

NFSv4
Internet-Draft
Updates: 8881, 7530 (if approved)
Intended status: Standards Track
Expires: 12 January 2022

D. Noveck, Ed.
NetApp
11 July 2021

Security for the NFSv4 Protocols
draft-dnoveck-nfsv4-security-00

Abstract

This document describes the core security features of the NFSv4 family of protocols, applying to all minor versions. The discussion includes the use of security features provided at the RPC transport level.

This preliminary version of the document, is intended, in large part, to result in working group discussion regarding NFSv4 security issues and identify issues on which the working group needs to work on achieving consensus.

When published as an RFC, it will supersede the description of security appearing in existing minor version specification documents such as RFC 7530 and RFC 8881.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 12 January 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Overview	4
2. Requirements Language	5
2.1. Keyword Definitions	5
2.2. Special Considerations	5
3. Introduction	6
3.1. Handling of Multiple Minor Versions	6
3.2. Handling of Minor-version-specific features	7
3.3. Transport-based Security Features	9
4. Authorization in General	10
5. User-based File Access Authorization	11
5.1. Handling of Multiple Parallel File Access Authorization Models	11
5.2. Attributes for User-based File Access Authorization . . .	13
5.3. Posix Authorization Model	14
5.3.1. Attribute 33: mode	14
5.3.2. V4.1 Attribute 74: mode_set_masked	15
5.4. ACL-based Authorization Model	15
5.4.1. Access control Entries	15
5.4.2. ACE Type	17
5.4.3. ACE Access Mask	19
5.4.4. Details Regarding Mask Bits	20
5.4.5. ACE4_DELETE vs. ACE4_DELETE_CHILD	27
5.4.6. ACE flag	27
5.4.7. Details Regarding ACE Flag Bits	28
5.4.8. ACE Who	29

5.4.9. Automatic Inheritance Features	31
5.5. Attribute 12: acl	33
5.6. Attribute 13: aclsupport	34
5.7. V4.1 Attribute 58: dacl	34
5.8. V4.1 Attribute 59: sacl	35
6. Common Considerations for Both File access Models	35
6.1. Server Considerations	35
6.2. Client Considerations	36
7. Combining Authorization Models	37
7.1. Combined Authorization Models for V4.0	37
7.2. Combined Authorization Model for V4.1 and Beyond	37
7.2.1. Computing a Mode Attribute from an ACL	38
7.2.2. Alternatives in Computing Mode Bits	39
7.2.3. Setting Multiple ACL Attributes	39
7.2.4. Setting Mode and not ACL	40
7.2.5. Setting ACL and Not Mode	41
7.2.6. Setting Both ACL and Mode	41
7.2.7. Retrieving the Mode and/or ACL Attributes	41
7.2.8. Creating New Objects	41
7.2.9. Use of Inherited ACL When Creating Objects	43
7.3. Combined Authorization Models for V4.2	43
8. Labelled NFS Authorization Model	43
9. State Modification Authorization	44
10. Identification and Authentication	45
10.1. Identification vs. Authentication	45
10.2. Items to be Identified	45
10.3. Authentication Provided by specific RPC Flavors	47
10.4. Authentication Provided by the RPC Transport	47
11. Security of Data in Flight	47
11.1. Data Security Provided by the Flavor-associated Services	47
11.2. Data Security Provided by the RPC Transport	48
12. Security Negotiation	48
12.1. Flavors and Pseudo-flavors	48
12.2. Negotiation of Security Flavors and Mechanisms	50
12.3. Negotiation of RPC Transports and Characteristics	50
12.4. Overall Interpretation of SECINFO Response Arrays	51
12.5. SECINFO	52
12.5.4. SECINFO IMPLEMENTATION (general)	54
12.5.4.1. SECINFO IMPLEMENTATION (for v4.0)	55
12.5.4.2. SECINFO IMPLEMENTATION (for v4.1 and v4.2)	55
12.6. Future Security Needs	56
13. Security Considerations	57
13.1. Changes in Security Considerations	57
13.1.1. Wider View of Threats	57
13.1.2. Transport-layer Security Facilities	58
13.1.3. Compatibility and Maturity Issues	58
13.1.4. Discussion of AUTHSYS	59

13.2.	Security Considerations Scope	60
13.2.1.	Discussion of Potential Classification of Environmets	60
13.2.2.	Discussion of Environments	60
13.3.	Major New Recommendations	61
13.3.1.	Recommendations Regarding Security of Data in Flight	61
13.3.2.	Recommendations Regarding Client Peer Authentication	61
13.3.3.	Issues Regarding Valid Reasons to Bypass Recommendations	62
13.4.	Data Security Threats	62
13.5.	Authentication-based threats	62
13.5.1.	Attacks based on the use of AUTH_SYS	62
13.5.2.	Attacks on Name/Userid Mapping Facilities	62
13.6.	Disruption and Denial-of-Service Attacks	63
13.6.1.	Attacks Based on the Disruption of Client-Server Shared State	63
13.6.2.	Attacks Based on Forcing the Misuse of Server Resources	63
14.	IANA Considerations	63
14.1.	New Authstat Values	63
14.2.	New Authentication Pseudo-Flavors	63
15.	References	63
15.1.	Normative References	64
15.2.	Informative References	65
Appendix A.	Acknowledgments	65
Author's Address	65

1. Overview

This document presents a new approach to security for the NFSv4 protocols, which, although building on previous treatments of these issues, diverges substantially from them in a number of respects.

A new treatment is necessary because:

- * Previous treatments paid insufficient attention to security issues regarding data in flight.
- * The presentation of AUTH_SYS as an "'OPTIONAL' means of authentication" obscured the severe security problems that come with its use.
- * The security considerations sections of existing minor version specifications contain no threat analyses and focus on particular security issues in a way that obscures, rather than clarifying, the security issues that need to be addressed.

- * The availability of RPC-with-TLS (described in [12]) provides facilities that NFSv4 clients and servers will need to use to provide security for data in flight and mitigate the lack of authentication when AUTH_SYS is used.

This preliminary document contains many notes with headers in brackets, requesting comments regarding confusing or otherwise dubious passages in existing documents and other choices that need to be made. Comments and working group discussion of those notes will be important in arriving at an adequate RFC candidate.

2. Requirements Language

2.1. Keyword Definitions

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 specified in BCP 14 [1] [5] when, and only when, they appear in all capitals, as shown here.

2.2. Special Considerations

Because this document needs to revise previous treatments of its subject, it will need to cite previous treatments of issues that now need to be dealt with in a different way. This will take the form of quotations from documents whose treatment of the subject is being obsoleted, most often direct but sometimes indirect as well.

Such treatments in quotations will involve use of these BCP14-defined terms in two noteworthy ways:

- * The term may have been used inappropriately (i.e not in accord with [1]), as has been the case for the "RECOMMENDED" attributes, which are in fact OPTIONAL.

In such cases, the surrounding text will make clear that the quoted text has no normative effect.

Some specific issues relating to this case are described below Section 5.2.

- * The term may have been used in accord with [1], although the resulting normative statement is now felt to be inappropriate.

In such cases, the surrounding text will need to make clear that the text quoted is no longer to be considered normative, often by providing new text that conflicts with the quoted, previously normative, text.

An important instance of this situation is the description of AUTH_SYS as an "OPTIONAL" means of authentication. For detailed discussion of this case, see Sections 10 and 13.1.4

3. Introduction

Because the basic approach to security issues is so similar for all minor versions, this document applies to all NFSv4 minor versions. The details of this transition are discussed in Sections 3.1 and 3.2

This document is able to take a new approach to security issues because of the work that has been done to provide security features at the transport level, rather than providing security features only by making choices with regard to the authentication flavor and associated mechanisms and services. The effect of this shift is summarized in Section 3.3.

3.1. Handling of Multiple Minor Versions

In some cases, there are differences between minor versions in that there are security-related features, not present in all minor versions.

To deal with this issue, this document will focus on a few major areas listed below which are common to all minor versions.

- * File access authorization (discussed in Section 5) is the same in all minor versions together with the identification/ authentication infrastructure supporting it (discussed in Section 10) provided by RPC and applying to all of NFS.

An exception is made regarding labelled NFS, an optional feature within NFSv4.2, described in [10]. This is discussed as a version-specific feature in this document in Section 8

- * Features to secure data in-flight, all provided by RPC, together with the negotiation infrastructure to support them are common to all NFSv4 minor versions, are discussed in Section 12

However, the use of SECINFO_NONAME, together with changes needed for transport level encryption, paralleling those proposed here for SECINFO, is treated as a version-specific feature and, while mentioned here, will be fully documented in new v4.1 specification documents.

- * The Protection of state data from unauthorized modification is discussed in Section 9) is the same in all minor versions together with the identification/ authentication infrastructure supporting it (discussed in Section 10) provided by secure transports such as RPC-over-TLS.

It should be noted that state protection based on RPCSEC_GSS is treated as a version-specific feature and will continue to be described by [8] or its successors. Also, it needs to be noted that the use of state protection is not discussed in [6].

3.2. Handling of Minor-version-specific features

There are a number of areas in which security features differ among minor versions, as discussed below. In some cases, a new feature requires specific security support while in others one version will have a new feature related to enhancing the security infrastructure.

How such features are dealt with in this document depends on the specific feature.

[Working group discussion advisable]: There's a lot of opportunities to make mistakes. I'd prefer to find out sooner rather than later.

- * In addition to SECINFO, whose enhanced description appears in this document, NFSv4.1 added a new SECINFO_NONAME operation, useful for pNFS file as well as having some non-pNFS uses.

While the enhanced description of SECINFO mentions SECINFO_NONAME, this is handled as one of a number of cases in which the description has to indicate that different actions need to be taken for different minor versions.

The definitive description of SECINFO_NONAME, now appearing in RFC8881 needs to be modified to match the description of SECINFO appearing in this document. It is expected that this will be done as part of the rfc5661bis process.

The security implications of the security negotiation facilities as a whole will be addressed in the security considerations section of this document.

- * The pNFS optional feature added in NFSv4.1 has its own security needs which parallel closely those of non-pNFS access. As a result, these needs and the means to satisfy them are not discussed in this document.

The definitive description of pNFS security will remain in RFC8881 and its successors (i.e. the rfc5661bis document suite). However, because pNFS security relies heavily on the infrastructure discussed here, it is anticipated that the new treatment of pNFS security will deal with many matters by referencing the overall NFS security document.

The security considerations section of rfc5661bis will deal with pNFS security issues.

- * In addition to the state protection facilities described in this document, NFS has another set of such facilities that are not usable in NFSv4.0.

While this document will discuss the security implications of protection against state modification, it will not discuss the details of the v4.1-specific features to accomplish it.

- * The additional v4.1 acl attributes, sacl and dacl, are mentioned in this document, but do not affect the treatment of security, since the acl entries, whether in the acl attribute or separated into sacl and dacl attributes continue to have the same effect.
- * Nfsv4.1 introduced detailed description of methods of co-ordinating the values of the authorization-related attributes mode and acl. this is in contrast to Nfsv4.0, which while expecting such co-ordination to be provide somehow, did not provide any details, allowing a number of different approaches to meet the goal of providing appropriate co-ordination.

This document will provide separate sections describing each of the approaches, together with a common section describing the goals of providing co-ordination when both are supported.

As a result, this document will override the treatment within RFC8881 and this material will be removed in the rfc5661bis document suite and replaced by a reference to the treatment in the NFSv4 security RFC.

Simiarly, this doument will supersede the corresponding treatment in RFC7530 as well. As a result, v4.0 server implementers reading any successor document will be aware of the possibility of the v4.1 approach even though it is clear that this only one of the possible approaches to this issue. V4.1 client implementers will be made aware of the goals and that there is little that can be relied upon with regard to v4.0 servers' attempts to address those goals.

- * The protocol extension defined in [13], now part of NFSv4.2, is also related to the issue of co-ordination of acl and mode attributes and will be discussed in that context.

Nevertheless, the description in [13] will remain definitive.

- * The the v4.1 attribute set-mode-masked attribute is mentioned together with the other attributes implementing the POSIX authorization model.

Because this attribute. while related to security, does not substantively modify the security properties of the protocol, the full description of this attribute, will continue to be the province of the v4.1 specification proper.

- * There is a brief description of the v4.2 Labelled NFS feature in Section 8. Part of that description discusses the limitations in the description of that feature within [10].

Because of some limitations in the description, it is not possible to provide an appropriate security considerations section for that feature in this document.

As a result, the responsibility to provide an appropriate Security Considerations section remains, unrealized for now, with the NFSv4.2 specification document.

3.3. Transport-based Security Features

There are a number of security-related facilities that can be provided at the transport layer eliminating the need for or or proving support to processing done as part of RPC proper.

These will initially be provided by RPC-over-TLS but similar facilities might be provided on new versions of existing transports or new RPC transports.

- * The transport might provide encryption of requests and replies, eliminating the need for privacy and integrity services to be negotiated later and applied on a per-request basis.

While clients might choose to establish connections with such encryption, servers can establish policies allowing access to certain pieces of the namespace using such transports, or limiting access to those providing privacy, allowing the use of either transport-based encryption or privacy services provided by RPCSEC_GSS.

- * The transport might provide mutual authentication of the client and server peers as part of the establishment of the connection. This authentication is distinct from the mutual authentication of the client user and server peer, implemented within the GSSSEC_RPC framework.

This form of authentication is of particular importance when the server allows the use of the flavors AUTH_SYS and AUTH_NONE, which have no provision for the authentication of the user requesting the operation.

While clients might choose to establish connections with such peer authentication, servers can establish policies limiting access to certain pieces of the namespace without such peer authentication or only allowing it using RPCSEC_GSS.

To enable server policies to be effectively communicated to clients, the security negotiation framework now allows connection characteristics to be specified using pseudo-flavors. See Section 12 for details.

4. Authorization in General

There are three distinct methods of checking whether NFSv4 requests are authorized:

- * The most important method of authorization is used to effect user-based file access control, as described in Section 5.

This requires the identification of the user making the request. Because of the central role of such access control in providing NFSv4 security, server implementations SHOULD NOT use such identifications when they are not authenticated. In this context, valid reasons to do otherwise are limited to the compatibility and maturity issues discussed in Section 13.1.3

- * NFSv4.2, via the labelled NFS feature, provides an additional potential requirement for request authorization.

For reasons made clear in Section 8, there is no realistic possibility of the server using the data defined by existing specifications of this feature to effect request authorization. While it is possible for clients to provide this authorization, the lack of detailed specification makes it impossible to determine the nature of the identification used and whether it can be described as "authentication".

- * Since undesired changes to server-maintained locking state (and, for NFSv4.1, session state) can result in denial of service attacks (see Section 13.6), server implementations SHOULD take steps to prevent unauthorized state changes. This can be done by implementing the state authorization restrictions discussed in Section 9

5. User-based File Access Authorization

5.1. Handling of Multiple Parallel File Access Authorization Models

ACLs and modes represent two well-established models for specifying user-based file access permissions. NFSv4 provides support for either or both depending on the attributes supported by the server:

- * When the attributes mode, owner, owner group are all supported, the posix-based authorization model, described in Section 5.3 can be used.
- * When the acl (or dacl) attribute is supported, the acl based authorization model, described in Section 5.4 can be used.

[Working group discussion needed]: Existing specifications neither require that owner and owner group be supported nor provide useful guidance about how clients might deal with servers that support the acl attribute but do not support owner or owner_group (e.g how such a server might deal with acl containing "OWNER@" or "GROUP="). In addition, the use of acl's to generate a corresponding mode seems pointless in the absence of the owner and owner-group attributes.

[Working group decision needed]: It appears that, despite what RFC7530 and RFC8881 say, owner and owner_group are essentially REQUIRED attributes, but while our successor RFC will update both RFC7530 and RFC8881, that document may not be the place to rectify that mistake. I'm thinking of something like the following:

While formally Recommended (essentially OPTIONAL) attributes, it appears that the owner and owner_group attributes need to be available to support any file access authorization model. As a result, this document will not discuss the possibility of attributes that do not support either of these attributes.

- * When both authorization models can be used, there are difficulties that can arise because the ACL-base model provides finer-grained access control than the POSIX model. Some goals for dealing with these difficulties appear later in this section while more detail on the appropriate handling of this situation, which may depend on the minor version used, appears in Section 7.

The following lists the goals of NFSv4 in supporting multiple authorization models for file access.

- * If a server supports the mode attribute, it should provide the appropriate POSIX semantics to clients that only set and retrieve the mode attribute.

[WG Discussion might be needed]: The above substitutes for the requirement that the server provide "reasonable semantics". Are there suggestions for other approaches to this passage?

- * If a server supports ACL attributes, it should provide reasonable semantics to clients that only set and retrieve those attributes.

[WG Discussion might be needed]: Similarly we need a replacement for "reasonable semantics". I'm supposing "the semantics described in this document" is a reasonable replacement.

- * On servers that support the mode attribute, if ACL attributes have never been set on an object, via inheritance or explicitly, the behavior should be the behavior mandated by POSIX.
- * On servers that support the mode attribute, if the ACL attributes have been previously set on an object, either explicitly or via inheritance:
 - Setting only the mode attribute should effectively control the traditional UNIX-like permissions of read, write, and execute on owner, owner_group, and other.

[Working group discussion needed]: It isn't really clear what the above paragraph means, especially as it governs the handling of aces designating specific users and groups which are not the owner and have no overlap with the owning

While it would be possible to substitute a three-ace acl with one ace for each of owner, group and others, I'm not sure that is what is intended. In particular, I'm unsure why would need a second bullet item if we had a clear statement to that effect.

It is particularly important to resolve this now, since this is part of the goals that apply to all minor versions.

- Setting only the mode attribute should provide reasonable security. For example, setting a mode of 000 should be enough to ensure that future OPEN operations for OPEN4_SHARE_ACCESS_READ or OPEN4_SHARE_ACCESS_WRITE by any principal fail, regardless of a previously existing or inherited ACL.

[Working group discussion needed]: We need some replacement for the subjective first sentence. While the specific example give is unexceptionable, it raises question about other case in which what constitutes "reasonable semantics" might subect to dispute.

- * NFSv4.1 describes specific semantic requirements relating to the interaction of mode and ACL attributes, which do contract the goals listed above. As a result, the handling provided by servers of different minor version might well be different, as discussed below.

In regard to the interaction of the mode and ACL attributes:

- * As discussed in Section 7.1, the specific semanric requirements specified for v4.1 could be adopted by v4.0, although some modification might be necessary to deal with the absence of the mode_set_masked attribute.

V4.0 clients would have to be able to interact with servers that different approaches to these goals.

- * V4.1 servers have specfic rules for handling these issues as discussed in Section 7.2. However, despite this, the rules are, in many places, fairly loosely drawn and allow a range of server behaviors.
- * To deal with issues related to the perceived overuse of modes derived from the umask in vitiating needed acl inheritance, a V4.2 extension was provided, allowing the separation of the application-specfed mode from the umask. This is discussed in Section 7.3

5.2. Attributes for User-based File Access Authorization

NFSv4.1 provides for multiple authentication models, controlled by the support for particular Recommended attributes implemented by the server, as discussed below:

- * The attributes owner, owning_group, and mode enable use of a POSIX-based authorization model, as described in Section 5.3. When all of these attributes are supported, this authorization model can be implemented. When none of these attributes or only a proper subset of them are supported, this attribute model is unavailable.
- * The acl attribute (or the attributes sacl and dacl in v4.1) provide an ACL-based authorization model as described in Section 5.4.

5.3. Posix Authorization Model

5.3.1. Attribute 33: mode

The NFSv4.1 mode attribute is based on the UNIX mode bits. The following bits are defined:

```
const MODE4_SUID = 0x800; /* set user id on execution */
const MODE4_SGID = 0x400; /* set group id on execution */
const MODE4_SVTX = 0x200; /* save text even after use */
const MODE4_RUSR = 0x100; /* read permission: owner */
const MODE4_WUSR = 0x080; /* write permission: owner */
const MODE4_XUSR = 0x040; /* execute permission: owner */
const MODE4_RGRP = 0x020; /* read permission: group */
const MODE4_WGRP = 0x010; /* write permission: group */
const MODE4_XGRP = 0x008; /* execute permission: group */
const MODE4_OTH = 0x004; /* read permission: other */
const MODE4_WOTH = 0x002; /* write permission: other */
const MODE4_XOTH = 0x001; /* execute permission: other */
```

Bits MODE4_RUSR, MODE4_WUSR, and MODE4_XUSR apply to the principal identified in the owner attribute. Bits MODE4_RGRP, MODE4_WGRP, and MODE4_XGRP apply to principals identified in the owner_group attribute but who are not identified in the owner attribute. Bits MODE4_OTH, MODE4_WOTH, and MODE4_XOTH apply to any principal that does not match that in the owner attribute and does not have a group matching that of the owner_group attribute.

Bits within a mode other than those specified above are not defined by this protocol. A server MUST NOT return bits other than those defined above in a GETATTR or REaddir operation, and it MUST return NFS4ERR_INVAL if bits other than those defined above are set in a SETATTR, CREATE, OPEN, VERIFY, or NVERIFY operation.

[Working group input needed]: It is my impression that the three high-order bits do not affect server behavior. Is this so? If it is, should that be stated explicitly? Is there a need to more clearly define what the client does with these or is this somehow out of scope.

5.3.2. V4.1 Attribute 74: mode_set_masked

The mode_set_masked attribute is a write-only attribute that allows individual bits in the mode attribute to be set or reset, without changing others. It allows, for example, the bits MODE4_SUID, MODE4_SGID, and MODE4_SVTX to be modified while leaving unmodified any of the nine low-order mode bits devoted to permissions.

In instances such that none of the nine low-order bits are subject to modification, then neither the acl nor the dacl attribute should be automatically modified as discussed in Sections 7.2.4 and 7.2.6.

The mode_set_masked attribute consists of two words, each in the form of a mode4. The first consists of the value to be applied to the current mode value and the second is a mask. Only bits set to one in the mask word are changed (set or reset) in the file's mode. All other bits in the mode remain unchanged. Bits in the first word that correspond to bits that are zero in the mask are ignored, except that undefined bits are checked for validity and can result in NFS4ERR_INVALID as described below.

The mode_set_masked attribute is only valid in a SETATTR operation. If it is used in a CREATE or OPEN operation, the server MUST return NFS4ERR_INVALID.

Bits not defined as valid in the mode attribute are not valid in either word of the mode_set_masked attribute. The server MUST return NFS4ERR_INVALID if any such bits are set to one in a SETATTR. If the mode and mode_set_masked attributes are both specified in the same SETATTR, the server MUST also return NFS4ERR_INVALID.

5.4. ACL-based Authorization Model

5.4.1. Access control Entries

The attributes acl, sacl (v4.1 only) and dacl (v4.1 only) each contain an array of Access Control Entries (ACEs) that are associated with the file system object. The client can set and get these attributes attribute, the server is responsible for using the ACL-related attributes to perform access control. The client can use the OPEN or ACCESS operations to check access without modifying or explicitly reading data or metadata.

The NFS ACE structure is defined as follows:

```
typedef uint32_t      acetype4;

typedef uint32_t aceflag4;

typedef uint32_t      acemask4;

struct nfsace4 {
    acetype4          type;
    aceflag4          flag;
    acemask4          access_mask;
    utf8str_mixed     who;
};
```

To determine if a request succeeds, the server processes each `nfsace4` entry in turn as ordered in the array. Only ACEs that have a "who" that matches the requester are considered. An ACE is considered to match a given requester if at least one of the following is true:

- * The "who" designates a specific user which is the specific user making the request.
- * The "who" specifies "OWNER@" and the user making the request is the owner of the file.
- * The "who" designates a specific group and which is the specific user making the request is a member of that group.
- * The "who" specifies "GROUP@" and the user making the request is a member of the group owning the file.
- * The "who" specifies "EVERYONE@".
- * The "who" specifies "INTERACTIVE@", "NETWORK@", "DIALUP@", "BATCH@", or "SERVICE@" and the requester, in the judgement of the server, feels that designation appropriately describes the requester.
- * The "who" specifies "ANONYMOUS@" or "AUTHENTICATED@" and the requester's authentication status matches the who, using the definitions in Section 5.4.8

Each ACE is processed until all of the bits of the requester's access have been ALLOWED. Once a bit (see below) has been ALLOWED by an ACCESS_ALLOWED_ACE, it is no longer considered in the processing of later ACEs. If an ACCESS_DENIED_ACE is encountered where the requester's access still has unALLOWED bits in common with the

"access_mask" of the ACE, the request is denied. When the ACL is fully processed, if there are bits in the requester's mask that have not been ALLOWED or DENIED, access is denied.

Unlike the ALLOW and DENY ACE types, the ALARM and AUDIT ACE types do not affect a requester's access, and instead are for triggering events as a result of a requester's access attempt. Therefore, AUDIT and ALARM ACEs are processed only after processing ALLOW and DENY ACEs.

The NFSv4.1 ACL model is quite rich. Some server platforms may provide access-control functionality that goes beyond the UNIX-style mode attribute, but that is not as rich as the NFS ACL model. So that users can take advantage of this more limited functionality, the server may support the acl attributes by mapping between its ACL model and the NFSv4.1 ACL model. Servers must ensure that the ACL they actually store or enforce is at least as strict as the NFSv4 ACL that was set. It is tempting to accomplish this by rejecting any ACL that falls outside the small set that can be represented accurately. However, such an approach can render ACLs unusable without special client-side knowledge of the server's mapping, which defeats the purpose of having a common NFSv4 ACL protocol. Therefore, servers should accept every ACL that they can without compromising security. To help accomplish this, servers may make a special exception, in the case of unsupported permission bits, to the rule that bits not ALLOWED or DENIED by an ACL must be denied. For example, a UNIX-style server might choose to silently allow read attribute permissions even though an ACL does not explicitly allow those permissions. (An ACL that explicitly denies permission to read attributes should still be rejected.)

The situation is complicated by the fact that a server may have multiple modules that enforce ACLs. For example, the enforcement for NFSv4.1 access may be different from, but not weaker than, the enforcement for local access, and both may be different from the enforcement for access through other protocols such as SMB (Server Message Block). So it may be useful for a server to accept an ACL even if not all of its modules are able to support it.

The guiding principle with regard to NFSv4 access is that the server must not accept ACLs that appear to make access to the file more restrictive than it really is.

5.4.2. ACE Type

The constants used for the type field (acetype4) are as follows:

```
const ACE4_ACCESS_ALLOWED_ACE_TYPE      = 0x00000000;  
const ACE4_ACCESS_DENIED_ACE_TYPE       = 0x00000001;  
const ACE4_SYSTEM_AUDIT_ACE_TYPE        = 0x00000002;  
const ACE4_SYSTEM_ALARM_ACE_TYPE        = 0x00000003;
```

Only the ALLOWED and DENIED bits may be used in the dacl attribute,
and only the AUDIT and ALARM bits may be used in the sacl attribute.
All four are permitted in the acl attribute.

Value	Abbreviation	Description
ACE4_ACCESS_ALLOWED_ACE_TYPE	ALLOW	Explicitly grants the access defined in acemask4 to the file or directory.
ACE4_ACCESS_DENIED_ACE_TYPE	DENY	Explicitly denies the access defined in acemask4 to the file or directory.
ACE4_SYSTEM_AUDIT_ACE_TYPE	AUDIT	Log (in a system-dependent way) any access attempt to a file or directory that uses any of the access methods specified in acemask4.
ACE4_SYSTEM_ALARM_ACE_TYPE	ALARM	Generate an alarm (in a system-dependent way) when any access attempt is made to a file or directory for the access methods specified in acemask4.

Table 1

The "Abbreviation" column denotes how the types will be referred to throughout the rest of this section.

5.4.3. ACE Access Mask

The bitmask constants used for the access mask field are as follows:

```
const ACE4_READ_DATA           = 0x00000001;
const ACE4_LIST_DIRECTORY      = 0x00000001;
const ACE4_WRITE_DATA          = 0x00000002;
const ACE4_ADD_FILE            = 0x00000002;
const ACE4_APPEND_DATA         = 0x00000004;
const ACE4_ADD_SUBDIRECTORY    = 0x00000004;
const ACE4_READ_NAMED_ATTRS    = 0x00000008;
const ACE4_WRITE_NAMED_ATTRS   = 0x00000010;
const ACE4_EXECUTE             = 0x00000020;
const ACE4_DELETE_CHILD        = 0x00000040;
const ACE4_READ_ATTRIBUTES     = 0x00000080;
const ACE4_WRITE_ATTRIBUTES    = 0x00000100;
const ACE4_WRITE_RETENTION     = 0x00000200;
const ACE4_WRITE_RETENTION_HOLD = 0x00000400;

const ACE4_DELETE              = 0x00010000;
const ACE4_READ_ACL            = 0x00020000;
const ACE4_WRITE_ACL           = 0x00040000;
const ACE4_WRITE_OWNER         = 0x00080000;
const ACE4_SYNCHRONIZE         = 0x00100000;
```

Note that some masks have coincident values, for example, ACE4_READ_DATA and ACE4_LIST_DIRECTORY. The mask entries ACE4_LIST_DIRECTORY, ACE4_ADD_FILE, and ACE4_ADD_SUBDIRECTORY are intended to be used with directory objects, while ACE4_READ_DATA, ACE4_WRITE_DATA, and ACE4_APPEND_DATA are intended to be used with non-directory objects.

5.4.4. Details Regarding Mask Bits

ACE4_READ_DATA

Operation(s) affected:

READ

OPEN

Discussion:

Permission to read the data of the file.

Servers SHOULD allow a user the ability to read the data of the file when only the ACE4_EXECUTE access mask bit is allowed.

ACE4_LIST_DIRECTORY

Operation(s) affected:

REaddir

Discussion:

Permission to list the contents of a directory.

ACE4_WRITE_DATA

Operation(s) affected:

WRITE

OPEN

SETATTR of size

Discussion:

Permission to modify a file's data.

ACE4_ADD_FILE

Operation(s) affected:

CREATE

LINK

OPEN

RENAME

Discussion:

Permission to add a new file in a directory. The CREATE operation is affected when nfs_ftype4 is NF4LNK, NF4BLK, NF4CHR, NF4SOCK, or NF4FIFO. (NF4DIR is not listed because it is covered by ACE4_ADD_SUBDIRECTORY.) OPEN is affected when used to create a regular file. LINK and RENAME are always affected.

ACE4_APPEND_DATA

Operation(s) affected:

WRITE

OPEN

SETATTR of size

Discussion:

The ability to modify a file's data, but only starting at EOF. This allows for the specification of append-only files, by allowing ACE4_APPEND_DATA and denying ACE4_WRITE_DATA to the same user or group. If a file has an ACL such as the one described above and a WRITE request is made for somewhere other than EOF, the server SHOULD return NFS4ERR_ACCESS.

ACE4_ADD_SUBDIRECTORY**Operation(s) affected:**

CREATE

RENAME

Discussion:

Permission to create a subdirectory in a directory. The CREATE operation is affected when nfs_ftype4 is NF4DIR. The RENAME operation is always affected.

ACE4_READ_NAMED_ATTRS**Operation(s) affected:**

OPENATTR

Discussion:

Permission to read the named attributes of a file or to look up the named attribute directory. OPENATTR is affected when it is not used to create a named attribute directory. This is when 1) createdir is TRUE, but a named attribute directory already exists, or 2) createdir is FALSE.

ACE4_WRITE_NAMED_ATTRS**Operation(s) affected:**

OPENATTR

Discussion:

Permission to write the named attributes of a file or to create a named attribute directory. OPENATTR is affected when it is used to create a named attribute directory. This is when createdir is TRUE and no named attribute directory exists. The ability to check whether or not a named attribute directory exists depends on the ability to look it up; therefore, users also need the ACE4_READ_NAMED_ATTRS permission in order to create a named attribute directory.

ACE4_EXECUTE

Operation(s) affected:

READ

OPEN

REMOVE

RENAME

LINK

CREATE

Discussion:

Permission to execute a file.

Servers SHOULD allow a user the ability to read the data of the file when only the ACE4_EXECUTE access mask bit is allowed. This is because there is no way to execute a file without reading the contents. Though a server may treat ACE4_EXECUTE and ACE4_READ_DATA bits identically when deciding to permit a READ operation, it SHOULD still allow the two bits to be set independently in ACLs, and MUST distinguish between them when replying to ACCESS operations. In particular, servers SHOULD NOT silently turn on one of the two bits when the other is set, as that would make it impossible for the client to correctly enforce the distinction between read and execute permissions.

As an example, following a SETATTR of the following ACL:

nfsuser:ACE4_EXECUTE:ALLOW

A subsequent GETATTR of ACL for that file SHOULD return:

nfsuser:ACE4_EXECUTE:ALLOW

Rather than:

nfsuser:ACE4_EXECUTE/ACE4_READ_DATA:ALLOW

ACE4_EXECUTE

Operation(s) affected:

LOOKUP

Discussion:

Permission to traverse/search a directory.

ACE4_DELETE_CHILD

Operation(s) affected:
REMOVE

RENAME

Discussion:

Permission to delete a file or directory within a directory.
See Section 5.4.5 for information on ACE4_DELETE and
ACE4_DELETE_CHILD interact.

ACE4_READ_ATTRIBUTES

Operation(s) affected:
GETATTR of file system object attributes

VERIFY

NVERIFY

READDIR

Discussion:

The ability to read basic attributes (non-ACLs) of a file. On a UNIX system, basic attributes can be thought of as the stat-level attributes. Allowing this access mask bit would mean that the entity can execute "ls -l" and stat. If a READDIR operation requests attributes, this mask must be allowed for the READDIR to succeed.

ACE4_WRITE_ATTRIBUTES

Operation(s) affected:
SETATTR of time_access_set, time_backup,
time_create, time_modify_set, mimetype, hidden, system

Discussion:

Permission to change the times associated with a file or directory to an arbitrary value. Also permission to change the mimetype, hidden, and system attributes. A user having ACE4_WRITE_DATA or ACE4_WRITE_ATTRIBUTES will be allowed to set the times associated with a file to the current server time.

ACE4_WRITE_RETENTION

Operation(s) affected:
SETATTR of retention_set, retentevt_set.

Discussion:
Permission to modify the durations of event and non-event-based retention. Also permission to enable event and non-event-based retention. A server MAY behave such that setting ACE4_WRITE_ATTRIBUTES allows ACE4_WRITE_RETENTION.

ACE4_WRITE_RETENTION_HOLD

Operation(s) affected:
SETATTR of retention_hold.

Discussion:
Permission to modify the administration retention holds. A server MAY map ACE4_WRITE_ATTRIBUTES to ACE_WRITE_RETENTION_HOLD.

ACE4_DELETE

Operation(s) affected:
REMOVE

Discussion:
Permission to delete the filexs or directory. See Section 5.4.5 for information on ACE4_DELETE and ACE4_DELETE_CHILD interact.

ACE4_READ_ACL

Operation(s) affected:
GETATTR of acl, dacl, or sacl

NVERIFY

VERIFY

Discussion:
Permission to read the ACL.

ACE4_WRITE_ACL

Operation(s) affected:
SETATTR of acl and mode

Discussion:
Permission to write the acl and mode attributes.

ACE4_WRITE_OWNER

Operation(s) affected:

SETATTR of owner and owner_group

Discussion:

Permission to write the owner and owner_group attributes. On UNIX systems, this is the ability to execute chown() and chgrp().

ACE4_SYNCHRONIZE

Operation(s) affected:

NONE

Discussion:

Permission to use the file object as a synchronization primitive for interprocess communication. This permission is not enforced or interpreted by the NFSv4.1 server on behalf of the client.

Typically, the ACE4_SYNCHRONIZE permission is only meaningful on local file systems, i.e., file systems not accessed via NFSv4.1. The reason that the permission bit exists is that some operating environments, such as Windows, use ACE4_SYNCHRONIZE.

For example, if a client copies a file that has ACE4_SYNCHRONIZE set from a local file system to an NFSv4.1 server, and then later copies the file from the NFSv4.1 server to a local file system, it is likely that if ACE4_SYNCHRONIZE was set in the original file, the client will want it set in the second copy. The first copy will not have the permission set unless the NFSv4.1 server has the means to set the ACE4_SYNCHRONIZE bit. The second copy will not have the permission set unless the NFSv4.1 server has the means to retrieve the ACE4_SYNCHRONIZE bit.

Server implementations need not provide the granularity of control that is implied by this list of masks. For example, POSIX-based systems might not distinguish ACE4_APPEND_DATA (the ability to append to a file) from ACE4_WRITE_DATA (the ability to modify existing contents); both masks would be tied to a single "write" permission bit. When such a server returns attributes to the client that contain such masks, it would show ACE4_APPEND_DATA and ACE4_WRITE_DATA if and only if the the write permission bit is enabled.

If a server receives a SETATTR request that it cannot accurately implement, it should err in the direction of more restricted access, except in the previously discussed cases of execute and read. For example, suppose a server cannot distinguish overwriting data from appending new data, as described in the previous paragraph. If a client submits an ALLOW ACE where ACE4_APPEND_DATA is set but ACE4_WRITE_DATA is not (or vice versa), the server should either turn off ACE4_APPEND_DATA or reject the request with NFS4ERR_ATTRNOTSUPP.

5.4.5. ACE4_DELETE vs. ACE4_DELETE_CHILD

Two access mask bits govern the ability to delete a directory entry: ACE4_DELETE on the object itself (the "target") and ACE4_DELETE_CHILD on the containing directory (the "parent").

Many systems also take the "sticky bit" (MODE4_SVTX) on a directory to allow unlink only to a user that owns either the target or the parent; on some such systems the decision also depends on whether the target is writable.

Servers SHOULD allow unlink if either ACE4_DELETE is permitted on the target, or ACE4_DELETE_CHILD is permitted on the parent. (Note that this is true even if the parent or target explicitly denies one of these permissions.)

If the ACLs in question neither explicitly ALLOW nor DENY either of the above, and if MODE4_SVTX is not set on the parent, then the server SHOULD allow the removal if and only if ACE4_ADD_FILE is permitted. In the case where MODE4_SVTX is set, the server may also require the remover to own either the parent or the target, or may require the target to be writable.

This allows servers to support something close to traditional UNIX-like semantics, with ACE4_ADD_FILE taking the place of the write bit.

5.4.6. ACE flag

The bitmask constants used for the flag field are as follows:

```
const ACE4_FILE_INHERIT_ACE           = 0x000000001;
const ACE4_DIRECTORY_INHERIT_ACE      = 0x000000002;
const ACE4_NO_PROPAGATE_INHERIT_ACE   = 0x000000004;
const ACE4_INHERIT_ONLY_ACE           = 0x000000008;
const ACE4_SUCCESSFUL_ACCESS_ACE_FLAG = 0x000000010;
const ACE4_FAILED_ACCESS_ACE_FLAG     = 0x000000020;
const ACE4_IDENTIFIER_GROUP           = 0x000000040;
const ACE4_INHERITED_ACE               = 0x000000080;
```

A server need not support any of these flags. If the server supports flags that are similar to, but not exactly the same as, these flags, the implementation may define a mapping between the protocol-defined flags and the implementation-defined flags.

For example, suppose a client tries to set an ACE with `ACE4_FILE_INHERIT_ACE` set but not `ACE4_DIRECTORY_INHERIT_ACE`. If the server does not support any form of ACL inheritance, the server should reject the request with `NFS4ERR_ATTRNOTSUPP`. If the server supports a single "inherit ACE" flag that applies to both files and directories, the server may reject the request (i.e., requiring the client to set both the file and directory inheritance flags). The server may also accept the request and silently turn on the `ACE4_DIRECTORY_INHERIT_ACE` flag.

5.4.7. Details Regarding ACE Flag Bits

`ACE4_FILE_INHERIT_ACE`

Any non-directory file in any sub-directory will get this ACE inherited.

`ACE4_DIRECTORY_INHERIT_ACE`

Can be placed on a directory and indicates that this ACE should be added to each new directory created.

If this flag is set in an ACE in an ACL attribute to be set on a non-directory file system object, the operation attempting to set the ACL SHOULD fail with `NFS4ERR_ATTRNOTSUPP`.

`ACE4_NO_PROPAGATE_INHERIT_ACE`

Can be placed on a directory. This flag tells the server that inheritance of this ACE should stop at newly created child directories.

`ACE4_INHERIT_ONLY_ACE`

Can be placed on a directory but does not apply to the directory; `ALLOW` and `DENY` ACEs with this bit set do not affect access to the directory, and `AUDIT` and `ALARM` ACEs with this bit set do not trigger log or alarm events. Such ACEs only take effect once they are applied (with this bit cleared) to newly created files and directories as specified by the `ACE4_FILE_INHERIT_ACE` and `ACE4_DIRECTORY_INHERIT_ACE` flags.

If this flag is present on an ACE, but neither `ACE4_DIRECTORY_INHERIT_ACE` nor `ACE4_FILE_INHERIT_ACE` is present, then an operation attempting to set such an attribute SHOULD fail with `NFS4ERR_ATTRNOTSUPP`.

ACE4_SUCCESSFUL_ACCESS_ACE_FLAG and ACE4_FAILED_ACCESS_ACE_FLAG

The ACE4_SUCCESSFUL_ACCESS_ACE_FLAG (SUCCESS) and ACE4_FAILED_ACCESS_ACE_FLAG (FAILED) flag bits may be set only on ACE4_SYSTEM_AUDIT_ACE_TYPE (AUDIT) and ACE4_SYSTEM_ALARM_ACE_TYPE (ALARM) ACE types. If during the processing of the file's ACL, the server encounters an AUDIT or ALARM ACE that matches the principal attempting the OPEN, the server notes that fact, and the presence, if any, of the SUCCESS and FAILED flags encountered in the AUDIT or ALARM ACE. Once the server completes the ACL processing, it then notes if the operation succeeded or failed. If the operation succeeded, and if the SUCCESS flag was set for a matching AUDIT or ALARM ACE, then the appropriate AUDIT or ALARM event occurs. If the operation failed, and if the FAILED flag was set for the matching AUDIT or ALARM ACE, then the appropriate AUDIT or ALARM event occurs. Either or both of the SUCCESS or FAILED can be set, but if neither is set, the AUDIT or ALARM ACE is not useful.

The previously described processing applies to ACCESS operations even when they return NFS4_OK. For the purposes of AUDIT and ALARM, we consider an ACCESS operation to be a "failure" if it fails to return a bit that was requested and supported.

ACE4_IDENTIFIER_GROUP

Indicates that the "who" refers to a GROUP as defined under UNIX or a GROUP ACCOUNT as defined under Windows. Clients and servers MUST ignore the ACE4_IDENTIFIER_GROUP flag on ACEs with a who value equal to one of the special identifiers outlined in Section 5.4.8.

ACE4_INHERITED_ACE

Indicates that this ACE is inherited from a parent directory. A server that supports automatic inheritance will place this flag on any ACEs inherited from the parent directory when creating a new object. Client applications will use this to perform automatic inheritance. Clients and servers MUST clear this bit in the acl attribute; it may only be used in the dacl and sacl attributes.

5.4.8. ACE Who

The "who" field of an ACE is an identifier that specifies the principal or principals to whom the ACE applies. It may refer to a user or a group, with the flag bit ACE4_IDENTIFIER_GROUP specifying which.

There are several special identifiers that need to be understood universally, rather than in the context of a particular DNS domain. Some of these identifiers cannot be understood when an NFS client

accesses the server, but have meaning when a local process accesses the file. The ability to display and modify these permissions is permitted over NFS, even if none of the access methods on the server understands the identifiers.

Who	Description
OWNER	The owner of the file.
GROUP	The group associated with the file.
EVERYONE	The world, including the owner and owning group.
INTERACTIVE	Accessed from an interactive terminal.
NETWORK	Accessed via the network.
DIALUP	Accessed as a dialup user to the server.
BATCH	Accessed from a batch job.
ANONYMOUS	Accessed without any authentication.
AUTHENTICATED	Any authenticated user (opposite of ANONYMOUS).
SERVICE	Access from a system service.

Table 2

To avoid conflict, these special identifiers are distinguished by an appended "@" and should appear in the form "xxxx@" (with no domain name after the "@"), for example, ANONYMOUS@.

The ACE4_IDENTIFIER_GROUP flag MUST be ignored on entries with these special identifiers. When encoding entries with these special identifiers, the ACE4_IDENTIFIER_GROUP flag SHOULD be set to zero.

[working group input needed]: I don't understand what might be valid reasons to ignore this. Would "MUST" be appropriate here?

It is important to note that "EVERYONE@" is not equivalent to the UNIX "other" entity. This is because, by definition, UNIX "other" does not include the owner or owning group of a file. "EVERYONE@" means literally everyone, including the owner or owning group.

[working group input needed]: Some of these require that changes be made as discussed below:

- * For "INTERACTIVE", "NETWORK", "DIALUP", "BATCH", and "SERVICE" it needs to be specified that server's ability to make these distinctions is limited, making their use where security is an issue quite problematic.
- * For "ANONYMOUS", clearly requests using AUTH_NONE fit but what else?

Request by nobody and by users root-squashed to nobody are probably OK, although you could argue about the case of a user "nobody" authenticated by ROCSEC_GSS>

ON a more contentious note, I would argue that users "authenticated" using AUTH_SYS, in the clear, without client-peer authentication fit here, but we need to get to consensus on this point.

- * Issues regarding "AUTHENTICATED" will be the mirror image of those for "ANONYMOUS".

5.4.9. Automatic Inheritance Features

The acl attribute consists only of an array of ACEs, but the sacl (Section 5.8) and dacl (Section 5.7) attributes also include an additional flag field.

```
struct nfsacl4l {
    aclflag4      na4l_flag;
    nfsace4       na4l_aces<>;
};
```

The flag field applies to the entire sacl or dacl; three flag values are defined:

```
const ACL4_AUTO_INHERIT      = 0x00000001;
const ACL4_PROTECTED         = 0x00000002;
const ACL4_DEFAULTED         = 0x00000004;
```

and all other bits must be cleared. The ACE4_INHERITED_ACE flag may be set in the ACEs of the sacl or dacl (whereas it must always be cleared in the acl).

Together these features allow a server to support automatic inheritance, which we now explain in more detail.

Inheritable ACEs are normally inherited by child objects only at the time that the child objects are created; later modifications to inheritable ACEs do not result in modifications to inherited ACEs on descendants.

However, the `dacl` and `sacl` provide an OPTIONAL mechanism that allows a client application to propagate changes to inheritable ACEs to an entire directory hierarchy.

A server that supports this feature performs inheritance at object creation time in the normal way, and SHOULD set the `ACE4_INHERITED_ACE` flag on any inherited ACEs as they are added to the new object.

A client application such as an ACL editor may then propagate changes to inheritable ACEs on a directory by recursively traversing that directory's descendants and modifying each ACL encountered to remove any ACEs with the `ACE4_INHERITED_ACE` flag and to replace them by the new inheritable ACEs (also with the `ACE4_INHERITED_ACE` flag set). It uses the existing ACE inheritance flags in the obvious way to decide which ACEs to propagate. (Note that it may encounter further inheritable ACEs when descending the directory hierarchy and that those will also need to be taken into account when propagating inheritable ACEs to further descendants.)

The reach of this propagation may be limited in two ways: first, automatic inheritance is not performed from any directory ACL that has the `ACL4_AUTO_INHERIT` flag cleared; and second, automatic inheritance stops wherever an ACL with the `ACL4_PROTECTED` flag is set, preventing modification of that ACL and also (if the ACL is set on a directory) of the ACL on any of the object's descendants.

This propagation is performed independently for the `sacl` and the `dacl` attributes; thus, the `ACL4_AUTO_INHERIT` and `ACL4_PROTECTED` flags may be independently set for the `sacl` and the `dacl`, and propagation of one type of `acl` may continue down a hierarchy even where propagation of the other `acl` has stopped.

New objects should be created with a `dacl` and a `sacl` that both have the `ACL4_PROTECTED` flag cleared and the `ACL4_AUTO_INHERIT` flag set to the same value as that on, respectively, the `sacl` or `dacl` of the parent object.

Both the `dacl` and `sacl` attributes are Recommended, and a server may support one without supporting the other.

A server that supports both the old acl attribute and one or both of the new dacl or sacl attributes must do so in such a way as to keep all three attributes consistent with each other. Thus, the ACEs reported in the acl attribute should be the union of the ACEs reported in the dacl and sacl attributes, except that the ACE4_INHERITED_ACE flag must be cleared from the ACEs in the acl. And of course a client that queries only the acl will be unable to determine the values of the sacl or dacl flag fields.

When a client performs a SETATTR for the acl attribute, the server SHOULD set the ACL4_PROTECTED flag to true on both the sacl and the dacl. By using the acl attribute, as opposed to the dacl or sacl attributes, the client signals that it may not understand automatic inheritance, and thus cannot be trusted to set an ACL for which automatic inheritance would make sense.

