NFSv4 Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 3, 2009

Tom Talpey
NetApp
Brent Callaghan
Apple
December 3, 2008

Remote Direct Memory Access Transport for Remote Procedure Call
draft-ietf-nfsv4-rpcrdma-09

Status of this Memo

Abstract

A protocol is described providing Remote Direct Memory Access (RDMA) as a new transport for Remote Procedure Call (RPC). The RDMA transport binding conveys the benefits of efficient, bulk data transport over high speed networks, while providing for minimal change to RPC applications and with no required revision of the application RPC protocol, or the RPC protocol itself.

Table of Contents

Requirements Language

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

1. Introduction

Remote Direct Memory Access (RDMA) [RFC5040, RFC5041] [IB] is a technique for efficient movement of data between end nodes, which

becomes increasingly compelling over high speed transports. By directing data into destination buffers as it is sent on a network, and placing it via direct memory access by hardware, the double benefit of faster transfers and reduced host overhead is obtained.

Open Network Computing Remote Procedure Call (ONC RPC, or simply, RPC) [RFC1831bis] is a remote procedure call protocol that has been run over a variety of transports. Most RPC implementations today use UDP or TCP. RPC messages are defined in terms of an eXternal Data Representation (XDR) [RFC4506] which provides a canonical data representation across a variety of host architectures. An XDR data stream is conveyed differently on each type of transport. On UDP, RPC messages are encapsulated inside datagrams, while on a TCP byte stream, RPC messages are delineated by a record marking protocol. An RDMA transport also conveys RPC messages in a unique fashion that must be fully described if client and server implementations are to interoperate.

RDMA transports present new semantics unlike the behaviors of either UDP or TCP alone. They retain message delineations like UDP while also providing a reliable, sequenced data transfer like TCP. And, they provide the new efficient, bulk transfer service of RDMA. RDMA transports are therefore naturally viewed as a new transport type by RPC.

RDMA as a transport will benefit the performance of RPC protocols that move large "chunks" of data, since RDMA hardware excels at moving data efficiently between host memory and a high speed network with little or no host CPU involvement. In this context, the NFS protocol, in all its versions [RFC1094] [RFC1813] [RFC3530] [NFSv4.1], is an obvious beneficiary of RDMA. A complete problem statement is discussed in [NFSRDMAPS], and related NFSv4 issues are discussed in [NFSv4.1]. Many other RPC-based protocols will also benefit.

Although the RDMA transport described here provides relatively transparent support for any RPC application, the proposal goes further in describing mechanisms that can optimize the use of RDMA with more active participation by the RPC application.

2. Abstract RDMA Requirements

An RPC transport is responsible for conveying an RPC message from a sender to a receiver. An RPC message is either an RPC call from a client to a server, or an RPC reply from the server back to the client. An RPC message contains an RPC call header followed by arguments if the message is an RPC call, or an RPC reply header followed by results if the message is an RPC reply. The call

header contains a transaction ID (XID) followed by the program and procedure number as well as a security credential. An RPC reply header begins with an XID that matches that of the RPC call message, followed by a security verifier and results. All data in an RPC message is XDR encoded. For a complete description of the RPC protocol and XDR encoding, see [RFC1831bis] and [RFC4506].

This protocol assumes the following abstract model for RDMA transports. These terms, common in the RDMA lexicon, are used in this document. A more complete glossary of RDMA terms can be found in [RFC5040].

o Registered Memory

All data moved via tagged RDMA operations is resident in registered memory at its destination. This protocol assumes that each segment of registered memory MUST be identified with a steering tag of no more than 32 bits and memory addresses of up to 64 bits in length.

o RDMA Send

The RDMA provider supports an RDMA Send operation with completion signalled at the receiver when data is placed in a pre-posted buffer. The amount of transferred data is limited only by the size of the receiver's buffer. Sends complete at the receiver in the order they were issued at the sender.

o RDMA Write

The RDMA provider supports an RDMA Write operation to directly place data in the receiver's buffer. An RDMA Write is initiated by the sender and completion is signalled at the sender. No completion is signalled at the receiver. The sender uses a steering tag, memory address and length of the remote destination buffer. RDMA Writes are not necessarily ordered with respect to one another, but are ordered with respect to RDMA Sends; a subsequent RDMA Send completion obtained at the receiver guarantees that prior RDMA Write data has been successfully placed in the receiver's memory.

o RDMA Read

The RDMA provider supports an RDMA Read operation to directly place peer source data in the requester's buffer. An RDMA Read is initiated by the receiver and completion is signalled at the receiver. The receiver provides steering tags, memory addresses and a length for the remote source and local destination buffers. Since the peer at the data source receives no notification of RDMA Read completion, there is an assumption that on receiving the data the receiver will signal completion with an RDMA Send message, so that the peer can

free the source buffers and the associated steering tags.

This protocol is designed to be carried over all RDMA transports meeting the stated requirements. This protocol conveys to the RPC peer, information sufficient for that RPC peer to direct an RDMA layer to perform transfers containing RPC data, and to communicate their result(s). For example, it is readily carried over RDMA transports such as iWARP [RFC5040, RFC5041] or Infiniband [IB].

3. Protocol Outline

An RPC message can be conveyed in identical fashion, whether it is a call or reply message. In each case, the transmission of the message proper is preceded by transmission of a transport-specific header for use by RPC over RDMA transports. This header is analogous to the record marking used for RPC over TCP, but is more extensive, since RDMA transports support several modes of data transfer and it is important to allow the upper layer protocol to specify the most efficient mode for each of the segments in a message. Multiple segments of a message may thereby be transferred in different ways to different remote memory destinations.

All transfers of a call or reply begin with an RDMA Send which transfers at least the RPC over RDMA header, usually with the call or reply message appended, or at least some part thereof. Because the size of what may be transmitted via RDMA Send is limited by the size of the receiver's pre-posted buffer, the RPC over RDMA transport provides a number of methods to reduce the amount transferred by means of the RDMA Send, when necessary, by transferring various parts of the message using RDMA Read and RDMA Write.

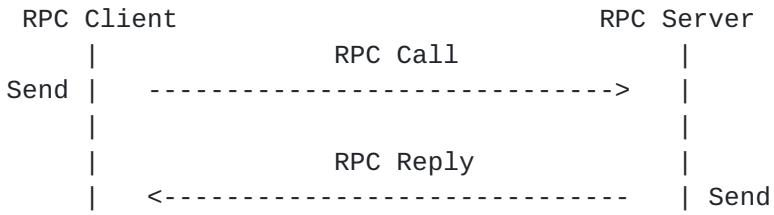
RPC over RDMA framing replaces all other RPC framing (such as TCP record marking) when used atop an RPC/RDMA association, even though the underlying RDMA protocol may itself be layered atop a protocol with a defined RPC framing (such as TCP). It is however possible for RPC/RDMA to be dynamically enabled, in the course of negotiating the use of RDMA via an upper layer exchange. Because RPC framing delimits an entire RPC request or reply, the resulting shift in framing must occur between distinct RPC messages, and in concert with the transport.

3.1. Short Messages

Many RPC messages are quite short. For example, the NFS version 3 GETATTR request, is only 56 bytes: 20 bytes of RPC header, plus a 32 byte file handle argument and 4 bytes of length. The reply to this common request is about 100 bytes.

