INTERNET-DRAFT
Expires: December 2005

Tom Talpey
Network Appliance, Inc.

Spencer Shepler
Sun Microsystems, Inc.

Jon Bauman
University of Michigan

July, 2005

NFSv4 Session Extensions
draft-ietf-nfsv4-sess-02

Abstract

Extensions are proposed to NFS version 4 which enable it to support long-lived sessions, endpoint management, and operation atop a variety of RPC transports, including TCP and RDMA. These extensions enable support for reliably implemented client response caching by NFSv4 servers, enhanced security, multipathing and trunking of transport connections. These extensions provide identical benefits over both TCP and RDMA connection types.

Table of Contents

1. Introduction

This draft proposes extensions to NFS version 4 [RFC3530] enabling it to support sessions and endpoint management, and to support operation atop RDMA-capable RPC over transports such as iWARP. [RDMAP, DDP] These extensions enable support for exactly-once semantics by NFSv4 servers, multipathing and trunking of transport connections, and enhanced security. The ability to operate over RDMA enables greatly enhanced performance. Operation over existing TCP is enhanced as well.

While discussed here with respect to IETF-chartered transports, the proposed protocol is intended to function over other standards, such as Infiniband. [IB]

The following are the major aspects of this proposal:

o   Changes are proposed within the framework of NFSv4 minor versioning. RPC, XDR, and the NFSv4 procedures and operations are preserved. The proposed extension functions equally well over existing transports and RDMA, and interoperates transparently with existing implementations, both at the local programmatic interface and over the wire.

o   An explicit session is introduced to NFSv4, and new operations are added to support it. The session allows for enhanced trunking, failover and recovery, and authentication efficiency, along with necessary support for RDMA. The session is implemented as operations within NFSv4 COMPOUND and does not impact layering or interoperability with existing NFSv4 implementations. The NFSv4 callback channel is dynamically associated and is connected by the client and not the server, enhancing security and operation through

firewalls. In fact, the callback channel will be enabled to share the same connection as the operations channel.

- An enhanced RPC layer enables NFSv4 operation atop RDMA. The session assists RDMA-mode connection, and additional facilities are provided for managing RDMA resources at both NFSv4 server and client. Existing NFSv4 operations continue to function as before, though certain size limits are negotiated. A companion draft to this document, "RDMA Transport for ONC RPC" [RPCRDMA] is to be referenced for details of RPC RDMA support.

- Support for exactly-once semantics ("EOS") is enabled by the new session facilities, by providing to the server a way to bound the size of the duplicate request cache for a single client, and to manage its persistent storage.

Block Diagram

```
+-----------------+-------------------------------------+
|     NFSv4       |     NFSv4 + session extensions      |
+-----------------+------+----------------+-------------+
|      Operations        |   Session      |             |
+------------------------+----------------+             |
|                RPC/XDR                  |             |
+-------------------------------+---------+             |
|       Stream Transport        |    RDMA Transport     |
+-------------------------------+-----------------------+
```

1.1. Motivation

NFS version 4 [RFC3530] has been granted "Proposed Standard" status. The NFSv4 protocol was developed along several design points, important among them: effective operation over wide-area networks, including the Internet itself; strong security integrated into the protocol; extensive cross-platform interoperability including integrated locking semantics compatible with multiple operating systems; and protocol extensibility.

The NFS version 4 protocol, however, does not provide support for certain important transport aspects. For example, the protocol does not address response caching, which is required to provide correctness for retried client requests across a network partition, nor does it provide an interoperable way to support trunking and multipathing of connections. This leads to inefficiencies, especially where trunking and multipathing are concerned, and presents additional difficulties in supporting RDMA fabrics, in which endpoints may require dedicated or specialized resources.

Sessions can be employed to unify NFS-level constructs such as the clientid, with transport-level constructs such as transport endpoints. Each transport endpoint draws on resources via its membership in a session. Resource management can be more strictly maintained, leading to greater server efficiency in implementing the protocol. The enhanced operation over a session affords an opportunity to the server to implement a highly reliable duplicate request cache, and thereby export exactly-once semantics.

NFSv4 advances the state of high-performance local sharing, by virtue of its integrated security, locking, and delegation, and its excellent coverage of the sharing semantics of multiple operating systems. It is precisely this environment where exactly-once semantics become a fundamental requirement.

Additionally, efforts to standardize a set of protocols for Remote Direct Memory Access, RDMA, over the Internet Protocol Suite have made significant progress. RDMA is a general solution to the problem of CPU overhead incurred due to data copies, primarily at the receiver. Substantial research has addressed this and has borne out the efficacy of the approach. An overview of this is the RDDP Problem Statement document, [RDDPPS].

Numerous upper layer protocols achieve extremely high bandwidth and low overhead through the use of RDMA. Products from a wide variety of vendors employ RDMA to advantage, and prototypes have demonstrated the effectiveness of many more. Here, we are concerned specifically with NFS and NFS-style upper layer protocols; examples from Network Appliance [DAFS, DCK+03], Fujitsu Prime Software Technologies [FJNFS, FJDAFS] and Harvard University [KM02] are all relevant.

By layering a session binding for NFS version 4 directly atop a standard RDMA transport, a greatly enhanced level of performance and transparency can be supported on a wide variety of operating system platforms. These combined capabilities alter the landscape between local filesystems and network attached storage, enable a new level of performance, and lead new classes of application to take advantage of NFS.

1.2. Problem Statement

Two issues drive the current proposal: correctness, and performance. Both are instances of "raising the bar" for NFS, whereby the desire to use NFS in new classes applications can be accommodated by providing the basic features to make such use feasible. Such applications include tightly coupled sharing environments such as cluster computing, high performance computing

(HPC) and information processing such as databases. These trends are explored in depth in [NFSPS].

The first issue, correctness, exemplified among the attributes of local filesystems, is support for exactly-once semantics. Such semantics have not been reliably available with NFS. Server-based duplicate request caches [CJ89] help, but do not reliably provide strict correctness. For the type of application which is expected to make extensive use of the high-performance RDMA-enabled environment, the reliable provision of such semantics is a fundamental requirement.

Introduction of a session to NFSv4 will address these issues. With higher performance and enhanced semantics comes the problem of enabling advanced endpoint management, for example high-speed trunking, multipathing and failover. These characteristics enable availability and performance. RFC3530 presents some issues in permitting a single clientid to access a server over multiple connections.

A second issue encountered in common by NFS implementations is the CPU overhead required to implement the protocol. Primary among the sources of this overhead is the movement of data from NFS protocol messages to its eventual destination in user buffers or aligned kernel buffers. The data copies consume system bus bandwidth and CPU time, reducing the available system capacity for applications. [RDDPPS] Achieving zero-copy with NFS has to date required sophisticated, "header cracking" hardware and/or extensive platform-specific virtual memory mapping tricks.

Combined in this way, NFSv4, RDMA and the emerging high-speed network fabrics will enable delivery of performance which matches that of the fastest local filesystems, preserving the key existing local filesystem semantics, while enhancing them by providing network filesystem sharing semantics.

RDMA implementations generally have other interesting properties, such as hardware assisted protocol access, and support for user space access to I/O. RDMA is compelling here for another reason; hardware offloaded networking support in itself does not avoid data copies, without resorting to implementing part of the NFS protocol in the NIC. Support of RDMA by NFS enables the highest performance at the architecture level rather than by implementation; this enables ubiquitous and interoperable solutions.

By providing file access performance equivalent to that of local file systems, NFSv4 over RDMA will enable applications running on a set of client machines to interact through an NFSv4 file system,

just as applications running on a single machine might interact through a local file system.

This raises the issue of whether additional protocol enhancements to enable such interaction would be desirable and what such enhancements would be. This is a complicated issue which the working group needs to address and will not be further discussed in this document.

1.3. NFSv4 Session Extension Characteristics

This draft will present a solution based upon minor versioning of NFSv4. It will introduce a session to collect transport endpoints and resources such as reply caching, which in turn enables enhancements such as trunking, failover and recovery. It will describe use of RDMA by employing support within an underlying RPC layer [RPCRDMA]. Most importantly, it will focus on making the best possible use of an RDMA transport.

These extensions are proposed as elements of a new minor revision of NFS version 4. In this draft, NFS version 4 will be referred to generically as "NFSv4", when describing properties common to all minor versions. When referring specifically to properties of the original, minor version 0 protocol, "NFSv4.0" will be used, and changes proposed here for minor version 1 will be referred to as "NFSv4.1".

This draft proposes only changes which are strictly upward-compatible with existing RPC and NFS Application Programming Interfaces (APIs).

2. Transport Issues

The Transport Issues section of the document explores the details of utilizing the various supported transports.

2.1. Session Model

The first and most evident issue in supporting diverse transports is how to provide for their differences. This draft proposes introducing an explicit session.

A session introduces minimal protocol requirements, and provides for a highly useful and convenient way to manage numerous endpoint-related issues. The session is a local construct; it represents a named, higher-layer object to which connections can refer, and encapsulates properties important to each associated client.

A session is a dynamically created, long-lived server object created by a client, used over time from one or more transport connections. Its function is to maintain the server's state relative to the connection(s) belonging to a client instance. This state is entirely independent of the connection itself. The session in effect becomes the object representing an active client on a connection or set of connections.

Clients may create multiple sessions for a single clientid, and may wish to do so for optimization of transport resources, buffers, or server behavior. A session could be created by the client to represent a single mount point, for separate read and write "channels", or for any number of other client-selected parameters.

The session enables several things immediately. Clients may disconnect and reconnect (voluntarily or not) without loss of context at the server. (Of course, locks, delegations and related associations require special handling, and generally expire in the extended absence of an open connection.) Clients may connect multiple transport endpoints to this common state. The endpoints may have all the same attributes, for instance when trunked on multiple physical network links for bandwidth aggregation or path failover. Or, the endpoints can have specific, special purpose attributes such as callback channels.

The NFSv4 specification does not provide for any form of flow control; instead it relies on the windowing provided by TCP to throttle requests. This unfortunately does not work with RDMA, which in general provides no operation flow control and will terminate a connection in error when limits are exceeded. Limits are therefore exchanged when a session is created; These limits then provide maxima within which each session's connections must operate, they are managed within these limits as described in [RPCRDMA]. The limits may also be modified dynamically at the server's choosing by manipulating certain parameters present in each NFSv4.1 request.

The presence of a maximum request limit on the session bounds the requirements of the duplicate request cache. This can be used to advantage by a server, which can accurately determine any storage needs and enable it to maintain duplicate request cache persistence and to provide reliable exactly-once semantics.

Finally, given adequate connection-oriented transport security semantics, authentication and authorization may be cached on a per-session basis, enabling greater efficiency in the issuing and processing of requests on both client and server. A proposal for transparent, server-driven implementation of this in NFSv4 has been

made. [CCM] The existence of the session greatly facilitates the implementation of this approach. This is discussed in detail in the Authentication Efficiencies section later in this draft.

2.1.1. Connection State

In RFC3530, the combination of a connected transport endpoint and a clientid forms the basis of connection state. While has been made to be workable with certain limitations, there are difficulties in correct and robust implementation. The NFSv4.0 protocol must provide a server-initiated connection for the callback channel, and must carefully specify the persistence of client state at the server in the face of transport interruptions. The server has only the client's transport address binding (the IP 4-tuple) to identify the client RPC transaction stream and to use as a lookup tag on the duplicate request cache. (A useful overview of this is in [RW96].) If the server listens on multiple adddresses, and the client connects to more than one, it must employ different clientid's on each, negating its ability to aggregate bandwidth and redundancy. In effect, each transport connection is used as the server's representation of client state. But, transport connections are potentially fragile and transitory.

In this proposal, a session identifier is assigned by the server upon initial session negotiation on each connection. This identifier is used to associate additional connections, to renegotiate after a reconnect, to provide an abstraction for the various session properties, and to address the duplicate request cache. No transport-specific information is used in the duplicate request cache implementation of an NFSv4.1 server, nor in fact the RPC XID itself. The session identifier is unique within the server's scope and may be subject to certain server policies such as being bounded in time.

It is envisioned that the primary transport model will be connection oriented. Connection orientation brings with it certain potential optimizations, such as caching of per-connection properties, which are easily leveraged through the generality of the session. However, it is possible that in future, other transport models could be accommodated below the session abstraction.

