read operation to pull the RPC call 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 arguments and results, and RDMA eligibility criteria is determined by a separate specification, and not 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, negating the benefit of RDMA data transfer. 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.

3.2.1. 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.

3.3. 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 bulk payload has been placed in read chunks.

A similar discussion applies to RDMA_NOMSG replies with large reply bodies and RDMA-eligible results. Such replies would use both the Write list and the Reply chunk simultaneously. However, write chunks do not have Position fields. It remains to be seen whether this is enough to enable requesters to re-assemble generic RPC replies correctly.

3.3.1. 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.

3.4. RDMA_MSG Call with Position Zero Read Chunk

An RPC header starts at XDR stream offset zero. 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 stream offset 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 stream offset 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 stream offset zero, but there is a Position Zero read chunk, which also starts at XDR stream offset zero.

3.4.1. Recommendations

RPC 5666bis should clearly define what is meant by an XDR stream. RFC 5666bis should state that XDR stream Position 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 are confusing and should be revised. Numbering chunks this way is not natural to the way read chunks and write chunks work.

3.5. 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 RPC argument.

A read chunk conveys a large RPC argument 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 argument after a read chunk should appear. XDR requires each argument in an RPC call to begin on 4-byte alignment [RFC4506]. But a read chunk's XDR padding is optional (see above). The next read chunk's position field determines where it is placed in the XDR stream. However inline content following a read chunk does not have a Position field to guide the receiver in the reassembly of the RPC call message.

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 open any requirements for XDR padding and alignment when a read chunk is followed in the XDR stream by more inline content.

The correct answer is that following a read chunk of an unaligned length, if the next argument in the XDR stream is in the inline buffer, it must begin on a 4-byte boundary in that buffer, even when XDR padding is not included in the preceding read chunk. This is because the object that follows a read chunk must always start on an XDR alignment boundary.

Furthermore, the XDR pad for the preceding read chunk cannot appear in the inline content, even if it was also not included in the chunk itself. This is because the RPC argument that preceded the read chunk will have been padded to 4-byte alignment. The next position in the inline buffer will already be on a 4-byte boundary.

3.5.1. 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.

A good generic rule is that all RPC objects in every call or reply message 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 RPC objects are placed in inline buffers. XDR alignment in inline buffers is always relative to Position Zero (or, where the RPC header starts).

3.6. Write List XDR Roundup

The final paragraph of RFC 5666 Section 3.7 says this:

> 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 never write XDR pad bytes, as the requester's upper layers does not reference them. However, 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 buffers following the advertised receive buffers.

In zero-copy scenarios, an extra segment must be provided separately to avoid overwriting existing data (case 2). In cases where live data follows the area where the responder writes the data payload, an extra registration is needed for just a handful of bytes of no value.

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 every RPC result object must begin on an XDR alignment boundary, the object following the write chunk in the reply's XDR stream must begin on an XDR alignment boundary. There should be no need for a XDR pad to be present for the receiver to re-assemble the RPC reply's XDR stream correctly.

Unfortunately at least one server implementation relies on the existence of that extra buffer, even though it does not write to it. Another server implementation does not rely on it (operation proceeds if it is missing) but when it is present, this server does write zeroes to it.

Therefore the extra buffer for a write chunk's XDR pad, either as a separate segment, or as an extension of the segment that represents the data payload buffer, must remain for now.

Note that because the Reply chunk is a write chunk, these roundup rules apply to it as well. However, a requester typically provides a single contiguous buffer for whole replies, which consist of XDR encoded content. A separate tail buffer to catch an XDR pad is unlikely to be needed.

3.6.1. Recommendations

RFC 5666bis should provide a discussion of the requirements around write chunk roundup, with examples. The discussion should be separate from the discussion of read chunk roundup.

Explicit RFC2119-style interoperability requirements should be provided in the text. For example, the requester MUST provide buffer space for XDR roundup of write chunks, and the responder SHOULD NOT write into that buffer.