There is no benefit in transferring such small messages with an RDMA Read or Write operation. The overhead in transferring steering tags and memory addresses is justified only by large transfers. The critical message size that justifies RDMA transfer will vary depending on the RDMA implementation and network, but is typically of the order of a few kilobytes. It is appropriate to transfer a short message with an RDMA Send to a pre-posted buffer. The RPC over RDMA header with the short message (call or reply) immediately following is transferred using a single RDMA Send operation.

Short RPC messages over an RDMA transport:



3.2. Data Chunks

Some protocols, like NFS, have RPC procedures that can transfer very large "chunks" of data in the RPC call or reply and would cause the maximum send size to be exceeded if one tried to transfer them as part of the RDMA Send. These large chunks typically range from a kilobyte to a megabyte or more. An RDMA transport can transfer large chunks of data more efficiently via the direct placement of an RDMA Read or RDMA Write operation. Using direct placement instead of inline transfer not only avoids expensive data copies, but provides correct data alignment at the destination.

3.3. Flow Control

It is critical to provide RDMA Send flow control for an RDMA connection. RDMA receive operations will fail if a pre-posted receive buffer is not available to accept an incoming RDMA Send, and repeated occurrences of such errors can be fatal to the connection. This is a departure from conventional TCP/IP networking where buffers are allocated dynamically on an as-needed basis, and where pre-posting is not required.

It is not practical to provide for fixed credit limits at the RPC server. Fixed limits scale poorly, since posted buffers are dedicated to the associated connection until consumed by receive operations. 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. (Note that the RPC server may in fact be a client at some other layer. For example, NFSv4 callbacks are processed by the NFSv4 client, acting as an RPC server. The credit discussions apply equally in either case.)

Flow control for RDMA Send operations is implemented as a simple request/grant protocol in the RPC over RDMA header associated with each RPC message. The RPC over RDMA header for RPC call messages contains a requested credit value for the RPC server, which MAY be dynamically adjusted by the caller to match its expected needs. The RPC over RDMA header for the RPC reply messages provides the granted result, which MAY have any value except it MUST NOT be zero when no in-progress operations are present at the server, since such a value would result in deadlock. The value MAY be adjusted up or down at each opportunity to match the server's needs or policies.

The RPC client MUST NOT send unacknowledged requests in excess of this granted RPC server credit limit. If the limit is exceeded, the RDMA layer may signal an error, possibly terminating the connection. Even if an error does not occur, it is OPTIONAL that the server handle the excess request(s), and it MAY return an RPC error to the client. Also note that the never-zero requirement implies that an RPC server MUST always provide at least one credit to each connected RPC client from which no requests are outstanding. The client would deadlock otherwise, unable to send another request.

While RPC calls complete in any order, the current flow control limit at the RPC server is known to the RPC client from the Send ordering properties. It is always the most recent server-granted credit value minus the number of requests in flight.

Certain RDMA implementations may impose additional flow control restrictions, such as limits on RDMA Read operations in progress at the responder. Because these operations are outside the scope of this protocol, they are not addressed and SHOULD be provided for by other layers. For example, a simple upper layer RPC consumer might perform single-issue RDMA Read requests, while a more sophisticated, multithreaded RPC consumer might implement its own FIFO queue of such operations. For further discussion of possible protocol implementations capable of negotiating these values, see section 6 "Connection Configuration Protocol" of this draft, or [NFSv4.1].

3.4. XDR Encoding with Chunks

The data comprising an RPC call or reply message is marshaled or serialized into a contiguous stream by an XDR routine. XDR data types such as integers, strings, arrays and linked lists are commonly implemented over two very simple functions that encode either an XDR data unit (32 bits) or an array of bytes.

Normally, the separate data items in an RPC call or reply are encoded as a contiguous sequence of bytes for network transmission over UDP or TCP. However, in the case of an RDMA transport, local routines such as XDR encode can determine that (for instance) an opaque byte array is large enough to be more efficiently moved via an RDMA data transfer operation like RDMA Read or RDMA Write.

Semantically speaking, the protocol has no restriction regarding data types which 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. One such example with rationale, for the NFS protocol family, is provided in [NFSDDP].

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. However, such implementations may lead to inefficiencies, either because they require the RPC layer to perform expensive registration and deregistration of memory "on the fly", or they may require using RDMA chunks in reply messages, along with the resulting additional handshaking with the RPC over RDMA peer. However, these issues are internal and generally confined to the local interface between RPC and its upper layers, one in which implementations are free to innovate. The only requirement is that the resulting RPC RDMA protocol sent to the peer is valid for the upper layer. See for example [NFSDDP].

When sending any message (request or reply) that contains an eligible large data chunk, the XDR encoding routine avoids moving

the data into the XDR stream. Instead, it does not encode the data portion, but records the address and size of each chunk in a separate "read chunk list" encoded within RPC RDMA transport-specific headers. Such chunks will be transferred via RDMA Read operations initiated by the receiver.

When the read chunks are to be moved via RDMA, the memory for each chunk is registered. This registration may take place within XDR itself, providing for full transparency to upper layers, or it may be performed by any other specific local implementation.

Additionally, when making an RPC call that can result in bulk data transferred in the reply, write chunks MAY be provided to accept the data directly via RDMA Write. These write chunks will therefore be pre-filled by the RPC server prior to responding, and XDR decode of the data at the client will not be required. These chunks undergo a similar registration and advertisement via "write chunk lists" built as a part of XDR encoding.

Some RPC client implementations are not able to determine where an RPC call's results reside during the "encode" phase. This makes it difficult or impossible for the RPC client layer to encode the write chunk list at the time of building the request. In this case, it is difficult for the RPC implementation to provide transparency to the RPC consumer, which may require recoding to provide result information at this earlier stage.

Therefore if the RPC client does not make a write chunk list available to receive the result, then the RPC server MAY return data inline in the reply, or if the upper layer specification permits, it MAY be returned via a read chunk list. 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. In the absence of a server-provided read chunk list in the reply, if the encoded reply overflows the posted receive buffer, the RPC will fail with an RDMA transport error.

When any data within a message is provided via either read or write chunks, the chunk itself refers only to the data portion of the XDR stream element. In particular, for counted fields (e.g., a "<>" encoding) the byte count which is encoded as part of the field remains in the XDR stream, and is also encoded in the chunk list. The data portion is however elided from the encoded XDR stream, and is transferred as part of chunk list processing. This is important to maintain upper layer implementation compatibility - both the

count and the data must be transferred as part of the logical XDR stream. While the chunk list processing results in the data being available to the upper layer peer for XDR decoding, the length present in the chunk list entries is not. Any byte count in the XDR stream MUST match the sum of the byte counts present in the corresponding read or write chunk list. If they do not agree, an RPC protocol encoding error results.

The following items are contained in a chunk list entry.

Handle
: Steering tag or handle obtained when the chunk memory is registered for RDMA.

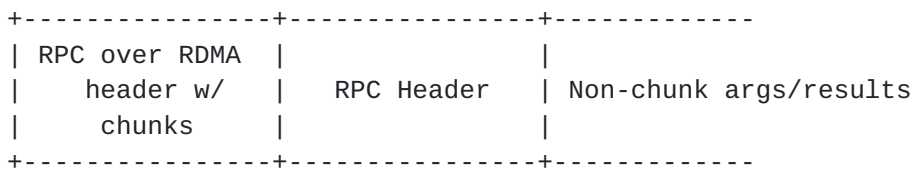
Length
: The length of the chunk in bytes.