2.1.2. NFSv4 Channels, Sessions and Connections

There are at least two types of NFSv4 channels: the "operations" channel used for ordinary requests from client to server, and the "back" channel, used for callback requests from server to client.

As mentioned above, different NFSv4 operations on these channels can lead to different resource needs. For example, server callback operations (CB_RECALL) are specific, small messages which flow from server to client at arbitrary times, while data transfers such as read and write have very different sizes and asymmetric behaviors. It is sometimes impractical for the RDMA peers (NFSv4 client and NFSv4 server) to post buffers for these various operations on a single connection. Commingling of requests with responses at the client receive queue is particularly troublesome, due both to the need to manage both solicited and unsolicited completions, and to provision buffers for both purposes. Due to the lack of any ordering of callback requests versus response arrivals, without any other mechanisms, the client would be forced to allocate all buffers sized to the worst case.

The callback requests are likely to be handled by a different task context from that handling the responses. Significant demultiplexing and thread management may be required if both are received on the same queue. However, if callbacks are relatively rare (perhaps due to client access patterns), many of these difficulties can be minimized.

Also, the client may wish to perform trunking of operations channel requests for performance reasons, or multipathing for availability. This proposal permits both, as well as many other session and connection possibilities, by permitting each operation to carry session membership information and to share session (and clientid) state in order to draw upon the appropriate resources. For example, reads and writes may be assigned to specific, optimized connections, or sorted and separated by any or all of size, idempotency, etc.

To address the problems described above, this proposal allows multiple sessions to share a clientid, as well as for multiple connections to share a session.

Single Connection model:

```
                    NFSv4.1 Session
                       /      \
        Operations_Channel   [Back_Channel]
                        \    /
                     Connection
                          |
```

Multi-connection trunked model (2 operations channels shown):

```
                    NFSv4.1 Session
                       /      \
         Operations_Channels  [Back_Channel]
             |          |               |
         Connection Connection     [Connection]
             |          |               |
```

Multi-connection split-use model (2 mounts shown):

```
                             NFSv4.1 Session
                               /                 \
                    (/home)        (/usr/local - readonly)
                    /      \                        |
      Operations_Channel  [Back_Channel]            |
              |                 |          Operations_Channel
          Connection       [Connection]             |
              |                 |            Connection
                                                    |
```

In this way, implementation as well as resource management may be optimized. Each session will have its own response caching and buffering, and each connection or channel will have its own transport resources, as appropriate. Clients which do not require certain behaviors may optimize such resources away completely, by using specific sessions and not even creating the additional channels and connections.

2.1.3. Reconnection, Trunking and Failover

Reconnection after failure references stored state on the server associated with lease recovery during the grace period. The session provides a convenient handle for storing and managing information regarding the client's previous state on a per-connection basis, e.g. to be used upon reconnection. Reconnection to a previously existing session, and its stored resources, are covered in the "Connection Models" section below.

One important aspect of reconnection is that of RPC library support. Traditionally, an Upper Layer RPC-based Protocol such as NFS leaves all transport knowledge to the RPC layer implementation below it. This allows NFS to operate over a wide variety of transports and has proven to be a highly successful approach. The session, however, introduces an abstraction which is, in a way, "between" RPC and NFSv4.1. It is important that the session

abstraction not have ramifications within the RPC layer.

One such issue arises within the reconnection logic of RPC. Previously, an explicit session binding operation, which established session context for each new connection, was explored. This however required that the session binding also be performed during reconnect, which in turn required an RPC request. This additional request requires new RPC semantics, both in implementation and the fact that a new request is inserted into the RPC stream. Also, the binding of a connection to a session required the upper layer to become "aware" of connections, something the RPC layer abstraction architecturally abstracts away. Therefore the session binding is not handled in connection scope but instead explicitly carried in each request.

For Reliability Availability and Serviceability (RAS) issues such as bandwidth aggregation and multipathing, clients frequently seek to make multiple connections through multiple logical or physical channels. The session is a convenient point to aggregate and manage these resources.

2.1.4. Server Duplicate Request Cache

Server duplicate request caches, while not a part of an NFS protocol, have become a standard, even required, part of any NFS implementation. First described in [CJ89], the duplicate request cache was initially found to reduce work at the server by avoiding duplicate processing for retransmitted requests. A second, and in the long run more important benefit, was improved correctness, as the cache avoided certain destructive non-idempotent requests from being reinvoked.

However, such caches do not provide correctness guarantees; they cannot be managed in a reliable, persistent fashion. The reason is understandable - their storage requirement is unbounded due to the lack of any such bound in the NFS protocol, and they are dependent on transport addresses for request matching.

As proposed in this draft, the presence of maximum request count limits and negotiated maximum sizes allows the size and duration of the cache to be bounded, and coupled with a long-lived session identifier, enables its persistent storage on a per-session basis.

This provides a single unified mechanism which provides the following guarantees required in the NFSv4 specification, while extending them to all requests, rather than limiting them only to a subset of state-related requests:

"It is critical the server maintain the last response sent to the client to provide a more reliable cache of duplicate non-idempotent requests than that of the traditional cache described in [CJ89]..." [RFC3530]

The maximum request count limit is the count of active operations, which bounds the number of entries in the cache. Constraining the size of operations additionally serves to limit the required storage to the product of the current maximum request count and the maximum response size. This storage requirement enables server-side efficiencies.

Session negotiation allows the server to maintain other state. An NFSv4.1 client invoking the session destroy operation will cause the server to denegotiate (close) the session, allowing the server to deallocate cache entries. Clients can potentially specify that such caches not be kept for appropriate types of sessions (for example, read-only sessions). This can enable more efficient server operation resulting in improved response times, and more efficient sizing of buffers and response caches.

Similarly, it is important for the client to explicitly learn whether the server is able to implement reliable semantics. Knowledge of whether these semantics are in force is critical for a highly reliable client, one which must provide transactional integrity guarantees. When clients request that the semantics be enabled for a given session, the session reply must inform the client if the mode is in fact enabled. In this way the client can confidently proceed with operations without having to implement consistency facilities of its own.

2.2. Session Initialization and Transfer Models

Session initialization issues, and data transfer models relevant to both TCP and RDMA are discussed in this section.

2.2.1. Session Negotiation

The following parameters are exchanged between client and server at session creation time. Their values allow the server to properly size resources allocated in order to service the client's requests, and to provide the server with a way to communicate limits to the client for proper and optimal operation. They are exchanged prior to all session-related activity, over any transport type. Discussion of their use is found in their descriptions as well as throughout this section.

Maximum Requests

The client's desired maximum number of concurrent requests is passed, in order to allow the server to size its reply cache storage. The server may modify the client's requested limit downward (or upward) to match its local policy and/or resources. Over RDMA-capable RPC transports, the per-request management of low-level transport message credits is handled within the RPC layer. [RPCRDMA]

Maximum Request/Response Sizes

The maximum request and response sizes are exchanged in order to permit allocation of appropriately sized buffers and request cache entries. The size must allow for certain protocol minima, allowing the receipt of maximally sized operations (e.g. RENAME requests which contains two name strings). Note the maximum request/response sizes cover the entire request/response message and not simply the data payload as traditional NFS maximum read or write size. Also note the server implementation may not, in fact probably does not, require the reply cache entries to be sized as large as the maximum response. The server may reduce the client's requested sizes.

Inline Padding/Alignment

The server can inform the client of any padding which can be used to deliver NFSv4 inline WRITE payloads into aligned buffers. Such alignment can be used to avoid data copy operations at the server for both TCP and inline RDMA transfers. For RDMA, the client informs the server in each operation when padding has been applied. [RPCRDMA]

Transport Attributes

A placeholder for transport-specific attributes is provided, with a format to be determined. Possible examples of information to be passed in this parameter include transport security attributes to be used on the connection, RDMA-specific attributes, legacy "private data" as used on existing RDMA fabrics, transport Quality of Service attributes, etc. This information is to be passed to the peer's transport layer by local means which is currently outside the scope of this draft, however one attribute is provided in the RDMA case:

RDMA Read Resources

RDMA implementations must explicitly provision resources to support RDMA Read requests from connected peers. These values must be explicitly specified, to provide adequate resources for matching the peer's expected needs and the connection's delay-bandwidth parameters. The

client provides its chosen value to the server in the initial session creation, the value must be provided in each client RDMA endpoint. The values are asymmetric and should be set to zero at the server in order to conserve RDMA resources, since clients do not issue RDMA Read operations in this proposal. The result is communicated in the session response, to permit matching of values across the connection. The value may not be changed in the duration of the session, although a new value may be requested as part of a new session.

2.2.2. RDMA Requirements

A complete discussion of the operation of RPC-based protocols atop RDMA transports is in [RPCRDMA]. Where RDMA is considered, this proposal assumes the use of such a layering; it addresses only the upper layer issues relevant to making best use of RPC/RDMA.

A connection oriented (reliable sequenced) RDMA transport will be required. There are several reasons for this. First, this model most closely reflects the general NFSv4 requirement of long-lived and congestion-controlled transports. Second, to operate correctly over either an unreliable or unsequenced RDMA transport, or both, would require significant complexity in the implementation and protocol not appropriate for a strict minor version. For example, retransmission on connected endpoints is explicitly disallowed in the current NFSv4 draft; it would again be required with these alternate transport characteristics. Third, the proposal assumes a specific RDMA ordering semantic, which presents the same set of ordering and reliability issues to the RDMA layer over such transports.

The RDMA implementation provides for making connections to other RDMA-capable peers. In the case of the current proposals before the RDDP working group, these RDMA connections are preceded by a "streaming" phase, where ordinary TCP (or NFS) traffic might flow. However, this is not assumed here and sizes and other parameters are explicitly exchanged upon a session entering RDMA mode.

2.2.3. RDMA Connection Resources

On transport endpoints which support automatic RDMA mode, that is, endpoints which are created in the RDMA-enabled state, a single, preposted buffer must initially be provided by both peers, and the client session negotiation must be the first exchange.

On transport endpoints supporting dynamic negotiation, a more sophisticated negotiation is possible, but is not discussed in the

current draft.

RDMA imposes several requirements on upper layer consumers. Registration of memory and the need to post buffers of a specific size and number for receive operations are a primary consideration.

Registration of memory can be a relatively high-overhead operation, since it requires pinning of buffers, assignment of attributes (e.g. readable/writable), and initialization of hardware translation. Preregistration is desirable to reduce overhead. These registrations are specific to hardware interfaces and even to RDMA connection endpoints, therefore negotiation of their limits is desirable to manage resources effectively.

Following the basic registration, these buffers must be posted by the RPC layer to handle receives. These buffers remain in use by the RPC/NFSv4 implementation; the size and number of them must be known to the remote peer in order to avoid RDMA errors which would cause a fatal error on the RDMA connection.

The session provides a natural way for the server to manage resource allocation to each client rather than to each transport connection itself. This enables considerable flexibility in the administration of transport endpoints.

2.2.4. TCP and RDMA Inline Transfer Model

The basic transfer model for both TCP and RDMA is referred to as "inline". For TCP, this is the only transfer model supported, since TCP carries both the RPC header and data together in the data stream.

For RDMA, the RDMA Send transfer model is used for all NFS requests and replies, but data is optionally carried by RDMA Writes or RDMA Reads. Use of Sends is required to ensure consistency of data and to deliver completion notifications. The pure-Send method is typically used where the data payload is small, or where for whatever reason target memory for RDMA is not available.

Inline message exchange

```
       Client                                Server
          :                Request              :
     Send :   ------------------------------>   : untagged
          :                                     :  buffer
          :               Response              :
 untagged :   <------------------------------   : Send
  buffer  :                                     :


       Client                                Server
          :            Read request             :
     Send :   ------------------------------>   : untagged
          :                                     :  buffer
          :       Read response with data       :
 untagged :   <------------------------------   : Send
  buffer  :                                     :


       Client                                Server
          :       Write request with data       :
     Send :   ------------------------------>   : untagged
          :                                     :  buffer
          :            Write response           :
 untagged :   <------------------------------   : Send
  buffer  :                                     :
```

Responses must be sent to the client on the same connection that the request was sent. It is important that the server does not assume any specific client implementation, in particular whether connections within a session share any state at the client. This is also important to preserve ordering of RDMA operations, and especially RMDA consistency. Additionally, it ensures that the RPC RDMA layer makes no requirement of the RDMA provider to open its memory registration handles (Steering Tags) beyond the scope of a single RDMA connection. This is an important security consideration.