When a client application queries an ACL, modifies it, and sets it again, it should leave any ACEs marked with ACE4_INHERITED_ACE unchanged, in their original order, at the end of the ACL. If the application is unable to do this, it should set the ACL4_PROTECTED flag. This behavior is not enforced by servers, but violations of this rule may lead to unexpected results when applications perform automatic inheritance.

If a server also supports the mode attribute, it SHOULD set the mode in such a way that leaves inherited ACEs unchanged, in their original order, at the end of the ACL. If it is unable to do so, it SHOULD set the ACL4_PROTECTED flag on the file's dacl.

Finally, in the case where the request that creates a new file or directory does not also set permissions for that file or directory, and there are also no ACEs to inherit from the parent's directory, then the server's choice of ACL for the new object is implementation-dependent. In this case, the server SHOULD set the ACL4_DEFAULTED flag on the ACL it chooses for the new object. An application performing automatic inheritance takes the ACL4_DEFAULTED flag as a sign that the ACL should be completely replaced by one generated using the automatic inheritance rules.

5.5. Attribute 12: acl

The acl attribute, as opposed to the sacl and dacl attributes, consists only of an ACE array and does not support automatic inheritance.

The `acl` attribute is recommended and there is no requirement that a server support it. However, when the `dacl` attribute is supported, it is a good idea to provide support for the `acl` attributes as well, in order to accommodate clients that have not been upgraded to use the `dacl` attribute.

5.6. Attribute 13: `aclsupport`

A server need not support all of the above ACE types. This attribute indicates which ACE types are supported for the current file system. The bitmask constants used to represent the above definitions within the `aclsupport` attribute are as follows:

```
const ACL4_SUPPORT_ALLOW_ACL      = 0x00000001;
const ACL4_SUPPORT_DENY_ACL       = 0x00000002;
const ACL4_SUPPORT_AUDIT_ACL      = 0x00000004;
const ACL4_SUPPORT_ALARM_ACL      = 0x00000008;
```

Servers that support either the `ALLOW` or `DENY` ACE type SHOULD support both `ALLOW` and `DENY` ACE types.

[Working group input needed]: What are the valid reasons not to do this?

Clients should not attempt to set an ACE unless the server claims support for that ACE type. If the server receives a request to set an ACE that it cannot store, it MUST reject the request with `NFS4ERR_ATTRNOTSUPP`. If the server receives a request to set an ACE that it can store but cannot enforce, the server SHOULD reject the request with `NFS4ERR_ATTRNOTSUPP`.

[Working group input needed]: I might be mistaken but this might contradict material in Section 5.4.1

Support for any of the ACL attributes is OPTIONAL, although Recommended. However, a server that supports either of the new ACL attributes (`dacl` or `sacl`) MUST allow use of the new ACL attributes to access all of the ACE types that it supports. In other words, if such a server supports `ALLOW` or `DENY` ACEs, then it MUST support the `dacl` attribute, and if it supports `AUDIT` or `ALARM` ACEs, then it MUST support the `sacl` attribute.

5.7. V4.1 Attribute 58: `dacl`

The `dacl` attribute is like the `acl` attribute, but `dacl` allows only `ALLOW` and `DENY` ACEs. The `dacl` attribute supports automatic inheritance (see Section 5.4.9).

5.8. V4.1 Attribute 59: sacl

The sacl attribute is like the acl attribute, but sacl allows only AUDIT and ALARM ACEs. The sacl attribute supports automatic inheritance (see Section 5.4.9).

6. Common Considerations for Both File access Models

[Important structural change to be noted]: This section is derived from Section 6.3 of 8881, entitled "Common Methods. However, its content is different because it has been rewritten to deal with issues common to both file access models, which now appears to have not been the original intention. Nevertheless, the following changes have been made:

- * The section "Server Considerations" has been revised to deal with both the mode and acl attributes, since the points being made apply, in almost all cases, to both attributes.
- * The section "Client Considerations" has been heavily revised, since what had been there did not make any sense to me.
- * The section "Computing a Mode Attribute from an ACL" has been moved to Section 7.2.1 since it deals with the co-ordination of the posix and acl authorization models.

6.1. Server Considerations

The server uses the mode attribute or the acl attribute applying the algorithm described in Section 5.4.1 to determine whether an ACL allows access to an object.

However, these attributes might not be the sole determiner of access. For example:

- * In the case of a file system exported as read-only, the server will deny write access even though an object's file access attributes would grant it.
- * Server implementations MAY grant ACE4_WRITE_ACL and ACE4_READ_ACL permissions to prevent a situation from arising in which there is no valid way to ever modify the ACL.
- * All servers will allow a user the ability to read the data of the file when only the execute permission is granted (e.g., if the ACL denies the user the ACE4_READ_DATA access and allows the user ACE4_EXECUTE, the server will allow the user to read the data of the file).

- * Many servers implement owner-override semantics in which the owner of the object is allowed to override accesses that are denied by the ACL. This may be helpful, for example, to allow users continued access to open files on which the permissions have changed.
- * Many servers provide for the existence of a "superuser" that has privileges beyond an ordinary user. The superuser may be able to read or write data or metadata in ways that would not be permitted by the ACL or mode attributes.
- * A retention attribute might also block access otherwise allowed by ACLs (see Section 5.13 of [8]).

6.2. Client Considerations

Clients SHOULD NOT do their own access checks based on their interpretation of the ACL, but rather use the OPEN and ACCESS operations to do access checks. This allows the client to act on the results of having the server determine whether or not access should be granted based on its interpretation of the ACL.

[Working group discussion needed]: With regard to the use of "SHOULD NOT" in the paragraph above, it is not clear what might be valid reasons to bypass this recommendation. Perhaps "MUST NOT" or "should not" would be more appropriate.

Clients must be aware of situations in which an object's ACL will define a certain access even though the server will not enforce it. In general, but especially in these situations, the client needs to do its part in the enforcement of access as defined by the ACL.

[Working group input needed]: Despite what is said later, the only such case I know of is the use of READ and EXECUTE where the client, but not the server, has any means of distinguishing these. don't know of any others. If there were, how could ACCESS or OPEN be used to verify access?

To do this, the client MAY send the appropriate ACCESS operation prior to servicing the request of the user or application in order to determine whether the user or application should be granted the access requested.

For examples in which the ACL may define accesses that the server doesn't enforce, see Section 6.1.

[Working group discussion needed]: The sentence above is clearly wrong since that section is about enforcement the server does do. Sigh!

7. Combining Authorization Models

[To be checked]: The section on computing mode from ACL in RFC3530 is quite similar to that in RFC8881. If they are essentially identical, Section 7.2.1 could be moved out of Section 7.2 and apply to all minor versions :-). In addition, there may be additional material that could be generalized in this way.

7.1. Combined Authorization Models for V4.0

V4.0 servers that support both the mode and acl attributes, like servers for other minor versions, have to appropriately co-ordinate the values. Because much of the material in Section 7.2 does not apply to v4.0, the server is relatively free about how it does this. Nevertheless, that material is one useful approach and v4.0 implementers may want to consult it, even though the requirements in it do not apply and threcommendations have no normative force.

One likely source of a need for different treatment is the absence of a set_mode_masked attribute in v4.0.

Clients need to be aware that the v4.1 handling may nor may not be adopted by the server and that the client needs to adapt accordingly.

7.2. Combined Authorization Model for V4.1 and Beyond

- * On servers that support both the mode and the acl or dacl attributes, the server must keep the two consistent with each other. The value of the mode attribute (with the exception of the three high-order bits described in Section 5.3.1) must be determined entirely by the value of the ACL, so that use of the mode is never required for anything other than setting the three high-order bits. See Sections 7.2.4 through 7.2.6 for detailed requirements.
- * When a mode attribute is set on an object, the ACL attributes may need to be modified in order to not conflict with the new mode. In such cases, it is desirable that the ACL keep as much information as possible. This includes information about inheritance, AUDIT and ALARM ACEs, and permissions granted and denied that do not conflict with the new mode.

The server that supports both mode and ACL must take care to synchronize the MODE4_*USR, MODE4_*GRP, and MODE4_*OTH bits with the ACEs that have respective who fields of "OWNER@", "GROUP@", and "EVERYONE@". This way, the client can see if semantically equivalent access permissions exist whether the client asks for the owner, owner_group, and mode attributes or for just the ACL.

In this section, much depends on the method in specified Section 7.2.1. Many requirements refer to this section. It should be noted that the methods have behaviors specified with "SHOULD" and that alternate approaches are discussed in Section 7.2.2. This is intentional, to avoid invalidating existing implementations that compute the mode according to the withdrawn POSIX ACL draft (1003.1e draft 17), rather than by actual permissions on owner, group, and other.

[Working group discussion needed]: Given the mixture of RFC2219 terms, I think all of them in Section 7 need review. Further, given the effort that has gone into Section 7, to accommodate these implementations of a draft that was withdrawn decades ago. The idea of trying to make mode and acl match is undercut when there are different valid ways of computing the mode. There shouldn't be. To specify one way to do this is necessary to accomplish the goal here and to do so would not "invalidate" anything. Rather, it would establish, correctly, that such implementations are not implementations of the NFSv4 ACL model, but of the withdrawn POSIX ACL draft.

7.2.1. Computing a Mode Attribute from an ACL

The following method can be used to calculate the MODE4_R*, MODE4_W*, and MODE4_X* bits of a mode attribute, based upon an ACL.

[Working group discussion needed]: "can be used" says essentially "do whatever you choose" and would make Section 7 essentially pointless. Would prefer "is to be used", "MUST", or "SHOULD" if valid reasons to do otherwise can be found.

First, for each of the special identifiers OWNER@, GROUP@, and EVERYONE@, evaluate the ACL in order, considering only ALLOW and DENY ACEs for the identifier EVERYONE@ and for the identifier under consideration. The result of the evaluation will be an NFSv4 ACL mask showing exactly which bits are permitted to that identifier.

Then translate the calculated mask for OWNER@, GROUP@, and EVERYONE@ into mode bits for, respectively, the user, group, and other, as follows:

1. Set the read bit (MODE4_RUSR, MODE4_RGRP, or MODE4_ROTH) if and only if ACE4_READ_DATA is set in the corresponding mask.
2. Set the write bit (MODE4_WUSR, MODE4_WGRP, or MODE4_WOTH) if and only if ACE4_WRITE_DATA and ACE4_APPEND_DATA are both set in the corresponding mask.
3. Set the execute bit (MODE4_XUSR, MODE4_XGRP, or MODE4_XOTH), if and only if ACE4_EXECUTE is set in the corresponding mask.

7.2.2. Alternatives in Computing Mode Bits

Some server implementations also add bits permitted to named users and groups to the group bits (MODE4_RGRP, MODE4_WGRP, and MODE4_XGRP).

Implementations are discouraged from doing this, because it has been found to cause confusion for users who see members of a file's group denied access that the mode bits appear to allow. (The presence of DENY ACEs may also lead to such behavior, but DENY ACEs are expected to be more rarely used.)

[Working group decision needed]; the text does not seem to really discourage this practice and makes no reference to the need to standardize behavior or any impediment to doing so. "SHOULD NOT" needs to be considered if valid reasons to do otherwise can be found.

The same user confusion seen when fetching the mode also results if setting the mode does not effectively control permissions for the owner, group, and other users; this motivates some of the requirements that follow.

7.2.3. Setting Multiple ACL Attributes

In the case where a server supports the sacl or dacl attribute, in addition to the acl attribute, the server MUST fail a request to set the acl attribute simultaneously with a dacl or sacl attribute. The error to be given is NFS4ERR_ATTRNOTSUPP.

7.2.4. Setting Mode and not ACL

When any of the nine low-order mode bits are subject to change, either because the mode attribute was set or because the mode_set_masked attribute was set and the mask included one or more bits from the nine low-order mode bits, and no ACL attribute is explicitly set, the acl and dacl attributes must be modified in accordance with the updated value of those bits. This must happen even if the value of the low-order bits is the same after the mode is set as before.

Note that any AUDIT or ALARM ACEs (hence any ACEs in the sacl attribute) are unaffected by changes to the mode.

In cases in which the permissions bits are subject to change, the acl and dacl attributes MUST be modified such that the mode computed via the method in Section 7.2.1 yields the low-order nine bits (MODE4_R*, MODE4_W*, MODE4_X*) of the mode attribute as modified by the attribute change. The ACL attributes SHOULD also be modified such that:

1. If MODE4_RGRP is not set, entities explicitly listed in the ACL other than OWNER@ and EVERYONE@ SHOULD NOT be granted ACE4_READ_DATA.
2. If MODE4_WGRP is not set, entities explicitly listed in the ACL other than OWNER@ and EVERYONE@ SHOULD NOT be granted ACE4_WRITE_DATA or ACE4_APPEND_DATA.
3. If MODE4_XGRP is not set, entities explicitly listed in the ACL other than OWNER@ and EVERYONE@ SHOULD NOT be granted ACE4_EXECUTE.

Access mask bits other than those listed above, appearing in ALLOW ACEs, MAY also be disabled.

Note that ACEs with the flag ACE4_INHERIT_ONLY_ACE set do not affect the permissions of the ACL itself, nor do ACEs of the type AUDIT and ALARM. As such, it is desirable to leave these ACEs unmodified when modifying the ACL attributes.

Also note that the requirement may be met by discarding the acl and dacl, in favor of an ACL that represents the mode and only the mode. This is permitted, but it is preferable for a server to preserve as much of the ACL as possible without violating the above requirements. Discarding the ACL makes it effectively impossible for a file created with a mode attribute to inherit an ACL (see Section 7.2.8).

7.2.5. Setting ACL and Not Mode

When setting the `acl` or `dacl` and not setting the mode or `mode_set_masked` attributes, the permission bits of the mode need to be derived from the ACL. In this case, the ACL attribute SHOULD be set as given. The nine low-order bits of the mode attribute (`MODE4_R*`, `MODE4_W*`, `MODE4_X*`) MUST be modified to match the result of the method in Section 7.2.1. The three high-order bits of the mode (`MODE4_SUID`, `MODE4_SGID`, `MODE4_SVTX`) SHOULD remain unchanged.

7.2.6. Setting Both ACL and Mode

When setting both the mode (includes use of either the mode attribute or the `mode_set_masked` attribute) and the `acl` or `dacl` attributes in the same operation, the attributes MUST be applied in this order: mode (or `mode_set_masked`), then ACL. The mode-related attribute is set as given, then the ACL attribute is set as given, possibly changing the final mode, as described above in Section 7.2.5.

7.2.7. Retrieving the Mode and/or ACL Attributes

Some server implementations may provide for the existence of "objects without ACLs", meaning that all permissions are granted and denied according to the mode attribute and that no ACL attribute is stored for that object. If an ACL attribute is requested of such a server, the server SHOULD return an ACL that does not conflict with the mode; that is to say, the ACL returned SHOULD represent the nine low-order bits of the mode attribute (`MODE4_R*`, `MODE4_W*`, `MODE4_X*`) as described in Section 7.2.1.

For other server implementations, the ACL attribute is always present for every object. Such servers SHOULD store at least the three high-order bits of the mode attribute (`MODE4_SUID`, `MODE4_SGID`, `MODE4_SVTX`). The server SHOULD return a mode attribute if one is requested, and the low-order nine bits of the mode (`MODE4_R*`, `MODE4_W*`, `MODE4_X*`) MUST match the result of applying the method in Section 7.2.1 to the ACL attribute.

7.2.8. Creating New Objects

If a server supports any ACL attributes, it may use the ACL attributes on the parent directory to compute an initial ACL attribute for a newly created object. This will be referred to as the inherited ACL within this section. The act of adding one or more ACEs to the inherited ACL that are based upon ACEs in the parent directory's ACL will be referred to as inheriting an ACE within this section.

Implementors should standardize the behavior of CREATE and OPEN depending on the presence or absence of the mode and ACL attributes by following the directions below:

1. If just the mode is given in the call:

In this case, inheritance SHOULD take place, but the mode MUST be applied to the inherited ACL as described in Section 7.2.4, thereby modifying the ACL.

2. If just the ACL is given in the call:

In this case, inheritance SHOULD NOT take place, and the ACL as defined in the CREATE or OPEN will be set without modification, and the mode modified as in Section 7.2.5.

3. If both mode and ACL are given in the call:

In this case, inheritance SHOULD NOT take place, and both attributes will be set as described in Section 7.2.6.

4. If neither mode nor ACL is given in the call:

In the case where an object is being created without any initial attributes at all, e.g., an OPEN operation with an opentype4 of OPEN4_CREATE and a createmode4 of EXCLUSIVE4, inheritance SHOULD NOT take place (note that EXCLUSIVE4_1 is a better choice of createmode4, since it does permit initial attributes). Instead, the server SHOULD set permissions to deny all access to the newly created object. It is expected that the appropriate client will set the desired attributes in a subsequent SETATTR operation, and the server SHOULD allow that operation to succeed, regardless of what permissions the object is created with. For example, an empty ACL denies all permissions, but the server should allow the owner's SETATTR to succeed even though WRITE_ACL is implicitly denied.

In other cases, inheritance SHOULD take place, and no modifications to the ACL will happen. The mode attribute, if supported, MUST be as computed in Section 7.2.1, with the MODE4_SUID, MODE4_SGID, and MODE4_SVTX bits clear. If no inheritable ACEs exist on the parent directory, the rules for creating acl, dacl, or sacl attributes are implementation defined. If either the dacl or sacl attribute is supported, then the ACL4_DEFAULTED flag SHOULD be set on the newly created attributes.

7.2.9. Use of Inherited ACL When Creating Objects

If the object being created is not a directory, the inherited ACL SHOULD NOT inherit ACEs from the parent directory ACL unless the ACE4_FILE_INHERIT_ACE flag is set.

If the object being created is a directory, the inherited ACL should inherit all inheritable ACEs from the parent directory, that is, those that have the ACE4_FILE_INHERIT_ACE or ACE4_DIRECTORY_INHERIT_ACE flag set. If the inheritable ACE has ACE4_FILE_INHERIT_ACE set but ACE4_DIRECTORY_INHERIT_ACE is clear, the inherited ACE on the newly created directory MUST have the ACE4_INHERIT_ONLY_ACE flag set to prevent the directory from being affected by ACEs meant for non-directories.

When a new directory is created, the server MAY split any inherited ACE that is both inheritable and effective (in other words, that has neither ACE4_INHERIT_ONLY_ACE nor ACE4_NO_PROPAGATE_INHERIT_ACE set), into two ACEs, one with no inheritance flags and one with ACE4_INHERIT_ONLY_ACE set. (In the case of a dacl or sacl attribute, both of those ACEs SHOULD also have the ACE4_INHERITED_ACE flag set.) This makes it simpler to modify the effective permissions on the directory without modifying the ACE that is to be inherited to the new directory's children.

7.3. Combined Authorization Models for V4.2

The v4.1 server implementation requirements described in Section 7.2 apply to v4.2 as well and v4.2 clients can assume that the server follows them.

V4.2 contains an OPTIONAL extension, defined in [13], which is intended to reduce the interference of modes, restricted by the umask mechanism, with the acl inheritance mechanism. The extension allows the client to specify the umask separately from the mask attribute.

8. Labelled NFS Authorization Model

The labelled NFS feature of NFSv4.2 is designed to support Mandatory Access control.

The attribute sec_label enables an authorization model focused on Mandatory Access Control and is described in Section 8.

Not much can be said about this feature because the specification, in the interest of flexibility has left important features undefined in order to allow future. As a result, we have something that is a framework to allow Mandatory Access Control rather than one to provide it. In particular,

- * The `sec_label` attribute, which provides the objects label has no existing specification.
- * There is no specification of the of the format of the subject label or way to authenticate them.
- * As a result, all authorization takes place on the client, and the server simply accepts the client's determination.

This arrangement shares important similarities with `AUTH_SYS`. As such it makes sense:

- * To require/recommend that an encrypted connection be used.
- * To require/recommend that client and server peers mutually authenticate as part of connection establishment.
- * That work be devoted to providing a replacement without the above issues.

9. State Modification Authorization

Modification of locking and session state data should not be done by a client other than the one that created the lock. For this form of authorization, the server needs to identify and authenticate client peers rather than client users.

Such authentication is not directly provided by any RPC authentication flavor. However, RPC-based transport, when suitable configured, can provide this authentication.

NFSv4.1 defines a number of ways to provide appropriate authorization facilities. These will not be discussed in detail here but the following points should be noted:

- * NFSv4.1 defines the `MACHCRED` mechanism which uses the `RPCSEC_GSS` infrastructure to provide authentication of the clients peer. However, this is of no value when `AUTH_SYS` is being used.
- * NFSv4.1 also defines the `SSV` mechanism which uses the `RPCSEC_GSS` infrastructure to enable it to be reliably determined whether two different client connections are connected to the same client. It

is unclear whether the word "authentication" is appropriate in this case. As with MACHCRED, this is of no value when AUTH_SYS is being used.

- * Because of the lack of support for AUTH_SYS and for NFSv4.0, it is quite desirable for clients to use and for servers to require the use of client-peer authentication as part of connection establishment.

When unauthenticated clients are allowed, their state is exposed to unwanted modification as part of disruption or denial-of-service attacks. As a result the potential burdens of such attacks are felt principally by client not to provide such authentication.

10. Identification and Authentication

Various objects and subjects need to be identified for a protocol to function. For it to be secure, many of these need to be authenticated so that incorrect identification is not the basis for attacks.

10.1. Identification vs. Authentication

It is necessary to be clear about this distinction which has been obscured in the past, by the use of the term "RPC Authentication Flavor" in connection with situation in which identification without authentication occurred or in which there was neither identification nor authentication involved. As a result, we will use the term "RPC Flavors" instead

10.2. Items to be Identified

Some identifiers are not security-relevant and can be used without authentication, given that, in the authorization decision, the object acted upon needs only to be properly identified

- * File names are of this type.

Unlike the case for some other protocols, confusion of names that result from internationalization issues, while an annoyance, are not relevant to security. If the confusion between LATIN CAPITAL LETTER O and CYRILLIC CAPITAL LETTER O, results in the wrong file being accessed, the mechanisms described in Section 5 prevent inappropriate access being granted.

Despite the above, it is desirable if file names together with similar are not transferred in the clear as the information exposed may give attackers useful information helpful in planning and executing attacks.

- * The case of file handles is similar.

Identifier that refer to state shared between client and server can be the basis of disruption attacks since clients and server necessarily assume that neither side will change the state corpus without appropriate notice.

While these identifiers do not need to be authenticated, they are associated with higher-level entities for which change of the state represented by those entities is subject to peer authentication.

- * Unexpected closure of stateids or changes in state sequence values can disrupt client access as no clients have provision to deal with this source of interference.

While encryption may make it more difficult to execute such attacks attackers can often guess stateid's since server generally not randomize them.

- * Similarly, modification to v4.1 session state information can result in confusion if an attacker changes the slot sequence by assuing spurious requests. Even if the request is rejected, the slot sequence is changed and clients may a difficult time getting back in sync with the server.

While encryption may make it more difficult to execute such attacks attackers can often guess slot id's and obtain sessinid's since server generally do not randomize them.

it is necessary that modification of the higher-level entities be restricted to the client that created them.

- * For v4.0, the relevant entity is the clientid.
- * for v4.1, the releant entity is the sessionid.

Identifiers describing the issuer of the request, whether in numeric or string form always require authentication.

10.3. Authentication Provided by specific RPC Flavors

Different flavors differ quite considerably, as discussed below;

- * When AUTH_NONE is used, the user making the request is neither authenticated nor identified to the server.

Also, the server is not authenticated to the client and has no way to determine whether the server it is communicating with is an imposter.

- * When AUTH_SYS is used, the user making is the request identified but there no authentication of that identification.

As in the previous case, the server is not authenticated to the client and has no way to determine whether the server it is communicating with is an imposter.

- * When RPCSEC_GSS is used, the user making the request is authenticated as is the server peer responding.

10.4. Authentication Provided by the RPC Transport

Different transports differ quite considerably, as discussed below. In contrast to the case of RPC flavors, any authentication happens once, at connection establishment, rather than on each RPC request. As a result, it is the client and server peers, rather than individual users that is authenticated.

- * For most transports, such as TCP and RPC-over-RDMA version 1, there is no provision for peer authentication.

As a result use of AUTH_SYS together with such transports is inherently problematic.

- * Some transports provide for the possibility of mutual peer authentication.

11. Security of Data in Flight

11.1. Data Security Provided by the Flavor-associated Services

The only flavor providing these facilities is RPCSEC_GSS. When this flavor is used, data security can be negotiated between client and server as described in Section 12.2. However, when data security is provided at the transport level, as described in Section 11.2, the negotiation of privacy and integrity support is unnecessary,

Other flavors, such as AUTH_SYS and AUTH_NONE have no such data security facilities. When these flavor are used, the only data security is provided by the transport.

11.2. Data Security Provided by the RPC Transport

Some transports provide data security for all transactions performed on them, eliminating the need for that security to be provided or negotiated by the selection of particular flavors, mechanism, or services.

12. Security Negotiation

As previously in NFSv4, we use the term "negotiation" to characterize the process of the server providing a set of options and the client selecting one.

The use of SECINFO, possibly with SECINFO_NONAME, remains the primary means by which the security parameters are determined. The addition of transports to flavors in providing security has resulted in the following changes:

- * Transport-related security choices are typically decided at connection-establishment so there needs to be provision for negotiation at this point.
- * Despite the above, because the choices of flavor and transport affect one another, SECINFO has been extended by the addition of pseudo-flavors, while retaining the existing XDR, to allow negotiation of transport choices and accompanying connection establishment options, in addition to selection of flavors and accompanying services. This allows server policies for such matters to be different for different portions of the namespace.

12.1. Flavors and Pseudo-flavors

The flavor field of the secinfo4 items returned by SECINFO and SECINFO_NONAME have always allowed pseudo flavors to be included. However, previous treatments of these operations have not provided information about how responses containing such pseudo-flavors are to be interpreted.

Those pseudo-flavors now provide a means of extending the negotiation process so it is capable of providing for the negotiation of the use particular RPC transports and security-related options for the connections established using those transports.

The flavors AUTH_NONE, AUTH_SYS and RPCSEC_GSS continue to indicate the acceptability of the corresponding method of user authentication, user identification, or user non-identification, when used with a particular RPC transport.

The flavor AUTH_TLS, which is not used as part of issuing requests is not included in this list and is treated as a connection-type--specifying pseudo-flavor.

secinfo4s for the flavor RPCSEC_GSS contains additional information describing the specific security algorithm to be used and the ancillary services to be provided (e.g. integrity, privacy) when these services are not provided by the transport.

Such flavors are referred to as "flavor-specifying flavors"

The classification below organizes the flavors and pseudo-flavors used in security negotiation while Section 12.4 describes how the set of secinfos in a response can be used by the client to select acceptable combinations of security flavor, security mechanism, security services, security-related transports, and security-related connection characteristics.

- * The pseudo-flavors designating a security-relevant transport type, together with AUTH_TLS, nominally a flavor but used as a pseudo-flavor in connection with SECINFO.

Such pseudo-flavors are referred to as "transport-specifying flavors".

- * The pseudo-flavors designating restrictions on acceptable connection characteristics include XPCH_ENCRYPT, XPCH_PEERAUTH, and XPCH_SECURE.

Such pseudo-flavors are referred to as "transport-restriction flavors".

- * The pseudo-flavors combining a flavor designation together with a restriction concerning the connections it is relevant to include AUTHXP_CLIENTPEER, AUTHXP_ENCRYPT, AUTHXP_SECURE.

Such pseudo-flavors are referred to as transport-hybridized flavors.

- * The special pseudo-flavors, XPBREAK and XPCURRENT

Such pseudo-flavors are referred to as connection-organizing flavors.

12.2. Negotiation of Security Flavors and Mechanisms

For the current connection, this proceeds as it has previously, when security-relevant transports were not available. Flavor entries, including those including mechanism information are listed in order of server preference and apply, by default, to the current connection, which normally is favored by the server.

When other transport-identifying pseudo-flavors appear before the flavor entries, then the server is indicating that these transport types, with the server preference following the ordering of the entries. In this case, any flavor entries that follow a transport entry specify that flavor is usable with the transport types denoted by that transport entry.

12.3. Negotiation of RPC Transports and Characteristics

First we define some necessary terminology.

- * A transport type specifies one of a small set of RPC transport types such as TCP or RDMA. Each such transport type has an associated pseudoflavor. There are also pseudo-flavors that specify a set of transport types such as XPT_ALL.
- * Connection characteristics are designations of security-relevant characteristics or sets of characteristics that connections might have.

There are pseudo-flavors associated with connection characteristics such as CONCH_CPAUTH, denoting client-peer authentication and CONCH_ENCRYPT, denoting the presence of an encrypted channel. The pseudo-flavor CONCH_SECURE denotes the presence of peer mutual authentication together with the use of an encrypted channel.

- * The combination of a transport type with a set of connection characteristics is considered a connection type..while many connection types are designated by a combination of a flavor designating a transport with on designating a set of connection characteristics, there are pseudo flavor that designate a connection type directly.

For example, the flavor AUTH_TLS is equivalent to XP_TCP combined with CONCH_ENCRYPT, while the pseudo-flavor XP_TCP_SECURE equivalent to XP_TCP combined with CONCH_SECURE.

- * A flavor specification designates a specific flavor, or, in the case of RPCSEC_GSS, a flavor combined with additional mechanism and service information.
- * A flavor assignment denote the association of a specific flavor specification with a connection type.

A secinfo response will designate a set of valid flavor assignments with an implied server ordering derived from the order that the entries appear in.

In interpreting the response array the client is to maintain sets of designated transport types, connection characteristics and connection types specified individually (i.e. without separately specifying transport types and connection characteristics). When a flavor specification is encountered, that flavor is considered valid when used with all currently active connection types, defined by the union of the individually specified connection types and the cartesian product of the current transport types and current connection types.

The presumed ordering of these assignments is as follows:

- * When one of the connection types was specified directly by a connection type, the position of that specification is compared to that of either the other individually-specified connection type or the earlier of the transport-type specification and the connection characteristics specification.
- * In other cases, the position of the transport type specifications are considered first with the position of the connection characteristics considered if necessary.
- * If neither of the above resolve issue, the position of the flavor specification is considered.
- * The type of the current connection is considered to be specified first, implicitly.
- * There are provisions, described in Section 12.4 to modify this ordering, as may be necessary, for example, when the current connection, while acceptable is of lower server preference.

12.4. Overall Interpretation of SECINFO Response Arrays

[TBD in -01]

12.5. SECINFO

The description in the sub-sections below, while it adheres to the XDR appearing [6], [7], [8], [9] and [11]. will supersede the descriptions in [7] and [8].

This is necessary to adapt the security negotiation process to the presence of transport-level security services such as encryption and peer authentication.

Sumilar changes are necessary in the parallel SECINFO_NONAME operation introduced in NFSv4.1. These are expected to be done as part of the rfc5661bis effort.

12.5.1. SECINFO ARGUMENTS

```
struct SECINFO4args {  
    /* CURRENT_FH: directory */  
    component4      name;  
};
```

Figure 1

12.5.2. SECINFO RESULTS

```

/*
 * From RFC 2203
 */
enum rpc_gss_svc_t {
    RPC_GSS_SVC_NONE          = 1,
    RPC_GSS_SVC_INTEGRITY     = 2,
    RPC_GSS_SVC_PRIVACY       = 3
};

struct rpcsec_gss_info {
    sec_oid4          oid;
    qop4             qop;
    rpc_gss_svc_t     service;
};

/* RPCSEC_GSS has a value of '6' - See RFC 2203 */
union secinfo4 switch (uint32_t flavor) {
    case RPCSEC_GSS:
        rpcsec_gss_info      flavor_info;
    default:
        void;
};

typedef secinfo4 SECINFO4resok<>;

union SECINFO4res switch (nfsstat4 status) {
    case NFS4_OK:
        /* CURRENTFH: consumed */
        SECINFO4resok resok4;
    default:
        void;
};

```

Figure 2

12.5.3. SECINFO DESCRIPTION

The SECINFO operation is used by the client determine the appropriate RPC authentication flavors, security mechanisms and encrypting transports to access a specific directory filehandle, file name pair. SECINFO should apply the same access approach used for LOOKUP when evaluating the name. In consequence, if the requester does not have the appropriate access to LOOKUP the name, then SECINFO will behave the same way and return NFS4ERR_ACCESS.

The result will contain an array that represents the security flavor, security mechanisms and transports available, with an order corresponding to the server's preferences, the most preferred being first in the array. The client is free to pick whatever security flavors, mechanisms and transports it both desires and supports, or to pick in the server's preference order the first one it supports. The array entries are represented by the secinfo4 structure. The field 'flavor' will contain one of the following sorts of values:

- * a value of AUTH_NONE, AUTH_SYS (as defined in RFC 5531 [4]).
- * AUTH_TLS as described in ...
- * A pseudo-flavor defined in Section 14.2
- * RPCSEC_GSS (as defined in RFC 2203 [2]).
- * Any other security flavor or pseudo-flavor registered with IANA.

For the flavors other than RPCSEC_GSS, no additional security information is returned. For a return value of RPCSEC_GSS, a security triple is returned that contains the mechanism object identifier (OID, as defined in RFC 2743 [3]), the quality of protection (as defined in RFC 2743 [3]), and the service type (as defined in RFC 2203 [2]). It is possible for SECINFO to return multiple entries with flavor equal to RPCSEC_GSS with different security triple values.

On success, the current filehandle is consumed, so that, if the operation following SECINFO tries to use the current filehandle, that operation will fail with the status NFS4ERR_NOFILEHANDLE.

If the name has a length of zero, or if the name does not obey the UTF-8 definition in circumstances in which UTF-8 names are required, the error NFS4ERR_INVALID will be returned.

See Sections 12.2 through 12.4 for additional information on the use of SECINFO.

12.5.4. SECINFO IMPLEMENTATION (general)

The SECINFO operation is expected to be used by the NFS client when the error value of NFS4ERR_WRONGSEC is returned from another NFS operation. This signifies to the client that the server's security policy is different from what the client is currently using. At this point, the client is expected to obtain a list of possible security flavors and choose what best suits its policies.

12.5.4.1. SECINFO IMPLEMENTATION (for v4.0)

[TBD in -01]

12.5.4.2. SECINFO IMPLEMENTATION (for v4.1 and v4.2)

As mentioned, the server's security policies will determine when a client request receives NFS4ERR_WRONGSEC.

See Table 14 of [8] for a list of operations that can return NFS4ERR_WRONGSEC. in the case of v4.2, there might be extensions alloed to return NFS4ERR_WRONGSEC. In addition, when READDIR returns attributes, the rdattnr_error (Section 5.8.1.12 of [8]) can contain NFS4ERR_WRONGSEC.

Note that CREATE and REMOVE MUST NOT return NFS4ERR_WRONGSEC. The rationale for CREATE is that unless the target name exists, it cannot have a separate security policy from the parent directory, and the security policy of the parent was checked when its filehandle was injected into the COMPOUND request's operations stream (for similar reasons, an OPEN operation that creates the target MUST NOT return NFS4ERR_WRONGSEC). If the target name exists, while it might have a separate security policy, that is irrelevant because CREATE MUST return NFS4ERR_EXIST. The rationale for REMOVE is that while that target might have a separate security policy, the target is going to be removed, and so the security policy of the parent trumps that of the object being removed. RENAME and LINK MAY return NFS4ERR_WRONGSEC, but the NFS4ERR_WRONGSEC error applies only to the saved filehandle (see Section 2.6.3.1.2 of [8]). Any NFS4ERR_WRONGSEC error on the current filehandle used by LINK and RENAME MUST be returned by the PUTFH, PUTPUBFH, PUTROOTFH, or RESTOREFH operation that injected the current filehandle.

With the exception of LINK and RENAME, the set of operations that can return NFS4ERR_WRONGSEC represents the point at which the client can inject a filehandle into the "current filehandle" at the server. The filehandle is either provided by the client (PUTFH, PUTPUBFH, PUTROOTFH), generated as a result of a name-to-filehandle translation (LOOKUP and OPEN), or generated from the saved filehandle via RESTOREFH. As oSection 2.6.3.1.1.1 of [8] states, a put filehandle operation followed by SAVEFH MUST NOT return NFS4ERR_WRONGSEC. Thus, the RESTOREFH operation, under certain conditions (see Section 2.6.3.1.1 of [8]), is permitted to return NFS4ERR_WRONGSEC so that security policies can be honored.

The READDIR operation will not directly return the NFS4ERR_WRONGSEC error. However, if the READDIR request included a request for attributes, it is possible that the READDIR request's security triple

did not match that of a directory entry. If this is the case and the client has requested the `rdattr_error` attribute, the server will return the `NFS4ERR_WRONGSEC` error in `rdattr_error` for the entry.

To resolve an error return of `NFS4ERR_WRONGSEC`, the client does the following:

- * For `LOOKUP` and `OPEN`, the client will use `SECINFO` with the same current filehandle and name as provided in the original `LOOKUP` or `OPEN` to enumerate the available security triples.
- * For the `rdattr_error`, the client will use `SECINFO` with the same current filehandle as provided in the original `READDIR`. The name passed to `SECINFO` will be that of the directory entry (as returned from `READDIR`) that had the `NFS4ERR_WRONGSEC` error in the `rdattr_error` attribute.
- * For `PUTFH`, `PUTROOTFH`, `PUTPUBFH`, `RESTOREFH`, `LINK`, and `RENAME`, the client will use `SECINFO_NO_NAME` { style = `SECINFO_STYLE4_CURRENT_FH` }. The client will prefix the `SECINFO_NO_NAME` operation with the appropriate `PUTFH`, `PUTPUBFH`, or `PUTROOTFH` operation that provides the filehandle originally provided by the `PUTFH`, `PUTPUBFH`, `PUTROOTFH`, or `RESTOREFH` operation.

NOTE: In NFSv4.0, the client was required to use `SECINFO`, and had to reconstruct the parent of the original filehandle and the component name of the original filehandle. The introduction in NFSv4.1 of `SECINFO_NO_NAME` obviates the need for reconstruction.

- * For `LOOKUPP`, the client will use `SECINFO_NO_NAME` { style = `SECINFO_STYLE4_PARENT` } and provide the filehandle that equals the filehandle originally provided to `LOOKUPP`.

12.6. Future Security Needs

[To be fleshed out in -01]: This section is basically an outline for now, to be filled out later based on Working Group input, particularly from Chuck lever who suggested this section and has ideas about many of the items in it.

- * Security for data-at-rest, most probably based on facilities defined within SAN.
- * Support for content signing.
- * Revision/extension of labelled NFS to provide true interoperability and server-based authorization.

- * Work to provide more security for RDMA-based transports. This would include the peer authentication infrastructure now being developed as part of RPC-over-RDMA version 2. In addition, there is a need for an RPC-based transport that provides for encryption, which might be provided in number of ways.

13. Security Considerations

13.1. Changes in Security Considerations

Beyond the needed inclusion of a threat analysis and the fact that all minor versions are dealt with together, there are a number of substantive changes in the approach to NFSv4 security presented in RFCs 7530 and 8881 and that appearing in this document.

This document will not seek to speculate how the previous treatment, now viewed as incorrect, came to be written, approved, and published. However, it will, for the benefit of those familiar with the previous treatment of these matters, draw attention to the important changes listed here.

- * There is a vastly expanded range of threats being considered as described in Section 13.1.1
- * New facilities available at the RPC transport level can be used to deal with security issues, as described in Section 13.1.2

13.1.1. Wider View of Threats

Although the absence of a threat analysis in previous treatments makes comparison most difficult, the security-related features described in previous specifications and the associated discussion in their security considerations sections makes it clear that earlier specifications took a quite narrow view of threats to be protected against.

One aspect of that narrow view that merits special attention is the handling of AUTH_SYS, at that time in the clear, with no client peer authentication.

With regard to specific threats, there is no mention in existing security consideration sections of:

- * Denial-of-service attacks.
- * Client-impersonation attacks.
- * Server-impersonation attacks.

The handling of data security in-flight is even more troubling.

- * Although there was considerable work in the protocol to allow use of encryption to be negotiated when using RPCSEC_GSS. The existing security considerations do not mention the potential need for encryption at all.

It is not clear why this was omitted but it is a pattern that cannot be repeated in this document.

- * The case of negotiation of integrity services is similar and uses the same negotiation infrastructure.

In this case, use of integrity is recommended but not to prevent the corruption of user data being read or written.

The use of integrity services is recommended in connection with issuing SECINFO (and for NFSv4.1, SECINFO_NONAME). The presence of this recommendation in the associated security considerations sections has the unfortunate effect of suggesting that the protection of user data is of relatively low importance.

13.1.2. Transport-layer Security Facilities

Such transport-level RPC facilities as RPC-over-TLS provide important ways of providing better security for all the NFSv4 minor versions.

In particular:

- * The presence of encryption by default will deal with security issues regarding data-in-flight, for both RPCSEC_GSS and AUTH_SYS.
- * Peer authentication provided by the server eliminates the possibility of a server-impersonation attack, even when using AUTH_SYS.
- * When mutual authentication is part of connection establishment, there is a possibility, where an appropriate trust relationship exists of treating the user's presented in AUTH_SYS, as effectively authenticated, based on the authentication of the client peer.

13.1.3. Compatibility and Maturity Issues

Given the need to drastically change the NFSv4 security approach from that specified previously, it is necessary for us to be mindful of:

- * The difficulty that might be faced in adapting to the newer guidance because the delays involved in designing, developing, and testing new transport- level security facilities such as RPC-over-TLS.
- * The difficulty in discarding or substantially modifying previous existing deployments and practices, developed on the basis of previous normative guidance.

For these reasons, we will not use the term "MUST NOT" in some situations in which the use of that term might have been justified earlier. In such cases, previous guidance together with the passage of time may have created a situation in which the considerations mentioned above in this section may be valid reasons to defer, for a limited time, correction of the current situation making the term "SHOULD NOT" appropriate, since the difficulties cited would constitute a valid reason to not allow what has been recommended against.

13.1.4. Discussion of AUTHSYS

An important change concerns the treatment of AUTH_SYS which is now divided into two subcases given the possible availability of support from the transport layer.

When such support is not available, AUTH_SYS SHOULD NOT be used, since it makes the following attacks quite easy to execute:

- * The absence of authentication of the server to the client allow server impersonation in which an imposter server can obtain data to be written by the user and supply corrupted data to read requests.
- * The absence of authentication of the client user to the server allow server impersonation in which an imposter client can issue requests and have them executed as a user designated by imposter client, vitiating the server's authorization policy.

With no authentication of the client peer, common approaches, such as using the source IP address can be easily defeated, allowing unauthenticated execution of requests made by the pseudo clients

- * The absence of any support to protect data-in-flight when AUTH_SYS is used result in further serious security weaknesses.

In connection with the use of the term "SHOULD NOT" above, it is understood that the "valid reasons" to use this form of access reflect the Compatibility and Maturity Issue discussed above in Section 13.1.3 and that it is expected that, over time, these will become less applicable.

13.2. Security Considerations Scope

13.2.1. Discussion of Potential Classification of Environments

[Working group discussion needed]: For now, we will not consider different security policies for different sorts of environments. This is because of:

- * Doing so would add considerable complexity to this document.
- * The additional complexity would undercut our main goal here, which is to discuss secure use on the internet, which remain an important NFSv4 goal.
- * The ubiquity of internet access makes it hard to treat corporate network separately from the internet per se.
- * While small networks might be sufficiently isolated to make it reasonable use NFSv4 without serious attention to security issues, the complexity of characterizing the necessary isolation makes it impractical to deal with such cases in this document.

13.2.2. Discussion of Environments

Although the security goal for NFSv4 has been and remains "secure use on the internet", much use of NFSv4 occurs on more restricted IP networks with NFS access from outside the owning organization prevented by firewalls.

This security considerations section will not deal separately with such environments since the threats that need to be discussed are essentially the same, despite the assumption by many that the restricted network access would eliminate the possibility of attacks originating inside the network by attackers who have some legitimate NFSv4 access within it.

In organizations of significant size, this sort of assumption of trusted access is usually not valid and this document will not deal with them explicitly. In any case, there is little point in doing so, since, if everyone can be trusted, there can be no attackers, rendering threat analysis superfluous.

This does not mean that NFSv4 use cannot, as a practical matter, be made secure through means outside the scope of this document including strict administrative controls on all software running within it, frequent polygraph tests, and threats of prosecution. However, this document is not prepared to discuss the details of such policies, their implementation, or legal issues associated with them and treats such matters as out-of-scope.

Nfsv4 can be used in very restrictive IP network environments where outside access is quite restricted and there is sufficient trust to allow, for example, every node to have the same root password. The case of a simple network only accessible by a single user is similar. In such networks, many things that this document says "SHOULD NOT" be done are unexceptionable but the responsibility for making that determination is one for those creating such networks to take on. This document will not deal further with NFSv4 use on such networks.

13.3. Major New Recommendations

13.3.1. Recommendations Regarding Security of Data in Flight

We RECOMMEND that requesters always issue requests with data security (i.e. with protection from disclosure or modification in flight) whether provided at the RPC request level or by the RPC transport, irrespective of the responder's requirements.

We RECOMMEND that implementers provide servers the ability to configure policies in which requests without data security will be rejected as having insufficient security.

We RECOMMEND that servers use such policies over either their entire local namespace or for all file systems except those clearly designed for the general dissemination of non-sensitive data.

13.3.2. Recommendations Regarding Client Peer Authentication

We RECOMMEND that clients provide authentication material whenever a connection is established with a server capable of using it to provide client peer authentication.

We RECOMMEND that implementers provide servers the ability to configure policies in which attempts to establish connections without client peer authentication will be rejected.

We RECOMMEND that servers adopt such policies whenever requests not using RPCSEC_GSS are allowed to be executed.

13.3.3. Issues Regarding Valid Reasons to Bypass Recommendations

Clearly, the maturity and compatibility issues mentioned in Section 13.1.3 are valid reasons to nypass the above recommenations, as loong as thse issues continue to exist.

[Working group discussion needed]: The question the working group needs to address is whether other valid reasons exist.

[Working group discussion needed]: In particular, some members of the group might feel that the performance cost of encrypted transports constitutes, in itself, a valid reason to ignore the above recommendations.

[Working group discussion needed]: I cannot agree and feel that accepting that as a valid reason would undercut Nfsv4 security improvement, and probably would not be acceptable to the security directorate. However, I do want to work out an a generally acceptable compromise. I propose something along the following lines:

The transport-based encryption facilities are designed to be compatible with facilities to offload the work of encryption and decryption. When such facilities are not available, at a reasonable cost, to NFSv4 servers and clients anticipating heavy use of NFSv4, then the lack of such facilities can be considered a valid reason to bypass the above recommendations, as long as that situation continues.

13.4. Data Security Threats

[TBD in -01]

13.5. Authentication-based threats

13.5.1. Attacks based on the use of AUTH_SYS

[TBD in -01]

13.5.2. Attacks on Name/Userid Mapping Facilities

[TBD in -01]

13.6. Disruption and Denial-of-Service Attacks

13.6.1. Attacks Based on the Disruption of Client-Server Shared State

[TBD in -01]

13.6.2. Attacks Based on Forcing the Misuse of Server Resources

[TBD in -01]

14. IANA Considerations

Because of the shift from implementing security-related services only in connection with RPCSEC_GSS to one in which transport-level security has a prominent role, a number of new values need to be assigned.

These include new authstat values to guide selection of a Transport acceptable to both client and server, presented in Section 14.1 and new pseudo-flavors to be used in the process of security negotiation, presented in Section 14.2.

14.1. New Authstat Values

The following new authstat values are necessary to enable a server to indicate that the server's policy does not allow requests to be made on the current connection because of security issues associated with the rpc transport. In the event they are received, the client needs to establish a new connection.

- * The value XP_CRYPT indicates that the server will not support access using unencrypted connections while the current connection is not encrypted.
- * The value XP_CPAUTH indicates that the server will not support access using connections for which the client peer has not authenticated itself as part of connection while the current connection has not been set up in that way.

14.2. New Authentication Pseudo-Flavors

The following new pseudo-flavors are made available to allow their return as part of the response to SECINFO operation described in Section 12.5 and for similar operations.

[TBD in -01]

15. References

15.1. Normative References

- [1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [2] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, DOI 10.17487/RFC2203, September 1997, <<https://www.rfc-editor.org/info/rfc2203>>.
- [3] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, DOI 10.17487/RFC2743, January 2000, <<https://www.rfc-editor.org/info/rfc2743>>.
- [4] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <<https://www.rfc-editor.org/info/rfc5531>>.
- [5] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [6] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <<https://www.rfc-editor.org/info/rfc7530>>.
- [7] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 External Data Representation Standard (XDR) Description", RFC 7531, DOI 10.17487/RFC7531, March 2015, <<https://www.rfc-editor.org/info/rfc7531>>.
- [8] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <<https://www.rfc-editor.org/info/rfc8881>>.
- [9] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, DOI 10.17487/RFC5662, January 2010, <<https://www.rfc-editor.org/info/rfc5662>>.
- [10] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <<https://www.rfc-editor.org/info/rfc7862>>.

