Network Working Group
Internet Draft
Document: draft-ietf-nfsv4-repl-mig-proto-01.txt

Robert Thurlow
May 2003

A Server-to-Server Replication/Migration Protocol

Status of this Memo

Abstract

NFS Version 4 [RFC3530] provided support for client/server interactions to support replication and migration, but left unspecified how replication and migration would be done. This document is an initial draft of a protocol which could be used to transfer filesystem data and metadata for use with replication and migration services for NFS Version 4.

Table of Contents

1. Introduction

This document describes a proposed protocol to perform the data transfer involved with replication and migration, as the problem was described in [DESIGN]; familiarity with that document is assumed. It is not yet proven by implementation experience, but is presented for collective work and discussion.

Though data replication and transfer are needed in many areas, this document will focus primarily on solving the problem of providing replication and migration support between NFS Version 4 servers. It is assumed that the reader has familiarity with NFS Version 4 [RFC3530].

1.1. Changes Since Last Revision

Since the -00 version of this draft, the following major changes have been made:

- The protocol no longer uses XDR-formatted messages sent via TCP; it now uses RPC calls and replies.
- The elements used to transfer data and metadata are now operations arguments to a unified SEND RPC, so that an array of information about a particular file may be sent in one RPC call.
- Session management has been simplified to a single OPEN_SESSION call and a single CLOSE_SESSION call. Sessions may also be multiplexed over the same connection.
- The protocol should now work in a continuous replication mode, where a transfer session stays up indefinitely and changes can be passed rapidly to replicas.
- Support for transferring delegation state has been added.
- Support for transferring hard links and symbolic links has been added.
- Zero-filled regions or "holes" are now sent as separate operations, rather than being treated as a special case of data transfers.
- ACLs and the object type are handled as part of the RMattrs type, rather than being separate.

1.2. Shortcomings

This draft has the following known shortcomings:

- it does not deal with [RSYNC]-like behaviour, which can compare source and destination files
- it introduces a capabilities negotiation feature which is not complete enough to be useful
- it does not fully specify compression algorithms which can be used
- it does not specify how it works with minor revisions to NFS Version 4

1.3. Rationale

The protocol presented below is a simple bulk-data transfer protocol with minimal traffic in the reverse direction. It is believed that optimal performance is best achieved by a well-implemented source server sending the smallest set of change information to the destination. The advantages in this protocol over data formats such as tar/pax/cpio (as defined by IEEE 1003.1 or ISO/IEC 9945-1) are:

- NFSv4 Access Control Lists (ACLs) and named attributes can be transferred
- The richer NFSv4 metadata set can be transferred
- Restarting of transfers can be achieved
- The bandwidth requirements approach the smallest possible.

1.4. Basic structure

This replication/migration protocol is optimized for bulk data transfer with a minimum of overhead. The ideal case is where the source server can stream filesystem data (or just the changes made) to the destination. An alternate [RSYNC]-like mode which supports both servers comparing files to determine differences has been discussed, but is not present in this draft.

Unlike the previous version of this draft, this version will specify RPC [RFC1831] rather than just XDR [RFC1832] formatted messages over TCP. Implementations MUST support operation over TCP and MAY support

UDP and other transports supported by RPC.

The protocol permits multiple "sessions" per TCP connection by using session identifiers in each RPC. Sessions can be terminated and restarted at a later time. Sessions used to update replicas can also be left in place continuously, so that changes to the master can be reflected on the replicas in near-real-time.

The SEND RPC has been optimized by permitting an array of data and metadata updates to be sent in one RPC call, while the response permits the source server to know how far the destination got in applying the updates.

2. Common data types

2.1. Session, file and checkpoint IDs

RMsession_id permits multiplexing transfer sessions on a single authenticated connection; the value is chosen arbitrarily by the source server. RMcheckpoint is used to track the last RPC known to the destination so that restart can be done; a timestamp is supplied to help choose the earliest checkpoint. RMfile_id is intended to be identical to the NFSv4 fileid attribute.

```
typedef uint64_t RMsession_id;

typedef uint64_t RMfile_id;

struct RMcheckpoint {
        nfstime4 time;
        uint64_t id;
};
```

2.2. Offset, length and cookies

These variables are chosen for compatibility with NFSv4.

```
typedef uint64_t        RMoffset;
typedef uint64_t        RMlength;
typedef uint64_t        RMcookie;
```