Offset
: The offset or beginning memory address of the chunk. In order to support the widest array of RDMA implementations, as well as the most general steering tag scheme, this field is unconditionally included in each chunk list entry.

  While zero-based offset schemes are available in many RDMA implementations, their use by RPC requires individual registration of each read or write chunk. On many such implementations this can be a significant overhead. By providing an offset in each chunk, many pre-registration or region-based registrations can be readily supported, and by using a single, universal chunk representation, the RPC RDMA protocol implementation is simplified to its most general form.

Position
: For data which is to be encoded, the position in the XDR stream where the chunk would normally reside. Note that the chunk therefore inserts its data into the XDR stream at this position, but its transfer is no longer "inline". Also note therefore that all chunks belonging to a single RPC argument or result will have the same position. For data which is to be decoded, no position is used.

When XDR marshaling is complete, the chunk list is XDR encoded, then sent to the receiver prepended to the RPC message. Any source data for a read chunk, or the destination of a write chunk, remain behind in the sender's registered memory and their actual payload is not marshaled into the request or reply.



Read chunk lists and write chunk lists are structured somewhat differently. This is due to the different usage - read chunks are decoded and indexed by their argument's or result's position in the XDR data stream; their size is always known. Write chunks on the other hand are used only for results, and have neither a preassigned offset in the XDR stream, nor a size until the results are produced, since the buffers may be only partially filled, or may not be used for results at all. Their presence in the XDR stream is therefore not known until the reply is processed. The mapping of Write chunks onto designated NFS procedures and their results is described in [NFSDDP].

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. Write chunks are encoded as a list of arrays of RDMA buffers, with each list element (an array) providing buffers for a separate result. 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. However, the RPC layer MUST NOT assume that the buffers have not been modified.

3.5. XDR Decoding with Read Chunks

The XDR decode process moves data from an XDR stream into a data structure provided by the RPC client or server application. Where elements of the destination data structure are buffers or strings, the RPC application can either pre-allocate storage to receive the data, or leave the string or buffer fields null and allow the XDR decode stage of RPC processing to automatically allocate storage of sufficient size.

When decoding a message from an RDMA transport, the receiver first XDR decodes the chunk lists from the RPC over RDMA header, then proceeds to decode the body of the RPC message (arguments or results). Whenever the XDR offset in the decode stream matches that of a chunk in the read chunk list, the XDR routine initiates an RDMA Read to bring over the chunk data into locally registered memory for the destination buffer.

When processing an RPC request, the RPC receiver (RPC server) acknowledges its completion of use of the source buffers by simply replying to the RPC sender (client), and the peer may then free all source buffers advertised by the request.

When processing an RPC reply, after completing such a transfer the RPC receiver (client) MUST issue an RDMA_DONE message (described in Section 3.8) to notify the peer (server) that the source buffers can be freed.

The read chunk list is constructed and used entirely within the RPC/XDR layer. Other than specifying the minimum chunk size, the management of the read chunk list is automatic and transparent to an RPC application.

3.6. XDR Decoding with Write Chunks

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 RPC reply conveys this by returning the write chunk list to the client with the lengths rewritten to match the actual transfer. The XDR "decode" of the reply therefore performs no local data transfer but merely returns the length obtained from the reply.

Each decoded result consumes one entry in the write chunk list, which in turn consists of an array of RDMA segments. The length is therefore the sum of all returned lengths in all segments comprising the corresponding list entry. As each list entry is "decoded", the entire entry is consumed.

The write chunk list is constructed and used by the RPC application. The RPC/XDR layer simply conveys the list between client and server and initiates the RDMA Writes back to the client. The mapping of write chunk list entries to procedure arguments MUST be determined for each protocol. An example of a mapping is described in [NFSDDP].

3.7. XDR Roundup and Chunks

The XDR protocol requires 4-byte alignment of each new encoded element in any XDR stream. This requirement is for efficiency and ease of decode/unmarshaling at the receiver - if the XDR stream buffer begins on a native machine boundary, then the XDR elements will lie on similarly predictable offsets in memory.

Within XDR, when non-4-byte encodes (such as an odd-length string or bulk data) are marshaled, their length is encoded literally,

while their data is padded to begin the next element at a 4-byte boundary in the XDR stream. For TCP or RDMA inline encoding, this minimal overhead is required because the transport-specific framing relies on the fact that the relative offset of the elements in the XDR stream from the start of the message determines the XDR position during decode.

On the other hand, RPC/RDMA Read chunks carry the XDR position of each chunked element and length of the Chunk segment, and can be placed by the receiver exactly where they belong in the receiver's memory without regard to the alignment of their position in the XDR stream. Since any rounded-up data is not actually part of the upper layer's message, the receiver will not reference it, and there is no reason to set it to any particular value in the receiver's memory.

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 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.

Some protocol operations over RPC/RDMA, for instance NFS writes of data encountered at the end of a file or in direct i/o situations, commonly yield these roundups within RDMA Read Chunks. Because any roundup bytes are not actually present in the data buffers being written, memory for these bytes would come from noncontiguous buffers, either as an additional memory registration segment, or as an additional Chunk. The overhead of these operations can be significant to both the sender to marshal them, and even higher to the receiver which to transfer them. Senders SHOULD therefore avoid encoding individual RDMA Read Chunks for roundup whenever possible. It is acceptable, but not necessary, to include roundup data in an existing RDMA Read Chunk, but only if it is already present in the XDR stream to carry upper layer data.

Note that there is no exposure of additional data at the sender due to eliding roundup data from the XDR stream, since any additional sender buffers are never exposed to the peer. The data is literally not there to be transferred.

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. Because of the requirement that the RDMA Write chunks are filled sequentially without gaps, this situation can only occur on the final chunk receiving data. Therefore there is no opportunity for roundup data to insert misalignment or positional gaps into the XDR stream.

3.8. RPC Call and Reply

The RDMA transport for RPC provides three methods of moving data between RPC client and server:

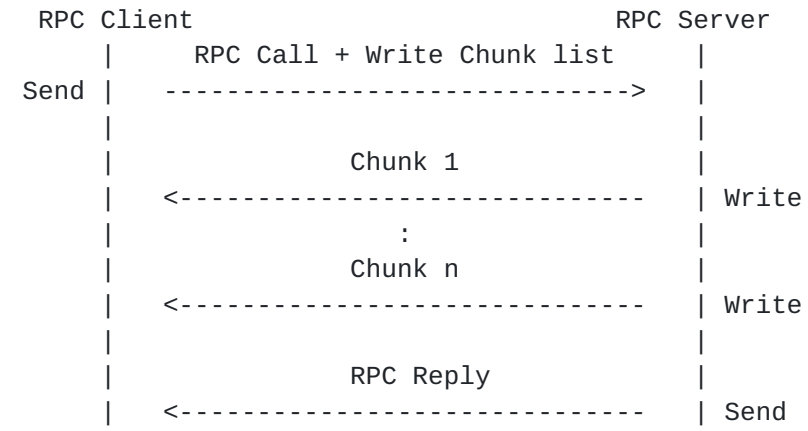
Inline
: Data are moved between RPC client and server within an RDMA Send.

RDMA Read
: Data are moved between RPC client and server via an RDMA Read operation via steering tag, address and offset obtained from a read chunk list.