- [11] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 External Data Representation Standard (XDR) Description", RFC 7863, DOI 10.17487/RFC7863, November 2016, <<https://www.rfc-editor.org/info/rfc7863>>.
- [12] Myklebust, T. and C. Lever, "Towards Remote Procedure Call Encryption By Default", Work in Progress, Internet-Draft, draft-ietf-nfsv4-rpc-tls-11, 23 November 2020, <<https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-rpc-tls-11>>.

15.2. Informative References

- [13] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <<https://www.rfc-editor.org/info/rfc8257>>.

Appendix A. Acknowledgments

The author wishes to thank Tom Haynes for his helpful suggestion to deal with security for all NFSv4 minor versions in the same document.

The author wishes to draw people's attention to Nico Williams' remark that NFSv4 security was not so bad, except that there was no provision for authentication of the client peer. This perceptive remark, which now seems like common sense, did not seem so when made, but it has served as a beacon for those putting NFSv4 security on a firmer footing. Our gratitude is appropriate.

The author wishes to acknowledge the important role of the authors of RPC-with-TLS, Chuck Lever and Trond Myklebust, in moving the NFS security agenda forward and thank them for all their efforts to improve NFS security.

The author wishes to thank Chuck Lever for his many helpful comments about NFSv4 security issues, his explanation of many unclear points, and much important guidance he provided that is reflected in this document.

The author wishes to thank Rick Macklem for his role in clarifying possible server policies regarding RPC-over-TLS and bringing possible approaches to the attention of the working group.

Author's Address

David Noveck (editor)
NetApp
1601 Trapelo Road, Suite 16
Waltham, MA 02451
United States of America

Phone: +1-781-572-8038
Email: davenoveck@gmail.com

Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 7 January 2022

C. Lever
Oracle
6 July 2021

Network File System (NFS) Upper-Layer Binding To RPC-Over-RDMA Version 2
draft-ietf-nfsv4-nfs-ulb-v2-05

Abstract

This document specifies Upper-Layer Bindings of Network File System (NFS) protocol versions to RPC-over-RDMA version 2.

Note

This note is to be removed before publishing as an RFC.

Discussion of this draft takes place on the NFSv4 working group mailing list (nfsv4@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/nfsv4/>. Working Group information can be found at <https://datatracker.ietf.org/wg/nfsv4/about/>.

The source for this draft is maintained in GitHub. Suggested changes can be submitted as pull requests at <https://github.com/chucklever/i-d-nfs-ulb-v2>. Instructions are on that page as well.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 7 January 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Requirements Language	3
3. Upper-Layer Binding for NFS Versions 2 and 3	4
3.1. Reply Size Estimation	4
3.2. RPC Binding Considerations	5
3.3. Transport Considerations	5
4. Upper-Layer Bindings for NFS Version 2 and 3 Auxiliary Protocols	6
4.1. MOUNT, NLM, and NSM Protocols	7
4.2. NFSACL Protocol	7
5. Upper-Layer Binding For NFS Version 4	7
5.1. DDP-Eligibility	7
5.2. Reply Size Estimation	9
5.3. RPC Binding Considerations	10
5.4. NFS COMPOUND Requests	10
5.5. NFS Callback Requests	12
5.6. Session-Related Considerations	13
5.7. Transport Considerations	14
6. Extending NFS Upper-Layer Bindings	15
7. Security Considerations	16
8. IANA Considerations	16
9. References	16
9.1. Normative References	16
9.2. Informative References	17
Acknowledgments	18
Author's Address	18

1. Introduction

The RPC-over-RDMA version 2 transport may employ direct data placement to convey data payloads associated with RPC transactions, as described in [I-D.ietf-nfsv4-rpcrdma-version-two]. RPC client and server implementations using RPC-over-RDMA version 2 must agree which XDR data items and RPC procedures are eligible to use direct data placement (DDP) to ensure successful interoperation.

An Upper-Layer Binding specifies this agreement for one or more versions of one RPC program. Other operational details, such as RPC binding assignments, pairing Write chunks with result data items, and reply size estimation, are also specified by such a Binding.

This document contains material required of Upper-Layer Bindings, as specified in Appendix A of [I-D.ietf-nfsv4-rpcrdma-version-two], for the following NFS protocol versions:

- * NFS version 2 [RFC1094]
- * NFS version 3 [RFC1813]
- * NFS version 4.0 [RFC7530]
- * NFS version 4.1 [RFC8881]
- * NFS version 4.2 [RFC7862]

The current document also provides Upper-Layer Bindings for auxiliary protocols used with NFS versions 2 and 3 (see Section 4).

This document assumes the reader is already familiar with concepts and terminology defined throughout [I-D.ietf-nfsv4-rpcrdma-version-two] and the documents it references.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Upper-Layer Binding for NFS Versions 2 and 3

The Upper-Layer Binding specification in this section applies to NFS version 2 [RFC1094] and NFS version 3 [RFC1813]. For brevity, in this document, a "Legacy NFS client" refers to an NFS client using version 2 or version 3 of the NFS RPC program (100003) to communicate with an NFS server. Likewise, a "Legacy NFS server" is an NFS server communicating with clients using NFS version 2 or NFS version 3.

The following XDR data items in NFS versions 2 and 3 are DDP-eligible:

- * The opaque file data argument in the NFS WRITE procedure
- * The pathname argument in the NFS SYMLINK procedure
- * The opaque file data result in the NFS READ procedure
- * The pathname result in the NFS READLINK procedure

All other argument or result data items in NFS versions 2 and 3 are not DDP-eligible.

Whether or not an NFS operation is considered non-idempotent, a transport error might not indicate whether the server has processed the arguments of the RPC Call, or whether the server has accessed or modified client memory associated with that RPC.

3.1. Reply Size Estimation

During the construction of each RPC Call message, a Requester is responsible for allocating appropriate transport resources to receive the corresponding Reply message. These resources must be capable of holding the entire Reply, therefore the Requester needs to estimate the maximum possible size of the expected Reply message.

- * In many cases, the expected Reply can fit in one or a few RDMA Send messages. The Requester need not provision any RDMA resources, relying instead on message continuation to handle the entire Reply message.
- * In cases where the Requester deems direct data placement to be the most efficient transfer mechanism, it provisions Write chunks wherein the Responder can place results. In these cases, the Requester must reliably estimate the maximum size of each result that is to be placed in a Write chunk.

- * When the Requester expects an especially large Reply message, it can provision a combination of a Reply chunk and Write chunks for result data items. In such cases, the Requester must reliably estimate the maximum size of each result that is to be placed in a Write chunk and the maximum size of the remainder to be placed in the Reply chunk.

A legacy NFS client needs to make every effort to avoid retransmission of non-idempotent NFS requests due to underestimated Reply resources. Thanks to the mechanism of message continuation in RPC-over-RDMA version 2, the need for such retransmission is greatly reduced.

3.2. RPC Binding Considerations

Legacy NFS servers traditionally listen for clients on UDP and TCP port 2049. Additionally, they register these ports with a local portmapper service [RFC1833].

A Legacy NFS server supporting RPC-over-RDMA version 2 and registering itself with the RPC portmapper MAY choose an arbitrary port, or MAY use the alternative well-known port number for its RPC-over-RDMA service (see Section 8). The chosen port MAY be registered with the RPC portmapper using the netids assigned in Section 12 of [I-D.ietf-nfsv4-rpcrdma-version-two].

3.3. Transport Considerations

Legacy NFS client implementations often rely on a transport-layer keep-alive mechanism to detect when a legacy server has become unresponsive. When an NFS server is no longer responsive, client-side keep-alive terminates the connection, which in turn triggers reconnection and retransmission of outstanding RPC transactions.

3.3.1. Keep-Alive

Some RDMA transports (such as the Reliable Connected QP type on InfiniBand) have no keep-alive mechanism. Without a disconnect or new RPC traffic, such connections can remain alive long after an NFS server has become unresponsive or unreachable. Once an NFS client has consumed all available RPC-over-RDMA version 2 credits on that transport connection, it awaits a reply indefinitely before sending another RPC request.

Legacy NFS clients SHOULD reserve one RPC-over-RDMA version 2 credit to use for periodic server or connection health assessment. Either peer can use this credit to drive an RPC request on an otherwise idle connection, triggering either an affirmative server response or a connection termination.

3.3.2. Replay Detection

Legacy NFS servers typically employ request replay detection to reduce the risk of data and file namespace corruption that could result when an NFS client retransmits a non-idempotent NFS request. A legacy NFS server can send a cached response when a replay is detected, rather than executing the request again. Replay detection is not perfect, but it is usually adequate.

For legacy NFS servers, replay detection commonly utilizes heuristic indicators such as the IP address of the NFS client, the source port of the connection, the transaction ID of the request, and the contents of the request's RPC and upper-layer protocol headers. In short, replay detection is typically based on a connection tuple and the request's XID. A legacy NFS client is careful to re-use the same source port, if practical, when reconnecting so that legacy NFS servers are better able to detect retransmissions.

However, a legacy NFS client operating over an RDMA transport has no control over connection source ports. It is almost certain that an RPC request that is retransmitted on a new connection can never be detected as a replay if the legacy NFS server includes the connection source port in its replay detection heuristics.

Therefore a legacy NFS server using an RDMA transport should never use a legacy NFS client connection's source port as part of its NFS request replay detection mechanism.

4. Upper-Layer Bindings for NFS Version 2 and 3 Auxiliary Protocols

Storage administrators typically deploy NFS versions 2 and 3 with several other protocols, sometimes referred to as the "NFS auxiliary protocols." These are distinct RPC programs that define procedures that are not part of the NFS RPC program (100003). The Upper-Layer Bindings in this section apply to:

- * Versions 2 and 3 of the MOUNT RPC program (100005) [RFC1813]
- * Versions 1, 3, and 4 of the NLM RPC program (100021) [RFC1813]
- * Version 1 of the NSM RPC program (100024), described in Chapter 11 of [XNFS]

- * Versions 2 and 3 of the NFSACL RPC program (100227). The NFSACL program does not have a public definition. In this document it is treated as a de facto standard, as there are several interoperating implementations.

4.1. MOUNT, NLM, and NSM Protocols

Historically, NFS/RDMA implementations have chosen to convey the MOUNT, NLM, and NSM protocols via TCP. A legacy NFS server implementation MUST provide support for these protocols via TCP to enable interoperation of these protocols when NFS/RDMA is in use.

4.2. NFSACL Protocol

Often legacy clients and servers that support the NFSACL RPC program convey NFSACL procedures on the same transport connection and port as the NFS RPC program (100003). Utilizing the same port obviates the need for separate a rpcbind query to discover server support for this RPC program.

ACLs are typically small, but even large ACLs must be encoded and decoded to some degree before being made available to users. Thus no data item in this Upper-Layer Protocol is DDP-eligible.

For procedures whose replies do not include an ACL object, the size of a reply is determined directly from the NFSACL RPC program's XDR definition. However, legacy client implementations should choose a maximum size for ACLs based on internal limits, and can rely on message continuation to handle the a priori unknown size of large ACL objects in Replies.

5. Upper-Layer Binding For NFS Version 4

The Upper-Layer Binding specification in this section applies to versions of the NFS RPC program defined in NFS version 4.0 [RFC7530] NFS version 4.1 [RFC8881] and NFS version 4.2 [RFC7862].

5.1. DDP-Eligibility

Only the following XDR data items in the COMPOUND procedure of all NFS version 4 minor versions are DDP-eligible:

- * The opaque data field in the WRITE4args structure
- * The linkdata field of the NF4LNK arm in the createtype4 union
- * The opaque data field in the READ4resok structure

- * The linkdata field in the READLINK4resok structure

5.1.1. The NFSv4.2 READ_PLUS operation

NFS version 4.2 introduces an enhanced READ operation called READ_PLUS [RFC7862]. READ_PLUS enables an NFS server to perform data reduction of READ results so that the returned READ data is more compact.

In a READ_PLUS result, returned file content appears as a list of one or more of the following items:

- * Regular data content: the same as the result of a traditional READ operation.
- * Unallocated space in a file: where no data has yet been written or previously-written data has been removed via a hole-punch operation.
- * A counted pattern.

Upon receipt of a READ_PLUS result, an NFSv4.2 client expands the returned list into the preferred local representation of the original file content.

Before receiving that result, an NFSv4.2 client typically does not know how the file's content is organized on the NFS server. Thus it is not possible to predict the size or structure of a READ_PLUS Reply in advance. The use of direct data placement is therefore challenging.

A READ_PLUS content list containing more than one segment of regular file data could be conveyed using multiple Write chunks, but only if the client knows in advance where those chunks appear in the Reply Payload stream. Moreover, the usual benefits of hardware-assisted data placement are entirely waived if the client-side transport must parse the result of each read I/O.

Therefore this Upper Layer Binding does not make any element of an NFSv4.2 READ_PLUS Reply DDP-eligible. Further, this Upper Layer Binding recommends that implementations avoid the use of the READ_PLUS operation on NFS/RDMA mount points.

5.2. Reply Size Estimation

Within NFS version 4, there are certain variable-length result data items whose maximum size cannot be estimated by clients reliably because there is no protocol-specified size limit on these result arrays. These include:

- * The attrlist4 field
- * Fields containing ACLs such as `fattr4_acl`, `fattr4_dacl`, and `fattr4_sacl`
- * Fields in the `fs_locations4` and `fs_locations_info4` data structures
- * Fields which pertain to pNFS layout metadata, such as `loc_body`, `loh_body`, `da_addr_body`, `lou_body`, `lrf_body`, `fattr_layout_types`, and `fs_layout_types`

5.2.1. Reply Size Estimation for Minor Version 0

The NFS version 4.0 protocol itself does not impose any bound on the size of NFS calls or replies.

Some of the data items enumerated in Section 5.2 (in particular, the items related to ACLs and `fs_locations`) make it difficult to predict the maximum size of NFS version 4.0 replies that interrogate variable-length `fattr4` attributes. Client implementations might rely upon internal architectural limits to constrain the reply size, but such limits are not always guaranteed to be reliable.

When an NFS version 4.0 client expects an especially sizeable `fattr4` result, it can rely on message continuation or provision a Reply chunk to enable that server to return that result via explicit RDMA.

5.2.2. Reply Size Estimation for Minor Version 1 and Newer

In NFS version 4.1 and newer minor versions, the `csa_fore_chan_attrs` argument of the `CREATE_SESSION` operation contains a `ca_maxresponsesize` field. The value in this field can be taken as the absolute maximum size of replies generated by an NFS version 4.1 server.

An NFS version 4 client can use this value in cases where it is not possible to estimate a reply size upper bound precisely. In practice, objects such as ACLs, named attributes, layout bodies, and security labels are much smaller than this maximum.

5.3. RPC Binding Considerations

NFS version 4 servers are required to listen on TCP port 2049, and are not required to register with an rpcbind service [RFC7530]. Therefore, an NFS version 4 server supporting RPC-over-RDMA version 2 MUST use the alternative well-known port number for its RPC-over-RDMA service (see Section 8 Clients SHOULD connect to this well-known port without consulting the RPC portmapper (as for NFS version 4 on TCP transports)).

5.4. NFS COMPOUND Requests

5.4.1. Multiple DDP-eligible Data Items

An NFS version 4 COMPOUND procedure can contain more than one operation that carries a DDP-eligible data item. An NFS version 4 client provides XDR Position values in each Read chunk to disambiguate which chunk is associated with which argument data item. However, NFS version 4 server and client implementations must agree in advance on how to pair Write chunks with returned result data items.

In the following lists, a "READ operation" refers to any NFS version 4 operation that has a DDP-eligible result data item. An NFS version 4 client applies the mechanism specified in Section 4.3.2 of [I-D.ietf-nfsv4-rpcrdma-version-two] to this class of operations as follows:

- * If an NFS version 4 client wishes all DDP-eligible items in an NFS reply to be conveyed inline, it leaves the Write list empty.

An NFS version 4 server acts as follows:

- * The first chunk in the Write list MUST be used by the first READ operation in an NFS version 4 COMPOUND procedure. The next Write chunk is used by the next READ operation, and so on.
- * If an NFS version 4 client has provided a matching non-empty Write chunk, then the corresponding READ operation MUST return its DDP-eligible data item using that chunk.
- * If an NFS version 4 client has provided an empty matching Write chunk, then the corresponding READ operation MUST return all of its result data items inline.

- * If a READ operation returns a union arm which does not contain a DDP-eligible result, and the NFS version 4 client has provided a matching non-empty Write chunk, an NFS version 4 server MUST return an empty Write chunk in that Write list position.
- * If there are more READ operations than Write chunks, then remaining NFS Read operations in an NFS version 4 COMPOUND that have no matching Write chunk MUST return their results inline.

5.4.2. Chunk List Complexity

By default, the RPC-over-RDMA version 2 protocol places limits on the number of chunks or segments that may appear in Read or Write lists (see Section 5.2 of [I-D.ietf-nfsv4-rpcrdma-version-two]).

These implementation limits are especially important when Kerberos integrity or privacy is in use [RFC7861]. GSS services increase the size of credential material in RPC headers, potentially requiring the use of a Long message, which increases the complexity of chunk lists independent of the particular NFS version 4 COMPOUND being conveyed.

In the absence of an explicit transport property exchange that alters these limits, NFS version 4 clients SHOULD follow the prescriptions listed below when constructing RPC-over-RDMA version 2 messages. NFS version 4 servers MUST accept and process all such requests.

- * The Read list can contain either a Position-Zero Read chunk, one Read chunk with a non-zero Position, or both.
- * The Write list can contain no more than one Write chunk.

NFS version 4 clients wishing to send more complex chunk lists can provide configuration interfaces to bound the complexity of NFS version 4 COMPOUNDS, limit the number of elements in scatter-gather operations, and avoid other sources of chunk overruns at the receiving peer.

If an NFS version 4 server receives an RPC request via RPC-over-RDMA version 2 that it cannot process due to chunk list complexity limits, it SHOULD return one of the following responses to the client:

- * A problem is detected by the transport layer while parsing the transport header in an RPC Call message. The server responds with an RDMA2_ERROR message with the err field set to ERR_CHUNK.

- * A problem is detected during XDR decoding of the RPC Call message while the RPC layer reassembles the call's XDR stream. The server responds with an RPC reply with its "reply_stat" field set to MSG_ACCEPTED and its "accept_stat" field set to GARBAGE_ARGS.

After receiving one of these errors, an NFS version 4 client SHOULD NOT retransmit the failing request, as the result would be the same error. It SHOULD terminate the RPC transaction associated with the XID in the reply without further processing, and report an error to the RPC consumer.

5.4.3. NFS Version 4 COMPOUND Example

The following example shows a Write list with three Write chunks, A, B, and C. The NFS version 4 server consumes the provided Write chunks by writing the results of the designated operations in the compound request (READ and READLINK) back to each chunk.

Write list:

A --> B --> C

NFS version 4 COMPOUND request:

PUTFH	LOOKUP	READ	PUTFH	LOOKUP	READLINK	PUTFH	LOOKUP	READ
		v			v			v
		A			B			C

If the NFS version 4 client does not want to have the READLINK result returned via RDMA, it provides an empty Write chunk for buffer B to indicate that the READLINK result must be returned inline.

5.5. NFS Callback Requests

The NFS version 4 family of protocols support server-initiated callbacks to notify NFS version 4 clients of events such as recalled delegations.

5.5.1. NFS Version 4.0 Callback

An NFS version 4.0 client uses the SETCLIENTID operation to advertise the IP address, port, and netid of its NFS version 4.0 callback service. When an NFS version 4.0 server provides a backchannel service to an NFS version 4.0 client that uses RPC-over-RDMA version 2 for its forward channel, the server MUST advertise the backchannel service using either the "tcp" or "tcp6" netid.

Because the backchannel does not operate on RPC-over-RDMA, no XDR data item in the NFS version 4.0 callback RPC program is DDP-eligible.

5.5.2. NFS Version 4.1 Callback

In NFS version 4.1 and newer minor versions, callback operations may appear on the same connection that is in use for NFS version 4 forward channel client requests. NFS version 4 clients and servers MUST use the mechanisms described in Section 4.5 of [I-D.ietf-nfsv4-rpcrdma-version-two] to convey backchannel operations on an RPC-over-RDMA version 2 transport.

The `csa_back_chan_attrs` argument of the `CREATE_SESSION` operation contains a `ca_maxresponsesize` field. The value in this field is the absolute maximum size of backchannel replies generated by a replying NFS version 4 client.

There are no DDP-eligible data items in callback procedures defined in NFS version 4.1 or NFS version 4.2. However, some callback operations, such as messages that convey device ID information, can be sizeable. A sender can use Message Continuation or a Long message in this situation.

When an NFS version 4.1 client can support Long Calls in its backchannel, it reports a backchannel `ca_maxrequestsize` that is larger than the connection's inline thresholds. Otherwise, an NFS version 4 server MUST use only Short messages to convey backchannel operations.

5.6. Session-Related Considerations

The presence of an NFS version 4 session (as defined in [RFC8881]) does not effect the operation of RPC-over-RDMA version 2. None of the operations introduced to support NFS sessions (e.g., the `SEQUENCE` operation) contain DDP-eligible data items. There is no need to match the number of session slots with the number of available RPC-over-RDMA version 2 credits.

However, there are a few new cases where an RPC transaction can fail. For example, a Requester might receive, in response to an RPC request, an `RDMA2_ERROR` message with a `rdma_err` value of `ERR_CHUNK`. These situations are not different from existing RPC errors, which an NFS session implementation can already handle for other transport types. Moreover, there might be no `SEQUENCE` result available to the Requester to distinguish whether failure occurred before or after the Responder executed the requested operations.

When a transport error occurs (e.g., an RDMA2_ERROR type message is received), the Requester proceeds, as usual, to match the incoming XID value to a waiting RPC Call. The Requester terminates the RPC transaction and reports the result status to the RPC consumer. The Requester's session implementation then determines the session ID and slot for the failed request and performs slot recovery to make that slot usable again. Otherwise, that slot could be rendered permanently unavailable.

When an NFS session is not present (for example, when NFS version 4.0 is in use), a transport error does not indicate whether the server has processed the arguments of the RPC Call, or whether the server has accessed or modified client memory associated with that RPC.

5.7. Transport Considerations

5.7.1. Congestion Avoidance

Section 3.1 of [RFC7530] states:

Where an NFS version 4 implementation supports operation over the IP network protocol, the supported transport layer between NFS and IP MUST be an IETF standardized transport protocol that is specified to avoid network congestion; such transports include TCP and the Stream Control Transmission Protocol (SCTP).

Section 2.9.1 of [RFC8881] further states:

Even if NFS version 4.1 is used over a non-IP network protocol, it is RECOMMENDED that the transport support congestion control.

It is permissible for a connectionless transport to be used under NFS version 4.1; however, reliable and in-order delivery of data combined with congestion control by the connectionless transport is REQUIRED. As a consequence, UDP by itself MUST NOT be used as an NFS version 4.1 transport.

RPC-over-RDMA version 2 utilizes only reliable, connection-oriented transports that guarantee in-order delivery, meeting all the above requirements for NFS version 4.0 and 4.1. See Section 4.2.1 of [I-D.ietf-nfsv4-rpcrdma-version-two] for more details.

5.7.2. Retransmission and Keep-alive

NFS version 4 client implementations often rely on a transport-layer keep-alive mechanism to detect when an NFS version 4 server has become unresponsive. When an NFS server is no longer responsive, client-side keep-alive terminates the connection, which in turn triggers reconnection and RPC retransmission.

Some RDMA transports (such as the Reliable Connected QP type on InfiniBand) have no keep-alive mechanism. Without a disconnect or new RPC traffic, such connections can remain alive long after an NFS server has become unresponsive. Once an NFS client has consumed all available RPC-over-RDMA version 2 credits on that transport connection, it indefinitely awaits a reply before sending another RPC request.

NFS version 4 clients SHOULD reserve one RPC-over-RDMA version 2 credit to use for periodic server or connection health assessment. Either peer can use this credit to drive an RPC request on an otherwise idle connection, triggering either a quick affirmative server response or immediate connection termination.

In addition to network partition and request loss scenarios, RPC-over-RDMA version 2 transport connections can be terminated when a Transport header is malformed, Reply messages exceed receive resources, or when too many RPC-over-RDMA messages are sent at once. In such cases:

- * If a transport error occurs (e.g., an RDMA2_ERROR type message is received) before the disconnect or instead of a disconnect, the Requester MUST respond to that error as prescribed by the specification of the RPC transport. Then the NFS version 4 rules for handling retransmission apply.
- * If there is a transport disconnect and the Responder has provided no other response for a request, then only the NFS version 4 rules for handling retransmission apply.

6. Extending NFS Upper-Layer Bindings

RPC programs such as NFS are required to have an Upper-Layer Binding specification to interoperate on RPC-over-RDMA version 2 transports [I-D.ietf-nfsv4-rpcrdma-version-two]. Via standards action, the Upper-Layer Binding specified in this document can be extended to cover versions of the NFS version 4 protocol specified after NFS version 4 minor version 2, or to cover separately published extensions to an existing NFS version 4 minor version, as described in [RFC8178].

7. Security Considerations

RPC-over-RDMA version 2 supports all RPC security models, including RPCSEC_GSS security and transport-level security [RFC7861]. The choice of what Direct Data Placement mechanism to convey RPC argument and results does not affect this since it changes only the method of data transfer. Because the current document defines only the binding of the NFS protocols atop RPC-over-RDMA version 2 [I-D.ietf-nfsv4-rpcrdma-version-two], all relevant security considerations are, therefore, described at that layer.

8. IANA Considerations

The use of direct data placement in NFS introduces a need for an additional port number assignment for networks that share traditional UDP and TCP port spaces with RDMA services. The iWARP protocol is such an example [RFC5040] [RFC5041].

For this purpose, the current document specifies a set of transport protocol port number assignments. IANA has assigned the following ports for NFS/RDMA in the IANA port registry, according to the guidelines described in [RFC6335].

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 current document should be added as a reference for the nfsrdma port assignments. The current document does not alter these assignments.

9. References

9.1. Normative References

- [I-D.ietf-nfsv4-rpcrdma-version-two]
Lever, C. and D. Noveck, "RPC-over-RDMA Version 2 Protocol", Work in Progress, Internet-Draft, draft-ietf-nfsv4-rpcrdma-version-two-05, 6 July 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-rpcrdma-version-two-05>>.
- [RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995,
<<https://www.rfc-editor.org/info/rfc1833>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <<https://www.rfc-editor.org/info/rfc6335>>.
- [RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <<https://www.rfc-editor.org/info/rfc7530>>.
- [RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <<https://www.rfc-editor.org/info/rfc7861>>.
- [RFC7862] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <<https://www.rfc-editor.org/info/rfc7862>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <<https://www.rfc-editor.org/info/rfc8881>>.

9.2. Informative References

- [RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <<https://www.rfc-editor.org/info/rfc1094>>.
- [RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <<https://www.rfc-editor.org/info/rfc1813>>.
- [RFC5040] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia, "A Remote Direct Memory Access Protocol Specification", RFC 5040, DOI 10.17487/RFC5040, October 2007, <<https://www.rfc-editor.org/info/rfc5040>>.

- [RFC5041] Shah, H., Pinkerton, J., Recio, R., and P. Culley, "Direct Data Placement over Reliable Transports", RFC 5041, DOI 10.17487/RFC5041, October 2007, <<https://www.rfc-editor.org/info/rfc5041>>.
- [RFC8178] Noveck, D., "Rules for NFSv4 Extensions and Minor Versions", RFC 8178, DOI 10.17487/RFC8178, July 2017, <<https://www.rfc-editor.org/info/rfc8178>>.
- [XNFS] The Open Group, "Protocols for Interworking: XNFS, Version 3W", February 1998.

Acknowledgments

Thanks to Tom Talpey, who contributed the text of Section 5.4.2. David Noveck contributed the text of Section 5.6 and Section 6. The author also wishes to thank Bill Baker and Greg Marsden for their support of this work.

Special thanks go to Transport Area Directors Zaheduzzaman Sarker, NFSV4 Working Group Chairs Brian Pawlowski, and David Noveck, and NFSV4 Working Group Secretary Thomas Haynes for their support.

Author's Address

Charles Lever
Oracle Corporation
United States of America

Email: chuck.lever@oracle.com

Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 14 November 2022

C. Lever
Oracle
13 May 2022

Network File System (NFS) Upper-Layer Binding To RPC-Over-RDMA Version 2
draft-ietf-nfsv4-nfs-ulb-v2-07

Abstract

This document specifies Upper-Layer Bindings of Network File System (NFS) protocol versions to RPC-over-RDMA version 2.

Note

Discussion of this draft takes place on the NFSv4 working group mailing list (nfsv4@ietf.org), archived at <https://mailarchive.ietf.org/arch/browse/nfsv4/>. Working Group information is available at <https://datatracker.ietf.org/wg/nfsv4/about/>.

Submit suggestions and changes as pull requests at <https://github.com/chucklever/i-d-nfs-ulb-v2>. Instructions are on that page.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 14 November 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Requirements Language	3
3. Upper-Layer Binding for NFS Versions 2 and 3	4
3.1. DDP-Eligibility	4
3.2. Reply Size Estimation	5
3.3. RPC Binding Considerations	5
3.4. Transport Considerations	5
3.4.1. Keep-Alive	6
3.4.2. Replay Detection	6
4. Upper-Layer Bindings for NFS Version 2 and 3 Auxiliary Protocols	7
4.1. MOUNT, NLM, and NSM Protocols	7
4.2. NFSACL Protocol	7
5. Upper-Layer Binding For NFS Version 4	8
5.1. DDP-Eligibility	8
5.1.1. The NFSv4.2 READ_PLUS operation	8
5.1.2. NFS Version 4 COMPOUND Requests	9
5.2. Reply Size Estimation	11
5.2.1. Reply Size Estimation for Minor Version 0	11
5.2.2. Reply Size Estimation for Minor Version 1 and Newer	12
5.3. RPC Binding Considerations	12
5.4. Transport Considerations	12
5.4.1. Congestion Avoidance	12
5.4.2. Retransmission and Keep-alive	13
5.5. Session-Related Considerations	13
6. Upper-Layer Binding For NFS Version 4 Callbacks	14
6.1. NFS Version 4.0 Callback	14
6.2. NFS Version 4.1 Callback	15
7. Extending NFS Upper-Layer Bindings	15
8. Security Considerations	15
9. IANA Considerations	16
10. References	16
10.1. Normative References	16
10.2. Informative References	17
Acknowledgments	18
Author's Address	18

1. Introduction

The RPC-over-RDMA version 2 transport can employ direct data placement to convey data payloads associated with RPC transactions, as described in [I-D.ietf-nfsv4-rpcrdma-version-two]. As mandated by that document, RPC client and server implementations using RPC-over-RDMA version 2 MUST agree in advance which XDR data items and RPC procedures are eligible for direct data placement (DDP).

An Upper-Layer Binding specifies this agreement for one or more versions of one RPC program. Other operational details, such as RPC binding assignments, pairing Write chunks with result data items, and reply size estimation, are also specified by such a Binding.

This document contains material required of Upper-Layer Bindings, as specified in Appendix A of [I-D.ietf-nfsv4-rpcrdma-version-two], for the following NFS protocol versions:

- * NFS version 2 [RFC1094]
- * NFS version 3 [RFC1813]
- * NFS version 4.0 [RFC7530]
- * NFS version 4.1 [RFC8881]
- * NFS version 4.2 [RFC7862]

The current document also provides Upper-Layer Bindings for auxiliary protocols used with NFS versions 2 and 3 (see Section 4).

This document assumes the reader is already familiar with concepts and terminology defined throughout [I-D.ietf-nfsv4-rpcrdma-version-two] and the documents it references.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Upper-Layer Binding for NFS Versions 2 and 3

The Upper-Layer Binding specification in this section applies to NFS version 2 [RFC1094] and NFS version 3 [RFC1813]. For brevity, in this document, a "Legacy NFS client" refers to an NFS client using version 2 or version 3 of the NFS RPC program (100003) to communicate with an NFS server. Likewise, a "Legacy NFS server" is an NFS server communicating with clients using NFS version 2 or NFS version 3.

3.1. DDP-Eligibility

Generally, storage protocols based on RDMA divide both read and write operations into two steps. This division enables the payload receiver to allocate the sink buffer for each I/O operation in advance of the network payload transfer. By allocating the sink buffer tactically, a good quality receiver implementation reduces the amount of data movement it must perform during and after the I/O operation.

During an NFS WRITE that involves explicit RDMA, first the NFS client sends a request that indicates where the NFS server can find the payload buffer, then the NFS server pulls the WRITE payload from that buffer. Likewise, during an NFS READ that involves explicit RDMA, the NFS client provides the location of the destination buffer, then the NFS server pushes the READ payload to that buffer.

Therefore, the following XDR data items in NFS versions 2 and 3 are DDP-eligible:

- * The opaque file data argument in the NFS WRITE procedure
- * The pathname argument in the NFS SYMLINK procedure
- * The opaque file data result in the NFS READ procedure
- * The pathname result in the NFS READLINK procedure

All other argument or result data items in NFS versions 2 and 3 are not DDP-eligible.

Regardless of whether an NFS operation is considered non-idempotent, a transport error might not indicate whether the server has processed the arguments of the RPC Call or whether the server has accessed or modified client memory associated with that RPC.

3.2. Reply Size Estimation

During the construction of each RPC Call message, a Requester is responsible for allocating appropriate RDMA resources to receive the corresponding Reply message. These resources must be capable of holding the entire Reply. Therefore the Requester needs to estimate the maximum possible size of the expected Reply message.

- * Often, the expected Reply can fit in a limited number of RDMA Send messages. The Requester need not provision any RDMA resources for the Reply, relying instead on message continuation to handle the entire Reply message.
- * In cases where the Upper Layer Binding permits direct data placement of the results (DDP), a Requester can provision Write chunks to receive those results. The Requester MUST reliably estimate the maximum size of each result receive via a Write chunk.
- * A Requester that expects a large Reply message can provision a Reply chunk. The Requester MUST reliably estimate the maximum size of the payload received via the Reply chunk.
- * If RDMA resources are not available to send a Reply, a Responder falls back to message continuation.

A correctly implemented Legacy NFS client thus avoids retransmission of non-idempotent NFS requests due to improperly estimated Reply resources.

3.3. RPC Binding Considerations

Legacy NFS servers typically listen for clients on UDP and TCP port 2049. Additionally, they register these ports with a local portmapper service [RFC1833].

A Legacy NFS server supporting RPC-over-RDMA version 2 and registering itself with the RPC portmapper MAY choose an arbitrary port or MAY use the alternative well-known port number for its RPC-over-RDMA service (see Section 9). The chosen port MAY be registered with the RPC portmapper using the netids assigned in Section 12 of [I-D.ietf-nfsv4-rpcrdma-version-two].

3.4. Transport Considerations

3.4.1. Keep-Alive

Legacy NFS client implementations can rely on connection keep-alive to detect when a Legacy NFS server has become unresponsive. When an NFS server is no longer responsive, client-side keep-alive terminates the connection, triggering reconnection and retransmission of outstanding RPC transactions.

Some RDMA transports (such as the Reliable Connected QP type on InfiniBand) have no keep-alive mechanism. Without a disconnect or new RPC traffic, such connections can remain alive long after an NFS server has become unresponsive or unreachable. Once an NFS client has consumed all available RPC-over-RDMA version 2 credits on that transport connection, it awaits a reply indefinitely before sending another RPC request.

Legacy NFS clients SHOULD reserve one RPC-over-RDMA version 2 credit to use for periodic server or connection health assessment. Either peer can use this credit to drive an RPC request on an otherwise idle connection, triggering either an affirmative server response or a connection termination.

3.4.2. Replay Detection

Like NFSv4.0, Legacy NFS servers typically employ request replay detection to reduce the risk of data and file namespace corruption that could result when an NFS client retransmits a non-idempotent NFS request. A Legacy NFS server can send a cached response when a replay is detected, rather than executing the request again. Replay detection is not perfect, but it is usually adequate.

For Legacy NFS servers, replay detection commonly utilizes heuristic indicators such as the IP address of the NFS client, the source port of the connection, the transaction ID of the request, and the contents of the request's RPC and upper-layer protocol headers. A Legacy NFS client is careful to re-use the same source port when reconnecting so that Legacy NFS servers can better detect RPC retransmission.

However, a Legacy NFS client operating over an RDMA transport has no control over connection source ports. It is almost certain that an RPC request retransmitted on a new connection can never be detected as a replay if the receiving Legacy NFS server includes the connection source port in its replay detection heuristics.

Therefore a Legacy NFS server using an RDMA transport should never use a connection's source port as part of its NFS request replay detection mechanism.

4. Upper-Layer Bindings for NFS Version 2 and 3 Auxiliary Protocols

Storage administrators typically deploy NFS versions 2 and 3 with several other protocols, sometimes called the "NFS auxiliary protocols." These are distinct RPC programs that define procedures not part of the NFS RPC program (100003). The Upper-Layer Bindings in this section apply to:

- * Versions 2 and 3 of the MOUNT RPC program (100005) [RFC1813]
- * Versions 1, 3, and 4 of the NLM RPC program (100021) [RFC1813]
- * Version 1 of the NSM RPC program (100024), described in Chapter 11 of [XNFS]
- * Versions 2 and 3 of the NFSACL RPC program (100227). The NFSACL program does not have a public definition. This document treats the NFSACL program as a de facto standard, as there are several interoperating implementations.

4.1. MOUNT, NLM, and NSM Protocols

Historically, NFS/RDMA implementations have conveyed the MOUNT, NLM, and NSM protocols via TCP. A Legacy NFS server implementation MUST provide support for these auxiliary protocols via TCP.

Moreover, there is little benefit from transporting these protocols via RDMA. Thus this document does not provide an Upper-Layer binding for them.

4.2. NFSACL Protocol

Legacy NFS clients and servers convey NFSACL procedures on the same transport connection and port as the NFS RPC program (100003). Utilizing the same port obviates the need for a separate rpcbind query to discover server support for this RPC program.

ACLs are typically small, but even large ACLs must be encoded and decoded to some degree before being stored in local filesystems. Thus no data item in this Upper-Layer Protocol is DDP-eligible.

For procedures whose replies do not include an ACL object, the size of each Reply is determined directly from the NFSACL RPC program's XDR definition.

The NFSACL protocol does not provide a mechanism to determine the size of a received ACL in advance. When preparing for responses that include ACLs, Legacy NFS clients estimate a maximum reply size based on limits within their local file systems. If that estimation is inadequate, a Responder falls back to message continuation.

5. Upper-Layer Binding For NFS Version 4

The Upper-Layer Binding specification in this section applies to versions of the NFS RPC program defined in NFS version 4.0 [RFC7530], NFS version 4.1 [RFC8881], and NFS version 4.2 [RFC7862].

5.1. DDP-Eligibility

Only the following XDR data items in the COMPOUND procedure of all NFS version 4 minor versions are DDP-eligible:

- * The opaque data field in the WRITE4args structure
- * The linkdata field of the NF4LNK arm in the createtype4 union
- * The opaque data field in the READ4resok structure
- * The linkdata field in the READLINK4resok structure

5.1.1. The NFSv4.2 READ_PLUS operation

NFS version 4.2 introduces an enhanced READ operation called READ_PLUS [RFC7862]. READ_PLUS enables an NFS server to compact returned READ data payloads. No part of a READ_PLUS Reply is DDP-eligible.

In a READ_PLUS result, returned file content appears as a list of one or more of the following items:

- * Regular data content, the same as the result of a traditional READ operation
- * Unallocated space in a file, where no data has been written, or previously-written data has been removed via a hole-punch operation
- * A counted pattern

Upon receipt of a READ_PLUS result, an NFSv4.2 client expands the returned list into its preferred representation of the original file content.

Before receiving that result, an NFSv4.2 client is unaware of how the NFS server has organized the file content. Thus it is not possible to predict the size or structure of a READ_PLUS Reply in advance. The use of direct data placement is therefore challenging. Moreover, the usual benefits of hardware-assisted data placement are entirely lost if the client must parse the result of each READ I/O.

Therefore this Upper Layer Binding does not make elements of an NFSv4.2 READ_PLUS Reply DDP-eligible. Further, this Upper Layer Binding recommends that NFS client implementations avoid using the READ_PLUS operation on NFS/RDMA mount points.

5.1.2. NFS Version 4 COMPOUND Requests

5.1.2.1. Multiple DDP-eligible Data Items

An NFS version 4 COMPOUND procedure can contain more than one operation that carries a DDP-eligible data item. An NFS version 4 client provides XDR Position values in each Read chunk to determine which chunk is associated with which argument data item. However, NFS version 4 server and client implementations must agree on how to pair Write chunks with returned result data items.

A "READ operation" refers to any NFS version 4 operation with a DDP-eligible result data item in the following lists. An NFS version 4 client applies the mechanism specified in Section 4.3.2 of [I-D.ietf-nfsv4-rpcrdma-version-two] to this class of operations as follows:

- * If an NFS version 4 client wishes all DDP-eligible items in an NFS reply to be conveyed inline, it leaves the Write list empty.

An NFS version 4 server acts as follows:

- * The first READ operation MUST use the first chunk in the Write list in an NFS version 4 COMPOUND procedure. The next READ operation uses the next Write chunk, and so on.
- * If an NFS version 4 client has provided a matching non-empty Write chunk, then the corresponding READ operation MUST return its DDP-eligible data item using that chunk.
- * If an NFS version 4 client has provided an empty matching Write chunk, then the corresponding READ operation MUST return all of its result data items inline.

- * If a READ operation returns a union arm which does not contain a DDP-eligible result, and the NFS version 4 client has provided a matching non-empty Write chunk, an NFS version 4 server MUST return an empty Write chunk in that Write list position.
- * If there are more READ operations than Write chunks, then remaining NFS Read operations in an NFS version 4 COMPOUND that have no matching Write chunk MUST return their results inline.

5.1.2.2. Chunk List Complexity

By default, the RPC-over-RDMA version 2 protocol limits the number of chunks or segments that may appear in Read or Write lists (see Section 5.2 of [I-D.ietf-nfsv4-rpcrdma-version-two]).

These implementation limits are significant when Kerberos integrity or privacy is in use [RFC7861]. GSS services increase the size of credential material in RPC headers, potentially requiring the more frequent use of less efficient Special Payload or Continued Payload messages.

NFS version 4 clients follow the prescriptions listed below when constructing RPC-over-RDMA version 2 messages in the absence of an explicit transport property exchange that alters these limits. NFS version 4 servers MUST accept and process all such requests.

- * The Read list can contain either a Call chunk, no more than one Read chunk, or both a Call chunk and one Read chunk.
- * The Write list can contain no more than one Write chunk.

NFS version 4 clients wishing to send more complex chunk lists can use transport properties to bound the complexity of NFS version 4 COMPOUNDS, limit the number of elements in scatter-gather operations, and avoid other sources of chunk overruns at the receiving peer.

5.1.2.3. NFS Version 4 COMPOUND Example

The following example shows a Write list with three Write chunks, A, B, and C. The NFS version 4 server consumes the provided Write chunks by writing the results of the designated operations in the compound request (READ and READLINK) back to each chunk.

Write list:

A --> B --> C

NFS version 4 COMPOUND request:

PUTFH	LOOKUP	READ	PUTFH	LOOKUP	READLINK	PUTFH	LOOKUP	READ
		v			v			v
		A			B			C

If the NFS version 4 client does not want the READLINK result returned via RDMA, it provides an empty Write chunk for buffer B to indicate that the READLINK result must be returned inline.

5.2. Reply Size Estimation

Within NFS version 4, there are certain variable-length result data items whose maximum size cannot be estimated by clients reliably because there is no protocol-specified size limit on these result arrays. These include:

- * The attrlist4 field
- * Fields containing ACLs such as `fattnr4_acl`, `fattnr4_dacl`, and `fattnr4_sacl`
- * Fields in the `fs_locations4` and `fs_locations_info4` data structures
- * Fields which pertain to pNFS layout metadata, such as `loc_body`, `loh_body`, `da_addr_body`, `lou_body`, `lrf_body`, `fattnr_layout_types`, and `fs_layout_types`

5.2.1. Reply Size Estimation for Minor Version 0

The NFS version 4.0 protocol itself does not impose any bound on the size of NFS Calls or Replies.

Variable-length `fattnr4` attributes make it particularly difficult for clients to predict the maximum size of some NFS version 4.0 Replies. Client implementations might rely upon internal architectural limits to constrain the reply size, but such limits are not always reliable. When an NFS version 4.0 client cannot predict the size of a Reply, it can rely on message continuation to enable a Reply under any circumstances.

5.2.2. Reply Size Estimation for Minor Version 1 and Newer

In NFS version 4.1 and newer minor versions, the `csa_fore_chan_attrs` argument of the `CREATE_SESSION` operation contains a `ca_maxresponsesize` field. The value in this field is the absolute maximum size of replies generated by an NFS version 4.1 server.

An NFS version 4 client can use this value when it is impossible to estimate a reply size upper bound precisely. In practice, objects such as ACLs, named attributes, layout bodies, and security labels are much smaller than this maximum.

5.3. RPC Binding Considerations

NFS version 4 servers are required to listen on TCP port 2049 and are not required to register with an `rpcbind` service [RFC7530]. Therefore, an NFS version 4 server supporting RPC-over-RDMA version 2 MUST use the alternative well-known port number for its RPC-over-RDMA service defined in Section 9.

5.4. Transport Considerations

5.4.1. Congestion Avoidance

Section 3.1 of [RFC7530] states:

Where an NFS version 4 implementation supports operation over the IP network protocol, the supported transport layer between NFS and IP MUST be an IETF standardized transport protocol that is specified to avoid network congestion; such transports include TCP and the Stream Control Transmission Protocol (SCTP).

Section 2.9.1 of [RFC8881] further states:

Even if NFS version 4.1 is used over a non-IP network protocol, it is RECOMMENDED that the transport support congestion control.

It is permissible for a connectionless transport to be used under NFS version 4.1; however, reliable and in-order delivery of data combined with congestion control by the connectionless transport is REQUIRED. As a consequence, UDP by itself MUST NOT be used as an NFS version 4.1 transport.

RPC-over-RDMA version 2 utilizes only reliable, connection-oriented transports that guarantee in-order delivery, meeting all the above requirements for NFS version 4.0 and 4.1. See Section 4.2.1 of [I-D.ietf-nfsv4-rpcrdma-version-two] for more details.

5.4.2. Retransmission and Keep-alive

NFS version 4 client implementations often rely on a transport-layer connection keep-alive mechanism to detect when an NFS version 4 server has become unresponsive. When an NFS server is no longer responsive, client-side keep-alive terminates the connection, triggering reconnection and RPC retransmission.

Some RDMA transports (such as the Reliable Connected QP type on InfiniBand) have no keep-alive mechanism. Without a disconnect or new RPC traffic, such connections can remain alive long after an NFS server has become unresponsive. Once an NFS client has consumed all available RPC-over-RDMA version 2 credits on that transport connection, it indefinitely awaits a reply before sending another RPC request.

NFS version 4 peers SHOULD reserve one RPC-over-RDMA version 2 credit for periodic server or connection health assessment. Either peer can use this credit to drive an RPC request on an otherwise idle connection, triggering either a quick affirmative server response or immediate connection termination.

In addition to network partition and request loss scenarios, RPC-over-RDMA version 2 peers can terminate a connection when a Transport header is malformed or when too many RPC-over-RDMA messages are sent without a credit update. In such cases:

- * If a transport error occurs (e.g., an RDMA2_ERROR type message is received) just before the disconnect or instead of a disconnect, the Requester MUST respond to that error as prescribed by the specification of the RPC transport. Then the NFS version 4 rules for handling retransmission apply.
- * If there is a transport disconnect and the Responder has provided no other response for a request, then only the NFS version 4 rules for handling retransmission apply.

5.5. Session-Related Considerations

The presence of an NFS version 4 session (as defined in [RFC8881]) does not affect the operation of RPC-over-RDMA version 2. None of the operations introduced to support NFS sessions (e.g., the SEQUENCE operation) contain DDP-eligible data items. There is no need to match the number of session slots with the available RPC-over-RDMA version 2 credits.