2.3. General status

Status responses for OPEN_SESSION and SEND responses and CLOSE_SESSION reasons shall return a value from this set.

```
enum RMstatus {
        RM_OK = 0,
        RMERR_PERM = 1,
        RMERR_IO = 5,
        RMERR_EXISTS = 17
};
```

2.4. From NFS Version 4 [RFC3530]

The following definitions are imported from NFS Version 4.

```
typedef uint32_t        bitmap4<>;
typedef opaque          attrlist4<>;
typedef opaque          utf8string<>;
typedef opaque          utf8str_mixed<>;
typedef opaque          utf8str_cis<>;

struct nfstime4 {
        int64_t         seconds;
        uint32_t        nseconds;
};

enum nfs_ftype4 {
        NF4REG          = 1,    /* Regular File */
        NF4DIR          = 2,    /* Directory */
        NF4BLK          = 3,    /* Special File - block device */
        NF4CHR          = 4,    /* Special File - character device */
        NF4LNK          = 5,    /* Symbolic Link */
        NF4SOCK         = 6,    /* Special File - socket */
        NF4FIFO         = 7,    /* Special File - fifo */
        NF4ATTRDIR      = 8,    /* Attribute Directory */
        NF4NAMEDATTR    = 9     /* Named Attribute */
};

typedef uint32_t        acetype4;
typedef uint32_t        aceflag4;
typedef uint32_t        acemask4;
struct nfsace4 {
        acetype4        type;
        aceflag4        flag;
        acemask4        access_mask;
        utf8string      who;
};

typedef nfsace4         fattr4_acl<>;
```

```
struct fattr4 {
        bitmap4         attrmask;
        attrlist4       attr_vals;
};
```

3. Session Management

Security flavors supported by the destination server may be known in advance, or may be discovered via an initial NULL RPC call which uses SNEGO GSS-API pseudo-mechanism as defined in [RFC2478]. A security flavor normally does not change through the life of the session.

A transfer session is created or resumed with the OPEN_SESSION call and terminated normally or abnormally with the CLOSE_SESSION call. This is simpler than the previous draft of this protocol. The OPEN_SESSION call permits negotiation of capabilities and of the checkpoint to be used for a restart, while CLOSE_SESSION permits abnormal as well as normal termination.

3.1. Capabilities negotiation

Parameters in the OPEN_SESSION call express certain capabilities of the source server and provide an indication of properties of the data to be transferred. The destination server is responsible for reacting to these capabilities. If the desired capabilities are not acceptable to the destination, the response can bid down capabilities by clearing capabilities bits, or reject the session by failing the RPC. If the lowered capabilities bid by the destination server are not acceptable to the source server, the session should be terminated with CLOSE_SESSION.

Currently, only three capabilities are specified; we expect to add more through working group effort. Specified so far are the following:

- RM_UTF8NAMES - source server supports and expects to send filenames encoded in UTF-8 format. If the destination server does not support UTF-8 filenames, it should convey that by clearing the flag.

- RM_FHPRESERVE - source server is willing to attempt to preserve filehandles by sending them as part of each SEND_METADATA operation. If the destination can issue filehandles which it did not generate, and can work with the filehandle format used by the implementation identified by RMimplementation field in the OPEN_SESSION arguments, it can accept this offer; otherwise it should clear the bit to indicate refusal. Since the source

server may be denied in attempting to preserve filehandles, it should either refuse to transfer data if the destination clears this flag, or should advise clients of the possibility that filehandles will change via the [RFC3530] FH4_VOL_MIGRATION bit.

- o RM_FILEID - in combination with RM_FHPRESERVE, the source server is willing to attempt to preserve file_ids as well. If the destination can issue file_ids which it did not generate, and can work with the file_id format used by the implementation identified by RMimplementation field in the OPEN_SESSION arguments, it can accept this offer; otherwise it should clear the bit to indicate refusal.

3.2. Security Negotiation

Security for this protocol is provided by the RPCSEC_GSS mechanism, defined in [RFC2203], with the same GSS-API mechanisms defined as mandatory-to-implement as [RFC3530], namely the Kerberos V5 and LIPKEY mechanisms defined in [RFC1964] and [RFC2847]. In the case of a client and server implementing more than one of these mechanisms, the first RPC call should be an RPC NULL procedure call with the RPCSEC_GSS auth flavor and the SNEGO GSS-API mechanism populated with the mechanisms acceptable to the client. The server should respond with the preferred mechanism, if any, and this mechanism will be used for all sessions on this connection.