Two values must be known to each peer prior to issuing Sends: the maximum number of sends which may be posted, and their maximum size. These values are referred to, respectively, as the message credits and the maximum message size. While the message credits might vary dynamically over the duration of the session, the maximum message size does not. The server must commit to preserving this number of duplicate request cache entires, and preparing a number of receive buffers equal to or greater than its

currently advertised credit value, each of the advertised size. These ensure that transport resources are allocated sufficient to receive the full advertised limits.

Note that the server must post the maximum number of session requests to each client operations channel. The client is not required to spread its requests in any particular fashion across connections within a session. If the client wishes, it may create multiple sessions, each with a single or small number of operations channels to provide the server with this resource advantage. Or, over RDMA the server may employ a "shared receive queue". The server can in any case protect its resources by restricting the client's request credits.

While tempting to consider, it is not possible to use the TCP window as an RDMA operation flow control mechanism. First, to do so would violate layering, requiring both senders to be aware of the existing TCP outbound window at all times. Second, since requests are of variable size, the TCP window can hold a widely variable number of them, and since it cannot be reduced without actually receiving data, the receiver cannot limit the sender. Third, any middlebox interposing on the connection would wreck any possible scheme. [MIDTAX] In this proposal, maximum request count limits are exchanged at the session level to allow correct provisioning of receive buffers by transports.

When operating over TCP or other similar transport, request limits and sizes are still employed in NFSv4.1, but instead of being required for correctness, they provide the basis for efficient server implementation of the duplicate request cache. The limits are chosen based upon the expected needs and capabilities of the client and server, and are in fact arbitrary. Sizes may be specified by the client as zero (requesting the server's preferred or optimal value), and request limits may be chosen in proportion to the client's capabilities. For example, a limit of 1000 allows 1000 requests to be in progress, which may generally be far more than adequate to keep local networks and servers fully utilized.

Both client and server have independent sizes and buffering, but over RDMA fabrics client credits are easily managed by posting a receive buffer prior to sending each request. Each such buffer may not be completed with the corresponding reply, since responses from NFSv4 servers arrive in arbitrary order. When an operations channel is also used for callbacks, the client must account for callback requests by posting additional buffers. Note that implementation-specific facilities such as a shared receive queue may also allow optimization of these allocations.

When a session is created, the client requests a preferred buffer size, and the server provides its answer. The server posts all buffers of at least this size. The client must comply by not sending requests greater than this size. It is recommended that server implementations do all they can to accommodate a useful range of possible client requests. There is a provision in [RPCRDMA] to allow the sending of client requests which exceed the server's receive buffer size, but it requires the server to "pull" the client's request as a "read chunk" via RDMA Read. This introduces at least one additional network roundtrip, plus other overhead such as registering memory for RDMA Read at the client and additional RDMA operations at the server, and is to be avoided.

An issue therefore arises when considering the NFSv4 COMPOUND procedures. Since an arbitrary number (total size) of operations can be specified in a single COMPOUND procedure, its size is effectively unbounded. This cannot be supported by RDMA Sends, and therefore this size negotiation places a restriction on the construction and maximum size of both COMPOUND requests and responses. If a COMPOUND results in a reply at the server that is larger than can be sent in an RDMA Send to the client, then the COMPOUND must terminate and the operation which causes the overflow will provide a TOOSMALL error status result.

2.2.5. RDMA Direct Transfer Model

Placement of data by explicitly tagged RDMA operations is referred to as "direct" transfer. This method is typically used where the data payload is relatively large, that is, when RDMA setup has been performed prior to the operation, or when any overhead for setting up and performing the transfer is regained by avoiding the overhead of processing an ordinary receive.

The client advertises RDMA buffers in this proposed model, and not the server. This means the "XDR Decoding with Read Chunks" described in [RPCRDMA] is not employed by NFSv4.1 replies, and instead all results transferred via RDMA to the client employ "XDR Decoding with Write Chunks". There are several reasons for this.

First, it allows for a correct and secure mode of transfer. The client may advertise specific memory buffers only during specific times, and may revoke access when it pleases. The server is not required to expose copies of local file buffers for individual clients, or to lock or copy them for each client access.

Second, client credits based on fixed-size request buffers are easily managed on the server, but for the server additional management of buffers for client RDMA Reads is not well-bounded.

For example, the client may not perform these RDMA Read operations in a timely fashion, therefore the server would have to protect itself against denial-of-service on these resources.

Third, it reduces network traffic, since buffer exposure outside the scope and duration of a single request/response exchange necessitates additional memory management exchanges.

There are costs associated with this decision. Primary among them is the need for the server to employ RDMA Read for operations such as large WRITE. The RDMA Read operation is a two-way exchange at the RDMA layer, which incurs additional overhead relative to RDMA Write. Additionally, RDMA Read requires resources at the data source (the client in this proposal) to maintain state and to generate replies. These costs are overcome through use of pipelining with credits, with sufficient RDMA Read resources negotiated at session initiation, and appropriate use of RDMA for writes by the client - for example only for transfers above a certain size.

A description of which NFSv4 operation results are eligible for data transfer via RDMA Write is in [NFSDDP]. There are only two such operations: READ and READLINK. When XDR encoding these requests on an RDMA transport, the NFSv4.1 client must insert the appropriate xdr_write_list entries to indicate to the server whether the results should be transferred via RDMA or inline with a Send. As described in [NFSDDP], a zero-length write chunk is used to indicate an inline result. In this way, it is unnecessary to create new operations for RDMA-mode versions of READ and READLINK.

Another tool to avoid creation of new, RDMA-mode operations is the Reply Chunk [RPCRDMA], which is used by RPC in RDMA mode to return large replies via RDMA as if they were inline. Reply chunks are used for operations such as READDIR, which returns large amounts of information, but in many small XDR segments. Reply chunks are offered by the client and the server can use them in preference to inline. Reply chunks are transparent to upper layers such as NFSv4.

In any very rare cases where another NFSv4.1 operation requires larger buffers than were negotiated when the session was created (for example extraordinarily large RENAMEs), the underlying RPC layer may support the use of "Message as an RDMA Read Chunk" and "RDMA Write of Long Replies" as described in [RPCRDMA]. No additional support is required in the NFSv4.1 client for this. The client should be certain that its requested buffer sizes are not so small as to make this a frequent occurrence, however.

All operations are initiated by a Send, and are completed with a Send. This is exactly as in conventional NFSv4, but under RDMA has a significant purpose: RDMA operations are not complete, that is, guaranteed consistent, at the data sink until followed by a successful Send completion (i.e. a receive). These events provide a natural opportunity for the initiator (client) to enable and later disable RDMA access to the memory which is the target of each operation, in order to provide for consistent and secure operation. The RDMAP Send with Invalidate operation may be worth employing in this respect, as it relieves the client of certain overhead in this case.

A "onetime" boolean advisory to each RDMA region might become a hint to the server that the client will use the three-tuple for only one NFSv4 operation. For a transport such as iWARP, the server can assist the client in invalidating the three-tuple by performing a Send with Solicited Event and Invalidate. The server may ignore this hint, in which case the client must perform a local invalidate after receiving the indication from the server that the NFSv4 operation is complete. This may be considered in a future version of this draft and [NFSDDP].

In a trusted environment, it may be desirable for the client to persistently enable RDMA access by the server. Such a model is desirable for the highest level of efficiency and lowest overhead.

RDMA message exchanges

```
       Client                                Server
          :         Direct Read Request         :
     Send :   ------------------------------>   : untagged
          :                                     :  buffer
          :               Segment               :
  tagged  :   <------------------------------   :  RDMA Write
  buffer  :                  :                  :
          :              [Segment]              :
  tagged  :   <------------------------------   : [RDMA Write]
  buffer  :                                     :
          :         Direct Read Response        :
 untagged :   <------------------------------   :  Send (w/Inv.)
  buffer  :                                     :
```

```
            Client                                Server
               :       Direct Write Request         :
          Send :   ------------------------------>  : untagged
               :                                    :  buffer
               :               Segment              :
        tagged :   v-----------------------------   :  RDMA Read
        buffer :   +----------------------------->  :
               :                  :                 :
               :              [Segment]             :
        tagged :   v-----------------------------   : [RDMA Read]
        buffer :   +----------------------------->  :
               :                                    :
               :       Direct Write Response        :
      untagged :   <------------------------------  :  Send (w/Inv.)
        buffer :                                    :
```

2.3. Connection Models

There are three scenarios in which to discuss the connection model. Each will be discussed individually, after describing the common case encountered at initial connection establishment.

After a successful connection, the first request proceeds, in the case of a new client association, to initial session creation, and then optionally to session callback channel binding, prior to regular operation.

Commonly, each new client "mount" will be the action which drives creation of a new session. However there are any number of other approaches. Clients may choose to share a single connection and session among all their mount points. Or, clients may support trunking, where additional connections are created but all within a single session. Alternatively, the client may choose to create multiple sessions, each tuned to the buffering and reliability needs of the mount point. For example, a readonly mount can sharply reduce its write buffering and also makes no requirement for the server to support reliable duplicate request caching.

Similarly, the client can choose among several strategies for clientid usage. Sessions can share a single clientid, or create new clientids as the client deems appropriate. For kernel-based clients which service multiple authenticated users, a single clientid shared across all mount points is generally the most appropriate and flexible approach. For example, all the client's file operations may wish to share locking state and the local client kernel takes the responsibility for arbitrating access locally. For clients choosing to support other authentication

models, perhaps example userspace implementations, a new clientid is indicated. Through use of session create options, both models are supported at the client's choice.

Since the session is explicitly created and destroyed by the client, and each client is uniquely identified, the server may be specifically instructed to discard unneeded presistent state. For this reason, it is possible that a server will retain any previous state indefinitely, and place its destruction under administrative control. Or, a server may choose to retain state for some configurable period, provided that the period meets other NFSv4 requirements such as lease reclamation time, etc. However, since discarding this state at the server may affect the correctness of the server as seen by the client across network partitioning, such discarding of state should be done only in a conservative manner.

Each client request to the server carries a new SEQUENCE operation within each COMPOUND, which provides the session context. This session context then governs the request control, duplicate request caching, and other persistent parameters managed by the server for a session.

2.3.1. TCP Connection Model

The following is a schematic diagram of the NFSv4.1 protocol exchanges leading up to normal operation on a TCP stream.

```
       Client                                Server
  TCPmode :   Create Clientid(nfs_client_id4)   : TCPmode
          :   ------------------------------->   :
          :                                      :
          :     Clientid reply(clientid, ...)   :
          :   <-------------------------------   :
          :                                      :
          :   Create Session(clientid, size S,  :
          :      maxreq N, STREAM, ...)          :
          :   ------------------------------->   :
          :                                      :
          :   Session reply(sessionid, size S', :
          :      maxreq N')                      :
          :   <-------------------------------   :
          :                                      :
          :          <normal operation>          :
          :   ------------------------------->   :
          :   <-------------------------------   :
          :                  :                   :
```

No net additional exchange is added to the initial negotiation by

this proposal. In the NFSv4.1 exchange, the CREATECLIENTID replaces SETCLIENTID (eliding the callback "clientaddr4" addressing) and CREATESESSION subsumes the function of SETCLIENTID_CONFIRM, as described elsewhere in this document. Callback channel binding is optional, as in NFSv4.0. Note that the STREAM transport type is shown above, but since the transport mode remains unchanged and transport attributes are not necessarily exchanged, DEFAULT could also be passed.

2.3.2. Negotiated RDMA Connection Model

One possible design which has been considered is to have a "negotiated" RDMA connection model, supported via use of a session bind operation as a required first step. However due to issues mentioned earlier, this proved problematic. This section remains as a reminder of that fact, and it is possible such a mode can be supported.

It is not considered critical that this be supported for two reasons. One, the session persistence provides a way for the server to remember important session parameters, such as sizes and maximum request counts. These values can be used to restore the endpoint prior to making the first reply. Two, there are currently no critical RDMA parameters to set in the endpoint at the server side of the connection. RDMA Read resources, which are in general not settable after entering RDMA mode, are set only at the client - the originator of the connection. Therefore as long as the RDMA provider supports an automatic RDMA connection mode, no further support is required from the NFSv4.1 protocol for reconnection.