However, there are a few new cases where an RPC transaction can fail. For example, a Requester might receive, in response to an RPC request, an RDMA2_ERROR message with a `rdma_err` value of `RDMA2_ERR_BADXDR`. These situations are not different from existing RPC errors, which an NFS session implementation can already handle for other transport types. Moreover, there might be no SEQUENCE result available to the Requester to distinguish whether failure occurred before or after the Responder executed the requested operations.

When a transport error occurs (e.g., an RDMA2_ERROR type message is received), the Requester proceeds, as usual, to match the incoming XID value to a waiting RPC Call. The Requester terminates the RPC transaction and reports the result status to the RPC consumer. The Requester's session implementation then determines the session ID and slot for the failed request and performs slot recovery to make that slot usable again. Otherwise, that slot is rendered permanently unavailable.

When an NFS session is not present (for example, when NFS version 4.0 is in use), a transport error does not indicate whether the server has processed the arguments of the RPC Call, or whether the server has accessed or modified client memory associated with that RPC.

6. Upper-Layer Binding For NFS Version 4 Callbacks

The NFS version 4 family of protocols supports server-initiated callbacks to notify NFS version 4 clients of events such as recalled delegations.

6.1. NFS Version 4.0 Callback

An NFS version 4.0 client uses the SETCLIENTID operation for advertising the IP address, port, and netid of its NFS version 4.0 callback service. When an NFS version 4.0 server provides a backchannel service to an NFS version 4.0 client that uses RPC-over-RDMA version 2 for its forward channel, the server MUST advertise the backchannel service using either the "tcp" or "tcp6" netid.

Because the NFSv4.0 backchannel does not operate on RPC-over-RDMA, this document does not specify an Upper-Layer binding for the NFSv4.0 backchannel RPC program.

6.2. NFS Version 4.1 Callback

In NFS version 4.1 and newer minor versions, callback operations may appear on the same connection that is in use for NFS version 4 forward channel client requests. NFS version 4 clients and servers MUST use the mechanisms described in Section 4.5 of [I-D.ietf-nfsv4-rpcrdma-version-two] to convey backchannel operations on an RPC-over-RDMA version 2 transport.

The `csa_back_chan_attrs` argument of the `CREATE_SESSION` operation contains a `ca_maxresponsesize` field. The value in this field is the absolute maximum size of backchannel replies generated by a replying NFS version 4 client.

There are no DDP-eligible data items in callback procedures defined in NFS version 4.1 or NFS version 4.2. However, some callback operations, such as messages that convey device ID information, can be sizeable. A sender can use Message Continuation or a Special Payload message in this situation.

When an NFS version 4.1 client can support Special Payload Calls in its backchannel, it reports a backchannel `ca_maxrequestsize` that is larger than the connection's inline thresholds. Otherwise, an NFS version 4 server MUST use only Simple Payload or Continued Payload messages to convey backchannel operations.

7. Extending NFS Upper-Layer Bindings

RPC programs such as NFS must have an Upper-Layer Binding specification to operate on an RPC-over-RDMA version 2 transport [I-D.ietf-nfsv4-rpcrdma-version-two]. Via standards action, the Upper-Layer Binding specified in this document can be extended to cover versions of the NFS version 4 protocol specified after NFS version 4 minor version 2, or to cover separately published extensions to an existing NFS version 4 minor version, as described in [RFC8178].

8. Security Considerations

RPC-over-RDMA version 2 supports all RPC security models, including `RPCSEC_GSS` security and transport-level security [RFC7861]. The choice of what Direct Data Placement mechanism to convey RPC argument and results does not affect this since it changes only the method of data transfer. Because the current document defines only the binding of the NFS protocols atop RPC-over-RDMA version 2 [I-D.ietf-nfsv4-rpcrdma-version-two], all relevant security considerations are, therefore, described at that layer.

9. IANA Considerations

The use of direct data placement in NFS introduces a need for an additional port number assignment for networks that share traditional UDP and TCP port spaces with RDMA services. The DDP protocol is such an example [RFC5041].

For this purpose, the current document lists a set of port number assignments that IANA has already assigned for NFS/RDMA in the IANA port registry, according to the guidelines described in [RFC6335].

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 author requests that IANA add the current document as a reference for the existing nfsrdma port assignments. This document does not alter these assignments.

10. References

10.1. Normative References

- [I-D.ietf-nfsv4-rpcrdma-version-two]
Lever, C. and D. Noveck, "RPC-over-RDMA Version 2 Protocol", Work in Progress, Internet-Draft, draft-ietf-nfsv4-rpcrdma-version-two-06, 2 January 2022,
<<https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-rpcrdma-version-two-06>>.
- [RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995,
<<https://www.rfc-editor.org/rfc/rfc1833>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997,
<<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011,
<<https://www.rfc-editor.org/rfc/rfc6335>>.

- [RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <<https://www.rfc-editor.org/rfc/rfc7530>>.
- [RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <<https://www.rfc-editor.org/rfc/rfc7861>>.
- [RFC7862] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <<https://www.rfc-editor.org/rfc/rfc7862>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <<https://www.rfc-editor.org/rfc/rfc8881>>.

10.2. Informative References

- [RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <<https://www.rfc-editor.org/rfc/rfc1094>>.
- [RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <<https://www.rfc-editor.org/rfc/rfc1813>>.
- [RFC5041] Shah, H., Pinkerton, J., Recio, R., and P. Culley, "Direct Data Placement over Reliable Transports", RFC 5041, DOI 10.17487/RFC5041, October 2007, <<https://www.rfc-editor.org/rfc/rfc5041>>.
- [RFC8178] Noveck, D., "Rules for NFSv4 Extensions and Minor Versions", RFC 8178, DOI 10.17487/RFC8178, July 2017, <<https://www.rfc-editor.org/rfc/rfc8178>>.
- [XNFS] The Open Group, "Protocols for Interworking: XNFS, Version 3W", January 1998.

Acknowledgments

Thanks to Tom Talpey, who contributed the text of Section 5.1.2.2. David Noveck contributed the text of Section 5.5 and Section 7. The author also wishes to thank Bill Baker and Greg Marsden for their support of this work.

Special thanks go to Transport Area Directors Zaheduzzaman Sarker, NFSV4 Working Group Chairs Brian Pawlowski, and David Noveck, and NFSV4 Working Group Secretary Thomas Haynes for their support.

Author's Address

Charles Lever
Oracle Corporation
United States of America
Email: chuck.lever@oracle.com

Network File System Version 4
Internet-Draft
Updates: 5531 (if approved)
Intended status: Standards Track
Expires: 27 May 2021

T. Myklebust
Hammerspace
C. Lever, Ed.
Oracle
23 November 2020

Towards Remote Procedure Call Encryption By Default
draft-ietf-nfsv4-rpc-tls-11

Abstract

This document describes a mechanism that, through the use of opportunistic Transport Layer Security (TLS), enables encryption of Remote Procedure Call (RPC) transactions while they are in-transit. The proposed mechanism interoperates with ONC RPC implementations that do not support it. This document updates RFC 5531.

Note

Discussion of this draft takes place on the NFSv4 working group mailing list (nfsv4@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/nfsv4/>. Working Group information can be found at <https://datatracker.ietf.org/wg/nfsv4/about/>.

This note is to be removed before publishing as an RFC.

The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/chucklever/i-d-rpc-tls>. Instructions are on that page as well.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 27 May 2021.

Copyright Notice

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

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

Table of Contents

1. Introduction	3
2. Requirements Language	5
3. Terminology	5
4. RPC-Over-TLS in Operation	5
4.1. Discovering Server-side TLS Support	6
4.2. Authentication	7
4.2.1. Using TLS with RPCSEC GSS	8
5. TLS Requirements	8
5.1. Base Transport Considerations	9
5.1.1. Protected Operation on TCP	10
5.1.2. Protected Operation on UDP	10
5.1.3. Protected Operation on Other Transports	11
5.2. TLS Peer Authentication	12
5.2.1. X.509 Certificates Using PKIX Trust	12
5.2.2. Pre-Shared Keys	14
6. Implementation Status	14
6.1. DESY NFS server	14
6.2. Hammerspace NFS server	15
6.3. Linux NFS server and client	15
6.4. FreeBSD NFS server and client	15
7. Security Considerations	16
7.1. The Limitations of Opportunistic Security	16
7.1.1. STRIPTLS Attacks	17
7.1.2. Privacy Leakage Before Session Establishment	17
7.2. TLS Identity Management on Clients	18
7.3. Security Considerations for AUTH_SYS on TLS	18
7.4. Best Security Policy Practices	19
8. IANA Considerations	19
8.1. RPC Authentication Flavor	19
8.2. ALPN Identifier for SUNRPC	20

8.3. Object Identifier for PKIX Extended Key Usage	20
9. References	20
9.1. Normative References	21
9.2. Informative References	22
Appendix A. Known Weaknesses of the AUTH_SYS Authentication Flavor	23
Appendix B. ASN.1 Module	25
Acknowledgments	25
Authors' Addresses	26

1. Introduction

In 2014 the IETF published a document entitled "Pervasive Monitoring Is an Attack" [RFC7258], which recognized that unauthorized observation of network traffic had become widespread and was a subversive threat to all who make use of the Internet at large. It strongly recommended that newly defined Internet protocols should make a genuine effort to mitigate monitoring attacks. Typically this mitigation includes encrypting data in transit.

The Remote Procedure Call version 2 protocol has been a Proposed Standard for three decades (see [RFC5531] and its antecedents). Over twenty years ago, Eisler et al. first introduced RPCSEC GSS as an in-transit encryption mechanism for RPC [RFC2203]. However, experience has shown that RPCSEC GSS with in-transit encryption can be challenging to use in practice:

- * Parts of each RPC header remain in clear-text, constituting a loss of metadata confidentiality.
- * Offloading the GSS privacy service is not practical in large multi-user deployments since each message is encrypted using a key based on the issuing RPC user.

However strong GSS-provided confidentiality is, it cannot provide any security if the challenges of using it result in choosing not to deploy it at all.

Moreover, the use of AUTH_SYS remains common despite the adverse effects that acceptance of UIDs and GIDs from unauthenticated clients brings with it. Continued use is in part because:

- * Per-client deployment and administrative costs for the only well-defined alternative to AUTH_SYS are expensive at scale. For instance, administrators must provide keying material for each RPC client, including transient clients.

- * GSS host identity management and user identity management typically must be enforced in the same security realm. However, cloud providers, for instance, might prefer to remain authoritative for host identity but allow tenants to manage user identities within their private networks.

In view of the challenges with the currently available mechanisms for authenticating and protecting the confidentiality of RPC transactions, this document specifies a transport-layer security mechanism that complements the existing ones. The Transport Layer Security [RFC8446] (TLS) and Datagram Transport Layer Security [I-D.ietf-tls-dtls13] (DTLS) protocols are a well-established Internet building blocks that protect many standard Internet protocols such as the Hypertext Transport Protocol (HTTP) [RFC2818].

Encrypting at the RPC transport layer accords several significant benefits:

Encryption By Default: Transport encryption can be enabled without additional administrative tasks such as identifying client systems to a trust authority and providing each with keying material.

Encryption Offload: Hardware support for the GSS privacy service has not appeared in the marketplace. However, the use of a well-established transport encryption mechanism that is employed by other ubiquitous network protocols makes it more likely that encryption offload for RPC is practicable.

Securing AUTH_SYS: Most critically, transport encryption can significantly reduce several security issues inherent in the current widespread use of AUTH_SYS (i.e., acceptance of UIDs and GIDs generated by an unauthenticated client).

Decoupled User and Host Identities: TLS can be used to authenticate peer hosts while other security mechanisms can handle user authentication.

Compatibility: The imposition of encryption at the transport layer protects any upper-layer protocol that employs RPC, without alteration of the upper-layer protocol.

Further, Section 7 of the current document defines policies in line with [RFC7435] which enable RPC-over-TLS to be deployed opportunistically in environments that contain RPC implementations that do not support TLS. However, specifications for RPC-based upper-layer protocols should choose to require even stricter policies that guarantee encryption and host authentication is used for all RPC transactions to mitigate against pervasive monitoring attacks

[RFC7258]. Enforcing the use of RPC-over-TLS is of particular importance for existing upper-layer protocols whose security infrastructure is weak.

The protocol specification in the current document assumes that support for ONC RPC [RFC5531], TLS [RFC8446], PKIX [RFC5280], DNSSEC/DANE [RFC6698], and optionally RPCSEC_GSS [RFC2203] is available within the platform where RPC-over-TLS support is to be added.

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

This document adopts the terminology introduced in Section 3 of [RFC6973] and assumes a working knowledge of the Remote Procedure Call (RPC) version 2 protocol [RFC5531] and the Transport Layer Security (TLS) version 1.3 protocol [RFC8446].

Note also that the NFS community long ago adopted the use of the term "privacy" from documents such as [RFC2203]. In the current document, the authors use the term "privacy" only when referring specifically to the historic GSS privacy service defined in [RFC2203]. Otherwise, the authors use the term "confidentiality", following the practices of contemporary security communities.

We adhere to the convention that a "client" is a network host that actively initiates an association, and a "server" is a network host that passively accepts an association request.

RPC documentation historically refers to the authentication of a connecting host as "machine authentication" or "host authentication". TLS documentation refers to the same as "peer authentication". In the current document there is little distinction between these terms.

The term "user authentication" in the current document refers specifically to the RPC caller's credential, provided in the "cred" and "verf" fields in each RPC Call.

4. RPC-Over-TLS in Operation

4.1. Discovering Server-side TLS Support

The mechanism described in the current document interoperates fully with RPC implementations that do not support RPC-over-TLS. When an RPC-over-TLS-enabled peer encounters a peer that does not support RPC-over-TLS, policy settings on the RPC-over-TLS-enabled peer determine whether RPC operation continues without the use of TLS, or RPC operation is not permitted.

To achieve this interoperability, we introduce a new RPC authentication flavor called AUTH_TLS. The AUTH_TLS authentication flavor signals that the client wants to initiate TLS negotiation if the server supports it. Except for the modifications described in this section, the RPC protocol is unaware of security encapsulation at the transport layer. The value of AUTH_TLS is defined in Section 8.1.

An RPC client begins its communication with an RPC server by selecting a transport and destination port. The choice of transport and port is typically based on the RPC program that is to be used. The RPC client might query the RPC server's RPCBIND service to make this selection (The RPCBIND service is described in [RFC1833]). The mechanism described in the current document does not support RPC transports other than TCP and UDP. In all cases, an RPC server MUST listen on the same ports for (D)TLS-protected RPC programs as the ports used when (D)TLS is not available.

To protect RPC traffic to a TCP port, the RPC client opens a TCP connection to that port and sends a NULL RPC procedure with an auth_flavor of AUTH_TLS on that connection. To protect RPC traffic to a UDP port, the RPC client sends a UDP datagram to that port containing a NULL RPC procedure with an auth_flavor of AUTH_TLS. The client constructs this RPC procedure as follows:

- * The length of the opaque data constituting the credential sent in the RPC Call message MUST be zero.
- * The verifier accompanying the credential MUST be an AUTH_NONE verifier of length zero.
- * The flavor value of the verifier in the RPC Reply message received from the server MUST be AUTH_NONE.
- * The length of the verifier's body field is eight.
- * The bytes of the verifier's body field encode the ASCII characters "STARTTLS" as a fixed-length opaque.

The RPC server signals its corresponding support for RPC-over-TLS by replying with a `reply_stat` of `MSG_ACCEPTED` and an `AUTH_NONE` verifier containing the "STARTTLS" token. The client SHOULD proceed with TLS session establishment, even if the Reply's `accept_stat` is not `SUCCESS`. If the `AUTH_TLS` probe was done via TCP, the RPC client MUST send the "ClientHello" message on the same connection. If the `AUTH_TLS` probe was done via UDP, the RPC client MUST send the "ClientHello" message to the same UDP destination port.

Conversely, if the Reply's `reply_stat` is not `MSG_ACCEPTED`, if its verifier flavor is not `AUTH_NONE`, or if its verifier does not contain the "STARTTLS" token, the RPC client MUST NOT send a "ClientHello" message. RPC operation may continue, depending on local policy, but without confidentiality, integrity, or peer authentication protection from (D)TLS.

If, after a successful RPC `AUTH_TLS` probe, the subsequent (D)TLS handshake should fail for any reason, the RPC client reports this failure to the upper-layer application the same way it reports an `AUTH_ERROR` rejection from the RPC server.

If an RPC client uses the `AUTH_TLS` authentication flavor on any procedure other than the `NULL` procedure, or an RPC client sends an RPC `AUTH_TLS` probe within an existing (D)TLS session, the RPC server MUST reject that RPC Call by returning a `reply_stat` of `MSG_DENIED` with a `reject_stat` of `AUTH_ERROR` and an `auth_stat` of `AUTH_BADCRED`.

Once the TLS session handshake is complete, the RPC client and server have established a secure channel for exchanging RPC transactions. A successful `AUTH_TLS` probe on one particular port/transport tuple does not imply that RPC-over-TLS is available on that same server using a different port/transport tuple, nor does it imply that RPC-over-TLS will be available in the future using the successfully probed port.

4.2. Authentication

There is some overlap between the authentication capabilities of RPC and TLS. The goal of interoperability with implementations that do not support TLS requires limiting the combinations that are allowed and precisely specifying the role that each layer plays.

Each RPC server that supports RPC-over-TLS MUST possess a unique global identity (e.g., a certificate that is signed by a well-known trust anchor). Such an RPC server MUST request a TLS peer identity from each client upon first contact. There are two different modes of client deployment:

Server-only Host Authentication

In this type of deployment, the client can authenticate the server host using the presented server peer TLS identity, but the server cannot authenticate the client. In this situation, RPC-over-TLS clients are anonymous. They present no globally unique identifier to the server peer.

Mutual Host Authentication

In this type of deployment, the client possesses an identity that is backed by a trusted entity (e.g. a pre-shared key or a certificate validated with a certification path). As part of the TLS handshake, both peers authenticate using the presented TLS identities. If authentication of either peer fails, or if authorization based on those identities blocks access to the server, the peers MUST reject the association. Further explanation appears in Section 5.2.

In either of these modes, RPC user authentication is not affected by the use of transport layer security. When a client presents a TLS peer identity to an RPC server, the protocol extension described in the current document provides no way for the server to know whether that identity represents one RPC user on that client, or is shared amongst many RPC users. Therefore, a server implementation cannot utilize the remote TLS peer identity to authenticate RPC users.

4.2.1. Using TLS with RPCSEC GSS

To use GSS, an RPC server has to possess a GSS service principal. On a TLS session, GSS mutual (peer) authentication occurs as usual, but only after a TLS session has been established for communication. Authentication of RPCSEC GSS users is unchanged by the use of TLS.

RPCSEC GSS can also perform per-request integrity or confidentiality protection. When operating over a TLS session, these GSS services become largely redundant. An RPC implementation capable of concurrently using TLS and RPCSEC GSS MUST use GSS-API channel binding, as defined in [RFC5056], to determine when an underlying transport provides a sufficient degree of confidentiality. RPC-over-TLS implementations MUST provide the "tls-exporter" channel binding type, as defined in [I-D.ietf-kitten-tls-channel-bindings-for-tls13].

5. TLS Requirements

When peers negotiate a TLS session that is to transport RPC, the following restrictions apply:

- * Implementations MUST NOT negotiate TLS versions prior to v1.3 (for TLS [RFC8446] or DTLS [I-D.ietf-tls-dtls13] respectively). Support for mandatory-to-implement ciphersuites for the negotiated TLS version is REQUIRED.
- * Implementations MUST conform to the recommendations for TLS usage specified in BCP 195 [RFC7525]. Although RFC 7525 permits the use of TLS v1.2, the requirement to use TLS v1.3 or later for RPC-over-TLS takes precedence. Further, because TLS v1.3 ciphers are qualitatively different than cipher suites in previous versions of TLS and RFC 7525 predates TLS v1.3, the cipher suite recommendations in RFC 7525 do not apply to RPC-over-(D)TLS. A strict TLS mode for RPC-over-TLS that protects against STRIPTLS attacks is discussed in detail in Section 7.1.1.
- * Implementations MUST support certificate-based mutual authentication. Support for PSK mutual authentication is OPTIONAL; see Section 5.2.2 for further details.
- * Negotiation of a ciphersuite providing confidentiality as well as integrity protection is REQUIRED.

Client implementations MUST include the "application_layer_protocol_negotiation(16)" extension [RFC7301] in their "ClientHello" message and MUST include the protocol identifier defined in Section 8.2 in that message's ProtocolNameList value.

Similarly, in response to the "ClientHello" message, server implementations MUST include the "application_layer_protocol_negotiation(16)" extension [RFC7301] in their "ServerHello" message and MUST include only the protocol identifier defined in Section 8.2 in that message's ProtocolNameList value.

If the server responds incorrectly (for instance, if the "ServerHello" message does not conform to the above requirements), the client MUST NOT establish a TLS session for use with RPC on this connection. See [RFC7301] for further details about how to form these messages properly.

5.1. Base Transport Considerations

There is traditionally a strong association between an RPC program and a destination port number. The use of TLS or DTLS does not change that association. Thus it is frequently -- though not always -- the case that a single TLS session carries traffic for only one RPC program.

5.1.1. Protected Operation on TCP

The use of the Transport Layer Security (TLS) protocol [RFC8446] protects RPC on TCP connections. Typically, once an RPC client completes the TCP handshake, it uses the mechanism described in Section 4.1 to discover RPC-over-TLS support for that RPC program on that connection. Until an AUTH_TLS probe is done on a connection, the RPC server treats all traffic as RPC messages. If spurious traffic appears on a TCP connection between the initial clear-text AUTH_TLS probe and the TLS session handshake, receivers MUST discard that data without response and then SHOULD drop the connection.

The protocol convention specified in the current document assumes there can be no more than one concurrent TLS session per TCP connection. This is true of current generations of TLS, but might be different in a future version of TLS.

Once a TLS session is established on a TCP connection, no further clear-text communication can occur on that connection until the session is terminated. The use of TLS does not alter RPC record framing used on TCP transports.

Furthermore, if an RPC server responds with PROG_UNAVAIL to an RPC Call within an established TLS session, that does not imply that RPC server will subsequently reject the same RPC program on a different TCP connection.

Reverse-direction operation occurs only on connected transports such as TCP (see Section 2 of [RFC8167]). To protect reverse-direction RPC operations, the RPC server does not establish a separate TLS session on the TCP connection, but instead uses the existing TLS session on that connection to protect these operations.

When operation is complete, an RPC peer terminates a TLS session by sending a TLS Closure Alert. It may then close the TCP connection.

5.1.2. Protected Operation on UDP

RFC Editor: In the following section, please replace TBD with the connection_id extension number that is to be assigned in [I-D.ietf-tls-dtls-connection-id]. And, please remove this Editor's Note before this document is published.

The use of the Datagram Transport Layer Security (DTLS) protocol [I-D.ietf-tls-dtls13] protects RPC carried in UDP datagrams. As soon as a client initializes a UDP socket for use with an RPC service, it uses the mechanism described in Section 4.1 to discover RPC-over-DTLS support for that RPC program on that port. If spurious traffic

appears on a 5-tuple between the initial clear-text AUTH_TLS probe and the DTLS association handshake, receivers MUST discard that traffic without response.

Using DTLS does not introduce reliable or in-order semantics to RPC on UDP. The use of DTLS record replay protection is REQUIRED when transporting RPC traffic.

Each RPC message MUST fit in a single DTLS record. DTLS encapsulation has overhead, which reduces the Packetization Layer Path MTU (PLPMTU) and thus the maximum RPC payload size. A possible PLPMTU discovery mechanism is offered in [RFC8899].

The current document does not specify a mechanism that enables a server to distinguish between DTLS traffic and unprotected RPC traffic directed to the same port. To make this distinction, each peer matches ingress datagrams that appear to be DTLS traffic to existing DTLS session state. A peer treats any datagram that fails the matching process as an RPC message.

Multi-homed RPC clients and servers may send protected RPC messages via network interfaces that were not involved in the handshake that established the DTLS session. Therefore, when protecting RPC traffic, each DTLS handshake MUST include the "connection_id(TBD)" extension described in Section 9 of [I-D.ietf-tls-dtls13], and RPC-over-DTLS peer endpoints MUST provide a ConnectionID with a non-zero length. Endpoints implementing RPC programs that expect a significant number of concurrent clients SHOULD employ ConnectionIDs of at least 4 bytes in length.

Sending a TLS Closure Alert terminates a DTLS session. Because neither DTLS nor UDP provide in-order delivery, after session closure there can be ambiguity as to whether a datagram should be interpreted as DTLS protected or not. Therefore receivers MUST discard datagrams exchanged using the same 5-tuple that just terminated the DTLS session for a sufficient length of time to ensure that retransmissions have ceased and packets already in the network have been delivered. In the absence of more specific data, a period of 60 seconds is expected to suffice.

5.1.3. Protected Operation on Other Transports

Transports that provide intrinsic TLS-level security (e.g., QUIC) need to be addressed separately from the current document. In such cases, the use of TLS is not opportunistic as it can be for TCP or UDP.

RPC-over-RDMA can make use of transport layer security below the RDMA transport layer [RFC8166]. The exact mechanism is not within the scope of the current document. Because there might not be other provisions to exchange client and server certificates, authentication material exchange needs to be provided by facilities within a future version of the RPC-over-RDMA transport protocol.

5.2. TLS Peer Authentication

TLS can perform peer authentication using any of the following mechanisms.

5.2.1. X.509 Certificates Using PKIX Trust

X.509 certificates are specified in [X.509]. [RFC5280] provides a profile of Internet PKI X.509 public key infrastructure. RPC-over-TLS implementations are REQUIRED to support the PKIX mechanism described in [RFC5280].

The rules and guidelines defined in [RFC6125] apply to RPC-over-TLS certificates with the following considerations:

- * The DNS-ID identifier type is a subjectAltName extension that contains a dNSName, as defined in Section 4.2.1.6 of [RFC5280]. Support for the DNS-ID identifier type is REQUIRED in RPC-over-TLS client and server implementations. Certification authorities that issue such certificates MUST support the DNS-ID identifier type.
- * To specify the identity of an RPC peer as a domain name, the certificate MUST contain a subjectAltName extension that contains a dNSName. DNS domain names in RPC-over-TLS certificates MUST NOT contain the wildcard character '*' within the identifier.
- * To specify the identity of an RPC peer as a network identifier (netid) or a universal network address (uaddr), the certificate MUST contain a subjectAltName extension that contains an iPAddress.

When validating a server certificate, an RPC-over-TLS client implementation takes the following into account:

- * Certificate validation MUST include the verification rules as per Section 6 of [RFC5280] and Section 6 of [RFC6125].
- * Server certificate validation MUST include a check on whether the locally configured expected DNS-ID or iPAddress subjectAltName of the server that is contacted matches its presented certificate.

- * For RPC services accessed by their network identifiers (netids) and universal network addresses (uaddr), the iPAddress subjectAltName MUST be present in the certificate and MUST exactly match the address represented by the universal network address.

An RPC client's domain name and IP address are often assigned dynamically, thus RPC servers cannot rely on those to verify client certificates. Therefore, when an RPC-over-TLS client presents a certificate to an RPC-over-TLS server, the server takes the following into account:

- * The server MUST use a procedure conformant to Section 6 of [RFC5280]) to validate the client certificate's certification path.
- * The tuple (serial number of the presented certificate; Issuer) uniquely identifies the RPC client. The meaning and syntax of these fields is defined in Section 4 of [RFC5280]).

RPC-over-TLS implementations MAY allow the configuration of a set of additional properties of the certificate to check for a peer's authorization to communicate (e.g., a set of allowed values in subjectAltName:URI, a set of allowed X.509v3 Certificate Policies, or a set of extended key usages).

When the configured set of trust anchors changes (e.g., removal of a CA from the list of trusted CAs; issuance of a new CRL for a given CA), implementations SHOULD reevaluate the certificate originally presented in the context of the new configuration and terminate the TLS session if the certificate is no longer trustworthy.

5.2.1.1. Extended Key Usage Values

Section 4.2.1.12 of [RFC5280] specifies the extended key usage X.509 certificate extension. This extension, which may appear in end-entity certificates, indicates one or more purposes for which the certified public key may be used in addition to or in place of the basic purposes indicated in the key usage extension.

The current document defines two new KeyPurposeId values: one that identifies the RPC-over-TLS peer as an RPC client, and one that identifies the RPC-over-TLS peer as an RPC server.

The inclusion of the RPC server value (id-kp-rpcTLSServer) indicates that the certificate has been issued for allowing the holder to process RPC transactions.

The inclusion of the RPC client value (id-kp-rpcTLSClient) indicates that the certificate has been issued for allowing the holder to request RPC transactions.

5.2.2. Pre-Shared Keys

This mechanism is OPTIONAL to implement. In this mode, the RPC peer can be uniquely identified by keying material that has been shared out-of-band (see Section 2.2 of [RFC8446]). The PSK Identifier SHOULD be exposed at the RPC layer.

6. Implementation Status

This section is to be removed before publishing as an RFC.

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

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

6.1. DESY NFS server

Organization: DESY

URL: <https://desy.de>

Maturity: Implementation will be based on mature versions of the current document.

Coverage: The bulk of this specification is implemented including DTLS.

Licensing: LGPL

Implementation experience: The implementer has read and commented on the current document.

6.2. Hammerspace NFS server

Organization: Hammerspace

URL: <https://hammerspace.com>

Maturity: Prototype software based on early versions of the current document.

Coverage: The bulk of this specification is implemented. The use of DTLS functionality is not implemented.

Licensing: Proprietary

Implementation experience: No comments from implementors.

6.3. Linux NFS server and client

Organization: The Linux Foundation

URL: <https://www.kernel.org>

Maturity: Not complete.

Coverage: The bulk of this specification has yet to be implemented. The use of DTLS functionality is not planned.

Licensing: GPLv2

Implementation experience: A Linux in-kernel prototype is underway, but implementation delays have resulted from the challenges of handling a TLS handshake in a kernel environment. Those issues stem from the architecture of TLS and the kernel, not from the design of the RPC-over-TLS protocol.

6.4. FreeBSD NFS server and client

Organization: The FreeBSD Project

URL: <https://www.freebsd.org>

Maturity: Prototype software based on early versions of the current document.

Coverage: The bulk of this specification is implemented. The use of DTLS functionality is not planned.

Licensing: BSD

Implementation experience: Implementers have read and commented on the current document.

7. Security Considerations

One purpose of the mechanism described in the current document is to protect RPC-based applications against threats to the confidentiality of RPC transactions and RPC user identities. A taxonomy of these threats appears in Section 5 of [RFC6973]. Also, Section 6 of [RFC7525] contains a detailed discussion of technologies used in conjunction with TLS. Section 8 of [RFC5280] covers important considerations about handling certificate material securely. Implementers should familiarize themselves with these materials.

Once a TLS session is established, the RPC payload carried on TLS version 1.3 is forward-secure. However, implementers need to be aware that replay attacks can occur during session establishment. Remedies for such attacks are discussed in detail in Section 8 of [RFC8446]. Further, the current document does not provide a profile that defines the use of 0-RTT data (see Appendix E.5 of [RFC8446]). Therefore, RPC-over-TLS implementations MUST NOT use 0-RTT data.

7.1. The Limitations of Opportunistic Security

Readers can find the definition of Opportunistic Security in [RFC7435]. A discussion of its underlying principals appears in Section 3 of that document.

The purpose of using an explicitly opportunistic approach is to enable interoperation with implementations that do not support RPC-over-TLS. A range of options is allowed by this approach, from "no peer authentication or encryption" to "server-only authentication with encryption" to "mutual authentication with encryption". The actual security level may indeed be selected based on policy and without user intervention.

In environments where interoperability is a priority, the security benefits of TLS are partially or entirely waived. Implementations of the mechanism described in the current document must take care to accurately represent to all RPC consumers the level of security that is actually in effect, and are REQUIRED to provide an audit log of RPC-over-TLS security mode selection.

In all other cases, the adoption, implementation, and deployment of RPC-based upper-layer protocols that enforce the use of TLS authentication and encryption (when similar RPCSEC GSS services are not in use) is strongly encouraged.

7.1.1. STRIPTLS Attacks

A classic form of attack on network protocols that initiate an association in plain-text to discover support for TLS is a man-in-the-middle that alters the plain-text handshake to make it appear as though TLS support is not available on one or both peers. Client implementers can choose from the following to mitigate STRIPTLS attacks:

- * A TLSA record [RFC6698] can alert clients that TLS is expected to work, and provide a binding of hostname to X.509 identity. If TLS cannot be negotiated or authentication fails, the client disconnects and reports the problem. When an opportunistic security policy is in place, a client SHOULD check for the existence of a TLSA record for the target server before initiating an RPC-over-TLS association.
- * Client security policy can require that a TLS session is established on every connection. If an attacker spoofs the handshake, the client disconnects and reports the problem. This policy prevents an attacker from causing the client to silently fall back to plaintext. If TLSA records are not available, this approach is strongly encouraged.

7.1.2. Privacy Leakage Before Session Establishment

As mentioned earlier, communication between an RPC client and server appears in the clear on the network prior to the establishment of a TLS session. This clear-text information usually includes transport connection handshake exchanges, the RPC NULL procedure probing support for TLS, and the initial parts of TLS session establishment. Appendix C of [RFC8446] discusses precautions that can mitigate exposure during the exchange of connection handshake information and TLS certificate material that might enable attackers to track the RPC client. Note that when PSK authentication is used, the PSK identifier is exposed during the TLS handshake, and can be used to track the RPC client.

Any RPC traffic that appears on the network before a TLS session has been established is vulnerable to monitoring or undetected modification. A secure client implementation limits or prevents any RPC exchanges that are not protected.

The exception to this edict is the initial RPC NULL procedure that acts as a STARTTLS message, which cannot be protected. This RPC NULL procedure contains no arguments or results, and the AUTH_TLS authentication flavor it uses does not contain user information, so there is negligible privacy impact from this exception.

7.2. TLS Identity Management on Clients

The goal of the RPC-over-TLS protocol extension is to hide the content of RPC requests while they are in transit. The RPC-over-TLS protocol by itself cannot protect against exposure of a user's RPC requests to other users on the same client.

Moreover, client implementations are free to transmit RPC requests for more than one RPC user using the same TLS session. Depending on the details of the client RPC implementation, this means that the client's TLS credentials are potentially visible to every RPC user that shares a TLS session. Privileged users may also be able to access this TLS identity.

As a result, client implementations need to carefully segregate TLS credentials so that local access to it is restricted to only the local users that are authorized to perform operations on the remote RPC server.

7.3. Security Considerations for AUTH_SYS on TLS

Using a TLS-protected transport when the AUTH_SYS authentication flavor is in use addresses several longstanding weaknesses in AUTH_SYS (as detailed in Appendix A). TLS augments AUTH_SYS by providing both integrity protection and confidentiality that AUTH_SYS lacks. TLS protects data payloads, RPC headers, and user identities against monitoring and alteration while in transit.

TLS guards against in-transit insertion and deletion of RPC messages, thus ensuring the integrity of the message stream between RPC client and server. DTLS does not provide full message stream protection, but it does enable receivers to reject non-participant messages. In particular, transport layer encryption plus peer authentication protects receiving XDR decoders from deserializing untrusted data, a common coding vulnerability. However, these decoders would still be exposed to untrusted input in the case of the compromise of a trusted peer or Certificate Authority.

The use of TLS enables strong authentication of the communicating RPC peers, providing a degree of non-repudiation. When AUTH_SYS is used with TLS, but the RPC client is unauthenticated, the RPC server still acts on RPC requests for which there is no trustworthy

authentication. In-transit traffic is protected, but the RPC client itself can still misrepresent user identity without server detection. TLS without authentication is an improvement from AUTH_SYS without encryption, but it leaves a critical security exposure.

In light of the above, when AUTH_SYS is used, the use of a TLS mutual authentication mechanism is RECOMMENDED to prove that the RPC client is known to the RPC server. The server can then determine whether the UIDs and GIDs in AUTH_SYS requests from that client can be accepted, based on the authenticated identity of the client.

The use of TLS does not enable RPC clients to detect compromise that leads to the impersonation of RPC users. Also, there continues to be a requirement that the mapping of 32-bit user and group ID values to user identities is the same on both the RPC client and server.

7.4. Best Security Policy Practices

RPC-over-TLS implementations and deployments are strongly encouraged to adhere to the following policies to achieve the strongest possible security with RPC-over-TLS.

- * When using AUTH_NULL or AUTH_SYS, both peers are RECOMMENDED to have DNSSEC TLSA records, keys with which to perform mutual peer authentication using one of the methods described in Section 5.2, and a security policy that requires mutual peer authentication and rejection of a connection when host authentication fails.
- * RPCSEC GSS provides integrity and privacy services which are largely redundant when TLS is in use. These services SHOULD be disabled in that case.

8. IANA Considerations

RFC Editor: In the following subsections, please replace RFC-TBD with the RFC number assigned to this document. And, please remove this Editor's Note before this document is published.

8.1. RPC Authentication Flavor

Following Appendix B of [RFC5531], the authors request a single new entry in the RPC Authentication Flavor Numbers registry. The purpose of the new authentication flavor is to signal the use of TLS with RPC. This new flavor is not a pseudo-flavor.

The fields in the new entry are assigned as follows:

Identifier String: AUTH_TLS

Flavor Name: TLS

Value: 7 (to be confirmed by IANA)

Description: Indicates support for RPC-over-TLS.

Reference: RFC-TBD

8.2. ALPN Identifier for SUNRPC

Following Section 6 of [RFC7301], the authors request the allocation of the following value in the "Application-Layer Protocol Negotiation (ALPN) Protocol IDs" registry. The "sunrpc" string identifies SunRPC when used over TLS.

Protocol: SunRPC

Identification Sequence: 0x73 0x75 0x6e 0x72 0x70 0x63 ("sunrpc")

Reference: RFC-TBD

8.3. Object Identifier for PKIX Extended Key Usage

RFC Editor: In the following subsection, please replace XXX and YYY with the IANA numbers assigned to these new entries. And, please remove this Editor's Note before this document is published.

Per the Specification Required policy as defined in Section 4.6 of [RFC8126], the authors request the reservation of the following new values:

- * The RPC-over-TLS ASN.1 module in the "SMI Security for PKIX Extended Key Purpose" registry (1.3.6.1.5.5.7.3) (see Section 5.2.1.1 and Appendix B.
- * The RPC-over-TLS client certificate extended key usage (1.3.6.1.5.5.7.3.XXX). The description of this new entry should be "id-kp-rpcTLSClient".
- * The RPC-over-TLS server certificate extended key usage (1.3.6.1.5.5.7.3.YYY). The description of this new entry should be "id-kp-rpcTLSServer".

IANA should use the current document (RFC-TBD) as the reference for the new entries.

9. References

9.1. Normative References

- [I-D.ietf-kitten-tls-channel-bindings-for-tls13]
Whited, S., "Channel Bindings for TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-kitten-tls-channel-bindings-for-tls13-00, 11 June 2020, <<https://tools.ietf.org/html/draft-ietf-kitten-tls-channel-bindings-for-tls13-00>>.
- [I-D.ietf-tls-dtls-connection-id]
Rescorla, E., Tschofenig, H., and T. Fossati, "Connection Identifiers for DTLS 1.2", Work in Progress, Internet-Draft, draft-ietf-tls-dtls-connection-id-07, 21 October 2019, <<https://tools.ietf.org/html/draft-ietf-tls-dtls-connection-id-07>>.
- [I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-38, 29 May 2020, <<https://tools.ietf.org/html/draft-ietf-tls-dtls13-38>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<https://www.rfc-editor.org/info/rfc5056>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <<https://www.rfc-editor.org/info/rfc5531>>.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<https://www.rfc-editor.org/info/rfc6125>>.

- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<https://www.rfc-editor.org/info/rfc7525>>.
- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [X.509] International Telephone and Telegraph Consultative Committee, "ITU-T X.509 - Information technology - The Directory: Public-key and attribute certificate frameworks.", ISO/IEC 9594-8, CCITT Recommendation X.509, October 2019.

9.2. Informative References

- [RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995, <<https://www.rfc-editor.org/info/rfc1833>>.
- [RFC2203] Eisler, M., Chiu, A., and L. Ling, "RPCSEC_GSS Protocol Specification", RFC 2203, DOI 10.17487/RFC2203, September 1997, <<https://www.rfc-editor.org/info/rfc2203>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/info/rfc2818>>.

- [RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012, <<https://www.rfc-editor.org/info/rfc6698>>.
- [RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <<https://www.rfc-editor.org/info/rfc6973>>.
- [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<https://www.rfc-editor.org/info/rfc7258>>.
- [RFC7435] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time", RFC 7435, DOI 10.17487/RFC7435, December 2014, <<https://www.rfc-editor.org/info/rfc7435>>.
- [RFC8166] Lever, C., Ed., Simpson, W., and T. Talpey, "Remote Direct Memory Access Transport for Remote Procedure Call Version 1", RFC 8166, DOI 10.17487/RFC8166, June 2017, <<https://www.rfc-editor.org/info/rfc8166>>.
- [RFC8167] Lever, C., "Bidirectional Remote Procedure Call on RPC-over-RDMA Transports", RFC 8167, DOI 10.17487/RFC8167, June 2017, <<https://www.rfc-editor.org/info/rfc8167>>.
- [RFC8899] Fairhurst, G., Jones, T., Tüxen, M., Rüngeler, I., and T. Völker, "Packetization Layer Path MTU Discovery for Datagram Transports", RFC 8899, DOI 10.17487/RFC8899, September 2020, <<https://www.rfc-editor.org/info/rfc8899>>.

Appendix A. Known Weaknesses of the AUTH_SYS Authentication Flavor

The ONC RPC protocol, as specified in [RFC5531], provides several modes of security, traditionally referred to as "authentication flavors". Some of these flavors provide much more than an authentication service. We refer to these as authentication flavors, security flavors, or simply, flavors. One of the earliest and most basic flavors is AUTH_SYS, also known as AUTH_UNIX. Appendix A of [RFC5531] specifies AUTH_SYS.

AUTH_SYS assumes that the RPC client and server both use POSIX-style user and group identifiers (each user and group can be distinctly represented as a 32-bit unsigned integer). It also assumes that the client and server both use the same mapping of user and group to an integer. One user ID, one primary group ID, and up to 16

supplemental group IDs are associated with each RPC request. The combination of these identifies the entity on the client that is making the request.

A string identifies peers (hosts) in each RPC request. [RFC5531] does not specify any requirements for this string other than that is no longer than 255 octets. It does not have to be the same from request to request. Also, it does not have to match the DNS hostname of the sending host. For these reasons, even though most implementations fill in their hostname in this field, receivers typically ignore its content.

Appendix A of [RFC5531] contains a brief explanation of security considerations:

It should be noted that use of this flavor of authentication does not guarantee any security for the users or providers of a service, in itself. The authentication provided by this scheme can be considered legitimate only when applications using this scheme and the network can be secured externally, and privileged transport addresses are used for the communicating end-points (an example of this is the use of privileged TCP/UDP ports in UNIX systems -- note that not all systems enforce privileged transport address mechanisms).

It should be clear, therefore, that AUTH_SYS by itself (i.e., without strong client authentication) offers little to no communication security:

1. It does not protect the confidentiality or integrity of RPC requests, users, or payloads, relying instead on "external" security.
2. It does not provide authentication of RPC peer machines, other than inclusion of an unprotected domain name.
3. The use of 32-bit unsigned integers as user and group identifiers is problematic because these data types are not cryptographically signed or otherwise verified by any authority. In addition, the mapping of these integers to users and groups has to be consistent amongst a server and its cohort of clients.
4. Because the user and group ID fields are not integrity-protected, AUTH_SYS does not provide non-repudiation.

Appendix B. ASN.1 Module

RFC Editor: In the following section, please replace XXX and YYY with the IANA numbers assigned to these new entries. And, please remove this Editor's Note before this document is published.

<CODE BEGINS>

-- OID Arc

```
id-kp OBJECT IDENTIFIER ::=
{ iso(1) identified-organization(3) dod(6) internet(1)
  security(5) mechanisms(5) pkix(7) kp(3) }
```

-- Extended Key Usage Values

```
id-kp-rpcTLSClient OBJECT IDENTIFIER ::= { id-kp XXX }
id-kp-rpcTLSServer OBJECT IDENTIFIER ::= { id-kp YYY }
<CODE ENDS>
```

Acknowledgments

Special mention goes to Charles Fisher, author of "Encrypting NFSv4 with Stunnel TLS" (<https://www.linuxjournal.com/content/encrypting-nfsv4-stunnel-tls>). His article inspired the mechanism described in the current document.

Many thanks to Tigran Mkrtchyan and Rick Macklem for their work on prototype implementations and feedback on the current document.

Thanks to Derrell Piper for numerous suggestions that improved both this simple mechanism and the current document's security-related discussion.

Many thanks to Transport Area Director Magnus Westerlund for his sharp questions and careful reading of the final revisions of the current document. The text of Section 5.1.2 is mostly his contribution. Also, thanks to Benjamin Kaduk for his expert guidance on the use of PKIX and TLS. In addition, the authors thank the other members of the IESG for their astute review comments. These contributors made this a significantly better document.

The authors are additionally grateful to Bill Baker, David Black, Alan DeKok, Lars Eggert, Olga Kornievskaia, Greg Marsden, Alex McDonald, Justin Mazzola Paluska, Tom Talpey, Martin Thomson, and Nico Williams, for their input and support of this work.

Finally, special thanks to NFSV4 Working Group Chair and document shepherd David Noveck, NFSV4 Working Group Chairs Spencer Shepler and Brian Pawlowski, and NFSV4 Working Group Secretary Thomas Haynes for their guidance and oversight.

Authors' Addresses

Trond Myklebust
Hammerspace Inc
4300 El Camino Real Ste 105
Los Altos, CA 94022
United States of America

Email: trond.myklebust@hammerspace.com

Charles Lever (editor)
Oracle Corporation
United States of America

Email: chuck.lever@oracle.com

Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 7 January 2022

C. Lever, Ed.
Oracle
D. Noveck
NetApp
6 July 2021

RPC-over-RDMA Version 2 Protocol
draft-ietf-nfsv4-rpcrdma-version-two-05

Abstract

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

Note

This note is to be removed before publishing as an RFC.

Discussion of this draft takes place on the NFSv4 working group mailing list (nfsv4@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/nfsv4/>. Working Group information can be found at <https://datatracker.ietf.org/wg/nfsv4/about/>.

The source for this draft is maintained in GitHub. Suggested changes can be submitted as pull requests at <https://github.com/chucklever/i-d-rpcrdma-version-two>. Instructions are on that page as well.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 7 January 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	4
1.1. Design Goals	4
1.2. Motivation for a New Version	5
2. Requirements Language	6
3. Terminology	6
3.1. Remote Procedure Calls	6
3.2. Remote Direct Memory Access	10
4. RPC-over-RDMA Framework	13
4.1. Message Framing	13
4.2. Reliable Message Delivery	13
4.3. Initial Connection State	16
4.4. Using Direct Data Placement	17
4.5. Encoding Chunks	19
4.6. Reverse-Direction Operation	23
5. Transport Properties	26
5.1. Transport Properties Model	26
5.2. Current Transport Properties	28
6. Transport Messages	31
6.1. Transport Header Types	32
6.2. Headers and Chunks	33

6.3.	Header Types	34
6.4.	Transport Header Prefix	42
6.5.	Remote Invalidation	42
6.6.	Payload Formats	43
7.	Error Handling	49
7.1.	Basic Transport Stream Parsing Errors	50
7.2.	XDR Errors	51
7.3.	Responder RDMA Operational Errors	52
7.4.	Other Operational Errors	54
7.5.	RDMA Transport Errors	55
8.	XDR Protocol Definition	56
8.1.	Code Component License	56
8.2.	Extraction of the XDR Definition	58
8.3.	XDR Definition for RPC-over-RDMA Version 2 Core Structures	59
8.4.	XDR Definition for RPC-over-RDMA Version 2 Base Header Types	61
8.5.	Use of the XDR Description	64
9.	RPC Bind Parameters	65
10.	Implementation Status	66
11.	Security Considerations	66
11.1.	Memory Protection	67
11.2.	RPC Message Security	68
11.3.	Transport Properties	71
11.4.	Host Authentication	72
12.	IANA Considerations	72
13.	References	72
13.1.	Normative References	72
13.2.	Informative References	74
Appendix A.	ULB Specifications	76
A.1.	DDP-Eligibility	76
A.2.	Maximum Reply Size	78
A.3.	Reverse-Direction Operation	78
A.4.	Additional Considerations	78
A.5.	ULP Extensions	79
Appendix B.	Extending RPC-over-RDMA Version 2	79
B.1.	Documentation Requirements	80
B.2.	Adding New Header Types to RPC-over-RDMA Version 2	80
B.3.	Adding New Transport properties to the Protocol	81
B.4.	Adding New Error Codes to the Protocol	82
Appendix C.	Differences from RPC-over-RDMA Version 1	82
C.1.	Changes to the XDR Definition	83
C.2.	Transport Properties	84
C.3.	Credit Management Changes	84
C.4.	Inline Threshold Changes	85
C.5.	Message Continuation Changes	86
C.6.	Host Authentication Changes	86
C.7.	Support for Remote Invalidation	87

C.8. Integration of Reverse-Direction Operation	88
C.9. Error Reporting Changes	89
C.10. Changes in Terminology	89
Acknowledgments	90
Authors' Addresses	90

1. Introduction

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

Open Network Computing Remote Procedure Call (ONC RPC, often shortened in NFSv4 documents to RPC) [RFC5531] is a Remote Procedure Call protocol that runs over a variety of transports. Most RPC implementations today use UDP [RFC0768] or TCP [RFC0793]. On UDP, a datagram encapsulates each RPC message. Within a TCP byte stream, a record marking protocol delineates RPC messages.