3.3. OPEN_SESSION call

SYNOPSIS

OPEN_SESSIONargs -> OPEN_SESSIONres

ARGUMENT

```
struct RMnewsession {
        utf8string src_path;
        utf8string dest_path;
        uint64_t fs_size;
        uint64_t tr_size;
        uint64_t tr_objs;
};

struct RMoldsession {
        RMcheckpoint check_id;
        uint64_t rem_size;
        uint64_t rem_objs;
};
```

```
union RMopeninfo switch (bool new) {
 case TRUE:
        RMnewsession newinfo;
 case FALSE:
        RMoldsession oldinfo;
};

typedef uint64_t RMcapability;
typedef utf8str_cis RMimplementation<>;

struct OPEN_SESSIONargs {
        RMsession_id session_id;
        RMcomp_type comp_list<>;
        RMcapability capabilities;
        RNimplementation implementation;
        RMopeninfo info;
};

RESULT

struct RMopenok {
        RMcheckpoint check_id;
        RMcomp_type comp_alg;
        RMcapability capabilities;
};

union RMopenresp switch (RMstatus status) {
 case RM_OK:
        RMopenok info;
 default:
        void;
};

struct OPEN_SESSIONres {
        RMsession_id session_id;
        RMopenresp response;
};
```

OPEN_SESSION is a request to create or resume a transfer session to send the full or incremental contents of one filesystem. For either new or resuming sessions, the source server supplies the following information:

- session_id - a unique number assigned by the source server to the transfer session, or the number of the session to be resumed.

- comp_list - a list of compression types the source server can use to compress data.

- capabilities - the bitmask used to negotiate as described in Section 4.3.

- implementation - a descriptor of the operating system and filesystem implementation, with version information, used by the source server; this is to permit preservation of filehandles and fileids if the destination server runs a compatible version. This field is constructed at the pleasure of the source server and need only be parsed properly by a destination server running the same operating system code.

For new sessions, the source server supplies the following information:

- src_path - full path name to the filesystem on source server

- dest_path - full path name to the filesystem on the destination server

- fs_size - total size of the filesystem data

- tr_size - amount of filesystem data to be sent during this transfer session

- tr_objs - number of objects to be sent or updated in this transfer session

For resuming sessions, the source server supplies the following information:

- check_id - checkpoint ID for the last RPC believed sent

- rem_size - remaining amount of filesystem data to be sent

- rem_objs - remaining number of objects to be sent or updated

The response from the destination server may reject the session proposal with an error code, may accept the proposal outright, or may bid down capabilities or state that it needs to start from an earlier checkpoint than that proposed by the source. The destination will also choose a compression algorithm from the list the source provided. The source may issue a CLOSE_SESSION call if capabilities negotiated down are not acceptable to it. Once the OPEN_SESSION RPC has been completed, SEND RPCs with data transfer operations will be sent until a CLOSE_SESSION RPC is sent.

3.4. CLOSE_SESSION call

SYNOPSIS

CLOSE_SESSIONargs -> CLOSE_SESSIONres

ARGUMENT

```
struct RMbadclose {
        RMcheckpoint check_id;
        bool_t restartable;
};

union RMcloseinfo switch (RMstatus status) {
 case RM_OK:
        void;
 default:
        RMbadclose info;
};

struct CLOSE_SESSIONargs {
        RMsession_id session_id;
        RMcloseinfo info;
};
```

RESULT

```
struct CLOSE_SESSIONres {
        RMsession_id session_id;
        RMcheckpoint check_id;
};
```

CLOSE_SESSION is used to terminate the session normally or abnormally by the source server.

A normal close is handled by setting the RMcloseinfo status to RM_OK. Upon a normal close, a migration event is considered complete and the source will begin to refer clients to the destination server.

An abnormal close is handled by setting the status to something other than RM_OK and supplying the last checkpoint the source server believes it sent plus an indication of whether it is possible to restart the transfer from that checkpoint. The destination server responds with the last checkpoint it has successfully committed. The destination server should attempt to save the state of the aborted session for a period of at least one hour.

4. Data transfer

4.1. Data transfer operations

Data transfer is accomplished by the SEND RPC, which takes an array of unions to permit a variety of transfer operations to be sent in each RPC. All operations must pertain to one filesystem object, since the RMfile_id is provided for each SEND RPC, not for each operation. Each operation in the array has an RMstatus in the response, so the source server can track how much was done if the call failed. Processingn stops at the first failure, and the SEND RPC response status is set to the first failure status.