RDMA Write
: Result data is moved from RPC server to client via an RDMA Write operation via steering tag, address and offset obtained from a write chunk list or reply chunk in the client's RPC call message.

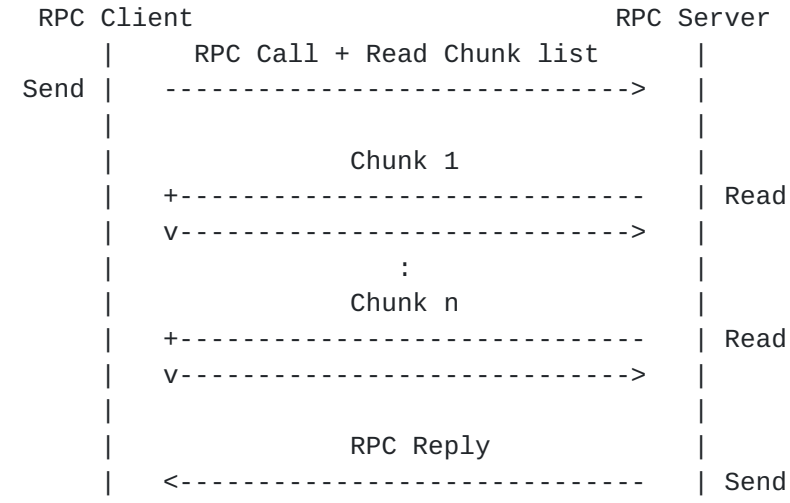
These methods of data movement may occur in combinations within a single RPC. For instance, an RPC call may contain some inline data along with some large chunks to be transferred via RDMA Read to the server. The reply to that call may have some result chunks that the server RDMA Writes back to the client. The following protocol interactions illustrate RPC calls that use these methods to move RPC message data:

An RPC with write chunks in the call message:

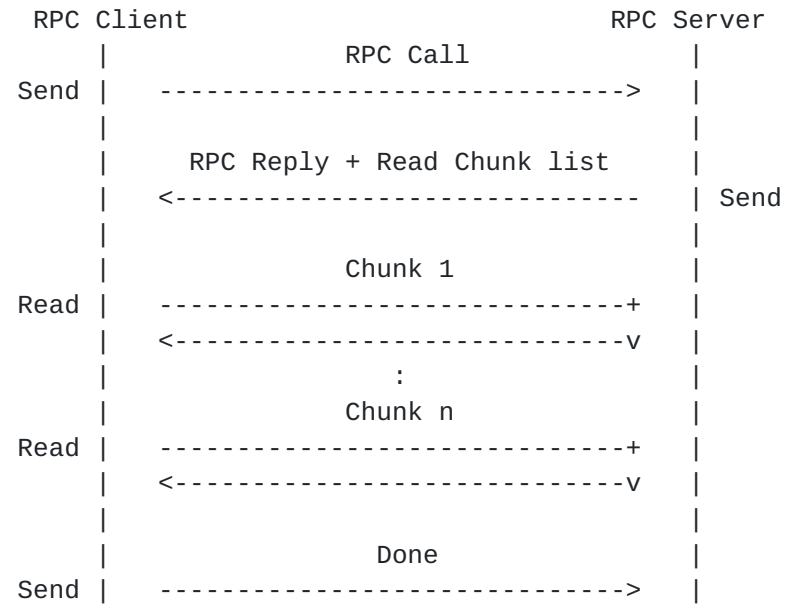


In the presence of write chunks, RDMA ordering provides the guarantee that all data in the RDMA Write operations has been placed in memory prior to the client's RPC reply processing.

An RPC with read chunks in the call message:



An RPC with read chunks in the reply message:



The final Done message allows the RPC client to signal the server that it has received the chunks, so the server can de-register and free the memory holding the chunks. A Done completion is not necessary for an RPC call, since the RPC reply Send is itself a receive completion notification. In the event that the client fails to return the Done message within some timeout period, the server MAY conclude that a protocol violation has occurred and close the RPC connection, or it MAY proceed with a de-register and free its chunk buffers. This may result in a fatal RDMA error if the client later attempts to perform an RDMA Read operation, which amounts to the same thing.

The use of read chunks in RPC reply messages is much less efficient than providing write chunks in the originating RPC calls, due to the additional message exchanges, the need for the RPC server to advertise buffers to the peer, the necessity of the server maintaining a timer for the purpose of recovery from misbehaving clients, and the need for additional memory registration. Their use is NOT RECOMMENDED by upper layers where efficiency is a primary concern. [NFSDDP] However, they MAY be employed by upper layer protocol bindings which are primarily concerned with transparency, since they can frequently be implemented completely within the RPC lower layers.

It is important to note that the Done message consumes a credit at the RPC server. The RPC server SHOULD provide sufficient credits

to the client to allow the Done message to be sent without deadlock (driving the outstanding credit count to zero). The RPC client MUST account for its required Done messages to the server in its accounting of available credits, and the server SHOULD replenish any credit consumed by its use of such exchanges at its earliest opportunity.

Finally, it is possible to conceive of RPC exchanges that involve any or all combinations of write chunks in the RPC call, read chunks in the RPC call, and read chunks in the RPC reply. Support for such exchanges is straightforward from a protocol perspective, but in practice such exchanges would be quite rare, limited to upper layer protocol exchanges which transferred bulk data in both the call and corresponding reply.

3.9. Padding

Alignment of specific opaque data enables certain scatter/gather optimizations. Padding leverages the useful property that RDMA transfers preserve alignment of data, even when they are placed into pre-posted receive buffers by Sends.

Many servers can make good use of such padding. Padding allows the chaining of RDMA receive buffers such that any data transferred by RDMA on behalf of RPC requests will be placed into appropriately aligned buffers on the system that receives the transfer. In this way, the need for servers to perform RDMA Read to satisfy all but the largest client writes is obviated.

The effect of padding is demonstrated below showing prior bytes on an XDR stream (XXX) followed by an opaque field consisting of four length bytes (LLLL) followed by data bytes (DDDD). The receiver of the RDMA Send has posted two chained receive buffers. Without padding, the opaque data is split across the two buffers. With the addition of padding bytes ("ppp" in the figure below) prior to the first data byte, the data can be forced to align correctly in the second buffer.

```
                                        Buffer 1       Buffer 2
     Unpadded                          --------------  --------------

      XXXXXXXLLLLDDDDDDDDDDDDDD    ---> XXXXXXXLLLLDDD  DDDDDDDDDD

     Padded

      XXXXXXXLLLLpppDDDDDDDDDDDDD ---> XXXXXXXLLLLppp  DDDDDDDDDDDDD
```

Padding is implemented completely within the RDMA transport encoding, flagged with a specific message type. Where padding is applied, two values are passed to the peer: an "rdma_align" which is the padding value used, and "rdma_thresh", which is the opaque data size at or above which padding is applied. For instance, if the server is using chained 4 KB receive buffers, then up to (4 KB - 1) padding bytes could be used to achieve alignment of the data. The XDR routine at the peer MUST consult these values when decoding opaque values. Where the decoded length exceeds the rdma_thresh, the XDR decode MUST skip over the appropriate padding as indicated by rdma_align and the current XDR stream position.