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

1.1. Design Goals

The general mission of RPC-over-RDMA transports is to leverage network hardware capabilities to reduce host CPU needs related to the transport of RPC messages. In particular, this includes mitigating host interrupt rates and limiting the necessity to copy RPC payload bytes on receivers.

These hardware capabilities benefit both RPC clients and servers. On balance, however, the RPC-over-RDMA protocol design approach has been to bolster clients more than servers, as the client is typically where applications are most hungry for CPU resources.

Additionally, RPC-over-RDMA transports are designed to support RPC applications transparently. However, such transports can also provide mechanisms that enable further optimization of data transfer when RPC applications are structured to exploit direct data placement. In this context, the Network File System (NFS) family of protocols (as described in [RFC1094], [RFC1813], [RFC7530], [RFC7862], [RFC8881], and subsequent NFSv4 minor versions) are all potential beneficiaries of RPC-over-RDMA.

A complete problem statement appears in [RFC5532].

1.2. Motivation for a New Version

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

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

To address these issues in a way that preserves interoperability with existing RPC-over-RDMA version 1 deployments, the current document presents an updated version of the RPC-over-RDMA transport protocol.

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

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

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

3.1. Remote Procedure Calls

This section highlights critical elements of the RPC protocol [RFC5531] and the External Data Representation (XDR) [RFC4506] it uses. RPC-over-RDMA version 2 enables the transmission of RPC messages built using XDR and also uses XDR internally to describe its header format.

3.1.1. Upper-Layer Protocols

RPCs are an abstraction used to implement the operations of an Upper-Layer Protocol (ULP). For RPC-over-RDMA, "ULP" refers to an RPC Program and Version tuple, which is a versioned set of procedure calls that comprise a single well-defined API. One example of a ULP is the Network File System Version 4.0 [RFC7530]. In the current document, the term "RPC consumer" refers to an implementation of a ULP running on an RPC client.

3.1.2. Requesters and Responders

Like a local procedure call, every RPC procedure has a set of "arguments" and a set of "results". A calling context invokes a procedure, passing arguments to it, and the procedure subsequently returns a set of results. Unlike a local procedure call, the called procedure is executed remotely rather than in the local application's execution context.

The RPC protocol as described in [RFC5531] is fundamentally a message-passing protocol between one or more clients, where RPC consumers are running, and a server, where a remote execution context is available to process RPC transactions on behalf of these consumers.

ONC RPC transactions consist of two types of messages:

- * A CALL message, or "Call", requests work. An RPC Call message is designated by the value zero (0) in the message's msg_type field. The sender places a unique 32-bit value in the message's XID field to match this RPC Call message to a corresponding RPC Reply message.
- * A REPLY message, or "Reply", reports the results of work requested by an RPC Call message. An RPC Reply message is designated by the value one (1) in the message's msg_type field. The sender copies the value contained in an RPC Reply message's XID field from the RPC Call message whose results the sender is reporting.

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

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

The Requester deserializes the results and allows the RPC consumer to proceed. At this point, the RPC transaction designated by the XID in the RPC Call message is complete, and the XID is retired.

In summary, Requesters send RPC Call messages to Responders to initiate RPC transactions. Responders send RPC Reply messages to Requesters to complete the processing on an RPC transaction.

3.1.3. RPC Transports

The role of an "RPC transport" is to mediate the exchange of RPC messages between Requesters and Responders. An RPC transport bridges the gap between the RPC message abstraction and the native operations of a network transport (e.g., a socket).

RPC-over-RDMA is a connection-oriented RPC transport. When a transport type is connection-oriented, clients initiate transport connections, while servers wait passively to accept incoming connection requests.

3.1.3.1. Transport Failure Recovery

So that appropriate and timely recovery action can be taken, the transport implementation is responsible for notifying a Requester when an RPC Call or Reply was not able to be conveyed. Recovery can take the form of establishing a new connection, re-sending RPC Calls, or terminating RPC transactions pending on the Requester.

For instance, a connection loss may occur after a Responder has received an RPC Call but before it can send the matching RPC Reply. Once the transport notifies the Requester of the connection loss, the Requester can re-send all pending RPC Calls on a fresh connection.

3.1.3.2. Forward Direction

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

3.1.3.3. Reverse-Direction

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

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

RPC clients and servers are usually optimized to perform and scale well when handling traffic in the forward direction. They might not be prepared to handle operation in the reverse direction. Not until NFS version 4.1 [RFC8881] has there been a strong need to handle reverse-direction operation.

3.1.3.4. Bi-directional Operation

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

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

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

3.1.3.5. XID Values

Section 9 of [RFC5531] introduces the RPC transaction identifier, or "XID" for short. A connection peer interprets the value of an XID in the context of the message's `msg_type` field.

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

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

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

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

3.1.4. External Data Representation

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

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

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

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

3.1.4.1. XDR Opaque Data

Sometimes, a data item is to be transferred as-is, without encoding or decoding. We refer to the contents of such a data item as "opaque data". XDR encoding places the content of opaque data items directly into an XDR stream without altering it in any way. ULPs or applications perform any needed data translation in this case. Examples of opaque data items include the content of files or generic byte strings.

3.1.4.2. XDR Roundup

The number of octets in a variable-length data item precedes that item in an XDR stream. If the size of an encoded data item is not a multiple of four octets, the sender appends octets containing zero after the end of the data item. These zero octets shift the next encoded data item in the XDR stream so that it always starts on a four-octet boundary. The addition of extra octets does not change the encoded size of the data item. Receivers do not expose the extra octets to ULPs.

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

3.2. Remote Direct Memory Access

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

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

3.2.1. Direct Data Placement

Typically, RPC implementations copy the contents of RPC messages into a buffer before being sent. An efficient RPC implementation sends bulk data without first copying it into a separate send buffer.

However, socket-based RPC implementations are often unable to receive data directly into its final place in memory. Receivers often need to copy incoming data to finish an RPC operation, if only to adjust data alignment.

Although it may not be efficient, before an RDMA transfer, a sender may copy data into an intermediate buffer. After an RDMA transfer, a receiver may copy that data again to its final destination. In this document, the term "DDP" refers to any optimized data transfer where a receiving host's CPU does not move transferred data to another location after arrival.

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

3.2.2. RDMA Transport Operation

RDMA transports require that RDMA consumers provision resources in advance to achieve good performance during receive operations. An RDMA consumer might provide Receive buffers in advance by posting an RDMA Receive Work Request for every expected RDMA Send from a remote peer. These buffers are provided before the remote peer posts RDMA Send Work Requests. Thus this is often referred to as "pre-posting" buffers.

An RDMA Receive Work Request remains outstanding until the RDMA provider matches it to an inbound Send operation. The resources associated with that Receive must be retained in host memory, or "pinned", until the Receive completes.

Given these tenets of operation, the RPC-over-RDMA version 2 protocol assumes each transport provides the following abstract operations. A more complete discussion of these operations appears in [RFC5040].

3.2.2.1. Memory Registration

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

3.2.2.2. RDMA Send

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

3.2.2.3. RDMA Receive

The RDMA provider supports an RDMA Receive operation to receive data conveyed by incoming RDMA Send operations. To reduce the amount of memory that must remain pinned awaiting incoming Sends, the amount of memory posted per Receive is limited. The RDMA consumer (in this case, the RPC-over-RDMA version 2 protocol) provides flow control to prevent overrunning receiver resources.

3.2.2.4. RDMA Write

The RDMA provider supports an RDMA Write operation to place data directly into a remote memory region. The local host initiates an RDMA Write and the RDMA provider signals completion there. The remote RDMA provider does not signal completion on the remote peer. The local host provides the steering tag, the memory address, and the length of the remote peer's memory region.

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

3.2.2.5. RDMA Read

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

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

4. RPC-over-RDMA Framework

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

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

4.1. Message Framing

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

- * The "Transport stream" contains a header that describes and controls the transfer of the Payload stream in this RPC-over-RDMA message. Every RDMA Send on an RPC-over-RDMA version 2 connection MUST begin with a Transport stream.
- * The "Payload stream" contains part or all of a single RPC message. The sender MAY divide an RPC message at any convenient boundary but MUST send RPC message fragments in XDR stream order and MUST NOT interleave Payload streams from multiple RPC messages.

The RPC-over-RDMA framing mechanism described in this section replaces all other RPC framing mechanisms. Connection peers use RPC-over-RDMA framing even when the underlying RDMA protocol runs on a transport type with well-defined RPC framing, such as TCP. However, a ULP can negotiate the use of RDMA, dynamically enabling the use of RPC-over-RDMA on a connection established on some other transport type. Because RPC framing delimits an entire RPC request or reply, the resulting shift in framing must occur between distinct RPC messages, and in concert with the underlying transport.

4.2. Reliable Message Delivery

RPC-over-RDMA provides a reliable and in-order data transport service for RPC Calls and Replies.

RPC-over-RDMA transports MUST operate only on a reliable Queue Pair (QP) such as the RDMA RC (Reliable Connected) QP type as defined in Section 9.7.7 of [IBA]. The Marker PDU Aligned (MPA) protocol

[RFC5044], when deployed on a reliable transport such as TCP, provides similar functionality. Using a reliable QP type ensures in-transit data integrity and proper recovery from packet loss in the lower layers.

If any pre-posted Receive buffer on the connection is not large enough to contain an incoming message, the receiving RDMA provider cannot deliver that message to the upper-layer consumer. Likewise, if no pre-posted Receive buffer is available to accept an incoming message, the receiving RDMA provide cannot pass that message to the consumer. Exceeding these limits results in a transition to a QP error state, the loss of an in-flight message, and the potential loss of the connection.

Therefore, senders need to respect peer receiver resource limits to ensure that the transport service can deliver every message reliably. Two operational parameters communicate these limits between RPC-over-RDMA peers: credits and inline threshold.

4.2.1. Flow Control

RPC-over-RDMA version 2 employs end-to-end credit-based flow control on each connection to prevent senders from transmitting more messages than a receiver is prepared to accept [CBFC]. Credit-based flow control is relatively simple, providing robust operation in the face of bursty traffic and automated management of receive buffer allocation while enabling effective pipelining. A simplified sliding window approach is all that is necessary for our purposes because the underlying RDMA transport service already guarantees reliable and in-order message delivery.

An RPC-over-RDMA version 2 credit represents the capability to convey exactly one RPC-over-RDMA version 2 message, regardless of its size, via an RDMA Send/Receive pair. This arrangement enables RPC-over-RDMA version 2 transport connections to support multiple unacknowledged messages in each direction.

Because an RPC-over-RDMA version 2 connection is full-duplex, each connection peer has its own set of credits. The two receivers manage their credits independently, although they typically communicate these values by piggy-backing them on a payload-bearing message in the opposite direction.

Each RPC-over-RDMA version 2 message header contains two fields that handle credit accounting:

- * The `rdma_credit` field contains the receiver's credit window size. This field functions in much the same way as the RPC-over-RDMA version 1 used "credit grants". When sending an RPC-over-RDMA message, a peer fills in this field with the number of unacknowledged messages it is prepared to receive on this connection.
- * A second field, which is yet to be defined, reports the number of messages received so far. When sending an RPC-over-RDMA message, a peer fills in this field with the count of messages it has already received on this connection.

A sender also keeps track of the count of messages it has sent on the connection. The sender **MUST** stop transmitting messages when the number of messages it has already sent is about to exceed the number of messages the receiver has acknowledged plus the receiver's credit window.

A receiver **MAY** adjust the credit window to match the needs or policies in effect on either peer. For instance, a peer may reduce the size of its credit window to accommodate the available resources in a Shared Receive Queue. Certain RDMA implementations may impose additional flow-control restrictions, such as limits on RDMA Read operations in progress at the Responder. Accommodation of such checks is considered the responsibility of each RPC-over-RDMA version 2 implementation.

The credit window size **MUST** be less than the total sequence number range (let's make this a more quantitative statement later). The receiver generally chooses a credit window that is large enough to maximize throughput, given the bandwidth-delay product of the connection, while not overwhelming memory resources on the local system.

4.2.1.1. Asynchronous Credit Grants

Credit accounting information is usually piggy-backed on data-bearing messages. However, on occasion, a receiver might need to refresh its credit window without sending an RPC payload. A receiving peer can use an alternate header type when the sender's credit window is exhausted during a stream of unacknowledged messages. See Section 6.3.2 for information about this header type.

Unlike RPC-over-RDMA version 1, the credit window on an RPC-over-RDMA version 2 connection **MAY** be zero. In that case, the sender waits until the receiver sends it an asynchronous credit refresh.

Therefore, receivers MUST always be in a position to receive one asynchronous credit update message, in addition to payload-bearing messages, to prevent transport deadlock. A receiver can do this is by posting one more RDMA Receive than the advertised credit window.

4.2.2. Inline Threshold

An "inline threshold" value is the largest message size (in octets) that can be conveyed in one direction between peer implementations using RDMA Send and Receive channel operations. An inline threshold value is less than or equal to the largest number of octets the sender can post in a single RDMA Send operation. It is also less than or equal to the largest number of octets the receiver can reliably accept via a single RDMA Receive operation.

Each connection has two inline threshold values. There is one for messages flowing from Requester-to-Responder (referred to as the "call inline threshold"), and one for messages flowing from Responder-to-Requester (referred to as the "reply inline threshold").

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

4.3. Initial Connection State

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

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

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

Immediately after the client establishes a connection, it sends a single valid RPC-over-RDMA message with the value two (2) in the `rdma_vers` field. Because the server might support only RPC-over-RDMA version 1, this initial message MUST NOT be larger than the version 1 default inline threshold of 1024 octets.

4.3.1. Server Supports RPC-over-RDMA Version 2

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

4.3.2. Server Does Not Support RPC-over-RDMA Version 2

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

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

4.3.3. Client Does Not Support RPC-over-RDMA Version 2

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

4.4. Using Direct Data Placement

RPC-over-RDMA version 2 provides a mechanism for moving part of an RPC message via a data transfer distinct from RDMA Send and Receive. For example, a sender can remove one or more XDR data items from the Payload stream. These items are then conveyed via other mechanisms, such as one or more RDMA Read or Write operations.

4.4.1. Chunks and Segments

A Requester records the location information for each registered memory region associated with an RPC payload in the transport header of an RPC-over-RDMA message. With this information, the Responder uses RDMA Read and Write operations to retrieve arguments contained in the specified region of the Requester's memory or place results in that region.

A "segment" is a transport header data object that contains the precise coordinates of a contiguous registered memory region. Each segment contains the following information:

Handle: A steering Tag (STag) or R_key generated by registering this memory with the RDMA provider.

Length: The length of the segment's memory region, in octets. The length of a segment MAY be aligned to a single octet. An "empty segment" is defined as a segment with the value zero (0) in its length field.

Offset: The offset or beginning memory address of the segment's memory region.

The meaning of the values contained in these fields is elaborated in [RFC5040].

A "chunk" is simply a set of segments that have a related purpose. A Requester MAY divide a chunk into segments using any convenient boundaries. The length of a chunk is defined as the sum of the lengths of the segments that comprise it.

4.4.2. Reducing a Payload Stream

We refer to a data item that a sender removes from a Payload stream to transmit separately as a "reduced" data item. After a sender has finished removing XDR data items from a Payload stream, we refer to it as a "reduced" Payload stream. A set of segments that describe memory regions containing a single reduced data item is categorized as a "data item chunk."

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

Specifying which data items are DDP-eligible is done in separate standards track documents known as "Upper Layer Bindings". A ULB identifies which XDR data items a peer MAY transfer using DDP. We refer to such data items as "DDP-eligible." Senders MUST NOT reduce any other XDR data items. Detailed requirements for ULB specifications appear in Appendix A of the current document.

4.4.3. Moving Whole RPC Messages using Explicit RDMA

RPC-over-RDMA version 2 also enables the movement of a whole RPC message via data transfer distinct from RDMA Send and Receive. A sender registers the memory containing a Payload stream without regard to data item boundaries or DDP-eligibility. The Payload stream is then conveyed via other mechanisms, such as one or more RDMA Read or Write operations. A set of segments that describe memory regions containing a Payload stream is categorized as a "body chunk".

A sender may first reduce that Payload stream if it contains one or more DDP-eligible data items. The sender moves these data items using data items chunks, and the reduced Payload stream using a body chunk.

4.5. Encoding Chunks

The RPC-over-RDMA version 2 transport protocol does not place a limit on chunk size. However, each ULP may cap the amount of data that can be transferred by a single RPC transaction. For example, NFS implementations typically have settings that restrict the payload size of NFS READ and WRITE operations. The Responder can use such limits to sanity check chunk sizes before using them in RDMA operations.

4.5.1. Read Chunks

A "Read chunk" contains data that its receiver pulls from the sender. Each Read chunk is a set of one or more "Read segments" encoded as a list. A Read segment consists of a Position field followed by a segment, as defined in Section 4.4.1.

Position: The byte offset in the unreduced Payload stream where the receiver reinserts the data item conveyed in the chunk. The sender MUST compute the Position value from the beginning of the unreduced Payload stream, which begins at Position zero. All segments in the same Read chunk share the same Position value, even if one or more of the segments have a non-four-byte-aligned length. The value in this field MUST be a multiple of four.

When constructing an RPC-over-RDMA message, the sender registers memory regions containing data intended for RDMA Read operations. It advertises the coordinates of these regions in Read chunks added to the transport header of an RPC-over-RDMA message.

The receiver of this message then pulls the chunk's data from the sender using RDMA Read operations. When receiving a Read chunk, the receiver inserts the first Read segment in a Read chunk into the Payload stream at the byte offset indicated by its Position field. The receiver concatenates Read segments whose Position field value matches this offset until there are no more Read segments at that Position value.

4.5.1.1. The Read List

Each RPC-over-RDMA message carries a list of Read segments that make up the set of Read chunks for that message. When no RDMA Read operations are needed to complete the transmission of the message's Payload stream, the message's Read list is empty.

If a Responder receives a Read list whose segment position values do not appear in monotonically increasing order, it MUST discard the message without processing it and respond with an RDMA2_ERROR message with the `rdma_xid` field set to the `XID` of the malformed message and the `rdma_err` field set to `RDMA2_ERR_BAD_XDR`.

4.5.1.2. The Call Chunk

The Call chunk is a Read chunk that acts as a body chunk containing an RPC Call message. A Requester can utilize a Call chunk at any time. However, using a Call chunk is less efficient than an RDMA Send.

A Read chunk may act as either a data item chunk or a body chunk. When the chunk's position is zero, it acts as a body chunk. Otherwise, it is a data item chunk containing exactly one XDR data item.

4.5.1.3. Read Completion

A Responder acknowledges that it is finished with the Requester's Read chunk memory regions when it sends the corresponding RPC Reply message. The Requester may then invalidate memory regions belonging to Read chunks associated with the associated RPC Call message.

4.5.2. Write Chunks

Each "Write chunk" consists of a counted array of zero or more segments, as defined in Section 4.4.1. The function of a Write chunk depends on the direction of the containing RPC-over-RDMA message. In a Call message, a Write chunk advertises registered memory regions into which the Responder may push data. In a Reply message, a Write chunk reports how much data has been pushed.

A Requester provisions Write chunks for an RPC transaction long before the Responder has constructed a corresponding Reply message. A Requester typically does not know the actual length of the result data items or Reply to be returned, since the Reply does not yet exist. Thus, a Requester MUST provision Write chunks large enough to accommodate the maximum possible size of each returned data item.

An "empty Write chunk" is a Write chunk with a zero segment count. By definition, the length of an empty Write chunk is zero. An "unused Write chunk" has a non-zero segment count, but all of its segments are empty segments.

4.5.2.1. The Write List

Each RPC-over-RDMA message carries a list of Write chunks. When no DDP-eligible data items are to appear in the Reply to an RPC transaction, the Requester provides an empty Write list in the RPC Call, and the Responder leaves the Write list empty in the matching RPC Reply. When a Write chunk appears in the Write list, it acts only as a data item chunk.

For each Write chunk in the Write list, the Responder pushes one DDP-eligible data item to the Requester. It fills the chunk contiguously and in segment array order until the Responder has written that data item to the Requester in its entirety. The Responder MUST copy the segment count and all segments from the Requester-provided Write chunk into the RPC Reply message's transport header. As it does so, the Responder updates each segment length field to reflect the actual amount of data returned in that segment.

The Responder then sends the RPC Reply message via an RDMA Send operation.

4.5.2.2. The Reply Chunk

The Reply chunk is a single Write chunk that acts as a body chunk. that contains an RPC Reply message. When a Requester estimates that the Reply message can exceed the connection's ability to convey that Reply using RDMA Send operations, it should provision a Reply chunk.

4.5.2.3. Write Completion

A Responder acknowledges that it is finished updating the Requester's Write chunk memory regions when it sends the corresponding RPC Reply message. The RDMA provider guarantees that the written data is at rest before the next Receive operation, which typically contains the corresponding RPC Reply, completes. The Requester may then invalidate memory regions belonging to Write chunks associated with the associated RPC Call message.

4.5.2.4. Write Chunk Roundup

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

4.5.3. Reducing Complex XDR Data Types

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

However, a data item chunk MUST contain only one XDR data item, and the chunk MUST occupy a four-byte aligned length in the Payload stream so that subsequent data items remain properly aligned once the reduced data item is removed from the Payload stream.

4.5.3.1. Variable-Length Data Items

When a sender reduces a variable-length XDR data item, the length of the item MUST remain in the Payload stream. The sender MUST omit the item's XDR roundup padding from the Payload stream and the chunk. The chunk's total length MUST be the same as the encoded length of the data item.

4.5.3.2. Counted Arrays

When reducing a data item that is a counted array data type, the count of array elements MUST remain in the Payload stream. The sender MUST move the array elements into the chunk. For example, when encoding an opaque byte array as a chunk, the count of bytes stays in the Payload stream, and the sender places the bytes in the array in the chunk.

Individual array elements appear in a chunk in their entirety. For example, when encoding an array of arrays as a chunk, the count of items in the enclosing array stays in the Payload stream. But each enclosed array, including its item count, is transferred as part of the chunk.

4.5.3.3. Optional-Data

Similar to a counted array, when reducing an optional-data data type, the discriminator field **MUST** remain in the Payload stream. The sender **MUST** place the data, when present, in the chunk.

4.5.3.4. XDR Unions

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

4.6. Reverse-Direction Operation

The terminology used in this section is introduced in Section 3.1.3.3.

4.6.1. Sending a Reverse-Direction RPC Call

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

- * The server generates a new XID value (see Section 3.1.3.5 for full requirements) and places it in the `rdma_xid` field of the transport header and the `xid` field of the RPC Call message. The RPC Call header **MUST** start with the same XID value that is present in the transport header.
- * The `rdma_vers` field of each reverse-direction Call **MUST** contain the same value as forward-direction Calls on the same connection.
- * The server fills in the `rdma_credit` field with the credit values for the connection, as described in Section 4.2.1.
- * The server determines the Payload format for the RPC message and fills in the `rdma_htype` field as appropriate (see Sections 6.6 and 4.6.4). Section 4.6.4 also covers the disposition of the chunk lists.

4.6.2. Sending a Reverse-Direction RPC Reply

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

- * The client copies the XID value from the matching RPC Call and places it in the `rdma_xid` field of the transport header and the `xid` field of the RPC Reply message. The RPC Reply header MUST start with the same XID value that is present in the transport header.
- * The `rdma_vers` field of each reverse-direction Call MUST contain the same value as forward-direction Replies on the same connection.
- * The client fills in the `rdma_credit` field with the credit values for the connection, as described in Section 4.2.1.
- * The client determines the Payload format for the RPC message and fills in the `rdma_htype` field as appropriate (see Sections 6.6 and 4.6.4). Section 4.6.4 also covers the disposition of the chunk lists.

4.6.3. When Reverse-Direction Operation is Not Supported

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

If an endpoint is unprepared to receive a reverse-direction message, loss of the RDMA connection might result. Thus a denial of service can occur if an RPC server continues to send reverse-direction messages after a client that is not prepared to receive them reconnects to that server.

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

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

4.6.4. Using Chunks During Reverse-Direction Operation

Reverse-direction operations can use chunks for DDP-eligible data items and Special payload formats the same way chunks are used in forward-direction operation. Connection peers indicate their support for using chunks in the reverse direction as part of the exchange of Transport Properties just after a connection is established (see Section 5.2.5).

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

When there is no Upper-Layer Protocol need for chunks in the reverse direction, implementers MAY choose not to provide support for chunks in the reverse direction, thus avoiding the complexity of implementing support for RDMA Reads and Writes in the reverse direction. When an RPC-over-RDMA transport implementation does not support chunks in the reverse direction, RPC endpoints use only the Simple Payload format without data item chunks or the Continued Payload format without data item chunks to send RPC messages in the reverse direction.

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

4.6.5. Reverse-Direction Retransmission

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

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

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

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

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

5. Transport Properties

RPC-over-RDMA version 2 enables connection endpoints to exchange information about implementation properties. Compatible endpoints use this information to optimize data transfer. Initially, only a small set of transport properties are defined. The protocol provides header types to exchange transport properties (see 6.3.3 and 6.3.4).

Both the set of transport properties and the operations used to communicate them may be extended. Within RPC-over-RDMA version 2, such extensions are OPTIONAL. A discussion of extending the set of transport properties appears in Appendix B.3.

5.1. Transport Properties Model

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

The following XDR types handle transport properties:

```
<CODE BEGINS>
typedef rpcrdma2_propid uint32;

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

typedef rpcrdma2_propval rpcrdma2_propset<>;

typedef uint32 rpcrdma2_propsubset<>;
<CODE ENDS>
```

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

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

Although the `rdma_data` field is opaque, receivers interpret its contents using the XDR type associated with the property specified by `rdma_which`. When the contents of the `rdma_data` field do not conform to that XDR type, the receiver MUST return the error `RDMA2_ERR_BAD_PROPVAL` using the header type `RDMA2_ERROR`, as described in Section 6.3.1.

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

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

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

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

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

5.2. Current Transport Properties

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

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

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

Table 1

5.2.1. Maximum Send Size

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_SBSIZ = 1;
typedef uint32 rpcrdma2_prop_sbsiz;
<CODE ENDS>
```

5.2.2. Receive Buffer Size

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_RBSIZ = 2;
typedef uint32 rpcrdma2_prop_rbsiz;
<CODE ENDS>
```

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

- * Requesters may use the value to determine when to use a Call chunk or Message Continuation when sending a Call.

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

5.2.3. Maximum Segment Size

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_RSSIZ = 3;
typedef uint32 rpcrdma2_prop_rssiz;
<CODE ENDS>
```

5.2.4. Maximum Segment Count

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_RCSIZ = 4;
typedef uint32 rpcrdma2_prop_rcsiz;
<CODE ENDS>
```

5.2.5. Reverse-Direction Support

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

```
<CODE BEGINS>
const uint32 RDMA_RVRSDIR_NONE = 0;
const uint32 RDMA_RVRSDIR_SIMPLE = 1;
const uint32 RDMA_RVRSDIR_CONT = 2;
const uint32 RDMA_RVRSDIR_GENL = 3;

const uint32 RDMA2_PROPID_BRS = 5;
typedef uint32 rpcrdma2_prop_brs;
<CODE ENDS>
```

Multiple levels of support are distinguished:

- * The value RDMA2_RVRSDIR_NONE indicates that the sender does not support reverse-direction operation.

- * The value RDMA2_RVRSDIR_SIMPLE indicates that the sender supports using only Simple Format messages without data item chunks for reverse-direction messages.
- * The value RDMA2_RVRSDIR_CONT indicates that the sender supports using either Simple Format without data item chunks or Continued Format messages without data item chunks for reverse-direction messages.
- * The value RDMA2_RVRSDIR_GENL indicates that the sender supports reverse-direction messages in the same way as forward-direction messages.

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

5.2.6. Host Authentication Message

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

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

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

6. Transport Messages

Each transport message consists of multiple sections.

- * A transport header prefix, as defined in Section 6.4. Among other things, this structure indicates the header type.
- * The transport header proper, as defined by one of the sub-sections below. See Section 6.1 for the mapping between header types and the corresponding header structure.
- * Potentially, all or part of an RPC message payload.

This organization differs from that presented in the definition of RPC-over-RDMA version 1 [RFC8166], which defined the first and second of the items above as a single XDR data structure. The new

organization is in keeping with RPC-over-RDMA version 2's extensibility model, which enables the definition of new header types without modifying the XDR definition of existing header types.

6.1. Transport Header Types

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

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

Operation	Code	XDR type	Msg	Section
Report Transport Error	4	rpcrdma2_hdr_error	No	6.3.1
Grant Credits	5	void	No	6.3.2
Specify Properties (Middle)	6	rpcrdma2_hdr_connprop	No	6.3.3
Specify Properties (Final)	7	rpcrdma2_hdr_connprop	No	6.3.4
Convey External RPC Call Message	8	rpcrdma2_hdr_call_external	No	6.3.5

Convey Continued RPC Call Message	9	rpcrdma2_hdr_call_middle	Yes	6.3.6
Convey Inline RPC Call Message	10	rpcrdma2_hdr_call_inline	Yes	6.3.7
Convey External RPC Reply Message	11	rpcrdma2_hdr_reply_external	No	6.3.8
Convey Continued RPC Reply Message	12	rpcrdma2_hdr_reply_middle	Yes	6.3.9
Convey Inline RPC Reply Message	13	rpcrdma2_hdr_reply_inline	Yes	6.3.10

Table 2

RPC-over-RDMA version 2 peers are REQUIRED to support all message header types in Table 2. RPC-over-RDMA version 2 implementations that receive an unrecognized header type MUST respond with an RDMA2_ERROR message with an `rdma_err` field containing RDMA2_ERR_INVALID_HTYPE and drop the incoming message without processing it further.

6.2. Headers and Chunks

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

6.2.1. Common Transport Header Prefix

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

```
<CODE BEGINS>
struct rpcrdma_common {
    uint32      rdma_xid;
    uint32      rdma_vers;
    uint32      rdma_credit;
    uint32      rdma_htype;
};
<CODE ENDS>
```

RPC-over-RDMA version 2's use of these first four words aligns with that of version 1 as required by Section 4.2 of [RFC8166]. However, there are crucial structural differences in the XDR definition of RPC-over-RDMA version 2: in the way that these words are described by the respective XDR descriptions:

- * The header type is represented as a `uint32` rather than as an enum type. An enum would need to be modified to reflect additions to the set of header types made by later extensions.
- * The header type field is part of an XDR structure devoted to representing the transport header prefix, rather than being part of a discriminated union, that includes the body of each transport header type.
- * There is now a prefix structure (see Section 6.4) of which the `rpcrdma_common` structure is the initial segment. This prefix is a newly defined XDR object within the protocol description, which constrains the universal portion of all header types to the four words in `rpcrdma_common`.

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

6.3. Header Types

The header types defined and used in RPC-over-RDMA version 1 are not carried over into RPC-over-RDMA version 2, although there are easy equivalents to the version 1 procedures:

- * The RDMA2_ERROR header (defined in Section 6.3.1) has an XDR definition that differs from that in RPC-over-RDMA version 1, and its modifications are all compatible extensions.
- * Senders use RDMA2_CALL_INLINE or RDMA2_REPLY_INLINE (defined in Sections 6.3.7 and 6.3.10) in place of RDMA_MSG. There are minor differences in the on-the-wire format between the version 1 procedure and the version 2 header types.
- * Senders use RDMA2_CALL_EXTERNAL or RDMA2_REPLY_EXTERNAL (defined in Sections 6.3.5 and 6.3.8) in place of RDMA_NOMSG. There are minor differences in the on-the-wire format between the version 1 procedure and the version 2 header types.
- * RDMA2_CONNPROP_MIDDLE and RDMA2_CONNPROP_FINAL (defined in Sections 6.3.3 and 6.3.4) are new header types devoted to enabling connection peers to exchange information about their transport properties.

6.3.1. RDMA2_ERROR: Report Transport Error

RDMA2_ERROR reports a transport layer error on a previous transmission.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_ERROR = 4;

struct rpcrdma2_err_vers {
    uint32 rdma_vers_low;
    uint32 rdma_vers_high;
};

struct rpcrdma2_err_write {
    uint32 rdma_chunk_index;
    uint32 rdma_length_needed;
};

union rpcrdma2_hdr_error switch (rpcrdma2_errcode rdma_err) {
    case RDMA2_ERR_VERS:
        rpcrdma2_err_vers rdma_vrange;
    case RDMA2_ERR_READ_CHUNKS:
        uint32 rdma_max_chunks;
    case RDMA2_ERR_WRITE_CHUNKS:
        uint32 rdma_max_chunks;
    case RDMA2_ERR_SEGMENTS:
        uint32 rdma_max_segments;
    case RDMA2_ERR_WRITE_RESOURCE:
        rpcrdma2_err_write rdma_writeres;
    case RDMA2_ERR_REPLY_RESOURCE:
        uint32 rdma_length_needed;
    default:
        void;
};
<CODE ENDS>
```

See Section 7 for details on the use of this header type.

6.3.2. RDMA2_GRANT: Grant Credits

The RDMA2_GRANT header type enables a connection peer to update credit information without conveying a payload.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_GRANT = 5;
<CODE ENDS>
```

This message carries no payload except for a struct `rpcrdma2_hdr_prefix`. The `rdma_xid` field is unused. Senders MUST set the `rdma_xid` field to zero and receivers MUST ignore the value in this field.

6.3.3. RDMA2_CONNPROP_MIDDLE: Exchange Transport Properties

The RDMA2_CONNPROP_MIDDLE header type enables a connection peer to publish the properties of its implementation to its remote peer.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CONNPROP_MIDDLE = 6;

struct rpcrdma2_hdr_connprop {
    rpcrdma2_propset rdma_props;
};
<CODE ENDS>
```

A peer sends an RDMA2_CONNPROP_MIDDLE header type when it has one or more properties to send that do not fit within the default inline threshold for the RPC-over-RDMA version that is in effect.

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

If a peer sends follows an RDMA2_CONNPROP_MIDDLE header type with anything other than another RDMA2_CONNPROP_MIDDLE message or an RDMA2_CONNPROP_FINAL message, the receiver MUST respond with an RDMA2_ERROR header type and set its rdma_err field to RDMA2_ERR_INVALID_CONT and drop the incoming message without processing it further.

6.3.4. RDMA2_CONNPROP_FINAL: Exchange Transport Properties

The RDMA2_CONNPROP_FINAL header type enables a connection peer to publish the properties of its implementation to its remote peer.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CONNPROP_FINAL = 7;

struct rpcrdma2_hdr_connprop {
    rpcrdma2_propset rdma_props;
};
<CODE ENDS>
```

Each peer sends an RDMA2_CONNPROP_FINAL header type as the final CONNPROP-type message after the client has established a connection. The size of this message is limited to the default inline threshold for the RPC-over-RDMA version that is in effect.

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

If a peer sends a CONNPROP-type message on a connection after it has sent an RDMA2_CONNPROP_FINAL message, the receiver MUST respond with an RDMA2_ERROR header type and set its `rdma_err` field to `RDMA2_ERR_INVALID_CONT` and drop the incoming message without processing it further.

6.3.5. RDMA2_CALL_EXTERNAL: Convey External RPC Call Message

RDMA2_CALL_EXTERNAL conveys an RPC Call message payload using explicit RDMA operations. The Responder reads the Payload stream from a memory area specified by the Call chunk. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CALL_EXTERNAL = 8;

struct rpcrdma2_hdr_call_external {
    uint32_t                                rdma_inv_handle;

    struct rpcrdma2_read_list               *rdma_call;
    struct rpcrdma2_read_list               *rdma_reads;
    struct rpcrdma2_write_list              *rdma_provisional_writes;
    struct rpcrdma2_write_chunk             *rdma_provisional_reply;
};
<CODE ENDS>
```

`rdma_inv_handle`: The `rdma_inv_handle` field contains a 32-bit RDMA handle that the Responder may use in a Send With Invalidation operation. See Section 6.5.

`rdma_call`: The `rdma_call` field anchors a list of one or more Read segments that contain the RPC Call's Payload stream.

`rdma_reads`: The `rdma_reads` field anchors a list of zero or more Read segments that contain data item chunks.

`rdma_provisional_writes`: The `rdma_writes` field anchors a list of zero or more provisional Write chunks.

`rdma_provisional_reply`: The `rdma_reply` field is a list containing zero or one provisional Reply chunk.

6.3.6. RDMA2_CALL_MIDDLE: Convey Continued RPC Call Message

RDMA2_CALL_MIDDLE conveys a beginning or middle portion of an RPC Call message immediately following the transport header in the send buffer. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload. The sender sets the `rdma_remaining` field to the number of bytes in the RPC Call message payload that remain to be sent. The `rdma_rpc_first_word` field demarks the first word of the Payload stream.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_CALL_MIDDLE = 9;
```

```
struct rpcrdma2_hdr_call_middle {  
    uint32                rdma_remaining;  
  
    /* The rpc message starts here and continues  
     * through the end of the transmission. */  
    uint32                rdma_rpc_first_word;  
};  
<CODE ENDS>
```

If a peer sends follows an RDMA2_CALL_MIDDLE header type with anything other than an RDMA2_CALL_MIDDLE message or an RDMA2_CALL_INLINE message, the receiver MUST respond with an RDMA2_ERROR header type and set its `rdma_err` field to RDMA2_ERR_INVALID_CONT and drop the incoming message without processing it further.

6.3.7. RDMA2_CALL_INLINE: Convey Inline RPC Call Message

RDMA2_CALL_INLINE conveys the only or final portion of an RPC Call message. The `rdma_rpc_first_word` field demarks the first word of this Payload stream.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CALL_INLINE = 10;

struct rpcrdma2_hdr_call_inline {
    uint32                rdma_inv_handle;

    struct rpcrdma2_read_list    *rdma_reads;
    struct rpcrdma2_write_list   *rdma_provisional_writes;
    struct rpcrdma2_write_chunk  *rdma_provisional_reply;

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

rdma_inv_handle: The `rdma_inv_handle` field contains a 32-bit RDMA handle that the Responder may use in a Send With Invalidation operation. See Section 6.5.

rdma_reads: The `rdma_reads` field anchors a list of zero or more Read segments that contain only data item chunks. A Requester MUST NOT insert Position-zero Read chunks in this list.

rdma_provisional_writes: The `rdma_writes` field anchors a list of zero or more provisional Write chunks.

rdma_provisional_reply: The `rdma_reply` field is a list containing zero or one provisional Reply chunk.

6.3.8. RDMA2_REPLY_EXTERNAL: Convey External RPC Reply Message

`RDMA2_REPLY_EXTERNAL` conveys an RPC Reply message payload using explicit RDMA operations. In particular, it is referred to as a Special Format Reply when the Responder writes the RPC payload into a memory area specified by a Reply chunk. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_REPLY_EXTERNAL = 11;

struct rpcrdma2_hdr_reply_external {
    struct rpcrdma2_write_list    *rdma_writes;
    struct rpcrdma2_write_chunk  *rdma_reply;
};
<CODE ENDS>
```

`rdma_writes`: The `rdma_writes` field anchors a list of zero or more Write chunks that are either empty or contain reduced data items.

`rdma_reply`: The `rdma_reply` field is a list that MUST contain exactly one Reply chunk.

6.3.9. RDMA2_REPLY_MIDDLE: Convey Continued RPC Reply Message

`RDMA2_REPLY_MIDDLE` conveys a beginning or middle portion of an RPC Reply message immediately following the transport header in the send buffer. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload. The sender sets the `rdma_remaining` field to the number of bytes in the RPC Call message payload that remain to be sent. The `rdma_rpc_first_word` field demarks the first word of the Payload stream.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_REPLY_MIDDLE = 12;
```

```
struct rpcrdma2_hdr_reply_middle {
    uint32                rdma_remaining;

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

<CODE ENDS>

If a peer sends follows an `RDMA2_REPLY_MIDDLE` header type with anything other than an `RDMA2_REPLY_MIDDLE` message or an `RDMA2_REPLY_INLINE` message, the receiver MUST respond with an `RDMA2_ERROR` header type and set its `rdma_err` field to `RDMA2_ERR_INVALID_CONT` and drop the incoming message without processing it further.

6.3.10. RDMA2_REPLY_INLINE: Convey RPC Reply Message Inline

`RDMA2_REPLY_INLINE` conveys the only or final portion of an RPC Reply message immediately following the transport header in the send buffer. If the Reply message payload has been reduced, the `rdma_chunks` object carries the reduced data item chunks.