The following transfer operations are supported:

- SEND_METADATA - send metadata about object
- SEND_FILE_DATA - send file data
- SEND_FILE_HOLE - send file data
- SEND_LOCK_STATE - send file lock state
- SEND_SHARE_STATE - send share modes state
- SEND_DELEG_STATE - send delegation state
- SEND_REMOVE - send an object removal transaction
- SEND_RENAME - send an object rename transaction
- SEND_LINK - send an object link transaction
- SEND_SYMLINK - send an object symlink transaction
- SEND_DIR_CONTENTS - send names of objects in a directory
- SEND_CLOSE - signal completion of object

4.2. Data transfer phase overview

The source server processes filesystem objects in some known order which will permit checkpointing and restarting in case of some problem or operator abort. Full transfers should be done in order such that objects which are needed, such as directories and link targets, are present when referrals are made to them. Incremental

transfers should be done in the order changes were made on the source server, if possible; if not possible, the order described for full transfers is acceptable.

For files which are to be created or updated, SEND_METADATA is sent first, then SEND_FILE_DATA operations will be sent. If outstanding lock, share or delegation state for an object exists on the source server, it will be sent via SEND_LOCK_STATE, SEND_SHARE_STATE or SEND_DELEG_STATE operations after all data has been transferred. SEND_CLOSE is used to signal that all changes to a file are complete. Directories are created with SEND_METADATA, but are not populated until its objects are created, so the SEND_METADATA is followed by SEND_CLOSE.

Ideally, the source server will track all filesystem changes via a mechanism such as [DMAPI], and will be able to reflect remove, rename and link changes via SEND_REMOVE, SEND_RENAME and SEND_LINK operations. If the source server cannot capture all create and remove operations on a directory reliably, SEND_DIR_CONTENTS should be used. This operation lists all directory entries for a source server, so that the destination server can compute what items should be removed. This is less reliable than being able to send SEND_REMOVE, SEND_RENAME and SEND_LINK operations, and should be used only when the underlying filesystem cannot record changes as they happen.

Named attributes for a filesystem object are handled with SEND_METADATA operations with file type NF4NAMEDATTR. This will be "nested", i.e. it will be understood that the named attribute is associated with the parent object handled. SEND_CLOSE is used to indicate that all data and metadata of the named attribute have been transferred, and must be issued before another named attribute can be handled and before the SEND_CLOSE for the parent object is issued. Named attributes may not themselves have named attributes.

4.3. SEND call

SYNOPSIS

SENDargs -> SENDres

ARGUMENT

```
union RMsendargs switch (RMoptype sendtype) {
 case OP_SEND_METADATA:
        SEND_METADATA metadata;
 case OP_SEND_FILE_DATA:
        SEND_FILE_DATA data;
```

```
    case OP_SEND_FILE_HOLE:
           SEND_FILE_HOLE hole;
    case OP_SEND_LOCK_STATE:
           SEND_LOCK_STATE lock;
    case OP_SEND_SHARE_STATE:
           SEND_SHARE_STATE share;
    case OP_SEND_DELEG_STATE:
           SEND_DELEG_STATE deleg;
    case OP_SEND_REMOVE:
           SEND_REMOVE remove;
    case OP_SEND_RENAME:
           SEND_RENAME rename;
    case OP_SEND_LINK:
           SEND_LINK link;
    case OP_SEND_SYMLINK:
           SEND_SYMLINK symlink;
    case OP_SEND_DIR_CONTENTS:
           SEND_DIR_CONTENTS dirc;
    case OP_SEND_CLOSE:
           void;
   };

   struct SEND1args {
           RMsession_id session_id;
           RMcheckpoint check_id;
           RMfile_id file_id;
           RMsendargs sendarray<>;
   };

   RESULT

   union RMsendres switch (RMoptype sendtype) {
    case OP_SEND_METADATA:
    case OP_SEND_FILE_DATA:
    case OP_SEND_FILE_HOLE:
    case OP_SEND_LOCK_STATE:
    case OP_SEND_SHARE_STATE:
    case OP_SEND_DELEG_STATE:
    case OP_SEND_REMOVE:
    case OP_SEND_RENAME:
    case OP_SEND_LINK:
    case OP_SEND_SYMLINK:
    case OP_SEND_DIR_CONTENTS:
    case OP_SEND_CLOSE:
           RMstatus status;
   };
```

```
struct SEND1res {
        RMsession_id session_id;
        RMcheckpoint check_id;
        RMfile_id file_id;
        RMsendres resarray<>;
        RMstatus status;
};
```

The SEND RPC batches data transfer operations together and sends them to the destination server to operate on one file and with one checkpoint. The destination server may fail a call in the middle of the array by setting the return status for that operation to something other than RM_OK, and will not process further operations. The call will be failed with that status as well.