4. RPC RDMA Message Layout

RPC call and reply messages are conveyed across an RDMA transport with a prepended RPC over RDMA header. The RPC over RDMA header includes data for RDMA flow control credits, padding parameters and lists of addresses that provide direct data placement via RDMA Read and Write operations. The layout of the RPC message itself is unchanged from that described in [RFC1831bis] except for the possible exclusion of large data chunks that will be moved by RDMA Read or Write operations. If the RPC message (along with the RPC over RDMA header) is too long for the posted receive buffer (even after any large chunks are removed), then the entire RPC message MAY be moved separately as a chunk, leaving just the RPC over RDMA header in the RDMA Send.

4.1. RPC over RDMA Header

The RPC over RDMA header begins with four 32-bit fields that are always present and which control the RDMA interaction including RDMA-specific flow control. These are then followed by a number of items such as chunk lists and padding which MAY or MUST NOT be

present depending on the type of transmission. The four fields which are always present are:

1. Transaction ID (XID).
   The XID generated for the RPC call and reply. Having the XID at the beginning of the message makes it easy to establish the message context. This XID MUST be the same as the XID in the RPC header. The receiver MAY perform its processing based solely on the XID in the RPC over RDMA header, and thereby ignore the XID in the RPC header, if it so chooses.

2. Version number.
   This version of the RPC RDMA message protocol is 1. The version number MUST be increased by one whenever the format of the RPC RDMA messages is changed.

3. Flow control credit value.
   When sent in an RPC call message, the requested value is provided. When sent in an RPC reply message, the granted value is returned. RPC calls SHOULD NOT be sent in excess of the currently granted limit.

4. Message type.

   - RDMA_MSG = 0 indicates that chunk lists and RPC message follow.

   - RDMA_NOMSG = 1 indicates that after the chunk lists there is no RPC message. In this case, the chunk lists provide information to allow the message proper to be transferred using RDMA Read or write and thus is not appended to the RPC over RDMA header.

   - RDMA_MSGP = 2 indicates that a chunk list and RPC message with some padding follow.

   0 RDMA_DONE = 3 indicates that the message signals the completion of a chunk transfer via RDMA Read.

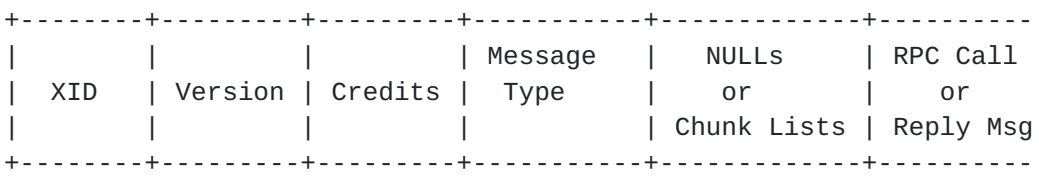
   - RDMA_ERROR = 4 is used to signal any detected error(s) in the RPC RDMA chunk encoding.

Because the version number is encoded as part of this header, and the RDMA_ERROR message type is used to indicate errors, these first four fields and the start of the following message body MUST always remain aligned at these fixed offsets for all versions of the RPC over RDMA header.

For a message of type RDMA_MSG or RDMA_NOMSG, the Read and Write chunk lists follow. If the Read chunk list is null (a 32 bit word of zeros), then there are no chunks to be transferred separately and the RPC message follows in its entirety. If non-null, then it's the beginning of an XDR encoded sequence of Read chunk list entries. If the Write chunk list is non-null, then an XDR encoded sequence of Write chunk entries follows.

If the message type is RDMA_MSGP, then two additional fields that specify the padding alignment and threshold are inserted prior to the Read and Write chunk lists.

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. The XID in the RDMA_MSG or RDMA_MSGP header MUST match this.



Note that in the case of RDMA_DONE and RDMA_ERROR, no chunk list or RPC message follows. As an implementation hint: a gather operation on the Send of the RDMA RPC message can be used to marshal the initial header, the chunk list, and the RPC message itself.

4.2. RPC over RDMA header errors

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.

Two types of errors are defined, version mismatch and invalid chunk format. When the peer detects an RPC over RDMA header version which it does not support (currently this draft 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. When other decoding errors are detected in the header or chunks, either an RPC decode error MAY be returned, or the ROC/RDMA error code ERR_CHUNK MUST be returned.

4.3. XDR Language Description

Here is the message layout in XDR language.

```
   struct xdr_rdma_segment {
      uint32 handle;          /* Registered memory handle */
      uint32 length;          /* Length of the chunk in bytes */
      uint64 offset;          /* Chunk virtual address or offset */
   };

   struct xdr_read_chunk {
      uint32 position;        /* Position in XDR stream */
      struct xdr_rdma_segment target;
   };

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

   struct xdr_write_chunk {
      struct xdr_rdma_segment target<>;
   };

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

   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;
   };

   enum rdma_proc {
      RDMA_MSG=0,   /* An RPC call or reply msg */
      RDMA_NOMSG=1, /* An RPC call or reply msg - separate body */
      RDMA_MSGP=2,  /* An RPC call or reply msg with padding */
      RDMA_DONE=3,  /* Client signals reply completion */
      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_MSGP:
             rpc_rdma_header_padded rdma_msgp;
           case RDMA_DONE:
             void;
           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;
        };

        struct rpc_rdma_header_padded {
           uint32                 rdma_align;   /* Padding alignment */
           uint32                 rdma_thresh;  /* Padding threshold */
           struct xdr_read_list   *rdma_reads;
           struct xdr_write_list  *rdma_writes;
           struct xdr_write_chunk *rdma_reply;
           /* rpc body follows */
        };
```

```
enum rpc_rdma_errcode {
   ERR_VERS = 1,
   ERR_CHUNK = 2
};

union rpc_rdma_error switch (rpc_rdma_errcode err) {
   case ERR_VERS:
     uint32               rdma_vers_low;
     uint32               rdma_vers_high;
   case ERR_CHUNK:
     void;
   default:
     uint32               rdma_extra[8];
};
```

5. Long Messages

The receiver of RDMA Send messages is required by RDMA to have previously posted one or more adequately sized buffers. The RPC client can inform the server of the maximum size of its RDMA Send messages via the Connection Configuration Protocol described later in this document.

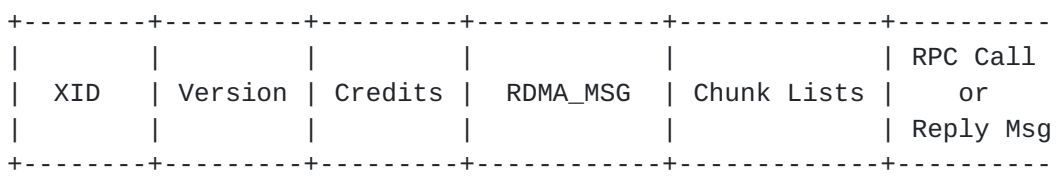
Since RPC messages are frequently small, memory savings can be achieved by posting small buffers. Even large messages like NFS READ or WRITE will be quite small once the chunks are removed from the message. However, there may be large messages that would demand a very large buffer be posted, where the contents of the buffer may not be a chunkable XDR element. A good example is an NFS READDIR reply which may contain a large number of small filename strings. Also, the NFS version 4 protocol [RFC3530] features COMPOUND request and reply messages of unbounded length.

Ideally, each upper layer will negotiate these limits. However, it is frequently necessary to provide a transparent solution.