Note, the client must provide at least as many RDMA Read resources to its local queue for the benefit of the server when reconnecting, as it used when negotiating the session. If this value is no longer appropriate, the client should resynchronize its session state, destroy the existing session, and start over with the more appropriate values.

2.3.3. Automatic RDMA Connection Model

The following is a schematic diagram of the NFSv4.1 protocol exchanges performed on an RDMA connection.

```
            Client                                Server
      RDMAmode :                  :                  : RDMAmode
               :                  :                  :
      Prepost  :                  :                  : Prepost
      receive  :                  :                  : receive
               :                                     :
               :   Create Clientid(nfs_client_id4)   :
               :   ------------------------------>   :
               :                                     : Prepost
               :     Clientid reply(clientid, ...)   : receive
               :   <------------------------------   :
      Prepost  :                                     :
      receive  :   Create Session(clientid, size S,  :
               :      maxreq N, RDMA ...)            :
               :   ------------------------------>   :
               :                                     : Prepost <=N'
               :   Session reply(sessionid, size S', :     receives of
               :      maxreq N')                     :     size S'
               :   <------------------------------   :
               :                                     :
               :          <normal operation>         :
               :   ------------------------------>   :
               :   <------------------------------   :
               :                  :                  :
```

2.4. Buffer Management, Transfer, Flow Control

Inline operations in NFSv4.1 behave effectively the same as TCP sends. Procedure results are passed in a single message, and its completion at the client signal the receiving process to inspect the message.

RDMA operations are performed solely by the server in this proposal, as described in the previous "RDMA Direct Model" section. Since server RDMA operations do not result in a completion at the client, and due to ordering rules in RDMA transports, after all required RDMA operations are complete, a Send (Send with Solicited Event for iWARP) containing the procedure results is performed from server to client. This Send operation will result in a completion which will signal the client to inspect the message.

In the case of client read-type NFSv4 operations, the server will have issued RDMA Writes to transfer the resulting data into client-advertised buffers. The subsequent Send operation performs two necessary functions: finalizing any active or pending DMA at the client, and signaling the client to inspect the message.

In the case of client write-type NFSv4 operations, the server will have issued RDMA Reads to fetch the data from the client-advertised buffers. No data consistency issues arise at the client, but the completion of the transfer must be acknowledged, again by a Send from server to client.

In either case, the client advertises buffers for direct (RDMA style) operations. The client may desire certain advertisement limits, and may wish the server to perform remote invalidation on its behalf when the server has completed its RDMA. This may be considered in a future version of this draft.

In the absence of remote invalidation, the client may perform its own, local invalidation after the operation completes. This invalidation should occur prior to any RPCSEC GSS integrity checking, since a validly remotely accessible buffer can possibly be modified by the peer. However, after invalidation and the contents integrity checked, the contents are locally secure.

Credit updates over RDMA transports are supported at the RPC layer as described in [RPCRDMA]. In each request, the client requests a desired number of credits to be made available to the connection on which it sends the request. The client must not send more requests than the number which the server has previously advertised, or in the case of the first request, only one. If the client exceeds its credit limit, the connection may close with a fatal RDMA error.

The server then executes the request, and replies with an updated credit count accompanying its results. Since replies are sequenced by their RDMA Send order, the most recent results always reflect the server's limit. In this way the client will always know the maximum number of requests it may safely post.

Because the client requests an arbitrary credit count in each request, it is relatively easy for the client to request more, or fewer, credits to match its expected need. A client that discovered itself frequently queuing outgoing requests due to lack of server credits might increase its requested credits proportionately in response. Or, a client might have a simple, configurable number. The protocol also provides a per-operation "maxslot" exchange to assist in dynamic adjustment at the session level, described in a later section.

Occasionally, a server may wish to reduce the total number of credits it offers a certain client on a connection. This could be encountered if a client were found to be consuming its credits slowly, or not at all. A client might notice this itself, and reduce its requested credits in advance, for instance requesting

only the count of operations it currently has queued, plus a few as a base for starting up again. Such mechanisms can, however, be potentially complicated and are implementation-defined. The protocol does not require them.

Because of the way in which RDMA fabrics function, it is not possible for the server (or client back channel) to cancel outstanding receive operations. Therefore, effectively only one credit can be withdrawn per receive completion. The server (or client back channel) would simply not replenish a receive operation when replying. The server can still reduce the available credit advertisement in its replies to the target value it desires, as a hint to the client that its credit target is lower and it should expect it to be reduced accordingly. Of course, even if the server could cancel outstanding receives, it cannot do so, since the client may have already sent requests in expectation of the previous limit.

This brings out an interesting scenario similar to the client reconnect discussed earlier in "Connection Models". How does the server reduce the credits of an inactive client?

One approach is for the server to simply close such a connection and require the client to reconnect at a new credit limit. This is acceptable, if inefficient, when the connection setup time is short and where the server supports persistent session semantics.

A better approach is to provide a back channel request to return the operations channel credits. The server may request the client to return some number of credits, the client must comply by performing operations on the operations channel, provided of course that the request does not drop the client's credit count to zero (in which case the connection would deadlock). If the client finds that it has no requests with which to consume the credits it was previously granted, it must send zero-length Send RDMA operations, or NULL NFSv4 operations in order to return the resources to the server. If the client fails to comply in a timely fashion, the server can recover the resources by breaking the connection.

While in principle, the back channel credits could be subject to a similar resource adjustment, in practice this is not an issue, since the back channel is used purely for control and is expected to be statically provisioned.

It is important to note that in addition to maximum request counts, the sizes of buffers are negotiated per-session. This permits the most efficient allocation of resources on both peers. There is an important requirement on reconnection: the sizes posted by the

server at reconnect must be at least as large as previously used, to allow recovery. Any replies that are replayed from the server's duplicate request cache must be able to be received into client buffers. In the case where a client has received replies to all its retried requests (and therefore received all its expected responses), then the client may disconnect and reconnect with different buffers at will, since no cache replay will be required.

2.5. Retry and Replay

NFSv4.0 forbids retransmission on active connections over reliable transports; this includes connected-mode RDMA. This restriction must be maintained in NFSv4.1.

If one peer were to retransmit a request (or reply), it would consume an additional credit on the other. If the server retransmitted a reply, it would certainly result in an RDMA connection loss, since the client would typically only post a single receive buffer for each request. If the client retransmitted a request, the additional credit consumed on the server might lead to RDMA connection failure unless the client accounted for it and decreased its available credit, leading to wasted resources.

RDMA credits present a new issue to the duplicate request cache in NFSv4.1. The request cache may be used when a connection within a session is lost, such as after the client reconnects. Credit information is a dynamic property of the connection, and stale values must not be replayed from the cache. This implies that the request cache contents must not be blindly used when replies are issued from it, and credit information appropriate to the channel must be refreshed by the RPC layer.

Finally, RDMA fabrics do not guarantee that the memory handles (Steering Tags) within each rdma three-tuple are valid on a scope outside that of a single connection. Therefore, handles used by the direct operations become invalid after connection loss. The server must ensure that any RDMA operations which must be replayed from the request cache use the newly provided handle(s) from the most recent request.

2.6. The Back Channel

The NFSv4 callback operations present a significant resource problem for the RDMA enabled client. Clearly, callbacks must be negotiated in the way credits are for the ordinary operations channel for requests flowing from client to server. But, for callbacks to arrive on the same RDMA endpoint as operation replies

would require dedicating additional resources, and specialized demultiplexing and event handling. Or, callbacks may not require RDMA sevice at all (they do not normally carry substantial data payloads). It is highly desirable to streamline this critical path via a second communications channel.

The session callback channel binding facility is designed for exactly such a situation, by dynamically associating a new connected endpoint with the session, and separately negotiating sizes and counts for active callback channel operations. The binding operation is firewall-friendly since it does not require the server to initiate the connection.

This same method serves as well for ordinary TCP connection mode. It is expected that all NFSv4.1 clients may make use of the session facility to streamline their design.

The back channel functions exactly the same as the operations channel except that no RDMA operations are required to perform transfers, instead the sizes are required to be sufficiently large to carry all data inline, and of course the client and server reverse their roles with respect to which is in control of credit management. The same rules apply for all transfers, with the server being required to flow control its callback requests.

The back channel is optional. If not bound on a given session, the server must not issue callback operations to the client. This in turn implies that such a client must never put itself in the situation where the server will need to do so, lest the client lose its connection by force, or its operation be incorrect. For the same reason, if a back channel is bound, the client is subject to revocation of its delegations if the back channel is lost. Any connection loss should be corrected by the client as soon as possible.

This can be convenient for the NFSv4.1 client; if the client expects to make no use of back channel facilities such as delegations, then there is no need to create it. This may save significant resources and complexity at the client.

For these reasons, if the client wishes to use the back channel, that channel must be bound first, before using the operations channel. In this way, the server will not find itself in a position where it will send callbacks on the operations channel when the client is not prepared for them.

There is one special case, that where the back channel is bound in fact to the operations channel's connection. This configuration

would be used normally over a TCP stream connection to exactly implement the NFSv4.0 behavior, but over RDMA would require complex resource and event management at both sides of the connection. The server is not required to accept such a bind request on an RDMA connection for this reason, though it is recommended.

2.7. COMPOUND Sizing Issues

Very large responses may pose duplicate request cache issues. Since servers will want to bound the storage required for such a cache, the unlimited size of response data in COMPOUND may be troublesome. If COMPOUND is used in all its generality, then the inclusion of certain non-idempotent operations within a single COMPOUND request may render the entire request non-idempotent. (For example, a single COMPOUND request which read a file or symbolic link, then removed it, would be obliged to cache the data in order to allow identical replay). Therefore, many requests might include operations that return any amount of data.

It is not satisfactory for the server to reject COMPOUNDs at will with NFS4ERR_RESOURCE when they pose such difficulties for the server, as this results in serious interoperability problems. Instead, any such limits must be explicitly exposed as attributes of the session, ensuring that the server can explicitly support any duplicate request cache needs at all times.

2.8. Data Alignment

A negotiated data alignment enables certain scatter/gather optimizations. A facility for this is supported by [RPCRDMA]. Where NFS file data is the payload, specific optimizations become highly attractive.

Header padding is requested by each peer at session initiation, and may be zero (no padding). Padding leverages the useful property that RDMA receives preserve alignment of data, even when they are placed into anonymous (untagged) buffers. If requested, client inline writes will insert appropriate pad bytes within the request header to align the data payload on the specified boundary. The client is encouraged to be optimistic and simply pad all WRITEs within the RPC layer to the negotiated size, in the expectation that the server can use them efficiently.

It is highly recommended that clients offer to pad headers to an appropriate size. Most servers can make good use of such padding, which allows them to chain receive buffers in such a way that any data carried by client requests will be placed into appropriate

buffers at the server, ready for filesystem processing. The receiver's RPC layer encounters no overhead from skipping over pad bytes, and the RDMA layer's high performance makes the insertion and transmission of padding on the sender a significant optimization. In this way, the need for servers to perform RDMA Read to satisfy all but the largest client writes is obviated. An added benefit is the reduction of message roundtrips on the network - a potentially good trade, where latency is present.

The value to choose for padding is subject to a number of criteria. A primary source of variable-length data in the RPC header is the authentication information, the form of which is client-determined, possibly in response to server specification. The contents of COMPOUNDs, sizes of strings such as those passed to RENAME, etc. all go into the determination of a maximal NFSv4 request size and therefore minimal buffer size. The client must select its offered value carefully, so as not to overburden the server, and vice-versa. The payoff of an appropriate padding value is higher performance.

```
            Sender gather:
|RPC Request|Pad bytes|Length| -> |User data...|
\------+---------------------/       \
        \                             \
         \    Receiver scatter:        \-----------+- ...
    /-----+----------------\            \           \
    |RPC Request|Pad|Length|   ->  |FS buffer|->|FS buffer|->...
```

In the above case, the server may recycle unused buffers to the next posted receive if unused by the actual received request, or may pass the now-complete buffers by reference for normal write processing. For a server which can make use of it, this removes any need for data copies of incoming data, without resorting to complicated end-to-end buffer advertisement and management. This includes most kernel-based and integrated server designs, among many others. The client may perform similar optimizations, if desired.