4.4. Data transfer operation description

4.4.1. SEND_METADATA operation

SYNOPSIS

```
struct SEND_METADATA {
        utf8string obj_name;
        RMattrs attrs;
};
```

SEND_METADATA announces that we are about to transfer information about a particular filesystem object. If an object does not exist on the destination, it will be created with the given obj_name and attributes supplied. If the object exists and is is the correct type, its attributes will be updated. If an object of the same name but a different type exists, it will be removed and recreated with this information. If a SEND_METADATA has not followed a SEND_CLOSE, it may have the is_named_attr flag set, in which case the object is a named attribute of the most recent object identified by a SEND_METADATA.

4.4.2. SEND_FILE_DATA operation

SYNOPSIS

```
struct SEND_FILE_DATA {
        RMoffset offset;
        RMlength length;
        opaque data<>;
};
```

SEND_FILE_DATA sends a block of data for a regular file. The range is identified by the offset, length pair as starting at seek position 'offset' and extending through 'offset+length-1', inclusive.

4.4.3. SEND_FILE_HOLE operation

SYNOPSIS

```
struct SEND_FILE_HOLE {
        RMoffset offset;
        RMlength length;
};
```

SEND_FILE_HOLE sends a description of a "hole", or a zero-filled and usually unallocated block of data. A source server which does sparse allocation and which can learn via APIs what parts of a file are unallocated can use this to describe the hole without transferring the block of zeros.

4.4.4. SEND_LOCK_STATE operation

SYNOPSIS

```
enum RMlocktype {
        RM_NOLOCK = 0,
        RM_READLOCK = 1,
        RM_WRITELOCK = 2
};

struct SEND_LOCK_STATE {
        RMowner owner;
        RMclientid clientid;
        RMoffset offset;
        RMlength length;
        RMlocktype type;
        RMstateid id;
};
```

SEND_LOCK_STATE transfers ownership and range information about outstanding byte-range locks to the destination server. The lock stateid is transferred so that the client need not reestablish the lock after migration. RM_NOLOCK is included to support continuous replication by permitting locks on replicas to be cleared.

4.4.5. SEND_SHARE_STATE operation

SYNOPSIS

```
typedef uint32_t RMaccess;
```

```
typedef uint32_t RMdeny;

struct SEND_SHARE_STATE {
        RMowner owner;
        RMclientid client;
        RMaccess accmode;
        RMdeny denymode;
};
```

SEND_SHARE_STATE transfers ownership and mode information about outstanding share reservations to the destination server.

4.4.6. SEND_DELEG_STATE operation

SYNOPSIS

```
enum RMdelegtype {
        RM_NODELEG = 0,
        RM_READDELEG = 1,
        RM_WRITEDELEG = 2
};

struct SEND_DELEG_STATE {
        RMclientid client;
        RMdelegtype type;
        RMstateid id;
};
```

SEND_DELEG_STATE transfers ownership and type information about outstanding file delegations to the destination server. RM_NODELEG is included to support continuous replication by permitting delegations on replicas to be cleared.

4.4.7. SEND_REMOVE operation

SYNOPSIS

```
struct SEND_REMOVE {
        utf8string name;
};
```

SEND_REMOVE documents a remove event on the object identified; upon receipt, the destination server will remove the object as well.

4.4.8. SEND_RENAME operation

SYNOPSIS

```
struct SEND_RENAME {
        utf8string old_name;
        utf8string new_name;
};
```

SEND_RENAME documents a rename event on the object identified by old_name; upon receipt, the destination server will rename the object in the destination filesystem. Full paths may be used relative to the root of the source filesystem.

4.4.9. SEND_LINK operation

SYNOPSIS

```
struct SEND_LINK {
        utf8string old_name;
        utf8string new_name;
};
```

SEND_LINK documents the creation of a hard link from the old_name to the new_name; upon receipt, the destination server will link the objects in the destination filesystem. Full paths may be used relative to the root of the source filesystem.