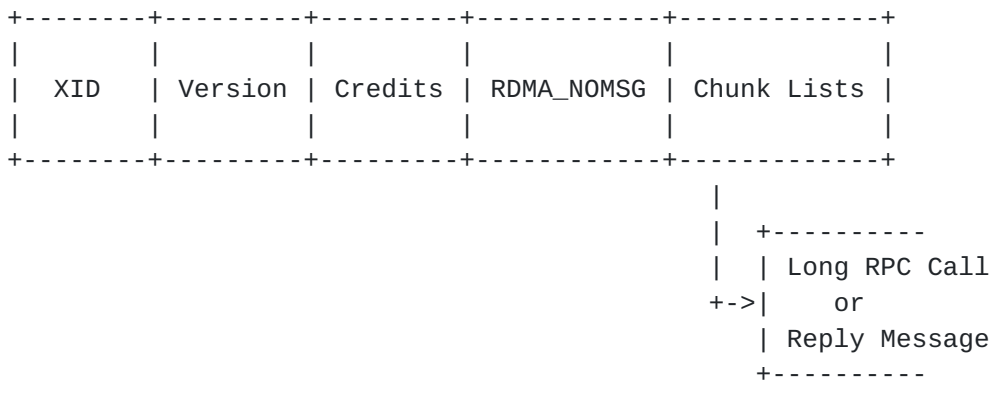
5.1. Message as an RDMA Read Chunk

One relatively simple method is to have the client identify any RPC message that exceeds the RPC server's posted buffer size and move it separately as a chunk, i.e., reference it as the first entry in the read chunk list with an XDR position of zero.

Normal Message



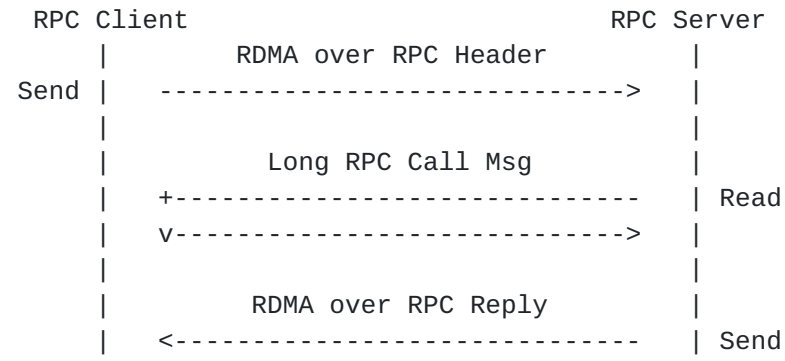
Long Message



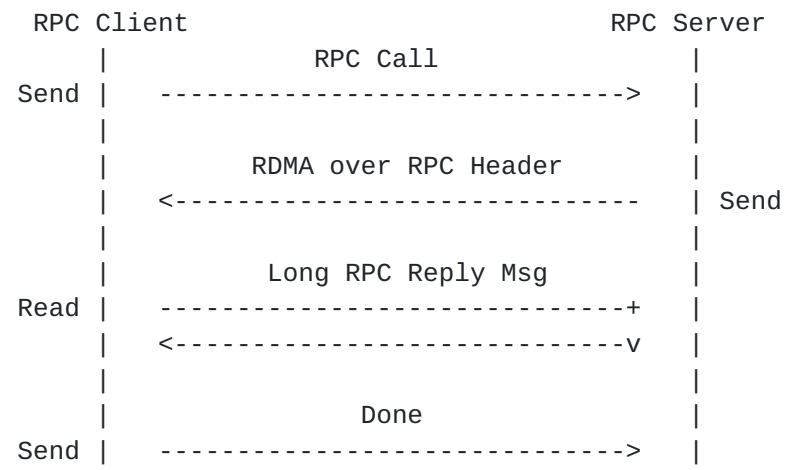
If the receiver gets an RPC over RDMA header with a message type of RDMA_NOMSG and finds an initial read chunk list entry with a zero XDR position, it allocates a registered buffer and issues an RDMA Read of the long RPC message into it. The receiver then proceeds to XDR decode the RPC message as if it had received it inline with the Send data. Further decoding may issue additional RDMA Reads to bring over additional chunks.

Although the handling of long messages requires one extra network turnaround, in practice these messages will be rare if the posted receive buffers are correctly sized, and of course they will be non-existent for RDMA-aware upper layers.

A long call RPC with request supplied via RDMA Read



An RPC with long reply returned via RDMA Read



It is possible for a single RPC procedure to employ both a long call for its arguments, and a long reply for its results. However, such an operation is atypical, as few upper layers define such exchanges.

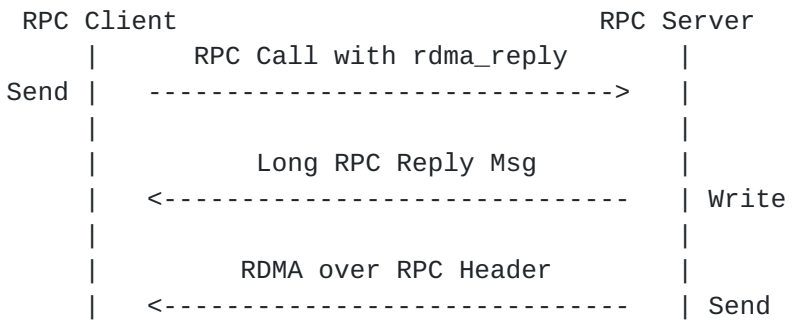
5.2. RDMA Write of Long Replies (Reply Chunks)

A superior method of handling long RPC replies is to have the RPC client post a large buffer into which the server can write a large RPC reply. This has the advantage that an RDMA Write may be slightly faster in network latency than an RDMA Read, and does not require the server to wait for the completion as it must for RDMA Read. Additionally, for a reply it removes the need for an RDMA_DONE message if the large reply is returned as a Read chunk.

This protocol supports direct return of a large reply via the

inclusion of an OPTIONAL rdma_reply write chunk after the read chunk list and the write chunk list. The client allocates a buffer sized to receive a large reply and enters its steering tag, address and length in the rdma_reply write chunk. If the reply message is too long to return inline with an RDMA Send (exceeds the size of the client's posted receive buffer), even with read chunks removed, then the RPC server performs an RDMA Write of the RPC reply message into the buffer indicated by the rdma_reply chunk. If the client doesn't provide an rdma_reply chunk, or if it's too small, then if the upper layer specification permits, the message MAY be returned as a Read chunk.

An RPC with long reply returned via RDMA Write



The use of RDMA Write to return long replies requires that the client applications anticipate a long reply and have some knowledge of its size so that an adequately sized buffer can be allocated. This is certainly true of NFS READDIR replies; where the client already provides an upper bound on the size of the encoded directory fragment to be returned by the server.

The use of these "reply chunks" is highly efficient and convenient for both RPC client and server. Their use is encouraged for eligible RPC operations such as NFS READDIR, which would otherwise require extensive chunk management within the results or use of RDMA Read and a Done message. [NFSDDP]

6. Connection Configuration Protocol

RDMA Send operations require the receiver to post one or more buffers at the RDMA connection endpoint, each large enough to receive the largest Send message. Buffers are consumed as Send messages are received. If a buffer is too small, or if there are no buffers posted, the RDMA transport MAY return an error and break the RDMA connection. The receiver MUST post sufficient, adequately buffers to avoid buffer overrun or capacity errors.