```

<CODE BEGINS>
const rpcrdma2_proc RDMA2_REPLY_INLINE = 13;

struct rpcrdma2_hdr_reply_inline {
    struct rpcrdma2_write_list  *rdma_writes;

    /* The rpc message starts here and continues
     * through the end of the transmission. */
    uint32                      rdma_rpc_first_word;
};
<CODE ENDS>

```

rdma_writes: The rdma_writes field anchors a list of zero or more Write chunks that are either empty or contain reduced data items.

6.4. Transport Header Prefix

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

```

<CODE BEGINS>
struct rpcrdma2_hdr_prefix {
    struct rpcrdma_common      rdma_start;
};
<CODE ENDS>

```

6.5. Remote Invalidation

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

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

A Responder never uses a Send With Invalidate Work Request when sending a control plane header type. This includes the RDMA2_ERROR header type, the RDMA2_GRANT header type, the RDMA2_CONNPROMP_MIDDLE header type, and the RDMA2_CONNPROMP_FINAL header type.

6.6. Payload Formats

RPC-over-RDMA version 2 provides several ways, known as "payload formats", to convey an RPC-over-RDMA message. A sender chooses the payload format for each message based on several factors:

- * The existence of DDP-eligible data items in the RPC message payload
- * The size of the RPC message payload
- * The direction of the RPC message (i.e., Call or Reply)
- * The available hardware resources
- * The arrangement of source and sink memory buffers

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

6.6.1. Simple Format

All RPC messages conveyed via RPC-over-RDMA version 2 need at least one RDMA Send operation to convey. Thus, the most efficient way to send an RPC message that is smaller than the inline threshold is to append the Payload stream directly to the Transport stream and use an RDMA Send to convey both. When no chunks are present, senders construct Calls and Replies the same way, and no other operations are needed.

6.6.1.1. Simple Format with Data Item Chunks

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

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

Simple Call

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

Simple Reply

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

6.6.1.2. Simple Format Examples

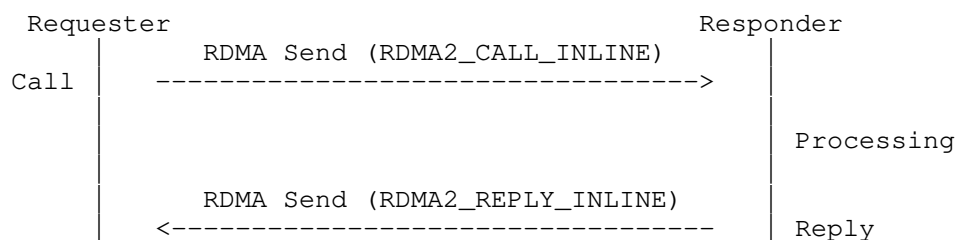


Figure 1: A Simple Call without data item chunks and a Simple Reply without data item chunks

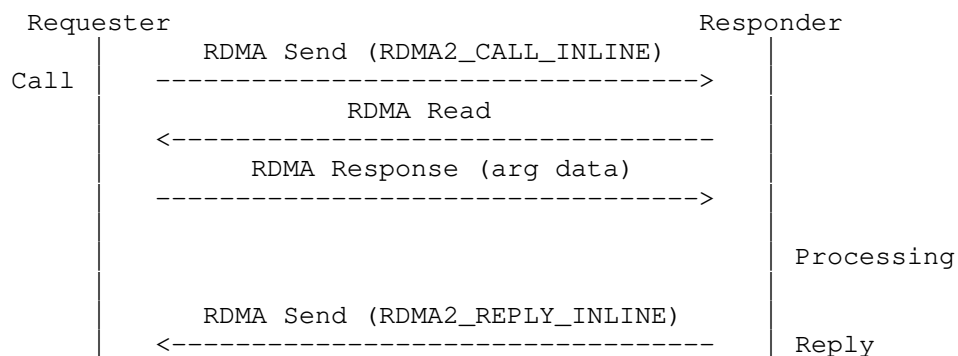


Figure 2: A Simple Call with a Read chunk and a Simple Reply without data item chunks

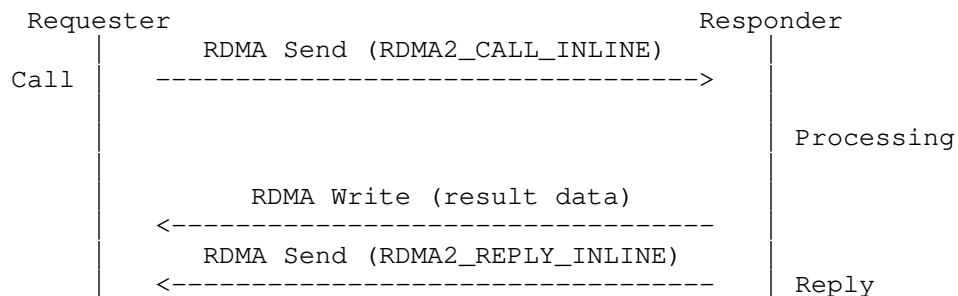


Figure 3: A Simple Call without data item chunks and a Simple Reply with a Write chunk

6.6.2. Continued Format

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

6.6.2.1. Continued Format with Data Item Chunks

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

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

Continued Call

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

Continued Reply

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

6.6.2.2. Continued Format Examples

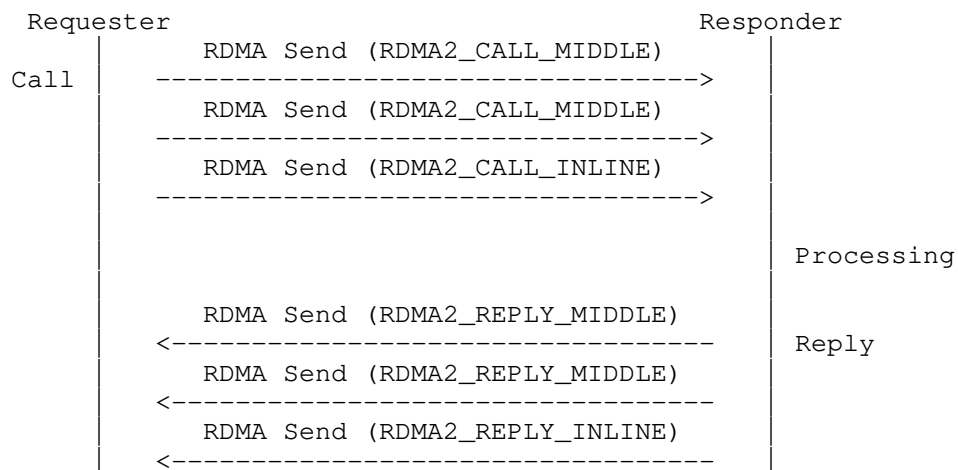


Figure 4: A Continued Call without data item chunks and a Continued Reply without data item chunks

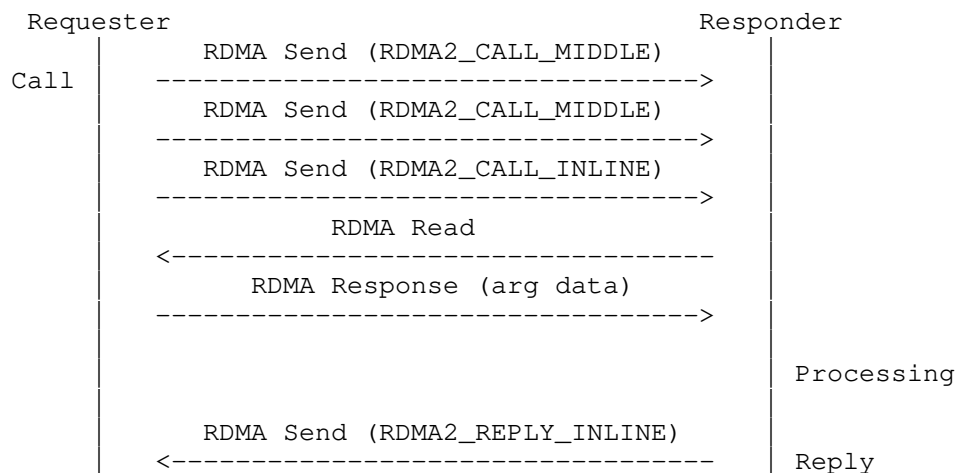


Figure 5: A Continued Call with a Read chunk and a Simple Reply without data item chunks

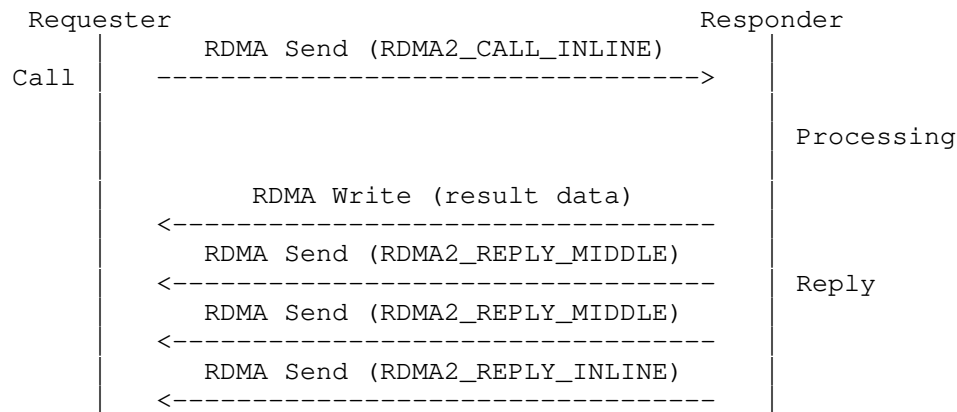


Figure 6: A Simple Call without data item chunks and a Continued Reply with a Write chunk

6.6.3. Special Format

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

To transmit a Special Format message, the sender transmits only the Transport stream with an RDMA Send operation. The sender does not include the Payload stream in the send buffer. Instead, the Requester provides a body chunk that the Responder uses to move the Payload stream.

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

Special Call

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

Special Reply

The Requester provisions a Reply chunk in advance. This body chunk is a Write chunk into which the Responder places the RPC Reply message's Payload stream. The Requester provisions the Reply chunk to accommodate the maximum expected reply size for that upper-layer operation.

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

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

6.6.3.1. Special Format Examples

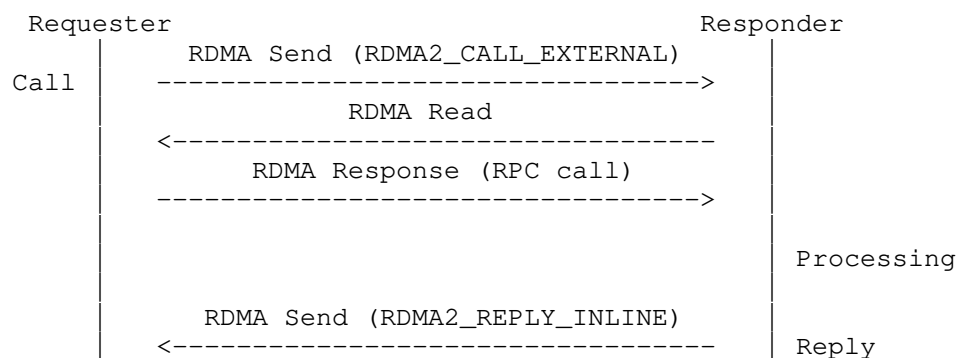


Figure 7: A Special Call and a Simple Reply without data item chunks

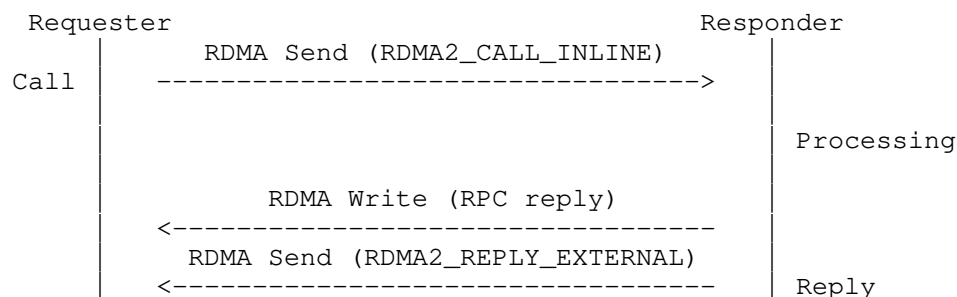


Figure 8: A Simple Call without data item chunks and a Special Reply

6.6.4. Choosing a Reply Payload Format

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

In RPC-over-RDMA version 2, a Requester can provide a Reply chunk for any transaction. The Responder can use the provided Reply chunk or it can decide to use another means to convey the RPC Reply. If the combination of the provided Write chunk list and Reply chunk is not adequate to convey a Reply, the Responder SHOULD use Message Continuation to send that Reply. If even that is not possible, the Responder sends an RDMA2_ERROR message to the Requester, as described in Section 6.3.1:

- * If the Write chunk list cannot accommodate the ULP's DDP-eligible data payload, the Responder sends an RDMA2_ERR_WRITE_RESOURCE error.
- * If the Reply chunk cannot accommodate the parts of the Reply that are not DDP-eligible, the Responder sends an RDMA2_ERR_REPLY_RESOURCE error.

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

7. Error Handling

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

When a request (for instance, an RPC Call or a control plane operation) is made, typically an RPC consumer blocks while waiting for the response. Thus when an incoming message conveys a request and that request cannot be acted upon, the receiver of that request

needs to report the problem to its sender in order to unblock waiters. Likewise, if, after processing a request, a sender is unable to transmit the response on an otherwise healthy connection, the sender needs to report that problem for the same reason.

The RDMA2_ERROR header type is used for this purpose. To form an RDMA2_ERROR type header:

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

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

7.1. Basic Transport Stream Parsing Errors

7.1.1. RDMA2_ERR_VERS

When a Responder detects an RPC-over-RDMA header version that it does not support (the current document defines version 2), it MUST respond with an RDMA2_ERROR message type and set its rdma_err field to RDMA2_ERR_VERS. The Responder then fills in the rpcrdma2_err_vers structure with the RPC-over-RDMA versions it supports. The Responder MUST silently discard the ingress message without passing it to the RPC layer.

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

7.1.2. RDMA2_ERR_VERS_MISMATCH

When a Responder receives a message with a transport protocol version that does not match the protocol version that was used in previous successful exchanges on the same connection, it MUST respond with an RDMA2_ERROR message type and set its rdma_err field to RDMA2_ERR_VERS_MISMATCH. The Responder MUST silently discard the ingress message without passing it to the RPC layer.

A Requester MUST NOT subsequently send a message that uses a protocol version that the Responder has indicated it does not recognize on this connection. The Requester can recover by sending the message again using a corrected protocol version, or it can terminate the RPC transaction associated with the XID in the rdma_xid field with an error.

7.1.3. RDMA2_ERR_INVALID_HTYPE

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

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

7.1.4. RDMA2_ERR_INVALID_CONT

If a Responder detects a problem with an ingress RPC-over-RDMA message that is part of a Message Continuation sequence, the Responder MUST set the rdma_err field to RDMA2_ERR_INVALID_CONT. The Responder MUST silently discard all ingress messages with an rdma_xid field that matches the failing message without reassembling the payload.

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

7.2. XDR Errors

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

- * The value of the `rdma_xid` field does not match the value of the `XID` field in the accompanying RPC message.
- * The receive buffer ends before the end of a data object contained in the Transport stream.

Moreover, when a Responder receives a valid RPC-over-RDMA header but the Responder's ULP implementation cannot parse the RPC arguments in the RPC Call, the Responder returns an RPC Reply with status `GARBAGE_ARGS`, using an `RDMA2_REPLY_INLINE` message type. This type of parsing failure might be due to mismatches between chunk sizes or offsets and the contents of the Payload stream, for example. In this case, the error is permanent, but the Requester has no way to know how much processing the Responder has completed for this RPC transaction.

7.2.1. `RDMA2_ERR_BAD_XDR`

If a Responder recognizes the values in the `rdma_vers` field, but it cannot otherwise parse the ingress Transport stream, it **MUST** set the `rdma_err` field to `RDMA2_ERR_BAD_XDR`. The Responder **MUST** silently discard the ingress message without passing it to the RPC layer.

`RDMA2_ERR_BAD_XDR` indicates a permanent error. Receipt of this error completes the RPC transaction associated with `XID` in the `rdma_xid` field.

7.2.2. `RDMA2_ERR_BAD_PROPVAL`

If a receiver recognizes the value in an ingress `rdma_which` field, but it cannot parse the accompanying `propval`, it **MUST** set the `rdma_err` field to `RDMA2_ERR_BAD_PROPVAL` (see Section 5.1). The receiver **MUST** silently discard the ingress message without applying any of its property settings.

7.3. Responder RDMA Operational Errors

In RPC-over-RDMA version 2, the Responder initiates RDMA Read and Write operations that target the Requester's memory. Problems might arise as the Responder attempts to use Requester-provided resources for RDMA operations. For example:

- * Usually, chunks can be validated only by using their contents to perform data transfers. If chunk contents are invalid (e.g., a memory region is no longer registered or a chunk length exceeds the end of the registered memory region), a Remote Access Error occurs.

- * If a Requester's Receive buffer is too small, the Responder's Send operation completes with a Local Length Error.
- * If the Requester-provided Reply chunk is too small to accommodate a large RPC Reply message, a Remote Access Error occurs. A Responder might detect this problem before attempting to write past the end of the Reply chunk.

RDMA operational errors can be fatal to the connection. To avoid a retransmission loop and repeated connection loss that deadlocks the connection, once the Requester has re-established a connection, the Responder SHOULD send an RDMA2_ERROR response to indicate that no RPC-level reply is possible for that transaction.

7.3.1. RDMA2_ERR_READ_CHUNKS

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

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

7.3.2. RDMA2_ERR_WRITE_CHUNKS

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

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

7.3.3. RDMA2_ERR_SEGMENTS

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

7.3.4. RDMA2_ERR_WRITE_RESOURCE

If a Requester has provided a Write chunk that is not large enough to contain a DDP-eligible result, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_WRITE_RESOURCE`. The Responder MUST set the `rdma_chunk_index` field to point to the first Write chunk in the transport header that is too short, or to zero to indicate that it was not possible to determine which chunk is too small. Indexing starts at one (1), which represents the first Write chunk. The Responder MUST set the `rdma_length_needed` to the number of bytes needed in that chunk to convey the result data item.

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

7.3.5. RDMA2_ERR_REPLY_RESOURCE

If a Responder cannot send an RPC Reply using Message Continuation and the Reply does not fit in the Reply chunk, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_REPLY_RESOURCE`. The Responder MUST set the `rdma_length_needed` to the number of Reply chunk bytes needed to convey the reply. The Requester can reconstruct the Call with more reply resources and resend it. If the Requester does not resend the Call (for instance, if the Responder set the length field to zero), it MUST terminate this RPC transaction with an error.

7.4. Other Operational Errors

While a Requester is constructing an RPC Call message, an unrecoverable problem might occur that prevents the Requester from posting further RDMA Work Requests on behalf of that message. As with other transports, if a Requester is unable to construct and transmit an RPC Call, the associated RPC transaction fails immediately.

After a Requester has received a Reply, if it is unable to invalidate a memory region due to an unrecoverable problem, the Requester MUST close the connection to protect that memory from Responder access before the associated RPC transaction is complete.

While a Responder is constructing an RPC Reply message or error message, an unrecoverable problem might occur that prevents the Responder from posting further RDMA Work Requests on behalf of that message. If a Responder is unable to construct and transmit an RPC Reply or RPC-over-RDMA error message, the Responder MUST close the connection to signal to the Requester that a reply was lost.

7.4.1. RDMA2_ERR_SYSTEM

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

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

7.5. RDMA Transport Errors

The RDMA connection and physical link provide some degree of error detection and retransmission. The Marker PDU Aligned Framing (MPA) protocol (as described in Section 7.1 of [RFC5044]) as well as the InfiniBand link layer [IBA] provide Cyclic Redundancy Check (CRC) protection of RDMA payloads. CRC-class protection is a general attribute of such transports.

Additionally, the RPC layer itself can accept errors from the transport and recover via retransmission. RPC recovery can typically handle complete loss and re-establishment of a transport connection.

The details of reporting and recovery from RDMA link-layer errors are described in specific link-layer APIs and operational specifications and are outside the scope of this protocol specification. See Section 11 for further discussion of RPC-level integrity schemes.

8. XDR Protocol Definition

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

8.1. Code Component License

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

```
<CODE BEGINS>
/// /*
///  * Copyright (c) 2010, 2020 IETF Trust and the persons
///  * identified as authors of the code.  All rights reserved.
///  *
///  * The authors of the code are:
///  * B. Callaghan, T. Talpey, C. Lever, and D. Noveck.
///  *
///  * Redistribution and use in source and binary forms, with
///  * or without modification, are permitted provided that the
///  * following conditions are met:
///  *
///  * - Redistributions of source code must retain the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer.
///  *
///  * - Redistributions in binary form must reproduce the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer in the documentation and/or other
///  *   materials provided with the distribution.
///  *
///  * - Neither the name of Internet Society, IETF or IETF
///  *   Trust, nor the names of specific contributors, may be
///  *   used to endorse or promote products derived from this
///  *   software without specific prior written permission.
///  *
///  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
///  * AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
///  * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
///  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
///  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
///  * EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
///  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
///  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
///  * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
///  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
///  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
///  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
///  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
///  * IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
///  * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
///  */
///
<CODE ENDS>
```

8.2. Extraction of the XDR Definition

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

```
<CODE BEGINS>
sed -n -e 's:^ */// ::p' -e 's:^ *///$::p'
<CODE ENDS>
```

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

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

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

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

Running the above script results in two files:

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

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

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

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

```
<CODE BEGINS>
/// /*****
///  *      Transport Header Prefixes
///  *****/
///
/// struct rpcrdma_common {
///     uint32      rdma_xid;
///     uint32      rdma_vers;
///     uint32      rdma_credit;
///     uint32      rdma_htype;
/// };
///
/// struct rpcrdma2_hdr_prefix {
///     struct rpcrdma_common      rdma_start;
/// };
///
/// /*****
///  *      Chunks and Chunk Lists
///  *****/
///
/// struct rpcrdma2_segment {
///     uint32 rdma_handle;
///     uint32 rdma_length;
///     uint64 rdma_offset;
/// };
///
```

```

/// struct rpcrdma2_read_segment {
///     uint32          rdma_position;
///     struct rpcrdma2_segment rdma_target;
/// };
///
/// struct rpcrdma2_read_list {
///     struct rpcrdma2_read_segment rdma_entry;
///     struct rpcrdma2_read_list    *rdma_next;
/// };
///
/// struct rpcrdma2_write_chunk {
///     struct rpcrdma2_segment rdma_target<>;
/// };
///
/// struct rpcrdma2_write_list {
///     struct rpcrdma2_write_chunk rdma_entry;
///     struct rpcrdma2_write_list  *rdma_next;
/// };
///
/// /*****
///  * Transport Properties
///  *****/
///
/// /*
///  * Types for transport properties model
///  */
/// typedef rpcrdma2_propid uint32;
///
/// struct rpcrdma2_propval {
///     rpcrdma2_propid rdma_which;
///     opaque          rdma_data<>;
/// };
///
/// typedef rpcrdma2_propval rpcrdma2_propset<>;
/// typedef uint32 rpcrdma2_propsubset<>;
///
/// /*
///  * Transport propid values for basic properties
///  */
/// const RDMA2_PROPID_SBSIZ = 1;
/// const RDMA2_PROPID_RBSIZ = 2;
/// const RDMA2_PROPID_RSSIZ = 3;
/// const RDMA2_PROPID_RCSIZ = 4;
/// const RDMA2_PROPID_BRS = 5;
/// const RDMA2_PROPID_HOSTAUTH = 6;
///
/// /*
///  * Types specific to particular properties

```

```

/// */
/// typedef uint32 rpcrdma2_prop_sbsiz;
/// typedef uint32 rpcrdma2_prop_rbsiz;
/// typedef uint32 rpcrdma2_prop_rssiz;
/// typedef uint32 rpcrdma2_prop_rcsiz;
/// typedef uint32 rpcrdma2_prop_brs;
/// typedef opaque rpcrdma2_prop_hostauth<>;
///
/// const RDMA2_RVRSDIR_NONE = 0;
/// const RDMA2_RVRSDIR_SIMPLE = 1;
/// const RDMA2_RVRSDIR_CONT = 2;
/// const RDMA2_RVRSDIR_GENL = 3;
///
/// /* FILE ENDS: common.x; */
<CODE ENDS>

```

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

```

<CODE BEGINS>
/// /*****
///  *      Descriptions of RPC-over-RDMA Header Types
///  *****/
///
/// /*
///  * Header Type Codes: Control plane operations.
///  */
/// const RDMA2_ERROR = 4;
/// const RDMA2_GRANT = 5;
/// const RDMA2_CONNPROP_MIDDLE = 6;
/// const RDMA2_CONNPROP_FINAL = 7;
///
/// /*
///  * Header Type Codes: Call messages.
///  */
/// const RDMA2_CALL_EXTERNAL = 8;
/// const RDMA2_CALL_MIDDLE = 9;
/// const RDMA2_CALL_INLINE = 10;
///
/// /*
///  * Header Type Codes: Reply messages.
///  */
/// const RDMA2_REPLY_EXTERNAL = 11;
/// const RDMA2_REPLY_MIDDLE = 12;
/// const RDMA2_REPLY_INLINE = 13;
///
/// /*
///  * Header Type to Report Errors.
///  */

```

```
/// const RDMA2_ERR_VERS = 1;
/// const RDMA2_ERR_BAD_XDR = 2;
/// const RDMA2_ERR_BAD_PROPVAL = 3;
/// const RDMA2_ERR_INVALID_HTYPE = 4;
/// const RDMA2_ERR_INVALID_CONT = 5;
/// const RDMA2_ERR_READ_CHUNKS = 6;
/// const RDMA2_ERR_WRITE_CHUNKS = 7;
/// const RDMA2_ERR_SEGMENTS = 8;
/// const RDMA2_ERR_WRITE_RESOURCE = 9;
/// const RDMA2_ERR_REPLY_RESOURCE = 10;
/// const RDMA2_ERR_VERS_MISMATCH = 11;
/// const RDMA2_ERR_SYSTEM = 100;
///
/// struct rpcrdma2_err_vers {
///     uint32 rdma_vers_low;
///     uint32 rdma_vers_high;
/// };
///
/// struct rpcrdma2_err_write {
///     uint32 rdma_chunk_index;
///     uint32 rdma_length_needed;
/// };
///
/// union rpcrdma2_hdr_error switch (rpcrdma2_errcode rdma_err) {
///     case RDMA2_ERR_VERS:
///         rpcrdma2_err_vers rdma_vrange;
///     case RDMA2_ERR_READ_CHUNKS:
///         uint32 rdma_max_chunks;
///     case RDMA2_ERR_WRITE_CHUNKS:
///         uint32 rdma_max_chunks;
///     case RDMA2_ERR_SEGMENTS:
///         uint32 rdma_max_segments;
///     case RDMA2_ERR_WRITE_RESOURCE:
///         rpcrdma2_err_write rdma_writeres;
///     case RDMA2_ERR_REPLY_RESOURCE:
///         uint32 rdma_length_needed;
///     default:
///         void;
/// };
///
/// /*
///  * Header Type to Exchange Transport Properties.
///  */
/// struct rpcrdma2_hdr_connprop {
///     rpcrdma2_propset rdma_props;
/// };
///
/// /*
```

```
/// * Header Types to Convey RPC Messages.
/// */
/// struct rpcrdma2_hdr_call_external {
///     uint32                rdma_inv_handle;
///
///     struct rpcrdma2_read_list  *rdma_call;
///     struct rpcrdma2_read_list  *rdma_reads;
///     struct rpcrdma2_write_list *rdma_provisional_writes;
///     struct rpcrdma2_write_chunk *rdma_provisional_reply;
/// };
///
/// struct rpcrdma2_hdr_call_middle {
///     uint32                rdma_remaining;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32                rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_hdr_call_inline {
///     uint32                rdma_inv_handle;
///
///     struct rpcrdma2_read_list  *rdma_reads;
///     struct rpcrdma2_write_list *rdma_provisional_writes;
///     struct rpcrdma2_write_chunk *rdma_provisional_reply;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32                rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_hdr_reply_external {
///     struct rpcrdma2_write_list *rdma_writes;
///     struct rpcrdma2_write_chunk *rdma_reply;
/// };
///
/// struct rpcrdma2_hdr_reply_middle {
///     uint32                rdma_remaining;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32                rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_hdr_reply_inline {
///     struct rpcrdma2_write_list *rdma_writes;
///
///     /* The rpc message starts here and continues
```