4.4.10. SEND_SYMLINK operation

SYNOPSIS

```
struct SEND_SYMLINK {
        utf8string old_name;
        utf8string new_name;
};
```

SEND_SYMLINK documents the creation of a symbolic link from the old_name to the new_name; upon receipt, the destination server will symlink the objects in the destination filesystem. The old_name value is not checked in any way and can be arbitrary textual data.

4.4.11. SEND_DIR_CONTENTS operation

SYNOPSIS

```
struct SEND_DIR_CONTENTS {
        RMcookie cookie;
        bool eof;
        utf8string names<>;
};
```

SEND_DIR_CONTENTS is used to account for removals and renames when source servers cannot record the events such that they may be sent with SEND_REMOVE and SEND_RENAME. The contents are listed in no predictable order so that the destination can what entries it has which are no longer found on the source. Each SEND_DIR_CONTENTS includes an opaque directory cookie to represent starting location of the block on the source server, and the eof flag is set on the last block. Any item existing on the destination that is not listed in a SEND_DIR_CONTENTS operation will be removed.

4.4.12. SEND_CLOSE operation

SYNOPSIS

```
void;
```

SEND_CLOSE is used to announce that all data and metadata changes for a particular object have been completed.

5. IANA Considerations

The replication/migration protocol will use a well-known RPC program number at which destination servers will register. The author will acquire an RPC program number for this purpose.

6. Security Considerations

NFS Version 4 is the primary impetus behind a replication/migration protocol, so this protocol should mandate a strong security scheme in a manner comparable with NFS Version 4. Implementations of this protocol MUST support the RPCSEC_GSS security flavor as defined in [RFC2203] and must also support the Kerberos V5 and LIPKEY mechanisms as defined in [RFC1964] and [RFC2847]. The particular mechanism chosen for sessions is determined by the use of SNEGO on the initial call, which should be a NULL RPC.