The protocol described above includes only a mechanism for managing the number of such receive buffers, and no explicit features to allow the RPC client and server to provision or control buffer sizing, nor any other session parameters.

In the past, this type of connection management has not been necessary for RPC. RPC over UDP or TCP does not have a protocol to negotiate the link. The server can get a rough idea of the maximum size of messages from the server protocol code. However, a protocol to negotiate transport features on a more dynamic basis is desirable.

The Connection Configuration Protocol allows the client to pass its connection requirements to the server, and allows the server to inform the client of its connection limits.

Use of the Connection Configuration Protocol by an upper layer is OPTIONAL.

6.1. Initial Connection State

This protocol MAY be used for connection setup prior to the use of another RPC protocol that uses the RDMA transport. It operates in-band, i.e., it uses the connection itself to negotiate the connection parameters. To provide a basis for connection negotiation, the connection is assumed to provide a basic level of interoperability: the ability to exchange at least one RPC message at a time that is at least 1 KB in size. The server MAY exceed this basic level of configuration, but the client MUST NOT assume more than one, and MUST receive a valid reply from the server carrying the actual number of available receive messages, prior to sending its next request.

6.2. Protocol Description

Version 1 of the Connection Configuration protocol consists of a single procedure that allows the client to inform the server of its connection requirements and the server to return connection information to the client.

The maxcall_sendsize argument is the maximum size of an RPC call message that the client MAY send inline in an RDMA Send message to the server. The server MAY return a maxcall_sendsize value that is smaller or larger than the client's request. The client MUST NOT send an inline call message larger than what the server will accept. The maxcall_sendsize limits only the size of inline RPC calls. It does not limit the size of long RPC messages transferred as an initial chunk in the Read chunk list.

The maxreply_sendsize is the maximum size of an inline RPC message that the client will accept from the server.

The maxrdmaread is the maximum number of RDMA Reads which may be active at the peer. This number correlates to the RDMA incoming RDMA Read count ("IRD") configured into each originating endpoint by the client or server. If more than this number of RDMA Read operations by the connected peer are issued simultaneously, connection loss or suboptimal flow control may result, therefore the value SHOULD be observed at all times. The peers' values need not be equal. If zero, the peer MUST NOT issue requests which require RDMA Read to satisfy, as no transfer will be possible.

The align value is the value recommended by the server for opaque data values such as strings and counted byte arrays. The client MAY use this value to compute the number of prepended pad bytes when XDR encoding opaque values in the RPC call message.

```
typedef unsigned int uint32;

struct config_rdma_req {
     uint32  maxcall_sendsize;
                 /* max size of inline RPC call */
     uint32  maxreply_sendsize;
                 /* max size of inline RPC reply */
     uint32  maxrdmaread;
                 /* max active RDMA Reads at client */
};

struct config_rdma_reply {
     uint32  maxcall_sendsize;
                 /* max call size accepted by server */
     uint32  align;
                 /* server's receive buffer alignment */
     uint32  maxrdmaread;
                 /* max active RDMA Reads at server */
};

program CONFIG_RDMA_PROG {
   version VERS1 {
      /*
       * Config call/reply
       */
      config_rdma_reply CONF_RDMA(config_rdma_req) = 1;
   } = 1;
} = 100417;
```

7. Memory Registration Overhead

RDMA requires that all data be transferred between registered memory regions at the source and destination. All protocol headers as well as separately transferred data chunks use registered memory. Since the cost of registering and de-registering memory can be a large proportion of the RDMA transaction cost, it is important to minimize registration activity. This is easily achieved within RPC controlled memory by allocating chunk list data and RPC headers in a reusable way from pre-registered pools.

The data chunks transferred via RDMA MAY occupy memory that persists outside the bounds of the RPC transaction. Hence, the default behavior of an RPC over RDMA transport is to register and de-register these chunks on every transaction. However, this is not a limitation of the protocol - only of the existing local RPC API. The API is easily extended through such functions as rpc_control(3) to change the default behavior so that the application can assume responsibility for controlling memory registration through an RPC-provided registered memory allocator.

8. Errors and Error Recovery

RPC RDMA protocol errors are described in section 4. RPC errors and RPC error recovery are not affected by the protocol, and proceed as for any RPC error condition. RDMA Transport error reporting and recovery are outside the scope of this protocol.

It is assumed that the link itself will provide some degree of error detection and retransmission. iWARP's MPA layer (when used over TCP), SCTP, as well as the Infiniband link layer all provide CRC protection of the RDMA payload, and CRC-class protection is a general attribute of such transports. Additionally, the RPC layer itself can accept errors from the link level and recover via retransmission. RPC recovery can handle complete loss and re-establishment of the link.

See section 11 for further discussion of the use of RPC-level integrity schemes to detect errors, and related efficiency issues.

9. Node Addressing

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

10. RPC Binding

RPC services normally register with a portmap or rpcbind [RFC1833] service, which associates an RPC program number with a service address. (In the case of UDP or TCP, the service address for NFS is normally port 2049.) This policy is no different with RDMA interconnects, although it may require the allocation of port numbers appropriate to each upper layer binding which uses the RPC framing defined here.

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

When mapped atop Infiniband [IB], which uses a GID-based service endpoint naming scheme, a translation MUST be employed. One such translation is defined in the Infiniband Port Addressing Annex [IBPORT], which is appropriate for translating IP port addressing to the Infiniband network. Therefore, in this case, IP port addressing may be readily employed by the upper layer.

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

One possibility is to have an upper layer server register its mapped IP port with the rpcbind service, under the netid (or netid's) defined here. An RPC/RDMA-aware client can then resolve its desired service to a mappable port, and proceed to connect. This is the most flexible and compatible approach, for those upper layers which are defined to use the rpcbind service.

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

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

Historically, different RPC protocols have taken different approaches to their port assignment, therefore the specific method is left to each RPC/RDMA-enabled upper layer binding, and not addressed here.

This specification defines two new "netid" values, to be used for registration of upper layers atop iWARP [RFC5040, RFC5041] and (when a suitable port translation service is available) Infiniband [IB] in section 12, "IANA Considerations." Additional RDMA-capable networks MAY define their own netids, or if they provide a port translation, MAY share the one defined here.

11. Security Considerations

RPC provides its own security via the RPCSEC_GSS framework [RFC2203]. RPCSEC_GSS can provide message authentication, integrity checking, and privacy. This security mechanism will be unaffected by the RDMA transport. The data integrity and privacy features alter the body of the message, presenting it as a single chunk. For large messages the chunk may be large enough to qualify for RDMA Read transfer. However, there is much data movement associated with computation and verification of integrity, or encryption/decryption, so certain performance advantages may be lost.

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 [RPCSECGSSV2] in conjunction with the Channel Binding mechanism [RFC5056] and IPsec Channel Connection Latching [BTNSLATCH]. Use of such mechanisms is REQUIRED where integrity and/or privacy is desired, and where efficiency is required.

An additional consideration is the protection of the integrity and privacy of local memory by the RDMA transport itself. The use of RDMA by RPC MUST NOT introduce any vulnerabilities to system memory contents, or to memory owned by user processes. These protections are provided by the RDMA layer specifications, and specifically their security models. It is REQUIRED that any RDMA provider used for RPC transport be conformant to the requirements of [RFC5042] in order to satisfy these protections.