```
///          * through the end of the transmission. */
///          uint32                      rdma_rpc_first_word;
/// };
///
/// /* FILE ENDS: baseops.x; */
<CODE ENDS>
```

8.5. Use of the XDR Description

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

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

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

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

9. RPC Bind Parameters

Before establishing a new connection, an RPC client obtains a transport address for the RPC server. The means used to obtain this address and to open an RDMA connection is dependent on the type of RDMA transport and is the responsibility of each RPC protocol binding and its local implementation.

RPC services typically register with a portmap or rpcbind service [RFC1833], which associates an RPC Program number with a service address. This policy is no different with RDMA transports. However, a distinct service address (port number) is sometimes required for operation on RPC-over-RDMA.

When mapped atop MPA [RFC5044], which uses IP port addressing due to its layering on TCP or SCTP, port mapping is trivial and consists merely of issuing the port in the connection process. The NFS/RDMA protocol service address has been assigned port 20049 by IANA for this deployment scenario [RFC8267].

When mapped atop InfiniBand [IBA], which uses a service endpoint naming scheme based on a Group Identifier (GID), a translation MUST be employed. One such translation is described in Annexes A3 (Application Specific Identifiers), A4 (Sockets Direct Protocol (SDP)), and A11 (RDMA IP CM Service) of [IBA], which is appropriate for translating IP port addressing to the InfiniBand network. Therefore, in this case, IP port addressing may be readily employed by the upper layer.

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

- * One possibility is to have the server register its mapped IP port with the rpcbind service under the netid (or netids) defined in [RFC8166]. An RPC-over-RDMA-aware RPC client can then resolve its desired service to a mappable port and proceed to connect. This method is the most flexible and compatible approach for those upper layers that are defined to use the rpcbind service.

- * A second possibility is to have the RPC server's portmapper register itself on the RDMA interconnect at a "well-known" service address (on UDP or TCP, this corresponds to port 111). An RPC client can connect to this service address and use the portmap protocol to obtain a service address in response to a program number (e.g., a TCP port number or an InfiniBand GID).
- * Alternately, an RPC client can connect to the mapped well-known port for the service itself, if it is appropriately defined. By convention, the NFS/RDMA service, when operating atop an InfiniBand fabric, uses the same 20049 assignment as for MPA.

Historically, different RPC protocols have taken different approaches to their port assignments. The current document leaves the specific method for each RPC-over-RDMA-enabled ULB.

[RFC8166] defines two new netid values to be used for registration of upper layers atop MPA and (when a suitable port translation service is available) InfiniBand. Additional RDMA-capable networks MAY define their own netids, or if they provide a port translation, they MAY share the one defined in [RFC8166].

10. Implementation Status

This section is to be removed before publishing as an RFC.

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

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

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

11. Security Considerations

11.1. Memory Protection

A primary consideration is the protection of the integrity and confidentiality of host memory by an RPC-over-RDMA transport. The use of an RPC-over-RDMA transport protocol MUST NOT introduce vulnerabilities to system memory contents nor memory owned by user processes. Any RDMA provider used for RPC transport MUST conform to the requirements of [RFC5042] to satisfy these protections.

11.1.1. Protection Domains

The use of a Protection Domain to limit the exposure of memory regions to a single connection is critical. Any attempt by an endpoint not participating in that connection to reuse memory handles needs to result in immediate failure of that connection. Because ULP security mechanisms rely on this aspect of Reliable Connected behavior, implementations SHOULD cryptographically authenticate connection endpoints.

11.1.2. Handle (STag) Predictability

Implementations should use unpredictable memory handles for any operation requiring exposed memory regions. Exposing a continuously registered memory region allows a remote host to read or write to that region even when an RPC involving that memory is not underway. Therefore, implementations should avoid the use of persistently registered memory.

11.1.3. Memory Protection

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

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

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

11.1.4. Denial of Service

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

A Responder is not obliged to pull unreasonably large Read chunks. A Responder can use an RDMA2_ERROR response to terminate RPCs with unreadable Read chunks. If a Responder transmits more data than a Requester is prepared to receive in a Write or Reply chunk, the RDMA provider typically terminates the connection. For further discussion, see Section 6.3.1. Such repeated connection termination can deny service to other users sharing the connection from the errant Requester.

An RPC-over-RDMA transport implementation is not responsible for throttling the RPC request rate, other than to keep the number of concurrent RPC transactions at or under the per connection credit window (see Section 4.2.1). A sender can trigger a self-denial of service by exceeding the credit window repeatedly.

When an RPC transaction terminates due to a signal or premature exit of an application process, a Requester should invalidate the RPC's Write and Reply chunks. Invalidation prevents the subsequent arrival of the Responder's Reply from altering the memory regions associated with those chunks after the Requester has released that memory.

On the Requester, a malfunctioning application or a malicious user can create a situation where RPCs initiate and abort continuously, resulting in Responder replies that terminate the underlying RPC-over-RDMA connection repeatedly. Such situations can deny service to other users sharing the connection from that Requester.

11.2. RPC Message Security

ONC RPC provides cryptographic security via the RPCSEC_GSS framework [RFC7861]. RPCSEC_GSS implements message authentication (rpc_gss_svc_none), per-message integrity checking (rpc_gss_svc_integrity), and per-message confidentiality (rpc_gss_svc_privacy) in a layer above the RPC-over-RDMA transport. The integrity and privacy services require significant computation and movement of data on each endpoint host. Some performance benefits enabled by RDMA transports can be lost.

11.2.1. RPC-over-RDMA Protection at Other Layers

For any RPC transport, utilizing RPCSEC_GSS integrity or privacy services has performance implications. Protection below the RPC implementation is often a better choice in performance-sensitive deployments, especially if it, too, can be offloaded. Certain implementations of IPsec can be co-located in RDMA hardware, for example, without change to RDMA consumers and with little loss of data movement efficiency. Such arrangements can also provide a higher degree of privacy by hiding endpoint identity or altering the frequency at which messages are exchanged, at a performance cost.

Implementations MAY negotiate the use of protection in another layer through the use of an RPCSEC_GSS security flavor defined in [RFC7861] in conjunction with the Channel Binding mechanism [RFC5056] and IPsec Channel Connection Latching [RFC5660].

11.2.2. RPCSEC_GSS on RPC-over-RDMA Transports

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

RPCSEC_GSS extends the ONC RPC protocol without changing the format of RPC messages. By observing the conventions described in this section, an RPC-over-RDMA transport can convey RPCSEC_GSS-protected RPC messages interoperably.

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

11.2.2.1. RPCSEC_GSS Context Negotiation

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

11.2.2.2. RPC-over-RDMA with RPCSEC_GSS Authentication

The RPCSEC_GSS authentication service has no impact on the DDP-eligibility of data items in a ULP.

However, RPCSEC_GSS authentication material appearing in an RPC message header can be larger than, say, an AUTH_SYS authenticator. In particular, when an RPCSEC_GSS pseudoflavor is in use, a Requester needs to accommodate a larger RPC credential when marshaling RPC Calls and needs to provide for a maximum size RPCSEC_GSS verifier when allocating reply buffers and Reply chunks.

RPC messages, and thus Payload streams, are larger on average as a result. ULP operations that fit in a Simple Format message when a simpler form of authentication is in use might need to be reduced or conveyed via a Special Format message when RPCSEC_GSS authentication is in use. It is therefore more likely that a Requester provisions both a Read list and a Reply chunk in the same RPC-over-RDMA Transport header to convey a Special Format Call and provision a receptacle for a Special Format Reply.

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

11.2.2.3. RPC-over-RDMA with RPCSEC_GSS Integrity or Privacy

The RPCSEC_GSS integrity service enables endpoints to detect the modification of RPC messages in flight. The RPCSEC_GSS privacy service prevents all but the intended recipient from viewing the cleartext content of RPC arguments and results. RPCSEC_GSS integrity and privacy services are end-to-end. They protect RPC arguments and results from application to server endpoint, and back.

The RPCSEC_GSS integrity and encryption services operate on whole RPC messages after they have been XDR encoded, and before they have been XDR decoded after receipt. Connection endpoints use intermediate buffers to prevent exposure of encrypted or unverified cleartext data to RPC consumers. After a sender has verified, encrypted, and wrapped a message, the transport layer MAY use RDMA data transfer between these intermediate buffers.

The process of reducing a DDP-eligible data item removes the data item and its XDR padding from an encoded Payload stream. In a non-protected RPC-over-RDMA message, a reduced data item does not include XDR padding. After reduction, the Payload stream contains fewer octets than the whole XDR stream did beforehand. XDR padding octets are often zero bytes, but they don't have to be. Thus, reducing DDP-eligible items affects the result of message integrity verification and encryption.

Therefore, a sender **MUST NOT** reduce a Payload stream when RPCSEC_GSS integrity or encryption services are in use. Effectively, no data item is DDP-eligible in this situation. Senders can use only Simple and Continued Formats without data item chunks, or Special Format. In this mode, an RPC-over-RDMA transport operates in the same manner as a transport that does not support DDP.

11.2.2.4. Protecting RPC-over-RDMA Transport Headers

Like the header fields in an RPC message (e.g., the xid and mtype fields), RPCSEC_GSS does not protect the RPC-over-RDMA Transport stream. XIDs, connection credit limits, and chunk lists (though not the content of the data items they refer to) are exposed to malicious behavior, which can redirect data that is transferred by the RPC-over-RDMA message, result in spurious retransmits, or trigger connection loss.

In particular, if an attacker alters the information contained in the chunk lists of an RPC-over-RDMA Transport header, data contained in those chunks can be redirected to other registered memory regions on Requesters. An attacker might alter the arguments of RDMA Read and RDMA Write operations on the wire to gain a similar effect. If such alterations occur, the use of RPCSEC_GSS integrity or privacy services enables a Requester to detect unexpected material in a received RPC message.

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

11.3. Transport Properties

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

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

11.4. Host Authentication

[cel: This subsection is unfinished.]

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

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

12. IANA Considerations

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

The following netid registry strings are already defined for this purpose:

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

The "rdma" netid is to be used when IPv4 addressing is employed by the underlying transport, and "rdma6" when IPv6 addressing is employed. The netid assignment policy and registry are defined in [RFC5665]. The current document does not alter these netid assignments.

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

13. References

13.1. Normative References

- [RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995, <<https://www.rfc-editor.org/info/rfc1833>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <<https://www.rfc-editor.org/info/rfc4506>>.
- [RFC5042] Pinkerton, J. and E. Deleganes, "Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMA) Security", RFC 5042, DOI 10.17487/RFC5042, October 2007, <<https://www.rfc-editor.org/info/rfc5042>>.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<https://www.rfc-editor.org/info/rfc5056>>.
- [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <<https://www.rfc-editor.org/info/rfc5531>>.
- [RFC5660] Williams, N., "IPsec Channels: Connection Latching", RFC 5660, DOI 10.17487/RFC5660, October 2009, <<https://www.rfc-editor.org/info/rfc5660>>.
- [RFC5665] Eisler, M., "IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats", RFC 5665, DOI 10.17487/RFC5665, January 2010, <<https://www.rfc-editor.org/info/rfc5665>>.
- [RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <<https://www.rfc-editor.org/info/rfc7861>>.
- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC8166] Lever, C., Ed., Simpson, W., and T. Talpey, "Remote Direct Memory Access Transport for Remote Procedure Call Version 1", RFC 8166, DOI 10.17487/RFC8166, June 2017, <<https://www.rfc-editor.org/info/rfc8166>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8267] Lever, C., "Network File System (NFS) Upper-Layer Binding to RPC-over-RDMA Version 1", RFC 8267, DOI 10.17487/RFC8267, October 2017, <<https://www.rfc-editor.org/info/rfc8267>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

13.2. Informative References

- [CBFC] Kung, H.T., Blackwell, T., and A. Chapman, "Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing", Proc. ACM SIGCOMM '94 Symposium on Communications Architectures, Protocols and Applications, pp. 101-114., August 1994.
- [I-D.ietf-nfsv4-rpc-tls] Myklebust, T. and C. Lever, "Towards Remote Procedure Call Encryption By Default", Work in Progress, Internet-Draft, draft-ietf-nfsv4-rpc-tls-11, 23 November 2020, <<https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-rpc-tls-11>>.
- [IBA] InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1", Release 1.3, March 2015. Available from <https://www.infinibandta.org/>
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <<https://www.rfc-editor.org/info/rfc1094>>.

- [RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <<https://www.rfc-editor.org/info/rfc1813>>.
- [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <<https://www.rfc-editor.org/info/rfc3552>>.
- [RFC5040] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia, "A Remote Direct Memory Access Protocol Specification", RFC 5040, DOI 10.17487/RFC5040, October 2007, <<https://www.rfc-editor.org/info/rfc5040>>.
- [RFC5041] Shah, H., Pinkerton, J., Recio, R., and P. Culley, "Direct Data Placement over Reliable Transports", RFC 5041, DOI 10.17487/RFC5041, October 2007, <<https://www.rfc-editor.org/info/rfc5041>>.
- [RFC5044] Culley, P., Elzur, U., Recio, R., Bailey, S., and J. Carrier, "Marker PDU Aligned Framing for TCP Specification", RFC 5044, DOI 10.17487/RFC5044, October 2007, <<https://www.rfc-editor.org/info/rfc5044>>.
- [RFC5532] Talpey, T. and C. Juszczak, "Network File System (NFS) Remote Direct Memory Access (RDMA) Problem Statement", RFC 5532, DOI 10.17487/RFC5532, May 2009, <<https://www.rfc-editor.org/info/rfc5532>>.
- [RFC5662] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, DOI 10.17487/RFC5662, January 2010, <<https://www.rfc-editor.org/info/rfc5662>>.
- [RFC5666] Talpey, T. and B. Callaghan, "Remote Direct Memory Access Transport for Remote Procedure Call", RFC 5666, DOI 10.17487/RFC5666, January 2010, <<https://www.rfc-editor.org/info/rfc5666>>.
- [RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <<https://www.rfc-editor.org/info/rfc7530>>.
- [RFC7862] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <<https://www.rfc-editor.org/info/rfc7862>>.

- [RFC8167] Lever, C., "Bidirectional Remote Procedure Call on RPC-over-RDMA Transports", RFC 8167, DOI 10.17487/RFC8167, June 2017, <<https://www.rfc-editor.org/info/rfc8167>>.
- [RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <<https://www.rfc-editor.org/info/rfc8881>>.

Appendix A. ULB Specifications

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

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

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

A.1. DDP-Eligibility

A ULB designates specific XDR data items as eligible for DDP. As a sender constructs an RPC-over-RDMA message, it can remove DDP-eligible data items from the Payload stream so that the RDMA provider can place them directly in the receiver's memory. An XDR data item should be considered for DDP-eligibility if there is a clear benefit to moving the contents of the item directly from the sender's memory to the receiver's memory.

Criteria for DDP-eligibility include:

- * The XDR data item is frequently sent or received, and its size is often much larger than typical inline thresholds.
- * If the XDR data item is a result, its maximum size must be predictable in advance by the Requester.

- * Transport-level processing of the XDR data item is not needed. For example, the data item is an opaque byte array, which requires no XDR encoding and decoding of its content.
- * The content of the XDR data item is sensitive to address alignment. For example, a data copy operation would be required on the receiver to enable the message to be parsed correctly, or to enable the data item to be accessed.
- * The XDR data item itself does not contain DDP-eligible data items.

In addition to defining the set of data items that are DDP-eligible, a ULB may limit the use of chunks to particular upper-layer procedures. If more than one data item in a procedure is DDP-eligible, the ULB may limit the number of chunks that a Requester can provide for a particular upper-layer procedure.

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

The programming interface by which an upper-layer implementation indicates the DDP-eligibility of a data item to the RPC transport is not described by this specification. The only requirements are that the receiver can re-assemble the transmitted RPC-over-RDMA message into a valid XDR stream and that DDP-eligibility rules specified by the ULB are respected.

There is no provision to express DDP-eligibility within the XDR language. The only definitive specification of DDP-eligibility is a ULB.

In general, a DDP-eligibility violation occurs when:

- * A Requester reduces a non-DDP-eligible argument data item. The Responder reports the violation as described in Section 6.3.1.
- * A Responder reduces a non-DDP-eligible result data item. The Requester terminates the pending RPC transaction and reports an appropriate permanent error to the RPC consumer.
- * A Responder does not reduce a DDP-eligible result data item into an available Write chunk. The Requester terminates the pending RPC transaction and reports an appropriate permanent error to the RPC consumer.

A.2. Maximum Reply Size

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

A.3. Reverse-Direction Operation

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

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

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

A.4. Additional Considerations

There may be other details provided in a ULB.

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

Each ULB needs to be designed to allow correct interoperation without regard to the transport parameters actually in use. Furthermore, implementations of ULPs must be designed to interoperate correctly regardless of the connection parameters in effect on a connection.

A.5. ULP Extensions

An RPC Program and Version tuple may be extensible. For instance, the RPC version number may not reflect a ULP minor versioning scheme, or the ULP may allow the specification of additional features after the publication of the original RPC Program specification. ULBs are provided for interoperable RPC Programs and Versions by extending existing ULBs to reflect the changes made necessary by each addition to the existing XDR.

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

Appendix B. Extending RPC-over-RDMA Version 2

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

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

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

- * They may introduce new OPTIONAL transport header types. Appendix B.2 covers such transport header types.
- * They may define new OPTIONAL transport properties. Appendix B.3 describes such transport properties.

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

- * They may create new error codes, as described in Appendix B.4.

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

B.1. Documentation Requirements

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

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

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

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

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

B.2. Adding New Header Types to RPC-over-RDMA Version 2

New transport header types are defined similar to Sections 6.3.5 through 6.3.10. In particular, what is needed is:

- * A description of the function and use of the new header type.
- * A complete XDR description of the new header type.
- * A description of how receivers report errors, including mechanisms for reporting errors outside the available choices already available in the base protocol or other extensions.

- * An indication of whether a Payload stream must be present, and a description of its contents and how receivers use such Payload streams to reconstruct RPC messages.
- * As appropriate, a statement of whether a Responder may use Remote Invalidation when sending messages that contain the new header type.

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

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

B.3. Adding New Transport properties to the Protocol

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

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

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

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

- * `rpcrdma2_propid` values specified in the document do not conflict with those currently assigned or in use by other pending working group documents defining transport properties.

- * rpcrdma2_propid values specified in the document do not conflict with the range reserved for experimental use, as defined in Section 8.2.

[cel: There is no longer a section 8.2 or an experimental range of propid values. Should we request the creation of an IANA registry for propid values?].

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

B.4. Adding New Error Codes to the Protocol

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

For error codes that do not require the return of additional information, a peer can use the existing RDMA_ERR2 header type to report the new error. The sender sets the new error code as the value of rdma_err with the result that the default switch arm of the rpcrdma2_error (i.e., void) is selected.

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

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

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

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

This section details specific changes in RPC-over-RDMA version 2 that address these constraints directly, in addition to other changes to make implementation easier.

C.1. Changes to the XDR Definition

Several XDR structural changes enable within-version protocol extensibility.

[RFC8166] defines the RPC-over-RDMA version 1 transport header as a single XDR object, with an RPC message potentially following it. In RPC-over-RDMA version 2, there are separate XDR definitions of the transport header prefix (see Section 6.4), which specifies the transport header type to be used, and the transport header itself (defined within one of the subsections of Section 6.3). This construction is similar to an RPC message, which consists of an RPC header (defined in [RFC5531]) followed by a message defined by an Upper-Layer Protocol.

As a new version of the RPC-over-RDMA transport protocol, RPC-over-RDMA version 2 exists within the versioning rules defined in [RFC8166]. In particular, it maintains the first four words of the protocol header, as specified in Section 4.2 of [RFC8166], even though, as explained in Section 6.2.1 of the current document, the XDR definition of those words is structured differently.

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

- * The first word of the header, the `rdma_xid` field, retains the format and function that it had in RPC-over-RDMA version 1. Because RPC-over-RDMA version 2 messages can convey non-RPC messages, a receiver should not use the contents of this field without consideration of the protocol version and header type.
- * The second word of the header, the `rdma_vers` field, retains the format and function that it had in RPC-over-RDMA version 1. To clearly distinguish version 1 and version 2 messages, senders need to fill in the correct version (fixed after version negotiation). Receivers should check that the content of the `rdma_vers` is correct before using the content of any other header field.
- * The third word of the header, the `rdma_credit` field, retains the size and general purpose that it had in RPC-over-RDMA version 1. However, RPC-over-RDMA version 2 divides this field into two 16-bit subfields. See Section 4.2.1 for further details.

- * The fourth word of the header, previously the union discriminator field `rdma_proc`, retains its format and general function even though the set of valid values has changed. Within RPC-over-RDMA version 2, this word is the `rdma_htype` field of the structure `rdma_start`. The value of this field is now an unsigned 32-bit integer rather than an enum type, to facilitate header type extension.

Beyond conforming to the restrictions specified in [RFC8166], RPC-over-RDMA version 2 attempts to limit the scope of the changes made to ensure interoperability. Although it introduces the Call chunk and splits the two version 1 workhorse procedure types `RDMA_MSG` and `RDMA_NOMSG` into several variants, RPC-over-RDMA version 2 otherwise expresses chunks in the same format and utilizes them the same way.

C.2. Transport Properties

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

C.3. Credit Management Changes

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

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

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

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

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

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

C.4. Inline Threshold Changes

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

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

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

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

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

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

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

C.5. Message Continuation Changes

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

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

C.6. Host Authentication Changes

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

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

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

C.7. Support for Remote Invalidation

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

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

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

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

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

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

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

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

C.8. Integration of Reverse-Direction Operation

Because [RFC5666] did not include specification of reverse-direction operation, [RFC8166] does not include it either. Reverse-direction operation in RPC-over-RDMA version 1 is specified by a separate standards track document [RFC8167].

Reverse-direction operation in RPC-over-RDMA version 1 was constrained by the limited ability to extend that version of the protocol. The most awkward issue is that a receiver needs to peek at ingress RPC message payloads to determine whether it is a Call or Reply message. This is necessary because the meaning of several fields in the RPC-over-RDMA transport header is determined by the direction of the RPC message payload:

- * The meaning of the value in the `rdma_xid` field is determined by the direction of the message because the XID spaces in the forward and reverse directions are distinct.
- * The meaning of the value in the `rdma_credit` field is determined by the direction of the message because credits are granted separately for forward and reverse direction operation.
- * The purpose of Write chunks and the meaning of their length fields is determined by the direction of the message because in Call messages, they are provisional, but in Reply messages, they represent returned results.

The current document remedies this awkwardness by integrating reverse-direction operation into RPC-over-RDMA version 2 so that it can make use of all facilities that are available in the forward-direction, including body chunks, remote invalidation, and message continuation. To enable this integration, the direction of the RPC message payload is encoded in each RPC-over-RDMA version 2 transport header.

C.9. Error Reporting Changes

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

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

C.10. Changes in Terminology

The RPC-over-RDMA version 2 specification makes the following changes in terminology. These changes do not result in changes in the behavior or operation of the protocol.

- * The current document explicitly acknowledges the different semantics and purpose of Write chunks appearing in Call messages and those appearing in Reply messages.
- * The current document introduces the term "payload format" to describe the selection of a mechanism for reducing and conveying an RPC message payload. It replaces the terms "short message" and "long message" with the terms "simple format" and "special format" because this selection is not based only on the size of the payload.
- * The current document introduces the terms "data item chunk" and "body chunk" in order to distinguish the purpose and operation of these two categories of chunk.
- * For improved readability, the current document replaces the terms "RDMA segment" and "plain segment" with the term "segment", and the term "RDMA read segment" with the term "Read segment".
- * The current document refers specifically to the RDMA, DDP, and MPA standards track protocols rather than using the nebulous term "iWARP".

Acknowledgments

The authors gratefully acknowledge the work of Brent Callaghan and Tom Talpey on the original RPC-over-RDMA version 1 specification [RFC5666]. The authors also wish to thank Bill Baker, Greg Marsden, and Matt Benjamin for their support of this work.

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

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

Authors' Addresses

Charles Lever (editor)
Oracle Corporation
United States of America

Email: chuck.lever@oracle.com

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

Phone: +1 781 572 8038
Email: davenoveck@gmail.com

Network File System Version 4
Internet-Draft
Intended status: Standards Track
Expires: 6 July 2022

C. Lever, Ed.
Oracle
D. Noveck
NetApp
2 January 2022

RPC-over-RDMA Version 2 Protocol
draft-ietf-nfsv4-rpcrdma-version-two-06

Abstract

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

Note

This note is to be removed before publishing as an RFC.

Discussion of this draft takes place on the NFSv4 working group mailing list (nfsv4@ietf.org), which is archived at <https://mailarchive.ietf.org/arch/browse/nfsv4/>. Working Group information can be found at <https://datatracker.ietf.org/wg/nfsv4/about/>.

The source for this draft is maintained in GitHub. Suggested changes can be submitted as pull requests at <https://github.com/chucklever/i-d-rpcrdma-version-two>. Instructions are on that page as well.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 6 July 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	4
1.1. Design Goals	4
1.2. Motivation for a New Version	5
2. Requirements Language	5
3. Terminology	6
3.1. Remote Procedure Calls	6
3.2. Remote Direct Memory Access	11
4. RPC-over-RDMA Framework	13
4.1. Message Framing	13
4.2. Reliable Message Delivery	14
4.3. Initial Connection State	17
4.4. Using Direct Data Placement	18
4.5. Encoding Chunks	20
4.6. Reverse-Direction Operation	24
4.7. Call-Only Operation	27
5. Transport Properties	27
5.1. Transport Properties Model	27
5.2. Current Transport Properties	28
6. Transport Messages	32
6.1. Transport Header Types	32

6.2.	Headers and Chunks	34
6.3.	Header Types	35
6.4.	Transport Header Prefix	42
6.5.	Remote Invalidation	42
6.6.	Payload Formats	43
7.	Error Handling	49
7.1.	Basic Transport Stream Parsing Errors	50
7.2.	XDR Errors	51
7.3.	Responder RDMA Operational Errors	52
7.4.	Other Operational Errors	54
7.5.	RDMA Transport Errors	55
8.	XDR Protocol Definition	56
8.1.	Code Component License	56
8.2.	Extraction of the XDR Definition	58
8.3.	XDR Definition for RPC-over-RDMA Version 2 Core Structures	59
8.4.	XDR Definition for RPC-over-RDMA Version 2 Base Header Types	61
8.5.	Use of the XDR Description	64
9.	RPC Bind Parameters	65
10.	Implementation Status	66
11.	Security Considerations	66
11.1.	Memory Protection	67
11.2.	RPC Message Security	68
11.3.	Transport Properties	71
11.4.	Host Authentication	72
12.	IANA Considerations	72
13.	References	72
13.1.	Normative References	72
13.2.	Informative References	74
Appendix A.	ULB Specifications	76
A.1.	DDP-Eligibility	76
A.2.	Maximum Reply Size	78
A.3.	Reverse-Direction Operation	78
A.4.	Additional Considerations	78
A.5.	ULP Extensions	79
Appendix B.	Extending RPC-over-RDMA Version 2	79
B.1.	Documentation Requirements	80
B.2.	Adding New Header Types to RPC-over-RDMA Version 2	80
B.3.	Adding New Transport properties to the Protocol	81
B.4.	Adding New Error Codes to the Protocol	82
Appendix C.	Differences from RPC-over-RDMA Version 1	82
C.1.	Changes to the XDR Definition	83
C.2.	Transport Properties	84
C.3.	Credit Management Changes	84
C.4.	Inline Threshold Changes	85
C.5.	Message Continuation Changes	86
C.6.	Host Authentication Changes	86

C.7. Support for Remote Invalidation	87
C.8. Integration of Reverse-Direction Operation	88
C.9. Error Reporting Changes	89
C.10. Changes in Terminology	89
Acknowledgments	90
Authors' Addresses	90

1. Introduction

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

Open Network Computing Remote Procedure Call (ONC RPC, often shortened in NFSv4 documents to RPC) [RFC5531] is a Remote Procedure Call protocol that runs over a variety of transports. Most RPC implementations today use UDP [RFC0768] or TCP [RFC0793]. On UDP, a datagram encapsulates each RPC message. Within a TCP byte stream, a record marking protocol delineates RPC messages.

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

1.1. Design Goals

The general mission of RPC-over-RDMA transports is to leverage network hardware capabilities to reduce host CPU needs related to the transport of RPC messages. In particular, this includes mitigating host interrupt rates and limiting the necessity to copy RPC payload bytes on receivers.

These hardware capabilities benefit both RPC clients and servers. On balance, however, the RPC-over-RDMA protocol design approach has been to bolster clients more than servers, as the client is typically where applications are most hungry for CPU resources.

Additionally, RPC-over-RDMA transports are designed to support RPC applications transparently. However, such transports can also provide mechanisms that enable further optimization of data transfer when RPC applications are structured to exploit direct data placement. In this context, the Network File System (NFS) family of

protocols (as described in [RFC1094], [RFC1813], [RFC7530], [RFC7862], [RFC8881], and subsequent NFSv4 minor versions) are all potential beneficiaries of RPC-over-RDMA.

A complete problem statement appears in [RFC5532].

1.2. Motivation for a New Version

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

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

To address these issues in a way that preserves interoperability with existing RPC-over-RDMA version 1 deployments, the current document presents an updated version of the RPC-over-RDMA transport protocol.

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

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

2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

3.1. Remote Procedure Calls

This section highlights critical elements of the RPC protocol [RFC5531] and the External Data Representation (XDR) [RFC4506] it uses. RPC-over-RDMA version 2 enables the transmission of RPC messages built using XDR and also uses XDR internally to describe its header format.

3.1.1. Upper-Layer Protocols

RPCs are an abstraction used to implement the operations of an Upper-Layer Protocol (ULP). For RPC-over-RDMA, "ULP" refers to an RPC Program and Version tuple, which is a versioned set of procedure calls that comprise a single well-defined API. One example of a ULP is the Network File System Version 4.0 [RFC7530]. In the current document, the term "RPC consumer" refers to an implementation of a ULP running on an RPC client.

3.1.2. RPC Procedures

Like a local procedure call, every RPC procedure has a set of "arguments" and a set of "results". A calling context invokes an RPC procedure, passing arguments to it, and the procedure subsequently returns a set of results. Unlike a local procedure call, an RPC procedure is executed remotely rather than in the local application's execution context.

3.1.3. RPC Transactions

The RPC protocol as described in [RFC5531] is fundamentally a message-passing protocol between one or more clients, where RPC consumers are running, and a server, where a remote execution context is available to process RPC transactions on behalf of these consumers.

ONC RPC transactions consist of two types of messages:

- * A CALL message, or "Call", requests work. An RPC Call message is designated by the value zero (0) in the message's msg_type field.
- * A REPLY message, or "Reply", reports the results of work requested by an RPC Call message. An RPC Reply message is designated by the value one (1) in the message's msg_type field.

Section 9 of [RFC5531] introduces the RPC transaction identifier, or "XID" for short. Each connection endpoint interprets the value of an XID in the context of the message's msg_type field.

- * The sender of a Call message generates an arbitrary XID value for each RPC that is unique among outstanding Calls from that sender.
- * The sender of a Reply message copies the XID of the initiating Call to the Reply containing the results of that procedure.

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

The ratio of Call messages to Reply messages is typically but not always one-to-one.

The most common operational paradigm is when a Requester sends a Call message to a Responder, who then sends a Reply message back to the Requester with the results of that procedure. One Call message elicits a single Reply message in response. A Responder never sends more than one Reply for each received Call message.

A "retransmission" occurs when a Requester sends exactly the same Call message, with the same arguments and XID, more than once. A Requester can retransmit if it believes the network layer or Responder has dropped a Call message, or if the Responder's Reply has been likewise lost. To prevent unnecessary network traffic or the execution of non-idempotent procedures multiple times, Requesters avoid retransmitting needlessly.

In rare cases, an RPC procedure may not require any results or even acknowledgement that the Responder has executed the procedure. In that case, the Requester sends a Call message but no Reply is returned. This document refers to that case as "Call-only".

3.1.4. Message Serialization

RPC messages are always transmitted atomically. RPC peers may interleave messages, but the contents of individual messages cannot be broken up or interleaved without making the messages illegible.

An RPC peer acting as a "Requester" serializes the procedure's arguments and conveys them to a "Responder" endpoint via an RPC Call message. A Call message contains an RPC protocol header with a unique XID, a header describing the requested upper-layer operation, and all arguments.

An RPC peer acting as a "Responder" deserializes these arguments and processes the requested procedure. It then serializes the procedure's results into an RPC Reply message. An RPC Reply message contains an RPC protocol header with the same XID, a header describing the upper-layer reply, and all results.

The Requester deserializes the results and allows the RPC consumer to proceed. At this point, the RPC transaction designated by the XID in the RPC Call message is complete, and the XID is retired.

3.1.5. RPC Transports

The role of an "RPC transport" is to mediate the exchange of RPC messages between Requesters and Responders, bridging the gap between the RPC message abstraction and the native operations of a network transport (e.g., a socket).

When an RPC transport type is connection-oriented, RPC client endpoints initiate transport connections, while RPC server endpoints wait passively to accept incoming connection requests. RPC messages may also be exchanged without a connection association. Because RPC-over-RDMA is a connection-oriented RPC transport, connectionless operation is not discussed further in the current document.

3.1.5.1. Transport Failure Recovery

So that appropriate and timely recovery action can be taken, the transport implementation is responsible for notifying a Requester when an RPC Call or Reply was not able to be conveyed. Recovery can take the form of establishing a new connection, re-sending RPC Calls, or terminating RPC transactions pending on the Requester.

For instance, a connection loss may occur after a Responder has received an RPC Call but before it can send the matching RPC Reply. Once the transport notifies the Requester of the connection loss, the Requester can re-send all pending RPC Calls on a fresh connection.

3.1.5.2. Forward Direction

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

3.1.5.3. Reverse-Direction

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

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

RPC clients and servers are usually optimized to perform and scale well when handling traffic in the forward direction. They might not be prepared to handle operation in the reverse direction. Not until NFS version 4.1 [RFC8881] has there been a strong need to handle reverse-direction operation.

3.1.5.4. Bi-directional Operation

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

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

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

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

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

3.1.6. External Data Representation

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

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

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

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

3.1.6.1. XDR Opaque Data

Sometimes, a data item is to be transferred as-is, without encoding or decoding. We refer to the contents of such a data item as "opaque data". XDR encoding places the content of opaque data items directly into an XDR stream without altering it in any way. ULPs or applications perform any needed data translation in this case. Examples of opaque data items include the content of files or generic byte strings.

3.1.6.2. XDR Roundup

The number of octets in a variable-length data item precedes that item in an XDR stream. If the size of an encoded data item is not a multiple of four octets, the sender appends octets containing zero after the end of the data item. These zero octets shift the next encoded data item in the XDR stream so that it always starts on a four-octet boundary. The addition of extra octets does not change the encoded size of the data item. Receivers do not expose the extra octets to ULPs.

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

3.2. Remote Direct Memory Access

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

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

3.2.1. Direct Data Placement

Typically, RPC implementations copy the contents of RPC messages into a buffer before being sent. An efficient RPC implementation sends bulk data without first copying it into a separate send buffer.

However, socket-based RPC implementations are often unable to receive data directly into its final place in memory. Receivers often need to copy incoming data to finish an RPC operation, if only to adjust data alignment.

Although it may not be efficient, before an RDMA transfer, a sender may copy data into an intermediate buffer. After an RDMA transfer, a receiver may copy that data again to its final destination. In this document, the term "DDP" refers to any optimized data transfer where a receiving host's CPU does not move transferred data to another location after arrival.

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

3.2.2. RDMA Transport Operation

RDMA transports require that RDMA consumers provision resources in advance to achieve good performance during receive operations. An RDMA consumer might provide Receive buffers in advance by posting an RDMA Receive Work Request for every expected RDMA Send from a remote peer. These buffers are provided before the remote peer posts RDMA Send Work Requests. Thus this is often referred to as "pre-posting" buffers.

An RDMA Receive Work Request remains outstanding until the RDMA provider matches it to an inbound Send operation. The resources associated with that Receive must be retained in host memory, or "pinned", until the Receive completes.

Given these tenets of operation, the RPC-over-RDMA version 2 protocol assumes each transport provides the following abstract operations. A more complete discussion of these operations appears in [RFC5040].

3.2.2.1. Memory Registration

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

3.2.2.2. RDMA Send

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

3.2.2.3. RDMA Receive

The RDMA provider supports an RDMA Receive operation to receive data conveyed by incoming RDMA Send operations. To reduce the amount of memory that must remain pinned awaiting incoming Sends, the amount of memory posted per Receive is limited. The RDMA consumer (in this case, the RPC-over-RDMA version 2 protocol) provides flow control to prevent overrunning receiver resources.

3.2.2.4. RDMA Write

The RDMA provider supports an RDMA Write operation to place data directly into a remote memory region. The local host initiates an RDMA Write and the RDMA provider signals completion there. The remote RDMA provider does not signal completion on the remote peer. The local host provides the steering tag, the memory address, and the length of the remote peer's memory region.

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

3.2.2.5. RDMA Read

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

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

4. RPC-over-RDMA Framework

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

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

4.1. Message Framing

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

- * The "Transport stream" contains a header that describes and controls the transfer of the Payload stream in this RPC-over-RDMA message. Every RDMA Send on an RPC-over-RDMA version 2 connection MUST begin with a Transport stream.
- * The "Payload stream" contains part or all of a single RPC message. The sender MAY divide an RPC message at any convenient boundary but MUST send RPC message fragments in XDR stream order and MUST NOT interleave Payload streams from multiple RPC messages.

The RPC-over-RDMA framing mechanism described in this section replaces all other RPC framing mechanisms. Connection peers use RPC-over-RDMA framing even when the underlying RDMA protocol runs on a

transport type with well-defined RPC framing, such as TCP. However, a ULP can negotiate the use of RDMA, dynamically enabling the use of RPC-over-RDMA on a connection established on some other transport type. Because RPC framing delimits an entire RPC request or reply, the resulting shift in framing must occur between distinct RPC messages, and in concert with the underlying transport.

4.2. Reliable Message Delivery

RPC-over-RDMA provides a reliable and in-order data transport service for RPC Calls and Replies.

RPC-over-RDMA transports MUST operate only on a reliable Queue Pair (QP) such as the RDMA RC (Reliable Connected) QP type as defined in Section 9.7.7 of [IBA]. The Marker PDU Aligned (MPA) protocol [RFC5044], when deployed on a reliable transport such as TCP, provides similar functionality. Using a reliable QP type ensures in-transit data integrity and proper recovery from packet loss in the lower layers.

If any pre-posted Receive buffer on the connection is not large enough to contain an incoming message, the receiving RDMA provider cannot deliver that message to the upper-layer consumer. Likewise, if no pre-posted Receive buffer is available to accept an incoming message, the receiving RDMA provider cannot pass that message to the consumer. Exceeding these limits results in a transition to a QP error state, the loss of an in-flight message, and the potential loss of the connection.

Therefore, senders need to respect peer receiver resource limits to ensure that the transport service can deliver every message reliably. Two operational parameters communicate these limits between RPC-over-RDMA peers: credits and inline threshold.

4.2.1. Flow Control

RPC-over-RDMA version 2 employs end-to-end credit-based flow control on each connection to prevent senders from transmitting more messages than a receiver is prepared to accept [CBFC]. Credit-based flow control is relatively simple, providing automated management of receive buffer allocation and robust operation in the face of bursty traffic while enabling effective pipelining. The RPC-over-RDMA version 2 flow control mechanism relies on reliable and in-order message delivery guarantees provided by the underlying RDMA transport service.

An RPC-over-RDMA version 2 credit represents the capability to convey exactly one RPC-over-RDMA version 2 message, regardless of its size, via an RDMA Send/Receive pair. Because an RPC-over-RDMA version 2 connection is full-duplex, each connection peer has its own set of credits. The two receivers manage their credit limits independently, although they communicate these values by piggy-backing them on a message in the opposite direction.

A peer tracks a few critical values for each connection. It uses these values to determine when it is safe to send a message on the connection.

Sent message count: The total number of RDMA Send operations it has posted. The peer MUST set this value to zero (0) when a connection is first established.

Received message count: The total number of RDMA Receive channel operations that have completed. The peer MUST set this value to zero (0) when a connection is first established.

Credit limit: The number of RDMA Receive operations that are currently posted.

Remote credits: The value in the `rdma_start.rdma_credit` field in the most recently received message from its peer. The peer MUST set this value to one (1) before it has received a message on a new connection.

When constructing a new RPC-over-RDMA header, each sender MUST set the header's `rdma_start.rdma_credit` field to its "Sent message count" plus its "Credit limit". The sender MUST NOT post this message if the sender's "Send message counter" is greater than the current "Remote credits" value. To handle counter wrapping, the sender uses appropriate modulo arithmetic to perform this comparison.

Because the `rdma_start.rdma_credit` field is 32 bits wide, a receiver's credit limit MUST be less than $2^{31} - 1$. Given the bandwidth-delay product of the connection, a receiver generally chooses a credit limit that is large enough to maximize throughput while not overwhelming memory resources on the local system.

A receiver MAY adjust its credit limit to match the needs or policies in effect on either peer. For instance, a peer may reduce its credit limit to accommodate the available resources in a Shared Receive Queue. Certain RDMA implementations may impose additional flow-control restrictions, such as limits on RDMA Read operations in progress at the Responder. Accommodation of such checks is considered the responsibility of each RPC-over-RDMA version 2 implementation.

4.2.1.1. Asynchronous Credit Grants

Credit accounting information is usually piggy-backed on payload-bearing messages. However, on occasion, a receiver might need to refresh its credit limit without sending an RPC payload. A receiving peer can send a message using a special header type when the sender's credit limit approaches exhaustion during a stream of unacknowledged messages. See Section 6.3.2 for information about this header type.

Unlike RPC-over-RDMA version 1, the credit limit on an RPC-over-RDMA version 2 connection MAY be zero. In that case, the sender waits until the receiver sends it an asynchronous credit refresh. To prevent a sender from ever having to wait for a credit limit refresh, a good receiver implementation provides a credit refresh before half its credit limit is exceeded.

To prevent transport deadlock, receivers MUST always be in a position to receive one asynchronous credit update message, in addition to payload-bearing messages. A receiver can do this by posting one more RDMA Receive than the credit limit it advertises to its connection peer.

4.2.2. Inline Threshold

An "inline threshold" value is the largest message size (in octets) that can be conveyed in one direction between peer implementations using RDMA Send and Receive channel operations. An inline threshold value is less than or equal to the largest number of octets the sender can post in a single RDMA Send operation. It is also less than or equal to the largest number of octets the receiver can reliably accept via a single RDMA Receive operation.

Each connection has two inline threshold values. There is one for messages flowing from Requester-to-Responder (referred to as the "call inline threshold"), and one for messages flowing from Responder-to-Requester (referred to as the "reply inline threshold").

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

4.3. Initial Connection State

Immediately upon connection establishment, both peers MUST allow only one outstanding RPC-over-RDMA message on the connection at a time until both the transport protocol version is established and both peers have received an initial credit limit. Note that because RPC-over-RDMA versions 1 and 2 each use a different flow control mechanism, the meaning of the value in the `rdma_start.rdma_credit` field depends on the value in the `rdma_start.vers` field.

The second word of each transport header conveys the transport protocol version. Immediately after the client establishes a connection, it sends a single valid RPC-over-RDMA message with the value two (2) in the `rdma_start.rdma_vers` field. Because the server might support only RPC-over-RDMA version 1, this initial message MUST NOT be larger than the version 1 default inline threshold of 1024 octets.

4.3.1. Server Supports RPC-over-RDMA Version 2

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

4.3.2. Server Does Not Support RPC-over-RDMA Version 2

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

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

4.3.3. Client Does Not Support RPC-over-RDMA Version 2

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

4.4. Using Direct Data Placement

RPC-over-RDMA version 2 provides a mechanism for moving part of an RPC message via a data transfer distinct from RDMA Send and Receive. For example, a sender can remove one or more XDR data items from the Payload stream. These items are then conveyed via other mechanisms, such as one or more RDMA Read or Write operations.

4.4.1. Chunks and Segments

A Requester records the location information for each registered memory region associated with an RPC payload in the transport header of an RPC-over-RDMA message. With this information, the Responder uses RDMA Read and Write operations to retrieve arguments contained in the specified region of the Requester's memory or place results in that region.

A "segment" is a transport header data object that contains the precise coordinates of a contiguous registered memory region. Each segment contains the following information:

Handle: A steering Tag (STag) or R_key generated by registering this memory with the RDMA provider.

Length: The length of the segment's memory region, in octets. The length of a segment MAY be aligned to a single octet. An "empty segment" is defined as a segment with the value zero (0) in its length field.

Offset: The offset or beginning memory address of the segment's

memory region.

The meaning of the values contained in these fields is elaborated in [RFC5040].

A "chunk" is simply a set of segments that have a related purpose. A Requester MAY divide a chunk into segments using any convenient boundaries. The length of a chunk is defined as the sum of the lengths of the segments that comprise it.

4.4.2. Reducing a Payload Stream

We refer to a data item that a sender removes from a Payload stream to transmit separately as a "reduced" data item. After a sender has finished removing XDR data items from a Payload stream, we refer to it as a "reduced" Payload stream. A set of segments that describe memory regions containing a single reduced data item is categorized as a "data item chunk."

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

Specifying which data items are DDP-eligible is done in separate standards track documents known as "Upper Layer Bindings". A ULB identifies which XDR data items a peer MAY transfer using DDP. We refer to such data items as "DDP-eligible." Senders MUST NOT reduce any other XDR data items. Detailed requirements for ULB specifications appear in Appendix A of the current document.

4.4.3. Moving Whole RPC Messages using Explicit RDMA

RPC-over-RDMA version 2 also enables the movement of a whole RPC message via data transfer distinct from RDMA Send and Receive. A sender registers the memory containing a Payload stream without regard to data item boundaries or DDP-eligibility. The Payload stream is then conveyed via other mechanisms, such as one or more RDMA Read or Write operations. A set of segments that describe memory regions containing a Payload stream is categorized as a "body chunk".

A sender may first reduce that Payload stream if it contains one or more DDP-eligible data items. The sender moves these data items using data items chunks, and the reduced Payload stream using a body chunk.

4.5. Encoding Chunks

The RPC-over-RDMA version 2 transport protocol does not place a limit on chunk size. However, each ULP may cap the amount of data that can be transferred by a single RPC transaction. For example, NFS implementations typically have settings that restrict the payload size of NFS READ and WRITE operations. The Responder can use such limits to sanity check chunk sizes before using them in RDMA operations.

4.5.1. Read Chunks

A "Read chunk" contains data that its receiver pulls from the sender. Each Read chunk is a set of one or more "Read segments" encoded as a list. A Read segment consists of a Position field followed by a segment, as defined in Section 4.4.1.

Position: The byte offset in the unreduced Payload stream where the receiver reinserts the data item conveyed in the chunk. The sender **MUST** compute the Position value from the beginning of the unreduced Payload stream, which begins at Position zero. All segments in the same Read chunk share the same Position value, even if one or more of the segments have a non-four-byte-aligned length. The value in this field **MUST** be a multiple of four.

When constructing an RPC-over-RDMA message, the sender registers memory regions containing data intended for RDMA Read operations. It advertises the coordinates of these regions in Read chunks added to the transport header of an RPC-over-RDMA message.

The receiver of this message then pulls the chunk's data from the sender using RDMA Read operations. When receiving a Read chunk, the receiver inserts the first Read segment in a Read chunk into the Payload stream at the byte offset indicated by its Position field. The receiver concatenates Read segments whose Position field value matches this offset until there are no more Read segments at that Position value.

4.5.1.1. The Read List

Each RPC-over-RDMA message carries a list of Read segments that make up the set of Read chunks for that message. When no RDMA Read operations are needed to complete the transmission of the message's Payload stream, the message's Read list is empty.

If a Responder receives a Read list whose segment position values do not appear in monotonically increasing order, it MUST discard the message without processing it and respond with an RDMA2_ERROR message with the `rdma_xid` field set to the XID of the malformed message and the `rdma_err` field set to RDMA2_ERR_BAD_XDR.

4.5.1.2. The Call Chunk

The Call chunk is a Read chunk that acts as a body chunk containing an RPC Call message. A Requester can utilize a Call chunk at any time. However, using a Call chunk is less efficient than an RDMA Send.

A Read chunk may act as either a data item chunk or a body chunk. When the chunk's position is zero, it acts as a body chunk. Otherwise, it is a data item chunk containing exactly one XDR data item.

4.5.1.3. Read Completion

A Responder acknowledges that it is finished with the Requester's Read chunk memory regions when it sends the corresponding RPC Reply message. The Requester may then invalidate memory regions belonging to Read chunks associated with the associated RPC Call message.

4.5.2. Write Chunks

Each "Write chunk" consists of a counted array of zero or more segments, as defined in Section 4.4.1. The function of a Write chunk depends on the direction of the containing RPC-over-RDMA message. In a Call message, a Write chunk advertises registered memory regions into which the Responder may push data. In a Reply message, a Write chunk reports how much data has been pushed.

A Requester provisions Write chunks for an RPC transaction long before the Responder has constructed a corresponding Reply message. A Requester typically does not know the actual length of the result data items or Reply to be returned, since the Reply does not yet exist. Thus, a Requester MUST provision Write chunks large enough to accommodate the maximum possible size of each returned data item.

An "empty Write chunk" is a Write chunk with a zero segment count. By definition, the length of an empty Write chunk is zero. An "unused Write chunk" has a non-zero segment count, but all of its segments are empty segments.

4.5.2.1. The Write List

Each RPC-over-RDMA message carries a list of Write chunks. When no DDP-eligible data items are to appear in the Reply to an RPC transaction, the Requester provides an empty Write list in the RPC Call, and the Responder leaves the Write list empty in the matching RPC Reply. When a Write chunk appears in the Write list, it acts only as a data item chunk.

For each Write chunk in the Write list, the Responder pushes one DDP-eligible data item to the Requester. It fills the chunk contiguously and in segment array order until the Responder has written that data item to the Requester in its entirety. The Responder **MUST** copy the segment count and all segments from the Requester-provided Write chunk into the RPC Reply message's transport header. As it does so, the Responder updates each segment length field to reflect the actual amount of data returned in that segment.

The Responder then sends the RPC Reply message via an RDMA Send operation.

4.5.2.2. The Reply Chunk

The Reply chunk is a single Write chunk that acts as a body chunk. that contains an RPC Reply message. When a Requester estimates that the Reply message can exceed the connection's ability to convey that Reply using RDMA Send operations, it should provision a Reply chunk.

4.5.2.3. Write Completion

A Responder acknowledges that it is finished updating the Requester's Write chunk memory regions when it sends the corresponding RPC Reply message. The RDMA provider guarantees that the written data is at rest before the next Receive operation, which typically contains the corresponding RPC Reply, completes. The Requester may then invalidate memory regions belonging to Write chunks associated with the associated RPC Call message.

4.5.2.4. Write Chunk Roundup

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

4.5.3. Reducing Complex XDR Data Types

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

However, a data item chunk MUST contain only one XDR data item, and the chunk MUST occupy a four-byte aligned length in the Payload stream so that subsequent data items remain properly aligned once the reduced data item is removed from the Payload stream.

4.5.3.1. Variable-Length Data Items

When a sender reduces a variable-length XDR data item, the length of the item MUST remain in the Payload stream. The sender MUST omit the item's XDR roundup padding from the Payload stream and the chunk. The chunk's total length MUST be the same as the encoded length of the data item.

4.5.3.2. Counted Arrays

When reducing a data item that is a counted array data type, the count of array elements MUST remain in the Payload stream. The sender MUST move the array elements into the chunk. For example, when encoding an opaque byte array as a chunk, the count of bytes stays in the Payload stream, and the sender places the bytes in the array in the chunk.

Individual array elements appear in a chunk in their entirety. For example, when encoding an array of arrays as a chunk, the count of items in the enclosing array stays in the Payload stream. But each enclosed array, including its item count, is transferred as part of the chunk.

4.5.3.3. Optional-Data

Similar to a counted array, when reducing an optional-data data type, the discriminator field MUST remain in the Payload stream. The sender MUST place the data, when present, in the chunk.

4.5.3.4. XDR Unions

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

4.6. Reverse-Direction Operation

The terminology used in this section is introduced in Section 3.1.5.3.

4.6.1. Sending a Reverse-Direction RPC Call

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

- * The server generates a new XID value (see Section 3.1.3 for full requirements) and places it in the `rdma_xid` field of the transport header and the `xid` field of the RPC Call message. The RPC Call header MUST start with the same XID value that is present in the transport header.
- * The `rdma_vers` field of each reverse-direction Call MUST contain the same value as forward-direction Calls on the same connection.
- * The server fills in the `rdma_credit` field with the credit values for the connection, as described in Section 4.2.1.
- * The server determines the Payload format for the RPC message and fills in the `rdma_htype` field as appropriate (see Sections 6.6 and 4.6.4). Section 4.6.4 also covers the disposition of the chunk lists.

4.6.2. Sending a Reverse-Direction RPC Reply

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

- * The client copies the XID value from the matching RPC Call and places it in the `rdma_xid` field of the transport header and the `xid` field of the RPC Reply message. The RPC Reply header MUST start with the same XID value that is present in the transport header.
- * The `rdma_vers` field of each reverse-direction Call MUST contain the same value as forward-direction Replies on the same connection.
- * The client fills in the `rdma_credit` field with the credit values for the connection, as described in Section 4.2.1.

- * The client determines the Payload format for the RPC message and fills in the `rdma_hype` field as appropriate (see Sections 6.6 and 4.6.4). Section 4.6.4 also covers the disposition of the chunk lists.

4.6.3. When Reverse-Direction Operation is Not Supported

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

If an endpoint is unprepared to receive a reverse-direction message, loss of the RDMA connection might result. Thus a denial of service can occur if an RPC server continues to send reverse-direction messages after a client that is not prepared to receive them reconnects to that server.

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

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

4.6.4. Using Chunks During Reverse-Direction Operation

Reverse-direction operations can use chunks for DDP-eligible data items and Special payload formats the same way chunks are used in forward-direction operation. Connection peers indicate their support for using chunks in the reverse direction as part of the exchange of Transport Properties just after a connection is established (see Section 5.2.5).

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

When there is no Upper-Layer Protocol need for chunks in the reverse direction, implementers MAY choose not to provide support for chunks in the reverse direction, thus avoiding the complexity of implementing support for RDMA Reads and Writes in the reverse direction. When an RPC-over-RDMA transport implementation does not support chunks in the reverse direction, RPC endpoints use only the Simple Payload format without data item chunks or the Continued Payload format without data item chunks to send RPC messages in the reverse direction.

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

4.6.5. Reverse-Direction Retransmission

In rare cases, an RPC server cannot complete an RPC transaction or cannot send a Reply. In these cases, the Requester may send the RPC transaction again using the same RPC XID.

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

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

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

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

4.7. Call-Only Operation

There is no corresponding Reply to a Call-only procedure. Thus there is no opportunity for the Responder to indicate it has completed its use of Read or Call chunks that hold arguments or the whole Call. In addition, Write and Reply chunks are not necessary because there are no results and no Reply message. Therefore, Requesters **MUST NOT** use chunks when sending Call-only RPC procedures.

5. Transport Properties

RPC-over-RDMA version 2 enables connection endpoints to exchange information about implementation properties. Compatible endpoints use this information to optimize data transfer. Initially, only a small set of transport properties are defined. The protocol provides header types to exchange transport properties (see 6.3.3 and 6.3.4).

Both the set of transport properties and the operations used to communicate them may be extended. Within RPC-over-RDMA version 2, such extensions are **OPTIONAL**. A discussion of extending the set of transport properties appears in Appendix B.3.

5.1. Transport Properties Model

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

The following XDR types handle transport properties:

```
<CODE BEGINS>
typedef rpcrdma2_propid uint32;

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

typedef rpcrdma2_propval rpcrdma2_propset<>;

typedef uint32 rpcrdma2_propsubset<>;
<CODE ENDS>
```

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

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

Although the `rdma_data` field is opaque, receivers interpret its contents using the XDR type associated with the property specified by `rdma_which`. When the contents of the `rdma_data` field do not conform to that XDR type, the receiver MUST return the error `RDMA2_ERR_BAD_PROPVAL` using the header type `RDMA2_ERROR`, as described in Section 6.3.1.

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

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

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

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

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

5.2. Current Transport Properties

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

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

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

Table 1

5.2.1. Maximum Send Size

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_SBSIZ = 1;
typedef uint32 rpcrdma2_prop_sbsiz;
<CODE ENDS>
```

5.2.2. Receive Buffer Size

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_RBSIZ = 2;
typedef uint32 rpcrdma2_prop_rbsiz;
<CODE ENDS>
```

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

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

5.2.3. Maximum Segment Size

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_RSSIZ = 3;
typedef uint32 rpcrdma2_prop_rssiz;
<CODE ENDS>
```

5.2.4. Maximum Segment Count

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

```
<CODE BEGINS>
const uint32 RDMA2_PROPID_RCSIZ = 4;
typedef uint32 rpcrdma2_prop_rcsiz;
<CODE ENDS>
```

5.2.5. Reverse-Direction Support

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

```
<CODE BEGINS>
const uint32 RDMA_RVRSDIR_NONE = 0;
const uint32 RDMA_RVRSDIR_SIMPLE = 1;
const uint32 RDMA_RVRSDIR_CONT = 2;
const uint32 RDMA_RVRSDIR_GENL = 3;

const uint32 RDMA2_PROPID_BRS = 5;
typedef uint32 rpcrdma2_prop_brs;
<CODE ENDS>
```

Multiple levels of support are distinguished:

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

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

5.2.6. Host Authentication Message

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

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

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

6. Transport Messages

Each transport message consists of multiple sections.

- * A transport header prefix, as defined in Section 6.4. Among other things, this structure indicates the header type.
- * The transport header proper, as defined by one of the sub-sections below. See Section 6.1 for the mapping between header types and the corresponding header structure.
- * Potentially, all or part of an RPC message payload.

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

6.1. Transport Header Types

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

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

Operation	Code	XDR type	Msg	Section
Report Transport Error	4	rpcrdma2_hdr_error	No	6.3.1
Grant Credits	5	void	No	6.3.2
Specify Properties (Middle)	6	rpcrdma2_hdr_connprop	No	6.3.3
Specify Properties (Final)	7	rpcrdma2_hdr_connprop	No	6.3.4
Convey External RPC Call Message	8	rpcrdma2_hdr_call_external	No	6.3.5
Convey Continued RPC Call Message	9	rpcrdma2_hdr_call_middle	Yes	6.3.6
Convey Inline RPC Call Message	10	rpcrdma2_hdr_call_inline	Yes	6.3.7
Convey External RPC Reply Message	11	rpcrdma2_hdr_reply_external	No	6.3.8
Convey Continued RPC Reply Message	12	rpcrdma2_hdr_reply_middle	Yes	6.3.9
Convey Inline RPC Reply Message	13	rpcrdma2_hdr_reply_inline	Yes	6.3.10

Table 2

RPC-over-RDMA version 2 peers are REQUIRED to support all message header types in Table 2. RPC-over-RDMA version 2 implementations that receive an unrecognized header type MUST respond with an RDMA2_ERROR message with an rdma_err field containing RDMA2_ERR_INVALID_HTYPE and drop the incoming message without processing it further.

6.2. Headers and Chunks

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

6.2.1. Common Transport Header Prefix

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

```
<CODE BEGINS>
struct rpcrdma_common {
    uint32      rdma_xid;
    uint32      rdma_vers;
    uint32      rdma_credit;
    uint32      rdma_htype;
};
<CODE ENDS>
```

RPC-over-RDMA version 2's use of these first four words aligns with that of version 1 as required by Section 4.2 of [RFC8166]. However, there are crucial structural differences in the XDR definition of RPC-over-RDMA version 2: in the way that these words are described by the respective XDR descriptions:

- * The header type is represented as a uint32 rather than as an enum type. An enum would need to be modified to reflect additions to the set of header types made by later extensions.
- * The header type field is part of an XDR structure devoted to representing the transport header prefix, rather than being part of a discriminated union, that includes the body of each transport header type.

- * There is now a prefix structure (see Section 6.4) of which the `rpcrdma_common` structure is the initial segment. This prefix is a newly defined XDR object within the protocol description, which constrains the universal portion of all header types to the four words in `rpcrdma_common`.

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

6.3. Header Types

The header types defined and used in RPC-over-RDMA version 1 are not carried over into RPC-over-RDMA version 2, although there are easy equivalents to the version 1 procedures:

- * The `RDMA2_ERROR` header (defined in Section 6.3.1) has an XDR definition that differs from that in RPC-over-RDMA version 1, and its modifications are all compatible extensions.
- * Senders use `RDMA2_CALL_INLINE` or `RDMA2_REPLY_INLINE` (defined in Sections 6.3.7 and 6.3.10) in place of `RDMA_MSG`. There are minor differences in the on-the-wire format between the version 1 procedure and the version 2 header types.
- * Senders use `RDMA2_CALL_EXTERNAL` or `RDMA2_REPLY_EXTERNAL` (defined in Sections 6.3.5 and 6.3.8) in place of `RDMA_NOMSG`. There are minor differences in the on-the-wire format between the version 1 procedure and the version 2 header types.
- * `RDMA2_CONNPROP_MIDDLE` and `RDMA2_CONNPROP_FINAL` (defined in Sections 6.3.3 and 6.3.4) are new header types devoted to enabling connection peers to exchange information about their transport properties.

6.3.1. `RDMA2_ERROR`: Report Transport Error

`RDMA2_ERROR` reports a transport layer error on a previous transmission.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_ERROR = 4;

struct rpcrdma2_err_vers {
    uint32 rdma_vers_low;
    uint32 rdma_vers_high;
};

struct rpcrdma2_err_write {
    uint32 rdma_chunk_index;
    uint32 rdma_length_needed;
};

union rpcrdma2_hdr_error switch (rpcrdma2_errcode rdma_err) {
    case RDMA2_ERR_VERS:
        rpcrdma2_err_vers rdma_vrange;
    case RDMA2_ERR_READ_CHUNKS:
        uint32 rdma_max_chunks;
    case RDMA2_ERR_WRITE_CHUNKS:
        uint32 rdma_max_chunks;
    case RDMA2_ERR_SEGMENTS:
        uint32 rdma_max_segments;
    case RDMA2_ERR_WRITE_RESOURCE:
        rpcrdma2_err_write rdma_writeres;
    case RDMA2_ERR_REPLY_RESOURCE:
        uint32 rdma_length_needed;
    default:
        void;
};
<CODE ENDS>
```

See Section 7 for details on the use of this header type.

6.3.2. RDMA2_GRANT: Grant Credits

The RDMA2_GRANT header type enables a connection peer to update credit information without conveying a payload.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_GRANT = 5;
<CODE ENDS>
```

This message carries no payload except for a struct `rpcrdma2_hdr_prefix`. The `rdma_xid` field is unused. Senders MUST set the `rdma_xid` field to zero and receivers MUST ignore the value in this field.

6.3.3. RDMA2_CONNPROP_MIDDLE: Exchange Transport Properties

The RDMA2_CONNPROP_MIDDLE header type enables a connection peer to publish the properties of its implementation to its remote peer.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CONNPROP_MIDDLE = 6;

struct rpcrdma2_hdr_connprop {
    rpcrdma2_propset rdma_props;
};
<CODE ENDS>
```

A peer sends an RDMA2_CONNPROP_MIDDLE header type when it has one or more properties to send that do not fit within the default inline threshold for the RPC-over-RDMA version that is in effect.

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

If a peer sends follows an RDMA2_CONNPROP_MIDDLE header type with anything other than another RDMA2_CONNPROP_MIDDLE message or an RDMA2_CONNPROP_FINAL message, the receiver MUST respond with an RDMA2_ERROR header type and set its rdma_err field to RDMA2_ERR_INVALID_CONT and drop the incoming message without processing it further.

6.3.4. RDMA2_CONNPROP_FINAL: Exchange Transport Properties

The RDMA2_CONNPROP_FINAL header type enables a connection peer to publish the properties of its implementation to its remote peer.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CONNPROP_FINAL = 7;

struct rpcrdma2_hdr_connprop {
    rpcrdma2_propset rdma_props;
};
<CODE ENDS>
```

Each peer sends an RDMA2_CONNPROP_FINAL header type as the final CONNPROP-type message after the client has established a connection. The size of this message is limited to the default inline threshold for the RPC-over-RDMA version that is in effect.

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

If a peer sends a CONNPROP-type message on a connection after it has sent an RDMA2_CONNPROP_FINAL message, the receiver MUST respond with an RDMA2_ERROR header type and set its `rdma_err` field to `RDMA2_ERR_INVALID_CONT` and drop the incoming message without processing it further.

6.3.5. RDMA2_CALL_EXTERNAL: Convey External RPC Call Message

RDMA2_CALL_EXTERNAL conveys an RPC Call message payload using explicit RDMA operations. The Responder reads the Payload stream from a memory area specified by the Call chunk. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CALL_EXTERNAL = 8;

struct rpcrdma2_hdr_call_external {
    uint32_t                                rdma_inv_handle;

    struct rpcrdma2_read_list               *rdma_call;
    struct rpcrdma2_read_list               *rdma_reads;
    struct rpcrdma2_write_list              *rdma_provisional_writes;
    struct rpcrdma2_write_chunk             *rdma_provisional_reply;
};
<CODE ENDS>
```

`rdma_inv_handle`: The `rdma_inv_handle` field contains a 32-bit RDMA handle that the Responder may use in a Send With Invalidation operation. See Section 6.5.

`rdma_call`: The `rdma_call` field anchors a list of one or more Read segments that contain the RPC Call's Payload stream.

`rdma_reads`: The `rdma_reads` field anchors a list of zero or more Read segments that contain data item chunks.

`rdma_provisional_writes`: The `rdma_writes` field anchors a list of zero or more provisional Write chunks.

`rdma_provisional_reply`: The `rdma_reply` field is a list containing zero or one provisional Reply chunk.

6.3.6. RDMA2_CALL_MIDDLE: Convey Continued RPC Call Message

RDMA2_CALL_MIDDLE conveys a beginning or middle portion of an RPC Call message immediately following the transport header in the send buffer. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload. The sender sets the `rdma_remaining` field to the number of bytes in the RPC Call message payload that remain to be sent. The `rdma_rpc_first_word` field demarks the first word of the Payload stream.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_CALL_MIDDLE = 9;
```

```
struct rpcrdma2_hdr_call_middle {
    uint32                rdma_remaining;

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

If a peer sends follows an RDMA2_CALL_MIDDLE header type with anything other than an RDMA2_CALL_MIDDLE message or an RDMA2_CALL_INLINE message, the receiver MUST respond with an RDMA2_ERROR header type and set its `rdma_err` field to RDMA2_ERR_INVALID_CONT and drop the incoming message without processing it further.

6.3.7. RDMA2_CALL_INLINE: Convey Inline RPC Call Message

RDMA2_CALL_INLINE conveys the only or final portion of an RPC Call message. The `rdma_rpc_first_word` field demarks the first word of this Payload stream.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_CALL_INLINE = 10;

struct rpcrdma2_hdr_call_inline {
    uint32                rdma_inv_handle;

    struct rpcrdma2_read_list    *rdma_reads;
    struct rpcrdma2_write_list   *rdma_provisional_writes;
    struct rpcrdma2_write_chunk *rdma_provisional_reply;

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

rdma_inv_handle: The `rdma_inv_handle` field contains a 32-bit RDMA handle that the Responder may use in a Send With Invalidation operation. See Section 6.5.

rdma_reads: The `rdma_reads` field anchors a list of zero or more Read segments that contain only data item chunks. A Requester MUST NOT insert Position-zero Read chunks in this list.

rdma_provisional_writes: The `rdma_writes` field anchors a list of zero or more provisional Write chunks.

rdma_provisional_reply: The `rdma_reply` field is a list containing zero or one provisional Reply chunk.

6.3.8. RDMA2_REPLY_EXTERNAL: Convey External RPC Reply Message

`RDMA2_REPLY_EXTERNAL` conveys an RPC Reply message payload using explicit RDMA operations. In particular, it is referred to as a Special Format Reply when the Responder writes the RPC payload into a memory area specified by a Reply chunk. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload.

```
<CODE BEGINS>
const rpcrdma2_proc RDMA2_REPLY_EXTERNAL = 11;

struct rpcrdma2_hdr_reply_external {
    struct rpcrdma2_write_list    *rdma_writes;
    struct rpcrdma2_write_chunk *rdma_reply;
};
<CODE ENDS>
```

`rdma_writes`: The `rdma_writes` field anchors a list of zero or more Write chunks that are either empty or contain reduced data items.

`rdma_reply`: The `rdma_reply` field is a list that MUST contain exactly one Reply chunk.

6.3.9. RDMA2_REPLY_MIDDLE: Convey Continued RPC Reply Message

RDMA2_REPLY_MIDDLE conveys a beginning or middle portion of an RPC Reply message immediately following the transport header in the send buffer. The sender MUST set the `rdma_xid` field to the same value as the `xid` of the RPC Reply message payload. The sender sets the `rdma_remaining` field to the number of bytes in the RPC Call message payload that remain to be sent. The `rdma_rpc_first_word` field demarks the first word of the Payload stream.

<CODE BEGINS>

```
const rpcrdma2_proc RDMA2_REPLY_MIDDLE = 12;
```

```
struct rpcrdma2_hdr_reply_middle {
    uint32                rdma_remaining;

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

<CODE ENDS>

If a peer sends follows an RDMA2_REPLY_MIDDLE header type with anything other than an RDMA2_REPLY_MIDDLE message or an RDMA2_REPLY_INLINE message, the receiver MUST respond with an RDMA2_ERROR header type and set its `rdma_err` field to RDMA2_ERR_INVALID_CONT and drop the incoming message without processing it further.

6.3.10. RDMA2_REPLY_INLINE: Convey RPC Reply Message Inline

RDMA2_REPLY_INLINE conveys the only or final portion of an RPC Reply message immediately following the transport header in the send buffer. If the Reply message payload has been reduced, the `rdma_chunks` object carries the reduced data item chunks.

```

<CODE BEGINS>
const rpcrdma2_proc RDMA2_REPLY_INLINE = 13;

struct rpcrdma2_hdr_reply_inline {
    struct rpcrdma2_write_list *rdma_writes;

    /* The rpc message starts here and continues
     * through the end of the transmission. */
    uint32_t rdma_rpc_first_word;
};
<CODE ENDS>

```

rdma_writes: The rdma_writes field anchors a list of zero or more Write chunks that are either empty or contain reduced data items.

6.4. Transport Header Prefix

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

```

<CODE BEGINS>
struct rpcrdma2_hdr_prefix {
    struct rpcrdma2_common rdma_start;
};
<CODE ENDS>

```

6.5. Remote Invalidation

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

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

A Responder never uses a Send With Invalidate Work Request when sending a control plane header type. This includes the RDMA2_ERROR header type, the RDMA2_GRANT header type, the RDMA2_CONNPROM_MIDDLE header type, and the RDMA2_CONNPROM_FINAL header type.

6.6. Payload Formats

RPC-over-RDMA version 2 provides several ways, known as "payload formats", to convey an RPC-over-RDMA message. A sender chooses the payload format for each message based on several factors:

- * The existence of DDP-eligible data items in the RPC message payload
- * The size of the RPC message payload
- * The direction of the RPC message (i.e., Call or Reply)
- * The available hardware resources
- * The arrangement of source and sink memory buffers

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

6.6.1. Simple Format

All RPC messages conveyed via RPC-over-RDMA version 2 need at least one RDMA Send operation to convey. Thus, the most efficient way to send an RPC message that is smaller than the inline threshold is to append the Payload stream directly to the Transport stream and use an RDMA Send to convey both. When no chunks are present, senders construct Calls and Replies the same way, and no other operations are needed.

6.6.1.1. Simple Format with Data Item Chunks

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

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

Simple Call

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

Simple Reply

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

6.6.1.2. Simple Format Examples

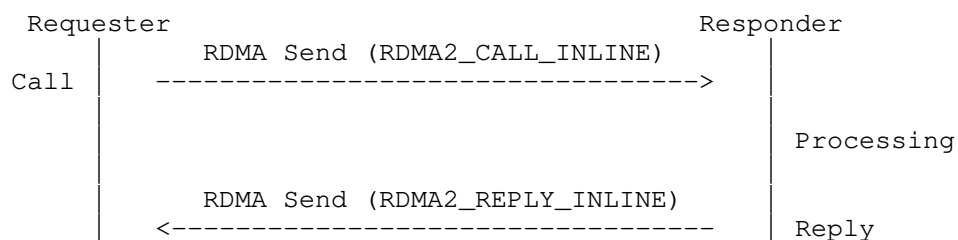


Figure 1: A Simple Call without data item chunks and a Simple Reply without data item chunks

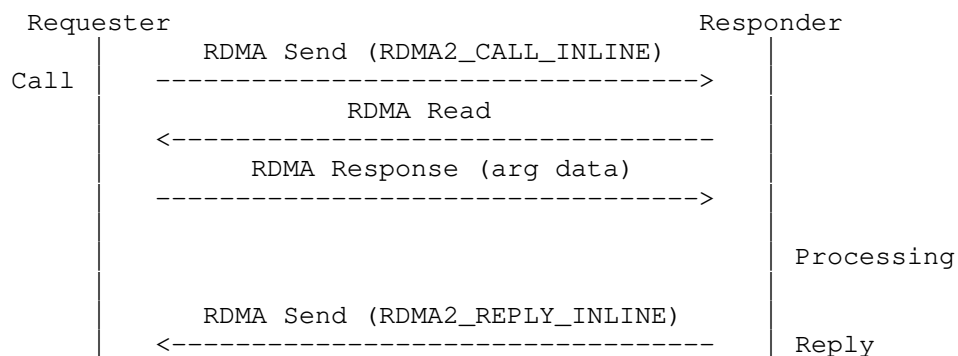


Figure 2: A Simple Call with a Read chunk and a Simple Reply without data item chunks

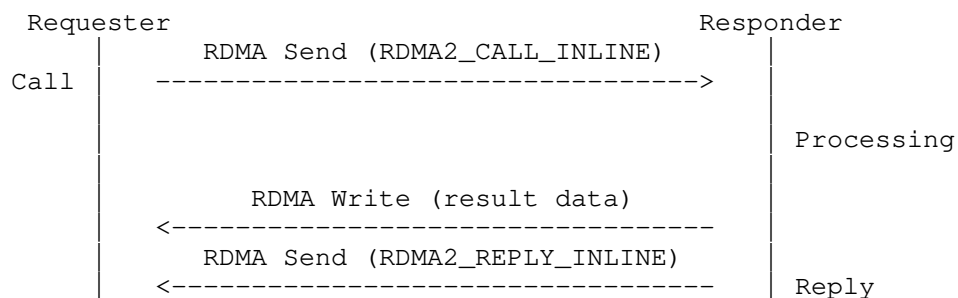


Figure 3: A Simple Call without data item chunks and a Simple Reply with a Write chunk

6.6.2. Continued Format

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

6.6.2.1. Continued Format with Data Item Chunks

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

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

Continued Call

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

Continued Reply

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

6.6.2.2. Continued Format Examples

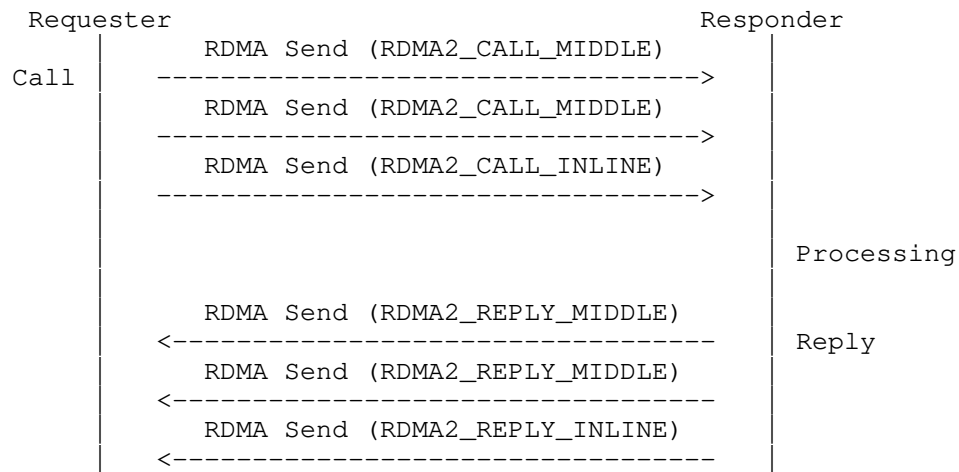


Figure 4: A Continued Call without data item chunks and a Continued Reply without data item chunks

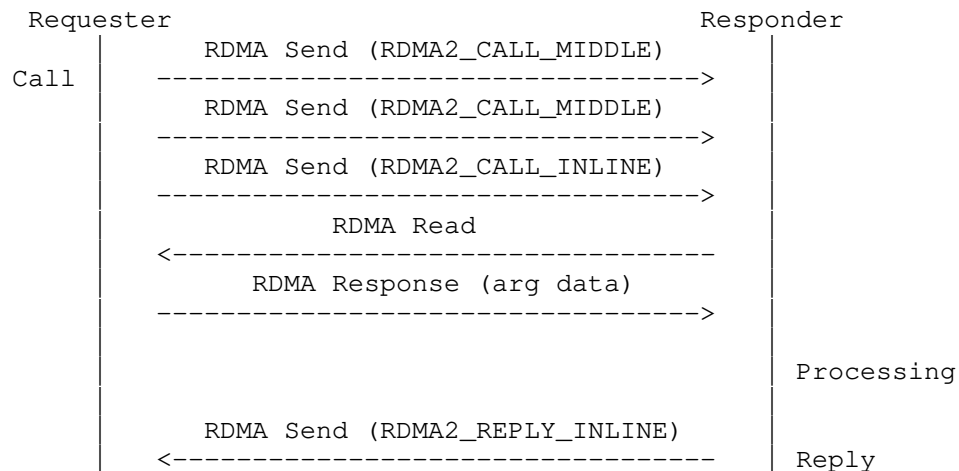


Figure 5: A Continued Call with a Read chunk and a Simple Reply without data item chunks

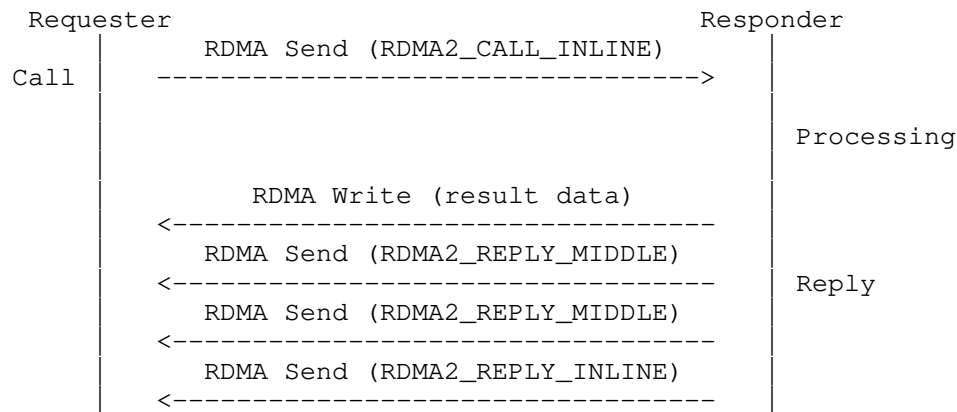


Figure 6: A Simple Call without data item chunks and a Continued Reply with a Write chunk

6.6.3. Special Format

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

To transmit a Special Format message, the sender transmits only the Transport stream with an RDMA Send operation. The sender does not include the Payload stream in the send buffer. Instead, the Requester provides a body chunk that the Responder uses to move the Payload stream.