7. Appendix A: XDR Protocol Definition File

```
/*
 * Copyright (C) The Internet Society (1998,1999,2000,2001,2002).
 *  All Rights Reserved.
 */

/*
 *      repl-mig.x
 */

%#pragma ident  "@(#)repl-mig.x 1.4     03/05/27"

/*
 * From RFC3530
 */
typedef uint32_t        bitmap4<>;
typedef opaque          attrlist4<>;
typedef opaque          utf8string<>;
typedef opaque          utf8str_mixed<>;
typedef opaque          utf8str_cis<>;

struct nfstime4 {
        int64_t         seconds;
        uint32_t        nseconds;
};

enum nfs_ftype4 {
        NF4REG          = 1,    /* Regular File */
        NF4DIR          = 2,    /* Directory */
        NF4BLK          = 3,    /* Special File - block device */
        NF4CHR          = 4,    /* Special File - character device */
        NF4LNK          = 5,    /* Symbolic Link */
        NF4SOCK         = 6,    /* Special File - socket */
        NF4FIFO         = 7,    /* Special File - fifo */
        NF4ATTRDIR      = 8,    /* Attribute Directory */
        NF4NAMEDATTR    = 9     /* Named Attribute */
};

typedef uint32_t        acetype4;
typedef uint32_t        aceflag4;
typedef uint32_t        acemask4;

struct nfsace4 {
        acetype4        type;
        aceflag4        flag;
        acemask4        access_mask;
```

```
        utf8str_mixed   who;
};

typedef nfsace4         fattr4_acl<>;

struct fattr4 {
        bitmap4         attrmask;
        attrlist4       attr_vals;
};

/*
 * For session, message, file and checkpoint IDs
 */
typedef uint64_t RMsession_id;

typedef uint64_t RMfile_id;

struct RMcheckpoint {
        nfstime4 time;
        uint64_t id;
};

/*
 * For compression algorithm negotiation
 */
enum RMcomp_type {
        RM_NULLCOMP = 0,
        RM_COMPRESS = 1,
        RM_ZIP = 2
};

/*
 * For capabilities negotiation
 */
typedef utf8str_cis RMimplementation<>;
typedef uint64_t RMcapability;
const   RM_UTF8NAMES = 0x00000001;
const   RM_FHPRESERVE = 0x00000002;

/*
 * For general status
 */
enum RMstatus {
        RM_OK = 0,
        RMERR_PERM = 1,
        RMERR_IO = 5,
        RMERR_EXISTS = 17
};
```

```
   /*
    * Attributes
    */
   struct RMattrs {
           fattr4  attr;
           nfs_ftype4 obj_type;
           fattr4_acl obj_acl;
           bool is_named_attr;
   };

   /*
    * Offset, length and cookies
    */
   typedef uint64_t        RMoffset;
   typedef uint64_t        RMlength;
   typedef uint64_t        RMcookie;

   /*
    * Owner
    */
   typedef utf8str_mixed   RMowner;

   /*
    * Lock and share supporting definitions
    */
   struct RMclientid {
           utf8string name;
           opaque address<>;
   };

   struct RMstateid {
           uint32_t        seqid;
           opaque          other[12];
   };

   enum RMlocktype {
           RM_NOLOCK = 0,
           RM_READLOCK = 1,
           RM_WRITELOCK = 2
   };

   typedef uint32_t RMaccess;
   typedef uint32_t RMdeny;

   enum RMdelegtype {
           RM_NODELEG = 0,
           RM_READDELEG = 1,
           RM_WRITEDELEG = 2
```

```
};

/*
 * Protocol elements - session control
 */
struct RMnewsession {
        utf8string src_path;
        utf8string dest_path;
        uint64_t fs_size;
        uint64_t tr_size;
        uint64_t tr_objs;
};

struct RMoldsession {
        RMcheckpoint check_id;
        uint64_t rem_size;
        uint64_t rem_objs;
};

union RMopeninfo switch (bool new) {
 case TRUE:
        RMnewsession newinfo;
 case FALSE:
        RMoldsession oldinfo;
};

struct OPEN_SESSIONargs {
        RMsession_id session_id;
        RMcomp_type comp_list<>;
        RMcapability capabilities;
        RNimplementation impl;
        RMopeninfo info;
};

struct RMopenok {
        RMcheckpoint check_id;
        RMcomp_type comp_alg;
        RMcapability capabilities;
};

union RMopenresp switch (RMstatus status) {
 case RM_OK:
        RMopenok info;
 default:
        void;
};

struct OPEN_SESSIONres {
```

```
        RMsession_id session_id;
        RMopenresp response;
};

struct RMbadclose {
        RMcheckpoint check_id;
        bool_t restartable;
};

union RMcloseinfo switch (RMstatus status) {
 case RM_OK:
        void;
 default:
        RMbadclose info;
};

struct CLOSE_SESSIONargs {
        RMsession_id session_id;
        RMcloseinfo info;
};

struct CLOSE_SESSIONres {
        RMsession_id session_id;
        RMcheckpoint check_id;
};

/*
 * Protocol elements - data transfer
 */
enum RMoptype {
        OP_SEND_METADATA = 1,
        OP_SEND_FILE_DATA = 2,
        OP_SEND_FILE_HOLE = 3,
        OP_SEND_LOCK_STATE = 4,
        OP_SEND_SHARE_STATE = 5,
        OP_SEND_DELEG_STATE = 6,
        OP_SEND_REMOVE = 7,
        OP_SEND_RENAME = 8,
        OP_SEND_LINK = 9,
        OP_SEND_SYMLINK = 10,
        OP_SEND_DIR_CONTENTS = 11,
        OP_SEND_CLOSE = 12
};

/*
 * Data and metadata send items
 */
struct SEND_METADATA {
```

```
        utf8string obj_name;
        RMattrs attrs;
};

struct SEND_FILE_DATA {
        RMoffset offset;
        RMlength length;
        opaque data<>;
};

struct SEND_FILE_HOLE {
        RMoffset offset;
        RMlength length;
};

struct SEND_LOCK_STATE {
        RMowner owner;
        RMclientid client;
        RMoffset offset;
        RMlength length;
        RMlocktype type;
        RMstateid id;
};

struct SEND_SHARE_STATE {
        RMowner owner;
        RMclientid client;
        RMaccess accmode;
        RMdeny denymode;
};

struct SEND_DELEG_STATE {
        RMclientid client;
        RMdelegtype type;
        RMstateid id;
};

struct SEND_REMOVE {
        utf8string name;
};

struct SEND_RENAME {
        utf8string old_name;
        utf8string new_name;
};

struct SEND_LINK {
        utf8string old_name;
```

```
        utf8string new_name;
};

struct SEND_SYMLINK {
        utf8string old_name;
        utf8string new_name;
};

struct SEND_DIR_CONTENTS {
        RMcookie cookie;
        bool eof;
        utf8string names<>;
};

/* no parameters for SEND_CLOSE */

union RMsendargs switch (RMoptype sendtype) {
 case OP_SEND_METADATA:
        SEND_METADATA metadata;
 case OP_SEND_FILE_DATA:
        SEND_FILE_DATA data;
 case OP_SEND_FILE_HOLE:
        SEND_FILE_HOLE hole;
 case OP_SEND_LOCK_STATE:
        SEND_LOCK_STATE lock;
 case OP_SEND_SHARE_STATE:
        SEND_SHARE_STATE share;
 case OP_SEND_DELEG_STATE:
        SEND_DELEG_STATE deleg;
 case OP_SEND_REMOVE:
        SEND_REMOVE remove;
 case OP_SEND_RENAME:
        SEND_RENAME rename;
 case OP_SEND_LINK:
        SEND_LINK link;
 case OP_SEND_SYMLINK:
        SEND_SYMLINK symlink;
 case OP_SEND_DIR_CONTENTS:
        SEND_DIR_CONTENTS dirc;
 case OP_SEND_CLOSE:
        void;
};

union RMsendres switch (RMoptype sendtype) {
 case OP_SEND_METADATA:
 case OP_SEND_FILE_DATA:
 case OP_SEND_FILE_HOLE:
 case OP_SEND_LOCK_STATE:
```

```
    case OP_SEND_SHARE_STATE:
    case OP_SEND_DELEG_STATE:
    case OP_SEND_REMOVE:
    case OP_SEND_RENAME:
    case OP_SEND_LINK:
    case OP_SEND_SYMLINK:
    case OP_SEND_DIR_CONTENTS:
    case OP_SEND_CLOSE:
           RMstatus status;
   };

   struct SEND1args {
           RMsession_id session_id;
           RMcheckpoint check_id;
           RMfile_id file_id;
           RMsendargs sendarray<>;
   };

   struct SEND1res {
           RMsession_id session_id;
           RMcheckpoint check_id;
           RMfile_id file_id;
           RMsendres resarray<>;
           RMstatus status;
   };

   program RM_PROGRAM {
           version RM_V1 {
                   void
                           RMPROC1_NULL(void) = 0;
                   OPEN_SESSIONres
                           RMPROC1_OPEN_SESSION(OPEN_SESSIONargs) = 1;
                   CLOSE_SESSIONres
                           RMPROC1_CLOSE_SESSION(CLOSE_SESSIONargs) = 2;
                   SEND1res
                           RMPROC1_SEND(SEND1args) = 3;
           } = 1;
   } = 100273;
```