Once delivered securely by the RDMA provider, any RDMA-exposed addresses will contain only RPC payloads in the chunk lists, transferred under the protection of RPCSEC_GSS integrity and privacy. By these means, the data will be protected end-to-end, as

required by the RPC layer security model.

Where upper layer protocols choose to supply results to the requester via Read chunks, a server resource deficit can arise if the client does not promptly acknowledge their status via the RDMA_DONE message. This can potentially lead to a denial of service situation, with a single client unfairly (and unnecessarily) consuming server RDMA resources. Servers for such upper layer protocols MUST protect against this situation, originating from one or many clients. For example, a time-based window of buffer availability may be offered, if the client fails to obtain the data within the window, it will simply retry using ordinary RPC retry semantics. Or, a more severe method would be for the server to simply close the client's RDMA connection, freeing the RDMA resources and allowing the server to reclaim them.

A fairer and more useful method is provided by the protocol itself. The server MAY use the rdma_credit value to limit the number of outstanding requests for each client. By including the number of outstanding RDMA_DONE completions in the computation of available client credits, the server can limit its exposure to each client, and therefore provide uninterrupted service as its resources permit.

However, the server must ensure that it does not decrease the credit count to zero with this method, since the RDMA_DONE message is not acknowledged. If the credit count were to drop to zero solely due to outstanding RDMA_DONE messages, the client would deadlock since it would never obtain a new credit with which to continue. Therefore, if the server adjusts credits to zero for outstanding RDMA_DONE, it MUST withhold its reply to at least one message in order to provide the next credit. The time-based window (or any other appropriate method) SHOULD be used by the server to recover resources in the event that the client never returns.

The "Connection Configuration Protocol", when used, MUST be protected by an appropriate RPC security flavor, to ensure it is not attacked in the process of initiating an RPC/RDMA connection.

12. IANA Considerations

Three new assignments are specified by this document:

- A new set of RPC "netids" for resolving RPC/RDMA services

- Optional service port assignments for upper layer bindings

- An RPC program number assignment for the configuration protocol

These assignments have been established, as below.

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

The following "nc_proto" registry strings are defined for this purpose:

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

These netids MAY be used for any RDMA network satisfying the requirements of section 2, and able to identify service endpoints using IP port addressing, possibly through use of a translation service as described above in section 10, RPC Binding. The "rdma" netid is to be used when IPv4 addressing is employed by the underlying transport, and "rdma6" for IPv6 addressing.

The netid assignment policy and registry are defined in [IANA-NETID].

As a new RPC transport, this protocol has no effect on RPC program numbers or existing registered port numbers. However, new port numbers MAY be registered for use by RPC/RDMA-enabled services, as appropriate to the new networks over which the services will operate.

For example, the NFS/RDMA service defined in [NFSDDP] has been assigned the port 20049, in the IANA registry:

```
nfsrdma 20049/tcp Network File System (NFS) over RDMA
nfsrdma 20049/udp Network File System (NFS) over RDMA
nfsrdma 20049/sctp Network File System (NFS) over RDMA
```

The OPTIONAL Connection Configuration protocol described herein requires an RPC program number assignment. The value "100417" has been assigned:

rdmaconfig 100417 rpc.rdmaconfig

The RPC program number assignment policy and registry are defined in [RFC1831bis].

13. Acknowledgments

The authors wish to thank Rob Thurlow, John Howard, Chet Juszczak, Alex Chiu, Peter Staubach, Dave Noveck, Brian Pawlowski, Steve Kleiman, Mike Eisler, Mark Wittle, Shantanu Mehendale, David Robinson and Mallikarjun Chadalapaka for their contributions to this document.

14. Normative References

[RFC2119]
S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", Best Current Practice, BCP 14, RFC 2119, March 1997.

[RFC1831bis]
R. Thurlow, Ed., "RPC: Remote Procedure Call Protocol Specification Version 2", Internet Draft Work in Progress, draft-ietf-nfsv4-rfc1831bis

[RFC4506]
M. Eisler Ed., "XDR: External Data Representation Standard", Standards Track RFC, http://www.ietf.org/rfc/rfc4506.txt

[RFC1833]
R. Srinivasan, "Binding Protocols for ONC RPC Version 2", Standards Track RFC, http://www.ietf.org/rfc/rfc1833.txt

[RFC2203]
M. Eisler, A. Chiu, L. Ling, "RPCSEC_GSS Protocol Specification", Standards Track RFC, http://www.ietf.org/rfc/rfc2203.txt

[RPCSECGSSV2]
M. Eisler, "RPCSEC_GSS Version 2", Internet Draft Work in Progress, draft-ietf-nfsv4-rpcsec-gss-v2

[RFC5056]
N. Williams, "On the Use of Channel Bindings to Secure Channels", Standards Track RFC
http://www.ietf.org/rfc/rfc5056.txt

[BTNSLATCH]
N. Williams, "IPsec Channels: Connection Latching", Internet Draft Work in Progress, draft-ietf-btns-connection-latching

[RFC5042]
J. Pinkerton, E. Deleganes, "Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security", Standards Track RFC, http://www.ietf.org/rfc/rfc5042.txt

[IANA-NETID]
M. Eisler, "IANA Considerations for RPC Net Identifiers and Universal Address Formats", Internet Draft Work in Progress, draft-ietf-nfsv4-rpc-netid

15. Informative References

[RFC1094]
Sun Microsystems, "NFS: Network File System Protocol Specification", (NFS version 2) Informational RFC, http://www.ietf.org/rfc/rfc1094.txt

[RFC1813]
B. Callaghan, B. Pawlowski, P. Staubach, "NFS Version 3 Protocol Specification", Informational RFC, http://www.ietf.org/rfc/rfc1813.txt

[RFC3530]
S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, "NFS version 4 Protocol", Standards Track RFC, http://www.ietf.org/rfc/rfc3530.txt

[NFSDDP]
B. Callaghan, T. Talpey, "NFS Direct Data Placement", Internet Draft Work in Progress, draft-ietf-nfsv4-nfsdirect

[RFC5040]
R. Recio et al., "A Remote Direct Memory Access Protocol Specification", Standards Track RFC, http://www.ietf.org/rfc/rfc5040.txt

[RFC5041]
H. Shah et al., "Direct Data Placement over Reliable Transports", Standards Track RFC, http://www.ietf.org/rfc/rfc5041.txt

[NFSRDMAPS]
T. Talpey, C. Juszczak, "NFS RDMA Problem Statement", Internet Draft Work in Progress, draft-ietf-nfsv4-nfs-rdma-problem-statement

[NFSv4.1]
S. Shepler et al., ed., "NFSv4 Minor Version 1", Internet Draft Work in Progress, draft-ietf-nfsv4-minorversion1

[IB]
Infiniband Architecture Specification, available from http://www.infinibandta.org

[IBPORT]
Infiniband Trade Association, "IP Addressing Annex", available from http://www.infinibandta.org

Authors' Addresses

Tom Talpey
Network Appliance, Inc.
1601 Trapelo Road, #16
Waltham, MA 02451 USA

Phone: +1 781 768 5329
EMail: thomas.talpey@netapp.com

Brent Callaghan
Apple Computer, Inc.
MS: 302-4K
2 Infinite Loop
Cupertino, CA 95014 USA

EMail: brentc@apple.com

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement