RPC-over-RDMA Version Two

Chuck Lever
<chuck.lever@oracle.com>

The High-order Bit

- This presentation does not propose new features, but does suggest changes to existing protocol elements.

- The I-D authors have striven to minimize on-the-wire changes to the RPC/RDMA version 2 protocol.

- Will an RPC/RDMA version 2 protocol with significant on-the-wire changes be embraced or ignored by implementers?

Implementation Experience

Preparing for Version 2

- Linux NFS server prototype converts chunk lists to an internal representation:
  - For more robust *input validation*
  - To make the bulk of the transport implementation *agnostic* to on-the-wire chunk format
  - To handle *multiple chunks* per chunk list
- Handles multiple Write chunks in a Write list. Pushes them from ULP XDR encoders without holding the transport send mutex.

Read Chunk Improvements

- RPC/RDMA version 2 now:
  - Forbids a position-zero Read chunk to appear in an RDMA_MSG type Call.
  - Requires an RDMA_NOMSG type Call to have a position-zero Read chunk.
  - Requires the client to pre-sort the Read list by position.

Overlapping Read Chunks

- Chunk overlap :: Assuming the Read list is sorted by position, the starting position and length of the NTH chunk in the Read list cause some of its content to fall after the starting position of the N+1TH chunk in the list.

- Chunk overlap can only occur when there is more than one normal Read chunk in the Read list.

- There is no protocol solution yet to prevent chunk overlap. Responders have to check ingress Read lists and throw an error when overlap is detected.

Over-sized Read Chunks

- A malicious or broken requester can create a Read chunk that asks the responder's RNIC to pull an enormous amount of data, resulting in a DoS. Responder ULP implementation must set a sane limit on chunk size.

- A similar issue does not exist for Write chunks:

  - The responder uses only as much of the Write chunk as it needs.

  - Hardware memory registration limits how much data the responder can write into the requester's memory.

Chunk List Parsing

- Write list parsing is efficient:
  - Each chunk's segments appear in a counted array.
  - List is always in order.
- Read list parsing is not efficient:
  - Receivers need to walk the list multiple times to count how many Read chunks and segments appear.
  - Segment position values don't have to be monotonic.

Pulling Chunks in XDR Decoders

- The original plan for RPC/RDMA version 1 was to have ULP XDR decoders pull Read chunks. This is not always feasible:
  - NFS servers may checksum a portion of ingress RPC messages to detect and avoid processing replayed Calls.
  - Position-zero Read chunks span XDR data items and therefore must be pulled by the transport, not by ULP XDR decoders.

Vestigial Reply Read Lists

- RPC/RDMA version 2 still requires a Read list to appear in a Reply, even though it's always empty. Do we want to continue to dream of using a Reply Read list someday?

- What if a Responder sends a Reply message that has both a Read list and a Reply chunk? The Reply chunk requires NOMSG, but a Reply Read list cannot have a PZRC.

- Allowing the Read list to appear in a Reply appears to be cumbersome at this point.

Wacky Ideas

Chunks On-the-wire

- Possible simplification: have a single on-the-wire chunk format.

- Except for the position field, both types of chunk carry the same information.

- Instead of different Read and Write chunk formats, can we replace Read chunks / segments with Write chunks by adding a position field to the Write chunk?

Whither PZRCs?

- Possible simplification: replace the Position-zero Read chunk.

- A “Call chunk” could work like a Reply chunk.

- Or, have one special “body chunk” that could be used for the RPC message body in both Calls or Replies.

  - Body chunks are always handled by the transport, not an XDR decoder.

Replace RDMA2_MSG?

- Instead, have distinct header types for Call messages and Reply messages, and distinct header types for handling message continuation.

- Simpler sender and receiver processing.

- The rdma2_flags field would no longer necessary.

- Some header types could leave out chunk lists, making more room for inline payload content or other header information.

RPC Call Messages

- Call_Last: Call with an inline body, actual arguments, provisioned results. Would also mean "last Send in message chain”. This would work like today’s RDMA2_MSG, but only for Calls.

- Call_Middle: Call with continuation, no chunk lists. All RPC message content is inline.

- Call_External: Call with a chunk body, no inline content. This would be like today’s RDMA2_NOMSG, but only for Calls.

- Last and External may carry provisional Write/Reply chunks.

RPC Reply Messages

- None of these would carry a Read list or provisioned but unused chunks:
  - Reply_Last: Reply with an inline body, actual results, and no Reply chunk. Would also mean "last Send in chain". This would work like today's RDMA2_MSG, but only for Replies.
  - Reply_Middle: Reply with continuation and no chunks. All RPC message content is inline.
  - Reply_External: Reply with a chunk body, no inline content. This would be like today's RDMA2_NOMSG, but only for Replies.

Message Continuation

- Last always terminates a sequence of Middles.

  - To send an RPC message whose inline body fits under the inline threshold, the sender would use a single Last.

  - To send an RPC message between 8KB and 12KB, it would be put on the wire with a sequence like Middle-Middle-Last (empty chunk lists).

  - That also works for an RPC message whose body is larger than the inline threshold but carries one or more chunks. So, Middle-Middle-Last (with populated chunk lists).

Control Plane Messages

- None of these header types need to have chunks:
  - Error response
  - Connprop_Last
  - Connprop_Middle
  - Asynchronous credit grant

Prototyping Next Steps

- Milestone states document delivery by December 2020. These as-yet-unprototyped features still feel risky to me:
  - Transport protocol version negotiation
  - The new credit management mechanism
  - Connection properties
  - Host authentication
  - Message continuation