Padding is negotiated by the session creation operation, and subsequently used by the RPC RDMA layer, as described in [RPCRDMA].

3. NFSv4 Integration

The following section discusses the integration of the proposed RDMA extensions with NFSv4.0.

3.1. Minor Versioning

Minor versioning is the existing facility to extend the NFSv4 protocol, and this proposal takes that approach.

Minor versioning of NFSv4 is relatively restrictive, and allows for tightly limited changes only. In particular, it does not permit adding new "procedures" (it permits adding only new "operations"). Interoperability concerns make it impossible to consider additional layering to be a minor revision. This somewhat limits the changes that can be proposed when considering extensions.

To support the duplicate request cache integrated with sessions and request control, it is desirable to tag each request with an identifier to be called a Slotid. This identifier must be passed by NFSv4 when running atop any transport, including traditional TCP. Therefore it is not desirable to add the Slotid to a new RPC transport, even though such a transport is indicated for support of RDMA. This draft and [RPCRDMA] do not propose such an approach.

Instead, this proposal conforms to the requirements of NFSv4 minor versioning, through the use of a new operation within NFSv4 COMPOUND procedures as detailed below.

If sessions are in use for a given clientid, this same clientid cannot be used for non-session NFSv4 operation, including NFSv4.0. Because the server will have allocated session-specific state to the active clientid, it would be an unnecessary burden on the server implementor to support and account for additional, non-session traffic, in addition to being of no benefit. Therefore this proposal prohibits a single clientid from doing this. Nevertheless, employing a new clientid for such traffic is supported.

3.2. Slot Identifiers and Server Duplicate Request Cache

The presence of deterministic maximum request limits on a session enables in-progress requests to be assigned unique values with useful properties.

The RPC layer provides a transaction ID (xid), which, while required to be unique, is not especially convenient for tracking requests. The transaction ID is only meaningful to the issuer (client), it cannot be interpreted at the server except to test for equality with previously issued requests. Because RPC operations may be completed by the server in any order, many transaction IDs may be outstanding at any time. The client may therefore perform a computationally expensive lookup operation in the process of

demultiplexing each reply.

In the proposal, there is a limit to the number of active requests. This immediately enables a convenient, computationally efficient index for each request which is designated as a Slot Identifier, or slotid.

When the client issues a new request, it selects a slotid in the range 0..N-1, where N is the server's current "totalrequests" limit granted the client on the session over which the request is to be issued. The slotid must be unused by any of the requests which the client has already active on the session. "Unused" here means the client has no outstanding request for that slotid. Because the slot id is always an integer in the range 0..N-1, client implementations can use the slotid from a server response to efficiently match responses with outstanding requests, such as, for example, by using the slotid to index into a outstanding request array. This can be used to avoid expensive hashing and lookup functions in the performace-critical receive path.

The sequenceid, which accompanies the slotid in each request, is important for a second, important check at the server: it must be able to be determined efficiently whether a request using a certain slotid is a retransmit or a new, never-before-seen request. It is not feasible for the client to assert that it is retransmitting to implement this, because for any given request the client cannot know the server has seen it unless the server actually replies. Of course, if the client has seen the server's reply, the client would not retransmit!

The sequenceid must increase monotonically for each new transmit of a given slotid, and must remain unchanged for any retransmission. The server must in turn compare each newly received request's sequenceid with the last one previously received for that slotid, to see if the new request is:

- A new request, in which the sequenceid is greater than that previously seen in the slot (accounting for sequence wraparound). The server proceeds to execute the new request.

- A retransmitted request, in which the sequenceid is equal to that last seen in the slot. Note that this request may be either complete, or in progress. The server performs replay processing in these cases.

- A misordered duplicate, in which the sequenceid is less than that previously seen in the slot. The server must drop the incoming request, which may imply dropping the connection if

the transport is reliable, as dictated by section 3.1.1 of [RFC3530].

This last condition is possible on any connection, not just unreliable, unordered transports. Delayed behavior on abandoned TCP connections which are not yet closed at the server, or pathological client implementations can cause it, among other causes. Therefore, the server may wish to harden itself against certain repeated occurrences of this, as it would for retransmissions in [RFC3530].

It is recommended, though not necessary for protocol correctness, that the client simply increment the sequenceid by one for each new request on each slotid. This reduces the wraparound window to a minimum, and is useful for tracing and avoidance of possible implementation errors.

The client may however, for implementation-specific reasons, choose a different algorithm. For example it might maintain a single sequence space for all slots in the session - e.g. employing the RPC XID itself. The sequenceid, in any case, is never interpreted by the server for anything but to test by comparison with previously seen values.

The server may thereby use the slotid, in conjunction with the sessionid and sequenceid, within the SEQUENCE portion of the request to maintain its duplicate request cache (DRC) for the session, as opposed to the traditional approach of ONC RPC applications that use the XID along with certain transport information [RW96].

Unlike the XID, the slotid is always within a specific range; this has two implications. The first implication is that for a given session, the server need only cache the results of a limited number of COMPOUND requests. The second implication derives from the first, which is unlike XID-indexed DRCs, the slotid DRC by its nature cannot be overflowed. Through use of the sequenceid to identify retransmitted requests, it is notable that the server does not need to actually cache the request itself, reducing the storage requirements of the DRC further. These new facilities makes it practical to maintain all the required entries for an effective DRC.

The slotid and sequenceid therefore take over the traditional role of the port number in the server DRC implementation, and the session replaces the IP address. This approach is considerably more portable and completely robust - it is not subject to the frequent reassignment of ports as clients reconnect over IP

networks. In addition, the RPC XID is not used in the reply cache, enhancing robustness of the cache in the face of any rapid reuse of XIDs by the client.

It is required to encode the slotid information into each request in a way that does not violate the minor versioning rules of the NFSv4.0 specification. This is accomplished here by encoding it in a control operation within each NFSv4.1 COMPOUND and CB_COMPOUND procedure. The operation easily piggybacks within existing messages. The implementation section of this document describes the specific proposal.

In general, the receipt of a new sequenced request arriving on any valid slot is an indication that the previous DRC contents of that slot may be discarded. In order to further assist the server in slot management, the client is required to use the lowest available slot when issuing a new request. In this way, the server may be able to retire additional entries.

However, in the case where the server is actively adjusting its granted maximum request count to the client, it may not be able to use receipt of the slotid to retire cache entries. The slotid used in an incoming request may not reflect the server's current idea of the client's session limit, because the request may have been sent from the client before the update was received. Therefore, in the downward adjustment case, the server may have to retain a number of duplicate request cache entries at least as large as the old value, until operation sequencing rules allow it to infer that the client has seen its reply.

The SEQUENCE (and CB_SEQUENCE) operation also carries a "maxslot" value which carries additional client slot usage information. The client must always provide its highest-numbered outstanding slot value in the maxslot argument, and the server may reply with a new recognized value. The client should in all cases provide the most conservative value possible, although it can be increased somewhat above the actual instantaneous usage to maintain some minimum or optimal level. This provides a way for the client to yield unused request slots back to the server, which in turn can use the information to reallocate resources. Obviously, maxslot can never be zero, or the session would deadlock.

The server also provides a target maxslot value to the client, which is an indication to the client of the maxslot the server wishes the client to be using. This permits the server to withdraw (or add) resources from a client that has been found to not be using them, in order to more fairly share resources among a varying level of demand from other clients. The client must always comply

with the server's value updates, since they indicate newly established hard limits on the client's access to session resources. However, because of request pipelining, the client may have active requests in flight reflecting prior values, therefore the server must not immediately require the client to comply.

It is worthwhile to note that Sprite RPC [BW87] defined a "channel" which in some ways is similar to the slotid proposed here. Sprite RPC used channels to implement parallel request processing and request/response cache retirement.

3.3. COMPOUND and CB_COMPOUND

Support for per-operation control can be piggybacked onto NFSv4 COMPOUNDs with full transparency, by placing such facilities into their own, new operation, and placing this operation first in each COMPOUND under the new NFSv4 minor protocol revision. The contents of the operation would then apply to the entire COMPOUND.

Recall that the NFSv4 minor revision is contained within the COMPOUND header, encoded prior to the COMPOUNDed operations. By simply requiring that the new operation always be contained in NFSv4 minor COMPOUNDs, the control protocol can piggyback perfectly with each request and response.

In this way, the NFSv4 RDMA Extensions may stay in compliance with the minor versioning requirements specified in section 10 of [RFC3530].

Referring to section 13.1 of the same document, the proposed session-enabled COMPOUND and CB_COMPOUND have the form:

```
+-----+--------------+-----------+------------+-----------+----
| tag | minorversion | numops    | control op | op + args | ...
|     |   (== 1)     | (limited) |  + args    |           |
+-----+--------------+-----------+------------+-----------+----
```

and the reply's structure is:

```
+------------+-----+--------+-------------------------------+--//
|last status | tag | numres | status + control op + results |  //
+------------+-----+--------+-------------------------------+--//
        //-----------------------+----
        // status + op + results | ...
        //-----------------------+----
```

The single control operation within each NFSv4.1 COMPOUND defines the context and operational session parameters which govern that

COMPOUND request and reply. Placing it first in the COMPOUND encoding is required in order to allow its processing before other operations in the COMPOUND.

3.4. eXternal Data Representation Efficiency

RDMA is a copy avoidance technology, and it is important to maintain this efficiency when decoding received messages. Traditional XDR implementations frequently use generated unmarshaling code to convert objects to local form, incurring a data copy in the process (in addition to subjecting the caller to recursive calls, etc). Often, such conversions are carried out even when no size or byte order conversion is necessary.

It is recommended that implementations pay close attention to the details of memory referencing in such code. It is far more efficient to inspect data in place, using native facilities to deal with word size and byte order conversion into registers or local variables, rather than formally (and blindly) performing the operation via fetch, reallocate and store.

Of particular concern is the result of the READDIR operation, in which such encoding abounds.

3.5. Effect of Sessions on Existing Operations

The use of a session replaces the use of the SETCLIENTID and SETCLIENTID_CONFIRM operations, and allows certain simplification of the RENEW and callback addressing mechanisms in the base protocol.

The cb_program and cb_location which are obtained by the server in SETCLIENTID_CONFIRM must not be used by the server, because the NFSv4.1 client performs callback channel designation with BIND_BACKCHANNEL. Therefore the SETCLIENTID and SETCLIENTID_CONFIRM operations becomes obsolete when sessions are in use, and a server should return an error to NFSv4.1 clients which might issue either operation.

Another favorable result of the session is that the server is able to avoid requiring the client to perform OPEN_CONFIRM operations. The existence of a reliable and effective DRC means that the server will be able to determine whether an OPEN request carrying a previously known open_owner from a client is or is not a retransmission. Because of this, the server no longer requires OPEN_CONFIRM to verify whether the client is retransmitting an open request. This in turn eliminates the server's reason for requesting OPEN_CONFIRM - the server can simply replace any

previous information on this open_owner. Client OPEN operations are therefore streamlined, reducing overhead and latency through avoiding the additional OPEN_CONFIRM exchange.

Since the session carries the client liveness indication with it implicitly, any request on a session associated with a given client will renew that client's leases. Therefore the RENEW operation is made unnecessary when a session is present, as any request (including a SEQUENCE operation with or without additional NFSv4 operations) performs its function. It is possible (though this proposal does not make any recommendation) that the RENEW operation could be made obsolete.

An interesting issue arises however if an error occurs on such a SEQUENCE operation. If the SEQUENCE operation fails, perhaps due to an invalid slotid or other non-renewal-based issue, the server may or may not have performed the RENEW. In this case, the state of any renewal is undefined, and the client should make no assumption that it has been performed. In practice, this should not occur but even if it did, it is expected the client would perform some sort of recovery which would result in a new, successful, SEQUENCE operation being run and the client assured that the renewal took place.

3.6. Authentication Efficiencies

NFSv4 requires the use of the RPCSEC_GSS ONC RPC security flavor [RFC2203] to provide authentication, integrity, and privacy via cryptography. The server dictates to the client the use of RPCSEC_GSS, the service (authentication, integrity, or privacy), and the specific GSS-API security mechanism that each remote procedure call and result will use.

If the connection's integrity is protected by an additional means than RPCSEC_GSS, such as via IPsec, then the use of RPCSEC_GSS's integrity service is nearly redundant (See the Security Considerations section for more explanation of why it is "nearly" and not completely redundant). Likewise, if the connection's privacy is protected by additional means, then the use of both RPCSEC_GSS's integrity and privacy services is nearly redundant.

Connection protection schemes, such as IPsec, are more likely to be implemented in hardware than upper layer protocols like RPCSEC_GSS. Hardware-based cryptography at the IPsec layer will be more efficient than software-based cryptography at the RPCSEC_GSS layer.

When transport integrity can be obtained, it is possible for server and client to downgrade their per-operation authentication, after

an appropriate exchange. This downgrade can in fact be as complete as to establish security mechanisms that have zero cryptographic overhead, effectively using the underlying integrity and privacy services provided by transport.

Based on the above observations, a new GSS-API mechanism, called the Channel Conjunction Mechanism [CCM], is being defined. The CCM works by creating a GSS-API security context using as input a cookie that the initiator and target have previously agreed to be a handle for GSS-API context created previously over another GSS-API mechanism.

NFSv4.1 clients and servers should support CCM and they must use as the cookie the handle from a successful RPCSEC_GSS context creation over a non-CCM mechanism (such as Kerberos V5). The value of the cookie will be equal to the handle field of the rpc_gss_init_res structure from the RPCSEC_GSS specification.

The [CCM] Draft provides further discussion and examples.

4. Security Considerations

The NFSv4 minor version 1 retains all of existing NFSv4 security; all security considerations present in NFSv4.0 apply to it equally.

Security considerations of any underlying RDMA transport are additionally important, all the more so due to the emerging nature of such transports. Examining these issues is outside the scope of this draft.

When protecting a connection with RPCSEC_GSS, all data in each request and response (whether transferred inline or via RDMA) continues to receive this protection over RDMA fabrics [RPCRDMA]. However when performing data transfers via RDMA, RPCSEC_GSS protection of the data transfer portion works against the efficiency which RDMA is typically employed to achieve. This is because such data is normally managed solely by the RDMA fabric, and intentionally is not touched by software. Therefore when employing RPCSEC_GSS under CCM, and where integrity protection has been "downgraded", the cooperation of the RDMA transport provider is critical to maintain any integrity and privacy otherwise in place for the session. The means by which the local RPCSEC_GSS implementation is integrated with the RDMA data protection facilities are outside the scope of this draft.

It is logical to use the same GSS context on a session's callback channel as that used on its operations channel(s), particularly when the connection is shared by both. The client must indicate to

the server:

- what security flavor(s) to use in the call back. A special callback flavor might be defined for this.

- if the flavor is RPCSEC_GSS, then the client must have previously created an RPCSEC_GSS session with the server. The client offers to the server the the opaque handle<> value from the rpc_gss_init_res structure, the window size of RPCSEC_GSS sequence numbers, and an opaque gss_cb_handle.

This exchange can be performed as part of session and clientid creation, and the issue warrants careful analysis before being specified.

If the NFS client wishes to maintain full control over RPCSEC_GSS protection, it may still perform its transfer operations using either the inline or RDMA transfer model, or of course employ traditional TCP stream operation. In the RDMA inline case, header padding is recommended to optimize behavior at the server. At the client, close attention should be paid to the implementation of RPCSEC_GSS processing to minimize memory referencing and especially copying. These are well-advised in any case!

The proposed session callback channel binding improves security over that provided by NFSv4 for the callback channel. The connection is client-initiated, and subject to the same firewall and routing checks as the operations channel. The connection cannot be hijacked by an attacker who connects to the client port prior to the intended server. The connection is set up by the client with its desired attributes, such as optionally securing with IPsec or similar. The binding is fully authenticated before being activated.

4.1. Authentication

Proper authentication of the principal which issues any session and clientid in the proposed NFSv4.1 operations exactly follows the similar requirement on client identifiers in NFSv4.0. It must not be possible for a client to impersonate another by guessing its session identifiers for NFSv4.1 operations, nor to bind a callback channel to an existing session. To protect against this, NFSv4.0 requires appropriate authentication and matching of the principal used. This is discussed in Section 16, Security Considerations of [RFC3530]. The same requirement when using a session identifier applies to NFSv4.1 here.

Going beyond NFSv4.0, the presence of a session associated with any clientid may also be used to enhance NFSv4.1 security with respect to client impersonation. In NFSv4.0, there are many operations which carry no clientid, including in particular those which employ a stateid argument. A rogue client which wished to carry out a denial of service attack on another client could perform CLOSE, DELEGRETURN, etc operations with that client's current filehandle, sequenceid and stateid, after having obtained them from eavesdropping or other approach. Locking and open downgrade operations could be similarly attacked.

When an NFSv4.1 session is in place for any clientid, countermeasures are easily applied through use of authentication by the server. Because the clientid and sessionid must be present in each request within a session, the server may verify that the clientid is in fact originating from a principal with the appropriate authenticated credentials, that the sessionid belongs to the clientid, and that the stateid is valid in these contexts. This is in general not possible with the affected operations in NFSv4.0 due to the fact that the clientid is not present in the requests.

In the event that authentication information is not available in the incoming request, for example after a reconnection when the security was previously downgraded using CCM, the server must require the client re-establish the authentication in order that the server may validate the other client-provided context, prior to executing any operation. The sessionid, present in the newly retransmitted request, combined with the retransmission detection enabled by the NFSv4.1 duplicate request cache, are a convenient and reliable context for the server to use for this contingency.

The server should take care to protect itself against denial of service attacks in the creation of sessions and clientids. Clients who connect and create sessions, only to disconnect and never use them may leave significant state behind. (The same issue applies to NFSv4.0 with clients who may perform SETCLIENTID, then never perform SETCLIENTID_CONFIRM.) Careful authentication coupled with resource checks is highly recommended.

5. IANA Considerations

As a proposal based on minor protocol revision, any new minor number might be registered and reserved with the agreed-upon specification. Assigned operation numbers and any RPC constants might undergo the same process.

There are no issues stemming from RDMA use itself regarding port number assignments not already specified by [RFC3530]. Initial connection is via ordinary TCP stream services, operating on the same ports and under the same set of naming services.

In the Automatic RDMA connection model described above, it is possible that a new well-known port, or a new transport type assignment (netid) as described in [RFC3530], may be desirable.

6. NFSv4 Protocol Extensions

This section specifies details of the extensions to NFSv4 proposed by this document. Existing NFSv4 operations (under minor version 0) continue to be fully supported, unmodified.

6.1. Operation: CREATECLIENTID - Instantiate Clientid

SYNOPSIS

client -> clientid

ARGUMENT

```
     struct CREATECLIENTID4args {
          nfs_client_id4  clientdesc;
     };
```

RESULT

```
     struct CREATECLIENTID4resok {
          clientid4       clientid;
          verifier4       clientid_confirm;
     };

     union SETCLIENTID4res switch (nfsstat4 status) {
     case NFS4_OK:
           CREATECLIENTID4resok      resok4;
     case NFS4ERR_CLID_INUSE:
           void;
     default:
           void;
     };
```

DESCRIPTION

The client uses the CREATECLIENTID operation to register a particular client identifier with the server. The clientid returned from this operation will be necessary for requests that create state on the server and will serve as a parent object to sessions created by the client. In order to verify the clientid it must first be used as an argument to CREATESESSION.

IMPLEMENTATION

A server's client record is a 5-tuple:

1. clientdesc.id:
   The long form client identifier, sent via the client.id subfield of the CREATECLIENTID4args structure

2. clientdesc.verifier:
   A client-specific value used to indicate reboots, sent via the clientdesc.verifier subfield of the CREATECLIENTID4args structure

3. principal:
   The RPCSEC_GSS principal sent via the RPC headers

4. clientid:
   The shorthand client identifier, generated by the server and returned via the clientid field in the CREATECLIENTID4resok structure

5. confirmed:
   A private field on the server indicating whether or not a client record has been confirmed. A client record is confirmed if there has been a successful CREATESESSION operation to confirm it. Otherwise it is unconfirmed. An unconfirmed record is established by a CREATECLIENTID call. Any unconfirmed record that is not confirmed within a lease period may be removed.

The following identifiers represent special values for the fields in the records.

id_arg:
   The value of the clientdesc.id subfield of the CREATECLIENTID4args structure of the current request.

verifier_arg:
   The value of the clientdesc.verifier subfield of the CREATECLIENTID4args structure of the current request.

old_verifier_arg:
A value of the clientdesc.verifier field of a client record received in a previous request; this is distinct from verifier_arg.

principal_arg:
The value of the RPCSEC_GSS principal for the current request.

old_principal_arg:
A value of the RPCSEC_GSS principal received for a previous request. This is distinct from principal_arg.

clientid_ret:
The value of the clientid field the server will return in the CREATECLIENTID4resok structure for the current request.

old_clientid_ret:
The value of the clientid field the server returned in the CREATECLIENTID4resok structure for a previous request. This is distinct from clientid_ret.

Since CREATECLIENTID is a non-idempotent operation, we must consider the possibility that replays may occur as a result of a client reboot, network partition, malfunctioning router, etc. Replays are identified by the value of the client field of CREATECLIENTID4args and the method for dealing with them is outlined in the scenarios below.

The scenarios are described in terms of what client records whose clientdesc.id subfield have value equal to id_arg exist in the server's set of client records. Any cases in which there is more than one record with identical values for id_arg represent a server implementation error. Operation in the potential valid cases is summarized as follows.

1) Common case
If no client records with clientdesc.id matching id_arg exist, a new shorthand client identifier clientid_ret is generated, and the following unconfirmed record is added to the server's state.

{ id_arg, verifier_arg, principal_arg, clientid_ret, FALSE }

Subsequently, the server returns clientid_ret.

2) Router Replay
If the server has the following confirmed record, then this request is likely the result of a replayed request due to a

faulty router or lost connection.

{ id_arg, verifier_arg, principal_arg, clientid_ret, TRUE }

Since the record has been confirmed, the client must have received the server's reply from the initial CREATECLIENTID request. Since this is simply a spurious request, there is no modification to the server's state, and the server makes no reply to the client.

3) Client Collision
If the server has the following confirmed record, then this request is likely the result of a chance collision between the values of the clientdesc.id subfield of CREATECLIENTID4args for two different clients.

{ id_arg, *, old_principal_arg, clientid_ret, TRUE }

Since the value of the clientdesc.id subfield of each client record must be unique, there is no modification of the server's state, and NFS4ERR_CLID_INUSE is returned to indicate the client should retry with a different value for the clientdesc.id subfield of CREATECLIENTID4args.

This scenario may also represent a malicious attempt to destroy a client's state on the server. For security reasons, the server MUST NOT remove the client's state when there is a principal mismatch.

4) Replay
If the server has the following unconfirmed record then this request is likely the result of a client replay due to a network partition or some other connection failure.

{ id_arg, verifier_arg, principal_arg, clientid_ret, FALSE }

Since the response to the CREATECLIENTID request that created this record may have been lost, it is not acceptable to drop this duplicate request. However, rather than processing it normally, the existing record is left unchanged and clientid_ret, which was generated for the previous request, is returned.

5) Change of Principal
If the server has the following unconfirmed record then this request is likely the result of a client which has for whatever reasons changed principals (possibly to change security flavor) after calling CREATECLIENTID, but before

calling CREATESESSION.

{ id_arg, verifier_arg, old_principal_arg, clientid_ret, FALSE}

Since the client has not changed, the principal field of the unconfirmed record is updated to principal_arg and clientid_ret is again returned. There is a small possibility that this is merely a collision on the client field of CREATECLIENTID4args between unrelated clients, but since that is unlikely, and an unconfirmed record does not generally have any filesystem pertinent state, we can assume it is the same client without risking loss of any important state.

After processing, the following record will exist on the server.

{ id_arg, verifier_arg, principal_arg, clientid_ret, FALSE}

6) Client Reboot
If the server has the following confirmed client record, then this request is likely from a previously confirmed client which has rebooted.