3.7. 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>

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

RFC 5666 does not prescribe how a responder should behave when the result object 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:

- 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

- 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.

In the future, RPC-over-RDMA Version One will have to handle RPC replies where multiple Write list entries are available but the responder has a choice about which result objects to return as bulk reply data. The arguments and results of an NFSv4 COMPOUND are a switched union, and some of the operations in a compound (such as READ, whose data payload reply is RDMA-eligible) also use a switched union.

For example, combining several READ operations in an NFSv4 COMPOUND might be problematic (if it weren't for the requirement that the entire compound should fail if just one operation in the compound fails).

3.7.1. 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. A good quality requester implementation would save the Write list and use that saved copy to invalidate the written memory region upon RPC completion. 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, each argument or result 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 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.

The Reply chunk is a write chunk (a degenerate write chunk list). It is not clear whether the authors intended this requirement to apply to the Reply chunk. Some server implementations regard the Reply chunk as optional.

Requesters may always provide a Reply chunk, at the cost of registering memory the server may choose not to use. Or a client may choose not to provide a Reply chunk when it believes there is no possibility the server will overrun the client's receive buffer when returning the RPC reply.

A server may always use a provided Reply chunk, even when it is more efficient to convey an RPC reply inline (for instance, if an RPC reply is very small). Or a server may choose to ignore the provided Reply chunk when it believes there is no possibility the RPC reply can overrun the client's receive buffer.

The choice of when to provide or utilize a reply chunk depends on whether the sender believes the RPC message will fit entirely within the inline buffer.

Section 3.6 of RFC 5667 says a server MUST use a Write list provided by a client. RFC 5666bis might prescribe that if the client provides a Reply chunk, the server MUST use it, as the client is telling the server that it believes the expected RPC reply may not fit in its receive buffer. That way the server cannot overrun client's receive buffer by choosing to Send an intermediate-sized inline request instead of using a supplied reply chunk.

Without CCP, however, both sides are guessing the other's inline threshold. To maintain 100% interoperability, a client endpoint must always provide a Reply chunk, and a server endpoint must always use it. However, this requirement can be very inefficient. A middle ground must be reached.

4.3.1. Recommendations

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

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.

5. Pre-requisites for NFSv4

5.1. Multiple RDMA-eligible Arguments and Results

One NFSv4 COMPOUND may include more than one NFSv4 operation that conveys RDMA-eligible arguments or replies. There may be additional considerations when marshaling or decoding such compounds on RPC-over-RDMA Version One transports.

5.1.1. Recommendations

Additional review of RFC 5666 and prototyping may be needed to understand if additional protocol requirements are necessary when multiple read chunks (Read list containing chunks with more than one Position value) or multiple write chunks (Write list containing multiple chunk arrays) are present.

More discussion and thought needs to go into handling an NFSv4 COMPOUND reply conveying more than one bulk data result object. When operation results are defined as XDR unions, it can be ambiguous which bulk data result object belongs to which Write list entry.

5.2. 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 unique requirements to pre-post receive resources, 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].

5.2.1. Recommendations

RFC 5666bis should reference or include an informational specification of backwards-direction RPC requests.

5.3. Missing NFS Binding Specifications

To fully support minor versions of NFSv4 on RDMA transports, RFC 5666 requires an Upper Layer Binding Specification for the following cases. This work is out of scope for RFC 5666bis.

- NFS ancillary protocols that are not specified in a published IETF standard, but that are typically conveyed on the same transport as NFS (e.g. NFSACL)

- NFS ancillary protocols that are not specified in a published IETF standard, but that can benefit from the low latency operation of RDMA transports (e.g. NLM)

- NFSv4 minor versions one and newer

- Existing and new pNFS layouts

- NFS protocol extensions that do not increment the minor version

6. Requirements 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 specify explicit generic requirements for what goes in an Upper Layer Binding specification in one separate section. 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

The third paragraph of Section 3.4 states that any object MAY be chosen for chunking (RDMA eligibility) in any given message. That paragraph also states:

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

Further advice about RDMA eligibility does not appear. However it is safe to say that object size is not the only consideration for RDMA eligibility.

For instance, an NFS READDIR result can be large, but typically a server copies this result piecemeal into place, encoding each section; and the receiving client must perform the converse actions. Though there is potentially a large amount of data, the benefit of an RDMA transfer is lost because of the need for both host CPUs to be involved in marshaling and decoding.

6.2.1. Recommendations

RFC 5666bis should define what an Upper Layer Binding is, and how it may be specified.

RFC 5666bis should explicitly specify that an Upper Layer Binding is required for every RPC program interested in operating on RDMA transports. Separate bindings may be required for different versions of that program.

RFC 5666bis should provide generic guidance about what makes a procedure argument or result eligible for RDMA transfer.

RFC 5666bis should state that the eligibility of any object not mentioned explicitly in an ULB is "not eligible." The exception is that Position Zero read chunks and Reply chunks may contain any and all argument and result objects regardless of their RDMA 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.

6.3. 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.

An Upper Layer Binding specification should consider each data type in the Upper Layer's XDR definition, in particular compound types such as arrays and lists, when restricting what arguments and results are eligible for RDMA transfer.

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.

6.3.1. Recommendations

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

7. Removal of Unimplemented Protocol Features

7.1. 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.

7.1.1. Recommendations

Remove Read-Read from RFC 5666bis, in particular from its equivalent of RFC 5666 Section 3.8. Reserve RDMA_DONE and make it unused.

7.2. RDMA_MSGP

RDMA_MSGP is typically difficult to implement in requesters, and the author has found none that do. Responders are required to accept RDMA_MSGP, though most do not take advantage of it.

Also, notably, without CCP, there is no way for peers to discover a server 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 a priori knowledge.

7.2.1. Recommendations

RFC 5666bis should allow implementations that choose not to implement CCP to not implement RDMA_MSGP. Or, RFC 5666bis should remove RDMA_MSGP.

8. Optional Additions To The Protocol

These items might be beyond the scope of RFC 5666bis because the required protocol changes could render existing implementations non-interoperable, or require a protocol version increment.

8.1. Support For GSS-API With RPC-Over-RDMA

Section 11 of RFC 5666 introduces the concept of employing RPCSEC_GSS [RFC2203] with an RPC-over-RDMA transport. However, it recommends using non-GSS-based security mechanisms to retain the efficiency benefits of RDMA transfer.

In some deployments, the use of GSS-based Kerberos integrity or privacy is a fixed requirement. One existing RPC-over-RDMA implementation has chosen to send all integrity and privacy-protected RPC calls and replies via long messages. The requirement to use long replies may present an interoperability problem to implementations that choose not to use long messages in most cases, even for GSS-wrapped RPC operations.

Another implementation is exploring the possibility of offloading integrity and privacy computation to the RNIC.

Further discussion and prototyping of RPC-over-RDMA with GSS-API [RFC2743] is needed. It is desirable to have done this before RFC 5666bis is complete, although it would be a large undertaking.

8.2. Remote Invalidation

On-the-fly memory registration must be performed when read or write chunks are transferred as part of an RPC request. A registration cost is incurred before the RPC call is sent, and an invalidation cost is incurred after the RPC reply is received.

To relieve the client of the cost of the latter, it is possible for the server endpoint to ask the client endpoint's RNIC to invalidate the registered memory associated with an RPC, as part of sending the RPC reply. When the server performs this invalidation, the client is no longer required to invalidate during RPC reply processing, avoiding the cost of that extra operation before retiring the RPC. We'll refer to the server invalidating a client's R_key as "remote invalidation."

To perform remote invalidation, the server uses a Send with Invalidate operation instead of a plain Send operation when conveying an RPC reply. A Send with Invalidate operation targets a single R_key that the client's HCA is to invalidate (see [RFC5040]). The server can invalidate memory regions associated with either read or write chunks sent by the client.

When an RDMA SEND operation arrives at a client, the client endpoint receives a completion for a pending receive operation. The completion indicates whether the server used a plain Send or a Send With Invalidate. The completion may also indicate which R_key the server chose to invalidate.

8.2.1. Hardware Support

Not all RNICs support receiving an RDMA SEND that requests an R_key invalidation. An RPC-over-RDMA client endpoint must somehow indicate to RPC-over-RDMA servers that Send with Invalidate may be used instead of Send. Otherwise the client's HCA would reject the Send with Invalidate conveying the RPC reply, and the RPC (and possibly the transport connection) would fail.

8.2.2. Avoiding Spurious Invalidation

In some cases, an RPC-over-RDMA client might not wish a server to perform R_key invalidation, ever. For instance, if the client uses a global or shared R_key to give the server access to its memory, other users of the R_key would be adversely affected if the R_key suddenly became invalid.

8.2.3. Invalidating Multiple R_keys

Send with Invalidate takes only a single R_key, but an RPC-over-RDMA client might have registered several memory regions for an RPC, each with its own R_key.

For example, an RPC request may require a Read list and a Reply chunk. Read chunks must be registered for read access, but a Reply chunk is registered to allow writes. Thus two R_keys are required in this case.

RPC-over-RDMA clients must be prepared to receive RPC replies where one R_key has been invalidated but others have not.

8.2.4. Invalidation Races

The protocol design must prevent the possibility that the server might invalidate an R_key that the client has recycled and is still actively using.

8.2.5. Backward Compatibility

For backward compatibility, RPC-over-RDMA servers must not use Send with Invalidate when an RPC-over-RDMA client endpoint is not prepared to sort out which memory regions have been remotely invalidated.

For backward compatibility, RPC-over-RDMA clients must not assume that an RPC-over-RDMA server endpoint has performed Send with Invalidate.

8.2.6. Conclusion

It is possible that to address all of the above issues, a change must be made to the on-the-wire RPC-over-RDMA Version One protocol. However if no change is required, remote invalidation could be successfully introduced to the RPC-over-RDMA Version One protocol.

8.3. Work Cancellation

Rarely, an RPC consumer might want to cancel outstanding work. An application might exit while there are pending RPC operations, for example, if a software fault or user-generated interrupt occurs. Or, the RPC service might be slow or unresponsive, and the application might have placed time-limits that pre-maturely retire RPCs that take too long to complete.

As a result of canceled work, the client endpoint may tear down registered RDMA memory regions before the server endpoint has performed RDMA operations on the memory. When the server endpoint attempts to complete the requested work, RDMA operations will encounter asynchronous memory protection errors because the R_keys the server has are no longer valid.

This can be fatal for the transport connection. Any other ongoing work on that connection must be redriven when a new transport connection has been established, opening a window for RPC requests to be repeated on the server.

Further, regular and repetitive cancellations of work (for example, an application repeatedly encountering a segmentation fault) could have an adverse impact on other work sharing the transport connection.

One way to avoid this hazard would be to provide a way for requesters to signal to responders that an RPC reply is no longer required. This is not a perfect solution, as a work cancellation request can race with the RPC reply it is trying to cancel.

A work cancellation request might be plumbed in as a new rdma_proc type. A client would be responsible for keeping receive buffers and memory regions associated with an RPC until the server has responded to the cancellation request.

9. Security Considerations

There are no security considerations at this time.

10. IANA Considerations

This document does not require actions by IANA.

11. 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 unwavering support of this work. Special thanks go to nfsv4 Working Group Chair Spencer Shepler and nfsv4 Working Group Secretary Tom Haynes for their support.

12. References

12.1. Normative References

[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>.

[RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, DOI 10.17487/RFC2203, September 1997, <http://www.rfc-editor.org/info/rfc2203>.

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <http://www.rfc-editor.org/info/rfc2743>.

[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>.

[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>.

[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>.

12.2. 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