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

Special Call

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

Special Reply

The Requester provisions a Reply chunk in advance. This body chunk is a Write chunk into which the Responder places the RPC Reply message's Payload stream. The Requester provisions the Reply chunk to accommodate the maximum expected reply size for that upper-layer operation.

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

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

6.6.3.1. Special Format Examples

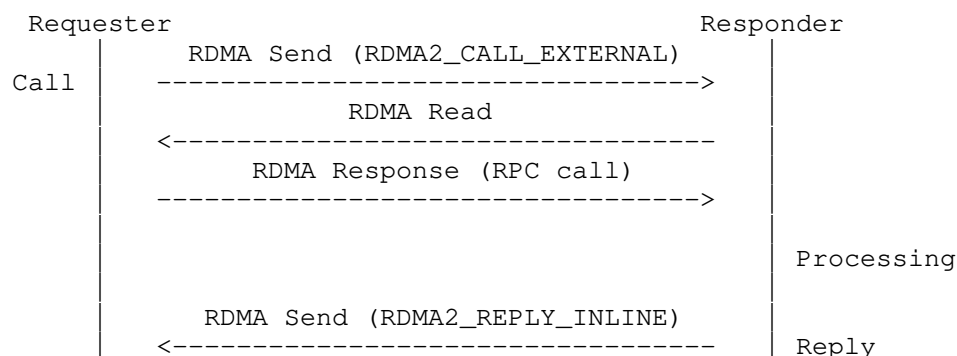


Figure 7: A Special Call and a Simple Reply without data item chunks

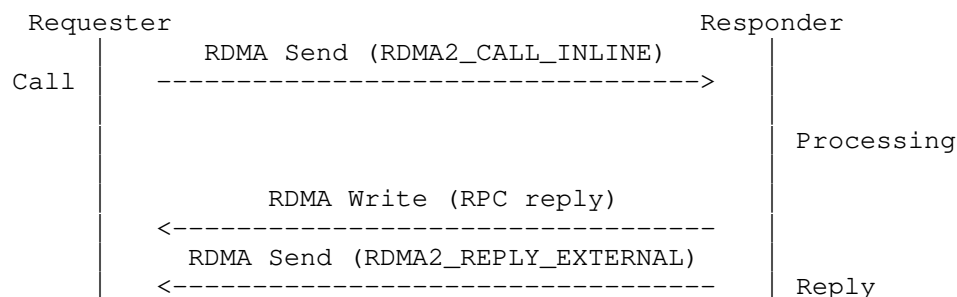


Figure 8: A Simple Call without data item chunks and a Special Reply

6.6.4. Choosing a Reply Payload Format

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

In RPC-over-RDMA version 2, a Requester can provide a Reply chunk for any transaction. The Responder can use the provided Reply chunk or it can decide to use another means to convey the RPC Reply. If the combination of the provided Write chunk list and Reply chunk is not adequate to convey a Reply, the Responder SHOULD use Message Continuation to send that Reply. If even that is not possible, the Responder sends an RDMA2_ERROR message to the Requester, as described in Section 6.3.1:

- * If the Write chunk list cannot accommodate the ULP's DDP-eligible data payload, the Responder sends an RDMA2_ERR_WRITE_RESOURCE error.
- * If the Reply chunk cannot accommodate the parts of the Reply that are not DDP-eligible, the Responder sends an RDMA2_ERR_REPLY_RESOURCE error.

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

7. Error Handling

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

When a request (for instance, an RPC Call or a control plane operation) is made, typically an RPC consumer blocks while waiting for the response. Thus when an incoming message conveys a request and that request cannot be acted upon, the receiver of that request

needs to report the problem to its sender in order to unblock waiters. Likewise, if, after processing a request, a sender is unable to transmit the response on an otherwise healthy connection, the sender needs to report that problem for the same reason.

The RDMA2_ERROR header type is used for this purpose. To form an RDMA2_ERROR type header:

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

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

7.1. Basic Transport Stream Parsing Errors

7.1.1. RDMA2_ERR_VERS

When a Responder detects an RPC-over-RDMA header version that it does not support (the current document defines version 2), it MUST respond with an RDMA2_ERROR message type and set its rdma_err field to RDMA2_ERR_VERS. The Responder then fills in the rpcrdma2_err_vers structure with the RPC-over-RDMA versions it supports. The Responder MUST silently discard the ingress message without passing it to the RPC layer.

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

7.1.2. RDMA2_ERR_VERS_MISMATCH

When a Responder receives a message with a transport protocol version that does not match the protocol version that was used in previous successful exchanges on the same connection, it MUST respond with an RDMA2_ERROR message type and set its rdma_err field to RDMA2_ERR_VERS_MISMATCH. The Responder MUST silently discard the ingress message without passing it to the RPC layer.

A Requester MUST NOT subsequently send a message that uses a protocol version that the Responder has indicated it does not recognize on this connection. The Requester can recover by sending the message again using a corrected protocol version, or it can terminate the RPC transaction associated with the XID in the rdma_xid field with an error.

7.1.3. RDMA2_ERR_INVALID_HTYPE

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

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

7.1.4. RDMA2_ERR_INVALID_CONT

If a Responder detects a problem with an ingress RPC-over-RDMA message that is part of a Message Continuation sequence, the Responder MUST set the rdma_err field to RDMA2_ERR_INVALID_CONT. The Responder MUST silently discard all ingress messages with an rdma_xid field that matches the failing message without reassembling the payload.

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

7.2. XDR Errors

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

- * The value of the `rdma_xid` field does not match the value of the `XID` field in the accompanying RPC message.
- * The receive buffer ends before the end of a data object contained in the Transport stream.

Moreover, when a Responder receives a valid RPC-over-RDMA header but the Responder's ULP implementation cannot parse the RPC arguments in the RPC Call, the Responder returns an RPC Reply with status `GARBAGE_ARGS`, using an `RDMA2_REPLY_INLINE` message type. This type of parsing failure might be due to mismatches between chunk sizes or offsets and the contents of the Payload stream, for example. In this case, the error is permanent, but the Requester has no way to know how much processing the Responder has completed for this RPC transaction.

7.2.1. `RDMA2_ERR_BAD_XDR`

If a Responder recognizes the values in the `rdma_vers` field, but it cannot otherwise parse the ingress Transport stream, it **MUST** set the `rdma_err` field to `RDMA2_ERR_BAD_XDR`. The Responder **MUST** silently discard the ingress message without passing it to the RPC layer.

`RDMA2_ERR_BAD_XDR` indicates a permanent error. Receipt of this error completes the RPC transaction associated with `XID` in the `rdma_xid` field.

7.2.2. `RDMA2_ERR_BAD_PROPVAL`

If a receiver recognizes the value in an ingress `rdma_which` field, but it cannot parse the accompanying `propval`, it **MUST** set the `rdma_err` field to `RDMA2_ERR_BAD_PROPVAL` (see Section 5.1). The receiver **MUST** silently discard the ingress message without applying any of its property settings.

7.3. Responder RDMA Operational Errors

In RPC-over-RDMA version 2, the Responder initiates RDMA Read and Write operations that target the Requester's memory. Problems might arise as the Responder attempts to use Requester-provided resources for RDMA operations. For example:

- * Usually, chunks can be validated only by using their contents to perform data transfers. If chunk contents are invalid (e.g., a memory region is no longer registered or a chunk length exceeds the end of the registered memory region), a Remote Access Error occurs.

- * If a Requester's Receive buffer is too small, the Responder's Send operation completes with a Local Length Error.
- * If the Requester-provided Reply chunk is too small to accommodate a large RPC Reply message, a Remote Access Error occurs. A Responder might detect this problem before attempting to write past the end of the Reply chunk.

RDMA operational errors can be fatal to the connection. To avoid a retransmission loop and repeated connection loss that deadlocks the connection, once the Requester has re-established a connection, the Responder SHOULD send an RDMA2_ERROR response to indicate that no RPC-level reply is possible for that transaction.

7.3.1. RDMA2_ERR_READ_CHUNKS

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

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

7.3.2. RDMA2_ERR_WRITE_CHUNKS

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

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

7.3.3. RDMA2_ERR_SEGMENTS

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

7.3.4. RDMA2_ERR_WRITE_RESOURCE

If a Requester has provided a Write chunk that is not large enough to contain a DDP-eligible result, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_WRITE_RESOURCE`. The Responder MUST set the `rdma_chunk_index` field to point to the first Write chunk in the transport header that is too short, or to zero to indicate that it was not possible to determine which chunk is too small. Indexing starts at one (1), which represents the first Write chunk. The Responder MUST set the `rdma_length_needed` to the number of bytes needed in that chunk to convey the result data item.

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

7.3.5. RDMA2_ERR_REPLY_RESOURCE

If a Responder cannot send an RPC Reply using Message Continuation and the Reply does not fit in the Reply chunk, the Responder MUST set the `rdma_err` field to `RDMA2_ERR_REPLY_RESOURCE`. The Responder MUST set the `rdma_length_needed` to the number of Reply chunk bytes needed to convey the reply. The Requester can reconstruct the Call with more reply resources and resend it. If the Requester does not resend the Call (for instance, if the Responder set the length field to zero), it MUST terminate this RPC transaction with an error.

7.4. Other Operational Errors

While a Requester is constructing an RPC Call message, an unrecoverable problem might occur that prevents the Requester from posting further RDMA Work Requests on behalf of that message. As with other transports, if a Requester is unable to construct and transmit an RPC Call, the associated RPC transaction fails immediately.

After a Requester has received a Reply, if it is unable to invalidate a memory region due to an unrecoverable problem, the Requester MUST close the connection to protect that memory from Responder access before the associated RPC transaction is complete.

While a Responder is constructing an RPC Reply message or error message, an unrecoverable problem might occur that prevents the Responder from posting further RDMA Work Requests on behalf of that message. If a Responder is unable to construct and transmit an RPC Reply or RPC-over-RDMA error message, the Responder MUST close the connection to signal to the Requester that a reply was lost.

7.4.1. RDMA2_ERR_SYSTEM

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

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

7.5. RDMA Transport Errors

The RDMA connection and physical link provide some degree of error detection and retransmission. The Marker PDU Aligned Framing (MPA) protocol (as described in Section 7.1 of [RFC5044]) as well as the InfiniBand link layer [IBA] provide Cyclic Redundancy Check (CRC) protection of RDMA payloads. CRC-class protection is a general attribute of such transports.

Additionally, the RPC layer itself can accept errors from the transport and recover via retransmission. RPC recovery can typically handle complete loss and re-establishment of a transport connection.

The details of reporting and recovery from RDMA link-layer errors are described in specific link-layer APIs and operational specifications and are outside the scope of this protocol specification. See Section 11 for further discussion of RPC-level integrity schemes.

8. XDR Protocol Definition

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

8.1. Code Component License

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

<CODE BEGINS>

```
/// /*
///  * Copyright (c) 2010, 2020 IETF Trust and the persons
///  * identified as authors of the code. All rights reserved.
///  *
///  * The authors of the code are:
///  * B. Callaghan, T. Talpey, C. Lever, and D. Noveck.
///  *
///  * Redistribution and use in source and binary forms, with
///  * or without modification, are permitted provided that the
///  * following conditions are met:
///  *
///  * - Redistributions of source code must retain the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer.
///  *
///  * - Redistributions in binary form must reproduce the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer in the documentation and/or other
///  *   materials provided with the distribution.
///  *
///  * - Neither the name of Internet Society, IETF or IETF
///  *   Trust, nor the names of specific contributors, may be
///  *   used to endorse or promote products derived from this
///  *   software without specific prior written permission.
///  *
///  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
///  * AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
///  * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
///  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
///  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
///  * EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
///  * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
///  * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
///  * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
///  * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
///  * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
///  * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
///  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
///  * IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
///  * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
///  */
///
<CODE ENDS>
```

8.2. Extraction of the XDR Definition

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

```
<CODE BEGINS>
sed -n -e 's:^ */// ::p' -e 's:^ *///$::p'
<CODE ENDS>
```

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

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

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

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

Running the above script results in two files:

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

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

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

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

```
<CODE BEGINS>
/// /*****
///  *      Transport Header Prefixes
///  *****/
///
/// struct rpcrdma_common {
///     uint32      rdma_xid;
///     uint32      rdma_vers;
///     uint32      rdma_credit;
///     uint32      rdma_htype;
/// };
///
/// struct rpcrdma2_hdr_prefix {
///     struct rpcrdma_common      rdma_start;
/// };
///
/// /*****
///  *      Chunks and Chunk Lists
///  *****/
///
/// struct rpcrdma2_segment {
///     uint32 rdma_handle;
///     uint32 rdma_length;
///     uint64 rdma_offset;
/// };
///
```

```

    /// struct rpcrdma2_read_segment {
    ///     uint32          rdma_position;
    ///     struct rpcrdma2_segment rdma_target;
    /// };
    ///
    /// struct rpcrdma2_read_list {
    ///     struct rpcrdma2_read_segment rdma_entry;
    ///     struct rpcrdma2_read_list   *rdma_next;
    /// };
    ///
    /// struct rpcrdma2_write_chunk {
    ///     struct rpcrdma2_segment rdma_target<>;
    /// };
    ///
    /// struct rpcrdma2_write_list {
    ///     struct rpcrdma2_write_chunk rdma_entry;
    ///     struct rpcrdma2_write_list  *rdma_next;
    /// };
    ///
    /// /*****
    ///  * Transport Properties
    ///  *****/
    ///
    /// /*
    ///  * Types for transport properties model
    ///  */
    /// typedef rpcrdma2_propid uint32;
    ///
    /// struct rpcrdma2_propval {
    ///     rpcrdma2_propid rdma_which;
    ///     opaque          rdma_data<>;
    /// };
    ///
    /// typedef rpcrdma2_propval rpcrdma2_propset<>;
    /// typedef uint32 rpcrdma2_propsubset<>;
    ///
    /// /*
    ///  * Transport propid values for basic properties
    ///  */
    /// const RDMA2_PROPID_SBSIZ = 1;
    /// const RDMA2_PROPID_RBSIZ = 2;
    /// const RDMA2_PROPID_RSSIZ = 3;
    /// const RDMA2_PROPID_RCSIZ = 4;
    /// const RDMA2_PROPID_BRS = 5;
    /// const RDMA2_PROPID_HOSTAUTH = 6;
    ///
    /// /*
    ///  * Types specific to particular properties

```

```

/// */
/// typedef uint32 rpcrdma2_prop_sbsiz;
/// typedef uint32 rpcrdma2_prop_rbsiz;
/// typedef uint32 rpcrdma2_prop_rssiz;
/// typedef uint32 rpcrdma2_prop_rcsiz;
/// typedef uint32 rpcrdma2_prop_brs;
/// typedef opaque rpcrdma2_prop_hostauth<>;
///
/// const RDMA2_RVRSDIR_NONE = 0;
/// const RDMA2_RVRSDIR_SIMPLE = 1;
/// const RDMA2_RVRSDIR_CONT = 2;
/// const RDMA2_RVRSDIR_GENL = 3;
///
/// /* FILE ENDS: common.x; */
<CODE ENDS>

```

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

```

<CODE BEGINS>
/// /*****
///  *      Descriptions of RPC-over-RDMA Header Types
///  *****/
///
/// /*
///  * Header Type Codes: Control plane operations.
///  */
/// const RDMA2_ERROR = 4;
/// const RDMA2_GRANT = 5;
/// const RDMA2_CONNPROP_MIDDLE = 6;
/// const RDMA2_CONNPROP_FINAL = 7;
///
/// /*
///  * Header Type Codes: Call messages.
///  */
/// const RDMA2_CALL_EXTERNAL = 8;
/// const RDMA2_CALL_MIDDLE = 9;
/// const RDMA2_CALL_INLINE = 10;
///
/// /*
///  * Header Type Codes: Reply messages.
///  */
/// const RDMA2_REPLY_EXTERNAL = 11;
/// const RDMA2_REPLY_MIDDLE = 12;
/// const RDMA2_REPLY_INLINE = 13;
///
/// /*
///  * Header Type to Report Errors.
///  */

```

```
/// const RDMA2_ERR_VERS = 1;
/// const RDMA2_ERR_BAD_XDR = 2;
/// const RDMA2_ERR_BAD_PROPVAL = 3;
/// const RDMA2_ERR_INVALID_HTYPE = 4;
/// const RDMA2_ERR_INVALID_CONT = 5;
/// const RDMA2_ERR_READ_CHUNKS = 6;
/// const RDMA2_ERR_WRITE_CHUNKS = 7;
/// const RDMA2_ERR_SEGMENTS = 8;
/// const RDMA2_ERR_WRITE_RESOURCE = 9;
/// const RDMA2_ERR_REPLY_RESOURCE = 10;
/// const RDMA2_ERR_VERS_MISMATCH = 11;
/// const RDMA2_ERR_SYSTEM = 100;
///
/// struct rpcrdma2_err_vers {
///     uint32 rdma_vers_low;
///     uint32 rdma_vers_high;
/// };
///
/// struct rpcrdma2_err_write {
///     uint32 rdma_chunk_index;
///     uint32 rdma_length_needed;
/// };
///
/// union rpcrdma2_hdr_error switch (rpcrdma2_errcode rdma_err) {
///     case RDMA2_ERR_VERS:
///         rpcrdma2_err_vers rdma_vrange;
///     case RDMA2_ERR_READ_CHUNKS:
///         uint32 rdma_max_chunks;
///     case RDMA2_ERR_WRITE_CHUNKS:
///         uint32 rdma_max_chunks;
///     case RDMA2_ERR_SEGMENTS:
///         uint32 rdma_max_segments;
///     case RDMA2_ERR_WRITE_RESOURCE:
///         rpcrdma2_err_write rdma_writeres;
///     case RDMA2_ERR_REPLY_RESOURCE:
///         uint32 rdma_length_needed;
///     default:
///         void;
/// };
///
/// /*
///  * Header Type to Exchange Transport Properties.
///  */
/// struct rpcrdma2_hdr_connprop {
///     rpcrdma2_propset rdma_props;
/// };
///
/// /*
```

```
/// * Header Types to Convey RPC Messages.
/// */
/// struct rpcrdma2_hdr_call_external {
///     uint32                rdma_inv_handle;
///
///     struct rpcrdma2_read_list  *rdma_call;
///     struct rpcrdma2_read_list  *rdma_reads;
///     struct rpcrdma2_write_list *rdma_provisional_writes;
///     struct rpcrdma2_write_chunk *rdma_provisional_reply;
/// };
///
/// struct rpcrdma2_hdr_call_middle {
///     uint32                rdma_remaining;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32                rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_hdr_call_inline {
///     uint32                rdma_inv_handle;
///
///     struct rpcrdma2_read_list  *rdma_reads;
///     struct rpcrdma2_write_list *rdma_provisional_writes;
///     struct rpcrdma2_write_chunk *rdma_provisional_reply;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32                rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_hdr_reply_external {
///     struct rpcrdma2_write_list *rdma_writes;
///     struct rpcrdma2_write_chunk *rdma_reply;
/// };
///
/// struct rpcrdma2_hdr_reply_middle {
///     uint32                rdma_remaining;
///
///     /* The rpc message starts here and continues
///      * through the end of the transmission. */
///     uint32                rdma_rpc_first_word;
/// };
///
/// struct rpcrdma2_hdr_reply_inline {
///     struct rpcrdma2_write_list *rdma_writes;
///
///     /* The rpc message starts here and continues
```

```
///          * through the end of the transmission. */
///          uint32                      rdma_rpc_first_word;
/// };
///
/// /* FILE ENDS: baseops.x; */
<CODE ENDS>
```

8.5. Use of the XDR Description

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

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

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

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

9. RPC Bind Parameters

Before establishing a new connection, an RPC client obtains a transport address for the RPC server. The means used to obtain this address and to open an RDMA connection is dependent on the type of RDMA transport and is the responsibility of each RPC protocol binding and its local implementation.

RPC services typically register with a portmap or rpcbind service [RFC1833], which associates an RPC Program number with a service address. This policy is no different with RDMA transports. However, a distinct service address (port number) is sometimes required for operation on RPC-over-RDMA.

When mapped atop MPA [RFC5044], which uses IP port addressing due to its layering on TCP or SCTP, port mapping is trivial and consists merely of issuing the port in the connection process. The NFS/RDMA protocol service address has been assigned port 20049 by IANA for this deployment scenario [RFC8267].

When mapped atop InfiniBand [IBA], which uses a service endpoint naming scheme based on a Group Identifier (GID), a translation MUST be employed. One such translation is described in Annexes A3 (Application Specific Identifiers), A4 (Sockets Direct Protocol (SDP)), and A11 (RDMA IP CM Service) of [IBA], which is appropriate for translating IP port addressing to the InfiniBand network. Therefore, in this case, IP port addressing may be readily employed by the upper layer.

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

- * One possibility is to have the server register its mapped IP port with the rpcbind service under the netid (or netids) defined in [RFC8166]. An RPC-over-RDMA-aware RPC client can then resolve its desired service to a mappable port and proceed to connect. This method is the most flexible and compatible approach for those upper layers that are defined to use the rpcbind service.

- * A second possibility is to have the RPC server's portmapper register itself on the RDMA interconnect at a "well-known" service address (on UDP or TCP, this corresponds to port 111). An RPC client can connect to this service address and use the portmap protocol to obtain a service address in response to a program number (e.g., a TCP port number or an InfiniBand GID).
- * Alternately, an RPC client can connect to the mapped well-known port for the service itself, if it is appropriately defined. By convention, the NFS/RDMA service, when operating atop an InfiniBand fabric, uses the same 20049 assignment as for MPA.

Historically, different RPC protocols have taken different approaches to their port assignments. The current document leaves the specific method for each RPC-over-RDMA-enabled ULB.

[RFC8166] defines two new netid values to be used for registration of upper layers atop MPA and (when a suitable port translation service is available) InfiniBand. Additional RDMA-capable networks MAY define their own netids, or if they provide a port translation, they MAY share the one defined in [RFC8166].

10. Implementation Status

This section is to be removed before publishing as an RFC.

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

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

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

11. Security Considerations

11.1. Memory Protection

A primary consideration is the protection of the integrity and confidentiality of host memory by an RPC-over-RDMA transport. The use of an RPC-over-RDMA transport protocol MUST NOT introduce vulnerabilities to system memory contents nor memory owned by user processes. Any RDMA provider used for RPC transport MUST conform to the requirements of [RFC5042] to satisfy these protections.

11.1.1. Protection Domains

The use of a Protection Domain to limit the exposure of memory regions to a single connection is critical. Any attempt by an endpoint not participating in that connection to reuse memory handles needs to result in immediate failure of that connection. Because ULP security mechanisms rely on this aspect of Reliable Connected behavior, implementations SHOULD cryptographically authenticate connection endpoints.

11.1.2. Handle (STag) Predictability

Implementations should use unpredictable memory handles for any operation requiring exposed memory regions. Exposing a continuously registered memory region allows a remote host to read or write to that region even when an RPC involving that memory is not underway. Therefore, implementations should avoid the use of persistently registered memory.

11.1.3. Memory Protection

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

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

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

11.1.4. Denial of Service

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

A Responder is not obliged to pull unreasonably large Read chunks. A Responder can use an RDMA2_ERROR response to terminate RPCs with unreadable Read chunks. If a Responder transmits more data than a Requester is prepared to receive in a Write or Reply chunk, the RDMA provider typically terminates the connection. For further discussion, see Section 6.3.1. Such repeated connection termination can deny service to other users sharing the connection from the errant Requester.

An RPC-over-RDMA transport implementation is not responsible for throttling the RPC request rate, other than to keep the number of concurrent RPC transactions within the per connection credit limits (see Section 4.2.1). A sender can trigger a self-denial of service by exceeding the credit limit repeatedly.

When an RPC transaction terminates due to a signal or premature exit of an application process, a Requester should invalidate the RPC's Write and Reply chunks. Invalidation prevents the subsequent arrival of the Responder's Reply from altering the memory regions associated with those chunks after the Requester has released that memory.

On the Requester, a malfunctioning application or a malicious user can create a situation where RPCs initiate and abort continuously, resulting in Responder replies that terminate the underlying RPC-over-RDMA connection repeatedly. Such situations can deny service to other users sharing the connection from that Requester.

11.2. RPC Message Security

ONC RPC provides cryptographic security via the RPCSEC_GSS framework [RFC7861]. RPCSEC_GSS implements message authentication (rpc_gss_svc_none), per-message integrity checking (rpc_gss_svc_integrity), and per-message confidentiality (rpc_gss_svc_privacy) in a layer above the RPC-over-RDMA transport. The integrity and privacy services require significant computation and movement of data on each endpoint host. Some performance benefits enabled by RDMA transports can be lost.

11.2.1. RPC-over-RDMA Protection at Other Layers

For any RPC transport, utilizing RPCSEC_GSS integrity or privacy services has performance implications. Protection below the RPC implementation is often a better choice in performance-sensitive deployments, especially if it, too, can be offloaded. Certain implementations of IPsec can be co-located in RDMA hardware, for example, without change to RDMA consumers and with little loss of data movement efficiency. Such arrangements can also provide a higher degree of privacy by hiding endpoint identity or altering the frequency at which messages are exchanged, at a performance cost.

Implementations MAY negotiate the use of protection in another layer through the use of an RPCSEC_GSS security flavor defined in [RFC7861] in conjunction with the Channel Binding mechanism [RFC5056] and IPsec Channel Connection Latching [RFC5660].

11.2.2. RPCSEC_GSS on RPC-over-RDMA Transports

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

RPCSEC_GSS extends the ONC RPC protocol without changing the format of RPC messages. By observing the conventions described in this section, an RPC-over-RDMA transport can convey RPCSEC_GSS-protected RPC messages interoperably.

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

11.2.2.1. RPCSEC_GSS Context Negotiation

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

11.2.2.2. RPC-over-RDMA with RPCSEC_GSS Authentication

The RPCSEC_GSS authentication service has no impact on the DDP-eligibility of data items in a ULP.

However, RPCSEC_GSS authentication material appearing in an RPC message header can be larger than, say, an AUTH_SYS authenticator. In particular, when an RPCSEC_GSS pseudoflavor is in use, a Requester needs to accommodate a larger RPC credential when marshaling RPC Calls and needs to provide for a maximum size RPCSEC_GSS verifier when allocating reply buffers and Reply chunks.

RPC messages, and thus Payload streams, are larger on average as a result. ULP operations that fit in a Simple Format message when a simpler form of authentication is in use might need to be reduced or conveyed via a Special Format message when RPCSEC_GSS authentication is in use. It is therefore more likely that a Requester provisions both a Read list and a Reply chunk in the same RPC-over-RDMA Transport header to convey a Special Format Call and provision a receptacle for a Special Format Reply.

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

11.2.2.3. RPC-over-RDMA with RPCSEC_GSS Integrity or Privacy

The RPCSEC_GSS integrity service enables endpoints to detect the modification of RPC messages in flight. The RPCSEC_GSS privacy service prevents all but the intended recipient from viewing the cleartext content of RPC arguments and results. RPCSEC_GSS integrity and privacy services are end-to-end. They protect RPC arguments and results from application to server endpoint, and back.

The RPCSEC_GSS integrity and encryption services operate on whole RPC messages after they have been XDR encoded, and before they have been XDR decoded after receipt. Connection endpoints use intermediate buffers to prevent exposure of encrypted or unverified cleartext data to RPC consumers. After a sender has verified, encrypted, and wrapped a message, the transport layer MAY use RDMA data transfer between these intermediate buffers.

The process of reducing a DDP-eligible data item removes the data item and its XDR padding from an encoded Payload stream. In a non-protected RPC-over-RDMA message, a reduced data item does not include XDR padding. After reduction, the Payload stream contains fewer octets than the whole XDR stream did beforehand. XDR padding octets are often zero bytes, but they don't have to be. Thus, reducing DDP-eligible items affects the result of message integrity verification and encryption.

Therefore, a sender **MUST NOT** reduce a Payload stream when RPCSEC_GSS integrity or encryption services are in use. Effectively, no data item is DDP-eligible in this situation. Senders can use only Simple and Continued Formats without data item chunks, or Special Format. In this mode, an RPC-over-RDMA transport operates in the same manner as a transport that does not support DDP.

11.2.2.4. Protecting RPC-over-RDMA Transport Headers

Like the header fields in an RPC message (e.g., the xid and mtype fields), RPCSEC_GSS does not protect the RPC-over-RDMA Transport stream. XIDs, connection credit limits, and chunk lists (though not the content of the data items they refer to) are exposed to malicious behavior, which can redirect data that is transferred by the RPC-over-RDMA message, result in spurious retransmits, or trigger connection loss.

In particular, if an attacker alters the information contained in the chunk lists of an RPC-over-RDMA Transport header, data contained in those chunks can be redirected to other registered memory regions on Requesters. An attacker might alter the arguments of RDMA Read and RDMA Write operations on the wire to gain a similar effect. If such alterations occur, the use of RPCSEC_GSS integrity or privacy services enables a Requester to detect unexpected material in a received RPC message.

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

11.3. Transport Properties

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

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

11.4. Host Authentication

[cel: This subsection is unfinished.]

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

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

12. IANA Considerations

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

The following netid registry strings are already defined for this purpose:

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

The "rdma" netid is to be used when IPv4 addressing is employed by the underlying transport, and "rdma6" when IPv6 addressing is employed. The netid assignment policy and registry are defined in [RFC5665]. The current document does not alter these netid assignments.

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

13. References

13.1. Normative References

- [RFC1833] Srinivasan, R., "Binding Protocols for ONC RPC Version 2", RFC 1833, DOI 10.17487/RFC1833, August 1995, <<https://www.rfc-editor.org/info/rfc1833>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <<https://www.rfc-editor.org/info/rfc4506>>.
- [RFC5042] Pinkerton, J. and E. Deleganes, "Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMA) Security", RFC 5042, DOI 10.17487/RFC5042, October 2007, <<https://www.rfc-editor.org/info/rfc5042>>.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<https://www.rfc-editor.org/info/rfc5056>>.
- [RFC5531] Thurlow, R., "RPC: Remote Procedure Call Protocol Specification Version 2", RFC 5531, DOI 10.17487/RFC5531, May 2009, <<https://www.rfc-editor.org/info/rfc5531>>.
- [RFC5660] Williams, N., "IPsec Channels: Connection Latching", RFC 5660, DOI 10.17487/RFC5660, October 2009, <<https://www.rfc-editor.org/info/rfc5660>>.
- [RFC5665] Eisler, M., "IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats", RFC 5665, DOI 10.17487/RFC5665, January 2010, <<https://www.rfc-editor.org/info/rfc5665>>.
- [RFC7861] Adamson, A. and N. Williams, "Remote Procedure Call (RPC) Security Version 3", RFC 7861, DOI 10.17487/RFC7861, November 2016, <<https://www.rfc-editor.org/info/rfc7861>>.
- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC8166] Lever, C., Ed., Simpson, W., and T. Talpey, "Remote Direct Memory Access Transport for Remote Procedure Call Version 1", RFC 8166, DOI 10.17487/RFC8166, June 2017, <<https://www.rfc-editor.org/info/rfc8166>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8267] Lever, C., "Network File System (NFS) Upper-Layer Binding to RPC-over-RDMA Version 1", RFC 8267, DOI 10.17487/RFC8267, October 2017, <<https://www.rfc-editor.org/info/rfc8267>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

13.2. Informative References

- [CBFC] Kung, H.T., Blackwell, T., and A. Chapman, "Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing", Proc. ACM SIGCOMM '94 Symposium on Communications Architectures, Protocols and Applications, pp. 101-114., August 1994.
- [I-D.ietf-nfsv4-rpc-tls] Myklebust, T. and C. Lever, "Towards Remote Procedure Call Encryption By Default", Work in Progress, Internet-Draft, draft-ietf-nfsv4-rpc-tls-11, 23 November 2020, <<https://datatracker.ietf.org/doc/html/draft-ietf-nfsv4-rpc-tls-11>>.
- [IBA] InfiniBand Trade Association, "InfiniBand Architecture Specification Volume 1", Release 1.3, March 2015. Available from <https://www.infinibandta.org/>
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC1094] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, DOI 10.17487/RFC1094, March 1989, <<https://www.rfc-editor.org/info/rfc1094>>.

- [RFC1813] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, DOI 10.17487/RFC1813, June 1995, <<https://www.rfc-editor.org/info/rfc1813>>.
- [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <<https://www.rfc-editor.org/info/rfc3552>>.
- [RFC5040] Recio, R., Metzler, B., Culley, P., Hilland, J., and D. Garcia, "A Remote Direct Memory Access Protocol Specification", RFC 5040, DOI 10.17487/RFC5040, October 2007, <<https://www.rfc-editor.org/info/rfc5040>>.
- [RFC5041] Shah, H., Pinkerton, J., Recio, R., and P. Culley, "Direct Data Placement over Reliable Transports", RFC 5041, DOI 10.17487/RFC5041, October 2007, <<https://www.rfc-editor.org/info/rfc5041>>.
- [RFC5044] Culley, P., Elzur, U., Recio, R., Bailey, S., and J. Carrier, "Marker PDU Aligned Framing for TCP Specification", RFC 5044, DOI 10.17487/RFC5044, October 2007, <<https://www.rfc-editor.org/info/rfc5044>>.
- [RFC5532] Talpey, T. and C. Juszczak, "Network File System (NFS) Remote Direct Memory Access (RDMA) Problem Statement", RFC 5532, DOI 10.17487/RFC5532, May 2009, <<https://www.rfc-editor.org/info/rfc5532>>.
- [RFC5662] Shepler, S., Ed., Eisler, M., Ed., and D. Noveck, Ed., "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, DOI 10.17487/RFC5662, January 2010, <<https://www.rfc-editor.org/info/rfc5662>>.
- [RFC5666] Talpey, T. and B. Callaghan, "Remote Direct Memory Access Transport for Remote Procedure Call", RFC 5666, DOI 10.17487/RFC5666, January 2010, <<https://www.rfc-editor.org/info/rfc5666>>.
- [RFC7530] Haynes, T., Ed. and D. Noveck, Ed., "Network File System (NFS) Version 4 Protocol", RFC 7530, DOI 10.17487/RFC7530, March 2015, <<https://www.rfc-editor.org/info/rfc7530>>.
- [RFC7862] Haynes, T., "Network File System (NFS) Version 4 Minor Version 2 Protocol", RFC 7862, DOI 10.17487/RFC7862, November 2016, <<https://www.rfc-editor.org/info/rfc7862>>.

- [RFC8167] Lever, C., "Bidirectional Remote Procedure Call on RPC-over-RDMA Transports", RFC 8167, DOI 10.17487/RFC8167, June 2017, <<https://www.rfc-editor.org/info/rfc8167>>.
- [RFC8881] Noveck, D., Ed. and C. Lever, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 8881, DOI 10.17487/RFC8881, August 2020, <<https://www.rfc-editor.org/info/rfc8881>>.

Appendix A. ULB Specifications

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

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

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

A.1. DDP-Eligibility

A ULB designates specific XDR data items as eligible for DDP. As a sender constructs an RPC-over-RDMA message, it can remove DDP-eligible data items from the Payload stream so that the RDMA provider can place them directly in the receiver's memory. An XDR data item should be considered for DDP-eligibility if there is a clear benefit to moving the contents of the item directly from the sender's memory to the receiver's memory.

Criteria for DDP-eligibility include:

- * The XDR data item is frequently sent or received, and its size is often much larger than typical inline thresholds.
- * If the XDR data item is a result, its maximum size must be predictable in advance by the Requester.

- * Transport-level processing of the XDR data item is not needed. For example, the data item is an opaque byte array, which requires no XDR encoding and decoding of its content.
- * The content of the XDR data item is sensitive to address alignment. For example, a data copy operation would be required on the receiver to enable the message to be parsed correctly, or to enable the data item to be accessed.
- * The XDR data item itself does not contain DDP-eligible data items.

In addition to defining the set of data items that are DDP-eligible, a ULB may limit the use of chunks to particular upper-layer procedures. If more than one data item in a procedure is DDP-eligible, the ULB may limit the number of chunks that a Requester can provide for a particular upper-layer procedure.

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

The programming interface by which an upper-layer implementation indicates the DDP-eligibility of a data item to the RPC transport is not described by this specification. The only requirements are that the receiver can re-assemble the transmitted RPC-over-RDMA message into a valid XDR stream and that DDP-eligibility rules specified by the ULB are respected.

There is no provision to express DDP-eligibility within the XDR language. The only definitive specification of DDP-eligibility is a ULB.

In general, a DDP-eligibility violation occurs when:

- * A Requester reduces a non-DDP-eligible argument data item. The Responder reports the violation as described in Section 6.3.1.
- * A Responder reduces a non-DDP-eligible result data item. The Requester terminates the pending RPC transaction and reports an appropriate permanent error to the RPC consumer.
- * A Responder does not reduce a DDP-eligible result data item into an available Write chunk. The Requester terminates the pending RPC transaction and reports an appropriate permanent error to the RPC consumer.

A.2. Maximum Reply Size

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

A.3. Reverse-Direction Operation

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

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

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

A.4. Additional Considerations

There may be other details provided in a ULB.

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

Each ULB needs to be designed to allow correct interoperation without regard to the transport parameters actually in use. Furthermore, implementations of ULPs must be designed to interoperate correctly regardless of the connection parameters in effect on a connection.

A.5. ULP Extensions

An RPC Program and Version tuple may be extensible. For instance, the RPC version number may not reflect a ULP minor versioning scheme, or the ULP may allow the specification of additional features after the publication of the original RPC Program specification. ULBs are provided for interoperable RPC Programs and Versions by extending existing ULBs to reflect the changes made necessary by each addition to the existing XDR.

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

Appendix B. Extending RPC-over-RDMA Version 2

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

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

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

- * They may introduce new OPTIONAL transport header types. Appendix B.2 covers such transport header types.
- * They may define new OPTIONAL transport properties. Appendix B.3 describes such transport properties.

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

- * They may create new error codes, as described in Appendix B.4.

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

B.1. Documentation Requirements

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

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

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

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

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

B.2. Adding New Header Types to RPC-over-RDMA Version 2

New transport header types are defined similar to Sections 6.3.5 through 6.3.10. In particular, what is needed is:

- * A description of the function and use of the new header type.
- * A complete XDR description of the new header type.
- * A description of how receivers report errors, including mechanisms for reporting errors outside the available choices already available in the base protocol or other extensions.

- * An indication of whether a Payload stream must be present, and a description of its contents and how receivers use such Payload streams to reconstruct RPC messages.
- * As appropriate, a statement of whether a Responder may use Remote Invalidation when sending messages that contain the new header type.

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

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

B.3. Adding New Transport properties to the Protocol

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

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

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

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

- * `rpcrdma2_propid` values specified in the document do not conflict with those currently assigned or in use by other pending working group documents defining transport properties.

- * rpcrdma2_propid values specified in the document do not conflict with the range reserved for experimental use, as defined in Section 8.2.

[cel: There is no longer a section 8.2 or an experimental range of propid values. Should we request the creation of an IANA registry for propid values?].

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

B.4. Adding New Error Codes to the Protocol

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

For error codes that do not require the return of additional information, a peer can use the existing RDMA_ERR2 header type to report the new error. The sender sets the new error code as the value of rdma_err with the result that the default switch arm of the rpcrdma2_error (i.e., void) is selected.

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

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

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

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

This section details specific changes in RPC-over-RDMA version 2 that address these constraints directly, in addition to other changes to make implementation easier.

C.1. Changes to the XDR Definition

Several XDR structural changes enable within-version protocol extensibility.

[RFC8166] defines the RPC-over-RDMA version 1 transport header as a single XDR object, with an RPC message potentially following it. In RPC-over-RDMA version 2, there are separate XDR definitions of the transport header prefix (see Section 6.4), which specifies the transport header type to be used, and the transport header itself (defined within one of the subsections of Section 6.3). This construction is similar to an RPC message, which consists of an RPC header (defined in [RFC5531]) followed by a message defined by an Upper-Layer Protocol.

As a new version of the RPC-over-RDMA transport protocol, RPC-over-RDMA version 2 exists within the versioning rules defined in [RFC8166]. In particular, it maintains the first four words of the protocol header, as specified in Section 4.2 of [RFC8166], even though, as explained in Section 6.2.1 of the current document, the XDR definition of those words is structured differently.

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

- * The first word of the header, the `rdma_xid` field, retains the format and function that it had in RPC-over-RDMA version 1. Because RPC-over-RDMA version 2 messages can convey non-RPC messages, a receiver should not use the contents of this field without consideration of the protocol version and header type.
- * The second word of the header, the `rdma_vers` field, retains the format and function that it had in RPC-over-RDMA version 1. To clearly distinguish version 1 and version 2 messages, senders need to fill in the correct version (fixed after version negotiation). Receivers should check that the content of the `rdma_vers` is correct before using the content of any other header field.
- * The third word of the header, the `rdma_credit` field, retains the size and general purpose that it had in RPC-over-RDMA version 1. However, RPC-over-RDMA version 2 divides this field into two 16-bit subfields. See Section 4.2.1 for further details.

- * The fourth word of the header, previously the union discriminator field `rdma_proc`, retains its format and general function even though the set of valid values has changed. Within RPC-over-RDMA version 2, this word is the `rdma_htype` field of the structure `rdma_start`. The value of this field is now an unsigned 32-bit integer rather than an enum type, to facilitate header type extension.

Beyond conforming to the restrictions specified in [RFC8166], RPC-over-RDMA version 2 attempts to limit the scope of the changes made to ensure interoperability. Although it introduces the Call chunk and splits the two version 1 workhorse procedure types `RDMA_MSG` and `RDMA_NOMSG` into several variants, RPC-over-RDMA version 2 otherwise expresses chunks in the same format and utilizes them the same way.

C.2. Transport Properties

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

C.3. Credit Management Changes

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

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

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

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

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

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

C.4. Inline Threshold Changes

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

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

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

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

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

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

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

C.5. Message Continuation Changes

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

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

C.6. Host Authentication Changes

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

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

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

C.7. Support for Remote Invalidation

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

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

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

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

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

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

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

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

C.8. Integration of Reverse-Direction Operation

Because [RFC5666] did not include specification of reverse-direction operation, [RFC8166] does not include it either. Reverse-direction operation in RPC-over-RDMA version 1 is specified by a separate standards track document [RFC8167].

Reverse-direction operation in RPC-over-RDMA version 1 was constrained by the limited ability to extend that version of the protocol. The most awkward issue is that a receiver needs to peek at ingress RPC message payloads to determine whether it is a Call or Reply message. This is necessary because the meaning of several fields in the RPC-over-RDMA transport header is determined by the direction of the RPC message payload:

- * The meaning of the value in the `rdma_xid` field is determined by the direction of the message because the XID spaces in the forward and reverse directions are distinct.
- * The meaning of the value in the `rdma_credit` field is determined by the direction of the message because credits are granted separately for forward and reverse direction operation.
- * The purpose of Write chunks and the meaning of their length fields is determined by the direction of the message because in Call messages, they are provisional, but in Reply messages, they represent returned results.

The current document remedies this awkwardness by integrating reverse-direction operation into RPC-over-RDMA version 2 so that it can make use of all facilities that are available in the forward-direction, including body chunks, remote invalidation, and message continuation. To enable this integration, the direction of the RPC message payload is encoded in each RPC-over-RDMA version 2 transport header.

C.9. Error Reporting Changes

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

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

C.10. Changes in Terminology

The RPC-over-RDMA version 2 specification makes the following changes in terminology. These changes do not result in changes in the behavior or operation of the protocol.

- * The current document explicitly acknowledges the different semantics and purpose of Write chunks appearing in Call messages and those appearing in Reply messages.
- * The current document introduces the term "payload format" to describe the selection of a mechanism for reducing and conveying an RPC message payload. It replaces the terms "short message" and "long message" with the terms "simple format" and "special format" because this selection is not based only on the size of the payload.
- * The current document introduces the terms "data item chunk" and "body chunk" in order to distinguish the purpose and operation of these two categories of chunk.
- * For improved readability, the current document replaces the terms "RDMA segment" and "plain segment" with the term "segment", and the term "RDMA read segment" with the term "Read segment".
- * The current document refers specifically to the RDMA, DDP, and MPA standards track protocols rather than using the nebulous term "iWARP".

Acknowledgments

The authors gratefully acknowledge the work of Brent Callaghan and Tom Talpey on the original RPC-over-RDMA version 1 specification [RFC5666].

We are deeply indebted to Jana Igeyar for contributing the RPC-over-RDMA version 2 flow control mechanism described in Section 4.2.1.

The authors also wish to thank Bill Baker, Greg Marsden, and Matt Benjamin for their support of this work.

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

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

Authors' Addresses

Charles Lever (editor)
Oracle Corporation
United States of America

Email: chuck.lever@oracle.com

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

Phone: +1 781 572 8038
Email: davenoveck@gmail.com