{ id_arg, old_verifier_arg, principal_arg, clientid_ret, TRUE }

Since the previous incarnation of the same client will no longer be making requests, lock and share reservations should be released immediately rather than forcing the new incarnation to wait for the lease time on the previous incarnation to expire. Furthermore, session state should be removed since if the client had maintained that information across reboot, this request would not have been issued. If the server does not support the CLAIM_DELEGATE_PREV claim type, associated delegations should be purged as well; otherwise, delegations are retained and recovery proceeds according to RFC3530. The client record is updated with the new verifier and its status is changed to unconfirmed.

After processing, clientid_ret is returned to the client and the following record will exist on the server.

{ id_arg, verifier_arg, principal_arg, clientid_ret, FALSE }

7) Reboot before confirmation
If the server has the following unconfirmed record, then this request is likely from a client which rebooted before sending

a CREATESESSION request.

```
{ id_arg, old_verifier_arg, *, clientid_ret, FALSE }
```

Since this is believed to be a request from a new incarnation of the original client, the server updates the value of clientdesc.verifier and returns the original clientid_ret. After processing, the following state exists on the server.

```
{ id_arg, verifier_arg, *, clientid_ret, FALSE }
```

ERRORS

NFS4ERR_BADXDR
NFS4ERR_CLID_INUSE
NFS4ERR_INVAL
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT

6.2. Operation: CREATESESSION - Create New Session and Confirm Clientid

SYNOPSIS

clientid, session_args -> sessionid, session_args

ARGUMENT

```
          struct CREATESESSION4args {
               clientid4       clientid;
               bool            persist;
               count4          maxrequestsize;
               count4          maxresponsesize;
               count4          maxrequests;
               count4          headerpadsize;
               switch (bool clientid_confirm) {
                case TRUE:
                    verifier4 setclientid_confirm;
                case FALSE:
                    void;
               }
               switch (channelmode4 mode) {
                case DEFAULT:
                    void;
                case STREAM:
                    streamchannelattrs4 streamchanattrs;
                case RDMA:
                    rdmachannelattrs4   rdmachanattrs;
               };
          };
```

RESULT

```
     typedef opaque sessionid4[16];

     struct CREATESESSION4resok {
          sessionid4      sessionid;
          bool            persist;
          count4          maxrequestsize;
          count4          maxresponsesize;
          count4          maxrequests;
          count4          headerpadsize;
          switch (channelmode4 mode) {
           case DEFAULT:
               void;
           case STREAM:
               streamchannelattrs4 streamchanattrs;
           case RDMA:
               rdmachannelattrs4   rdmachanattrs;
          };
     };

     union CREATESESSION4res switch (nfsstat4 status) {
     case NFS4_OK:
      CREATESESSION4resok     resok4;
     default:
      void;
     };
```

DESCRIPTION

This operation is used by the client to create new session objects on the server. Additionally the first session created with a new shorthand client identifier serves to confirm the creation of that client's state on the server. The server returns the parameter values for the new session.

IMPLEMENTATION

To describe the implementation, the same notation for client records introduced in the description of CREATECLIENTID is used with the following addition.

clientid_arg:
     The value of the clientid field of the CREATESESSION4args structure of the current request.

Since CREATESESSION is a non-idempotent operation, we must consider the possibility that replays may occur as a result of a client reboot, network partition, malfunctioning router, etc. Replays are

identified by the value of the clientid and sessionid fields of CREATESESSION4args and the method for dealing with them is outlined in the scenarios below.

The processing of this operation is divided into two phases: clientid confirmation and session creation. In case the state for the provided clientid has not been verified, it is confirmed before the session is created. Otherwise the clientid confirmation phase is skipped and only the session creation phase occurs. Note that since only confirmed clients may create sessions, the clientid confirmation stage does not depend upon sessionid_arg.

CLIENTID CONFIRMATION

The operational cases are described in terms of what client records whose clientid field have value equal to clientid_arg exist in the server's set of client records. Any cases in which there is more than one record with identical values for clientid represent a server implementation error. Operation in the potential valid cases is summarized as follows.

1) Common Case

If the server has the following unconfirmed record, then this is the expected confirmation of an unconfirmed record.

```
{ *, *, principal_arg, clientid_arg, FALSE }
```

The confirmed field of the record is set to TRUE and processing of the operation continues normally.

2) Stale Clientid

If the server contains no records with clientid equal to clientid_arg, then most likely the client's state has been purged during a period of inactivity, possibly due to a loss of connectivity. NFS4ERR_STALE_CLIENTID is returned, and no changes are made to any client records on the server.

3) Principal Change or Collision

If the server has the following record, then the client has changed principals after the previous CREATECLIENTID request, or there has been a chance collision between shortand client identifiers.

```
{ *, *, old_principal_arg, clientid_arg, * }
```

Neither of these cases are permissible. Processing stops and NFS4ERR_CLID_INUSE is returned to the client. No changes are made to any client records on the server.

SESSION CREATION

To determine whether this request is a replay, the server examines the sessionid argument provided by the client. If the sessionid matches the identifier of a previously created session, then this request must be interpreted as a replay. No new state is created and a reply with the parameters of the existing session is returned to the client. If a session corresponding to the sessionid does not already exist, then the request is not a replay and is processed as follows.

NOTE: It is the responsibility of the client to generate appropriate values for sessionid. Since the ordering of messages sent on different transport connections is not guaranteed, immediately reusing the sessionid of a previously destroyed session may yield unpredictable results. Client implementations should avoid recently used sessionids to ensure correct behavior.

The server examines the persist, maxrequestsize, maxresponsesize, maxrequests and headerpadsize arguments. For each argument, if the value is acceptable to the server, it is recommended that the server use the provided value to create the new session. If it is not acceptable, the server may use a different value, but must return the value used to the client. These parameters have the following interpretation.

persist:
    True if the client desires server support for "reliable" semantics. For sessions in which only idempotent operations will be used (e.g. a read-only session), clients should set this value to false. If the server does not or cannot provide "reliable" semantics this value must be set to false on return.

maxrequestsize:
    The maximum size of a COMPOUND request that will be sent by the client including RPC headers.

maxresponsesize:
    The maximum size of a COMPOUND reply that the client will accept from the server including RPC headers. The server must not increase the value of this parameter. If a client sends a COMPOUND request for which the size of the reply would exceed this value, the server will return NFS4ERR_RESOURCE.

maxrequests:
    The maximum number of concurrent COMPOUND requests that the client will issue on the session. Subsequent COMPOUND

requests will each be assigned a slot identifier by the client on the range 0 to maxrequests - 1 inclusive. A slot id cannot be reused until the previous request on that slot has completed.

headerpadsize:
The maximum amount of padding the client is willing to apply to ensure that write payloads are aligned on some boundary at the server. The server should reply with its preferred value, or zero if padding is not in use. The server may decrease this value but must not increase it.

The server creates the session by recording the parameter values used and if the persist parameter is true and has been accepted by the server, allocating space for the duplicate request cache (DRC).

If the session state is created successfully, the server associates it with the session identifier provided by the client. This identifier must be unique among the client's active sessions but there is no need for it to be globally unique. Finally, the server returns the negotiated values used to create the session to the client.

ERRORS

NFS4ERR_BADXDR
NFS4ERR_CLID_INUSE
NFS4ERR_RESOURCE
NFS4ERR_SERVERFAULT
NFS4ERR_STALE_CLIENTID

6.3. Operation: BIND_BACKCHANNEL - Create a callback channel binding

SYNOPSIS

Establish a callback channel on the connection.

ARGUMENTS

```
     struct BIND_BACKCHANNEL4args {
          clientid4 clientid;
          uint32_t  callback_program;
          uint32_t  callback_ident;
          count4         maxrequestsize;
          count4         maxresponsesize;
          count4         maxrequests;
          switch (channelmode4 mode) {
           case DEFAULT:
               void;
           case STREAM:
               streamchannelattrs4 streamchanattrs;
           case RDMA:
               rdmachannelattrs4   rdmachanattrs;
          };
     };
```

RESULTS

```
     struct BIND_BACKCHANNEL4resok {
          count4         maxrequestsize;
          count4         maxresponsesize;
          count4         maxrequests;
          switch (channelmode4 mode) {
           case DEFAULT:
               void;
           case STREAM:
               streamchannelattrs4 streamchanattrs;
           case RDMA:
               rdmachannelattrs4   rdmachanattrs;
          };
     };


     union BIND_BACKCHANNEL4res switch (nfsstat4 status) {
      case NFS4_OK:
          BIND_BACKCHANNEL4resok   resok4;
      default:
          void;
     };
```

DESCRIPTION

The BIND_BACKCHANNEL operation serves to establish the current connection as a designated callback channel for the specified session. Normally, only one callback channel is bound, however if more than one are established, they are used at the server's prerogative, no affinity or preference is specified by the client.

The arguments and results of the BIND_BACKCHANNEL call are a subset of the session parameters, and used identically to those values on the callback channel only. However, not all session operation channel parameters are relevant to the callback channel, for example header padding (since writes of bulk data are not performed in callbacks).

ERRORS

 ...

6.4. Operation: DESTROYSESSION - Destroy existing session

SYNOPSIS

```
void -> status
```

ARGUMENT

```
struct DESTROYSESSION4args {
     sessionid4     sessionid; };
```

RESULT

```
struct SESSION_DESTROYres {
     nfsstat status;
 };
```

DESCRIPTION

The SESSION_DESTROY operation closes the session and discards any active state such as locks, leases, and server duplicate request cache entries. Any remaining connections bound to the session are immediately unbound and may additionally be closed by the server.

This operation must be the final, or only operation in any request. Because the operation results in destruction of the session, any duplicate request caching for this request, as well as previously completed requests, will be lost. For this reason, it is advisable to not place this operation in a request with other state-modifying operations. In addition, a SEQUENCE operation is not required in the request.

Note that because the operation will never be replayed by the server, a client that retransmits the request may receive an error in response, even though the session may have been successfully destroyed.

...

ERRORS

<tbd>

6.5. Operation: SEQUENCE - Supply per-procedure sequencing and control

SYNOPSIS

```
     control -> control
```

ARGUMENT

```
     typedef uint32_t sequenceid4;
     typedef uint32_t slotid4;

     struct SEQUENCE4args {
          clientid4 clientid;
          sessionid4     sessionid;
          sequenceid4    sequenceid;
          slotid4        slotid;
          slotid4        maxslot;
     };
```

RESULT

```
     struct SEQUENCE4resok {
          clientid4 clientid;
          sessionid4     sessionid;
          sequenceid4    sequenceid;
          slotid4        slotid;
          slotid4        maxslot;
          slotid4        target_maxslot;
     };

     union SEQUENCE4res switch (nfsstat4 status) {
      case NFS4_OK:
          SEQUENCE4resok resok4;
      default:
          void;
     };
```

DESCRIPTION

The SEQUENCE operation is used to manage operational accounting for the session on which the operation is sent. The contents include the client and session to which this request belongs, slotid and sequenceid, used by the server to implement session request control and the duplicate reply cache semantics, and exchanged slot counts

which are used to adjust these values. This operation must appear once as the first operation in each COMPOUND sent after the channel is successfully bound, or a protocol error must result.

...

ERRORS

```
      NFS4ERR_BADSESSION
      NFS4ERR_BADSLOT
```

6.6. Callback operation: CB_RECALLCREDIT - change flow control limits

SYNOPSIS

```
     targetcount -> status
```

ARGUMENTS

```
     struct CB_RECALLCREDIT4args {
          sessionid4     sessionid;
          uint32_t  target;
     };
```

RESULT

```
     struct CB_RECALLCREDIT4res {
          nfsstat4   status;
     };
```

DESCRIPTION

The CB_RECALLCREDIT operation requests the client to return session and transport credits to the server, by zero-length RDMA Sends or NULL NFSv4 operations.

...