8. Normative References

[RFC1831]
R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2", RFC1831, August 1995.

[RFC1832]
R. Srinivasan, "XDR: External Data Representation Standard", RFC1832, August 1995.

[RFC1964]
J. Linn, "Kerberos Version 5 GSS-API Mechanism", RFC1964, June 1996

[RFC2203]
M. Eisler, A. Chiu, L. Ling, "RPCSEC_GSS Protocol Specification", RFC2203, September 1997

[RFC2478]
E. Baize, D. Pinkas, "The Simple and Protected GSS-API Negotiation Mechanism", RFC2478, December 1998.

[RFC2847]
M. Eisler, "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM", RFC2847, June 2000

[RFC3530]
S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, "Network File System (NFS) Version 4 Protocol", RFC3530, April 2003.

9. Informative References

[RDIST]
MagniComp, Inc., "RDist Home Page", http://www.magnicomp.com/rdist.

[RSYNC]
The Samba Team, "rsync web pages", http://samba.anu.edu.au/rsync.

[DESIGN]
R. Thurlow, "Server-to-Server Replication/Migration Protocol Design Principles" (work in progress), http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-repl-mig-design-00.txt, December 2002.

[DMAPI]
P. Lawthers, "The Data Management Applications Programming Interface", http://www.computer.org/conferences/mss95/lawthers/lawthers.htm, July 1995.

10. Author's Address

Address comments related to this memorandum to:

nfsv4-wg@sunroof.eng.sun.com

Robert Thurlow
Sun Microsystems, Inc.
500 Eldorado Boulevard, UBRM05-171
Broomfield, CO 80021

Phone: 877-718-3419
E-mail: robert.thurlow@sun.com