ERRORS

```
      <none>
```

6.7. Callback operation: CB_SEQUENCE - Supply callback channel sequencing and control

SYNOPSIS

```
     control -> control
```

ARGUMENT

```
     typedef uint32_t sequenceid4;
     typedef uint32_t slotid4;

     struct CB_SEQUENCE4args {
          clientid4 clientid;
          sessionid4     sessionid;
          sequenceid4    sequenceid;
          slotid4        slotid;
          slotid4        maxslot;
     };
```

RESULT

```
     struct CB_SEQUENCE4resok {
          clientid4 clientid;
          sessionid4     sessionid;
          sequenceid4    sequenceid;
          slotid4        slotid;
          slotid4        maxslot;
          slotid4        target_maxslot;
     };

     union CB_SEQUENCE4res switch (nfsstat4 status) {
      case NFS4_OK:
          CB_SEQUENCE4resok   resok4;
      default:
          void;
     };
```

DESCRIPTION

The CB_SEQUENCE operation is used to manage operational accounting for the callback channel of the session on which the operation is sent. The contents include the client and session to which this request belongs, slotid and sequenceid, used by the server to

implement session request control and the duplicate reply cache semantics, and exchanged slot counts which are used to adjust these values. This operation must appear once as the first operation in each CB_COMPOUND sent after the callback channel is successfully bound, or a protocol error must result.

...

ERRORS

NFS4ERR_BADSESSION
NFS4ERR_BADSLOT

7. NFSv4 Session Protocol Description

This section contains the proposed protocol changes in RPC description language. The constants named in this section are illustrative. When the working group decides on the full content of the NFSv4.1 minor revision, they may change in order to avoid conflict.

```
          NFS4ERR_BADSESSION      = 10049,/* invalid session  */
          NFS4ERR_BADSLOT         = 10050 /* invalid slotid   */




          /*
           * CREATECLIENTID: v4.1 setclientid for session use
           */

          struct CREATECLIENTID4args {
               nfs_client_id4 clientdesc;
          };

          struct CREATECLIENTID4resok {
               clientid4 clientid;
               verifier4 clientid_confirm;
          };

          union CREATECLIENTID4res switch (nfsstat4 status) {
           case NFS4_OK:
               CREATECLIENTID4resok     resok4;
           default:
               void;
          };




          /*
           * Channel attributes - TBD.
           */

          enum channelmode4 {
               DEFAULT   = 0,      /* don't change */
               STREAM    = 1,      /* TCP stream */
               RDMA = 2       /* upshift to RDMA */
          };

          struct streamchannelattrs4 {
               opaque nothing[0];  /* TBD */
          };

          struct rdmachannelattrs4 {
               count4    maxrdmareads;
               /* plus TBD */
          };
```

```
          /*
           * CREATESESSION: v4.1 session creation and optional
           * clientid confirm
           */

          typedef opaque sessionid4[16];

          union optverifier4 switch (bool clientid_confirm) {
           case TRUE:
               verifier4 setclientid_confirm;
           case FALSE:
               void;
          };

          union transportattrs4 switch (channelmode4 mode) {
           case DEFAULT:
               void;
           case STREAM:
               streamchannelattrs4 streamchanattrs;
           case RDMA:
               rdmachannelattrs4   rdmachanattrs;
          };

          struct CREATESESSION4args {
               clientid4 clientid;
               bool      persist;
               count4         maxrequestsize;
               count4         maxresponsesize;
               count4         maxrequests;
               count4         headerpadsize;
               optverifier4   verifier;
               transportattrs4     transportattrs;
          };

          struct CREATESESSION4resok {
               sessionid4     sessionid;
               bool      persist;
               count4         maxrequestsize;
               count4         maxresponsesize;
               count4         maxrequests;
               count4         headerpadsize;
               transportattrs4     transportattrs;
          };

          union CREATESESSION4res switch (nfsstat4 status) {
           case NFS4_OK:
               CREATESESSION4resok resok4;
           default:
```

```
                void;
           };




           /*
            * BIND_BACKCHANNEL: v4.1 callback binding
            */

           struct BIND_BACKCHANNEL4args {
                clientid4 clientid;
                uint32_t  callback_program;
                uint32_t  callback_ident;
                count4         maxrequestsize;
                count4         maxresponsesize;
                count4         maxrequests;
                transportattrs4     transportattrs;
           };

           struct BIND_BACKCHANNEL4resok {
                count4         maxrequestsize;
                count4         maxresponsesize;
                count4         maxrequests;
                transportattrs4     transportattrs;
           };


           union BIND_BACKCHANNEL4res switch (nfsstat4 status) {
            case NFS4_OK:
                BIND_BACKCHANNEL4resok   resok4;
            default:
                void;
           };




           /*
            * DESTROYSESSION: v4.1 session destruction
            */

           struct DESTROYSESSION4args {
                sessionid4     sessionid;
           };

           struct DESTROYSESSION4res {
                nfsstat4  status;
```

```
          };




          /*
           * SEQUENCE: v4.1 operation sequence control
           */

          typedef uint32_t sequenceid4;
          typedef uint32_t slotid4;

          struct SEQUENCE4args {
               clientid4 clientid;
               sessionid4     sessionid;
               sequenceid4    sequenceid;
               slotid4        slotid;
               slotid4        maxslot;
          };

          struct SEQUENCE4resok {
               clientid4 clientid;
               sessionid4     sessionid;
               sequenceid4    sequenceid;
               slotid4        slotid;
               slotid4        maxslot;
               slotid4        target_maxslot;
          };

          union SEQUENCE4res switch (nfsstat4 status) {
           case NFS4_OK:
               struct SEQUENCE4resok    resok4;
           default:
               void;
          };




           /* Operation values */
           OP_CREATECLIENTID  = 40,
           OP_CREATESESSION   = 41,
           OP_BIND_BACKCHANNEL= 42,
           OP_DESTROYSESSION  = 43,
           OP_SEQUENCE        = 44,

           /* Operation arguments */
           case OP_CREATECLIENTID:
```

```
                   CREATECLIENTID4args opcreateclientid;
          case OP_CREATESESSION:
                   CREATESESSION4args opcreatesession;
          case OP_BIND_BACKCHANNEL:
                   BIND_BACKCHANNEL4args opbind_backchannel;
          case OP_DESTROYSESSION:
                   DESTROYSESSION4args opdestroysession;
          case OP_SEQUENCE:
                   SEQUENCE4args opsequence;

          /* Operation results */
          case OP_CREATECLIENTID:
                   CREATECLIENTID4res opcreateclientid;
          case OP_CREATESESSION:
                   CREATESESSION4res opcreatesession;
          case OP_BIND_BACKCHANNEL:
                   BIND_BACKCHANNEL4res opbind_backchannel;
          case OP_DESTROYSESSION:
                   DESTROYSESSION4res opdestroysession;
          case OP_SEQUENCE:
                   SEQUENCE4res opsequence;



         /*
          * CB_RECALLCREDIT: Recall session credits from
          * operations channel(s)
          */

         struct CB_RECALLCREDIT4args {
              sessionid4     sessionid;
              uint32_t  target;
         };

         struct CB_RECALLCREDIT4res {
              nfsstat4   status;
         };


         /*
          * CB_SEQUENCE: v4.1 operation sequence control
          */

         struct CB_SEQUENCE4args {
              clientid4 clientid;
              sessionid4     sessionid;
              sequenceid4    sequenceid;
              slotid4        slotid;
```

```
                slotid4         maxslot;
           };

           struct CB_SEQUENCE4resok {
                clientid4 clientid;
                sessionid4      sessionid;
                sequenceid4     sequenceid;
                slotid4         slotid;
                slotid4         maxslot;
                slotid4         target_maxslot;
           };

           union CB_SEQUENCE4res switch (nfsstat4 status) {
            case NFS4_OK:
                struct CB_SEQUENCE4resok resok4;
            default:
                void;
           };


            /* Operation values */
            OP_CB_RECALL_CREDIT     = 5,
            OP_CB_SEQUENCE          = 6

            /* Operation arguments */
            case OP_CB_RECALLCREDIT:
                     CB_RECALLCREDIT4args opcbrecallcredit;
            case OP_CB_SEQUENCE:
                     CB_SEQUENCE4args opcbsequence;

            /* Operation results */
            case OP_CB_RECALLCREDIT:
                     CB_RECALLCREDIT4res opcbrecallcredit;
            case OP_CB_SEQUENCE:
                     CB_SEQUENCE4res opcbsequence;
```

8. Acknowledgements

The authors wish to acknowledge the valuable contributions and review of Charles Antonelli, Brent Callaghan, Mike Eisler, John Howard, Chet Juszczak, Trond Myklebust, Dave Noveck, John Scott, Mike Stolarchuk and Mark Wittle.

9. References

9.1. Normative References

[RFC3530]
S. Shepler, et al., "NFS Version 4 Protocol", Standards Track RFC, http://www.ietf.org/rfc/rfc3530

9.2. Informative References

[BW87]
B. Welch, "The Sprite Remote Procedure Call System", University of California Berkeley Technical Report CSD-87-302, ftp://sunsite.berkeley.edu/pub/techreps/CSD-87-302.html

[CCM]
M. Eisler, N. Williams, "The Channel Conjunction Mechanism (CCM) for GSS", Internet-Draft Work in Progress, http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-ccm

[CJ89]
C. Juszczak, "Improving the Performance and Correctness of an NFS Server," Winter 1989 USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Februry 1989, pages 53-63.

[DAFS]
Direct Access File System, available from http://www.dafscollaborative.org

[DCK+03]
M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, M. Wittle, "The Direct Access File System", in Proceedings of 2nd USENIX Conference on File and Storage Technologies (FAST '03), San Francisco, CA, March 31 - April 2, 2003

[DDP]
H. Shah, J. Pinkerton, R. Recio, P. Culley, "Direct Data Placement over Reliable Transports", Internet-Draft Work in Progress, http://www.ietf.org/internet-drafts/draft-ietf-rddp-ddp

[FJDAFS]
Fujitsu Prime Software Technologies, "Meet the DAFS Performance with DAFS/VI Kernel Implementation using cLAN", http://www.pst.fujitsu.com/english/dafsdemo/index.html

[FJNFS]
Fujitsu Prime Software Technologies, "An Adaptation of VIA to NFS on Linux", http://www.pst.fujitsu.com/english/nfs/index.html

[IB] InfiniBand Architecture Specification, Volume 1, Release 1.1. available from http://www.infinibandta.org

[KM02]
K. Magoutis, "Design and Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD", in Proceedings of USENIX BSDCon 2002 Conference, San Francisco, CA, February 11-14, 2002.

[MAF+02]
K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, E. Gabber, "Structure and Performance of the Direct Access File System (DAFS)", in Proceedings of 2002 USENIX Annual Technical Conference, Monterey, CA, June 9-14, 2002.

[MIDTAX]
B. Carpenter, S. Brim, "Middleboxes: Taxonomy and Issues", Informational RFC, http://www.ietf.org/rfc/rfc3234

[NFSDDP]
B. Callaghan, T. Talpey, "NFS Direct Data Placement", Internet-Draft Work in Progress, http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-nfsdirect

[NFSPS]
T. Talpey, C. Juszczak, "NFS RDMA Problem Statement", Internet-Draft Work in Progress, http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-nfs-rdma-problem-statement

[RDDP]
Remote Direct Data Placement Working Group charter, http://www.ietf.org/html.charters/rddp-charter.html

[RDDPPS]
A. Romanow, J. Mogul, T. Talpey, S. Bailey, Remote Direct Data Placement Working Group Problem Statement, Standards Track Informational RFC, http://www.ietf.org/internet-drafts/draft-ietf-rddp-problem-statement

[RDMAP]
R. Recio, P. Culley, D. Garcia, J. Hilland, "An RDMA Protocol Specification", Internet-Draft Work in Progress,

http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap

[RPCRDMA]
B. Callaghan, T. Talpey, "RDMA Transport for ONC RPC" Internet-Draft Work in Progress, http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-rpcrdma

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

[RW96]
R. Werme, "RPC XID Issues", Connectathon 1996, San Jose, CA, http://www.cthon.org/talks96/werme1.pdf

10. Authors' Addresses

Comments on this draft may be sent to the NFSv4 Working Group (nfsv4@ietf.org) and/or the authors.

Tom Talpey
Network Appliance, Inc.
375 Totten Pond Road
Waltham, MA 02451 USA

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

Spencer Shepler
Sun Microsystems, Inc.
7808 Moonflower Drive
Austin, TX 78750 USA

Phone: +1 512 349 9376
EMail: spencer.shepler@sun.com

Jon Bauman
University of Michigan
Center for Information Technology Integration
535 W. William St. Suite 3100
Ann Arbor, MI 48103 USA

Phone: +1 734 615-4782
Email: baumanj@umich.edu

11. Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.