# [MS-FRS2]:
# SD Microsoft Distributed File System Replication Protocol Specification



This protocol documentation is intended for use in conjunction with publicly available standard specifications, network programming art, and Microsoft Windows distributed systems concepts, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

A protocol specification does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them.

## Revision Summary

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 03/14/2007 | 1.0 | | Version 1.0 release |
| 04/10/2007 | 1.1 | | Version 1.1 release |
| 05/18/2007 | 1.2 | | Version 1.2 release |
| 06/08/2007 | 1.2.1 | Editorial | Revised and edited the technical content. |

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 07/10/2007 | 1.2.2 | Editorial | Revised and edited the technical content. |
| 08/17/2007 | 1.2.3 | Editorial | Revised and edited the technical content. |
| 09/21/2007 | 1.2.4 | Editorial | Revised and edited the technical content. |
| 10/26/2007 | 1.3 | Minor | Updated to use data types in MS-DTYP. |
| 01/25/2008 | 1.3.1 | Editorial | Revised and edited the technical content. |

# Table of Contents

# 1 Introduction

The Distributed File System: Replication (DFS-R) Protocol is an **RPC** interface that replicates files between **servers**. DFS-R enables creation of multimaster optimistic file replication systems. It is multimaster, as files may be changed by any member that participates in replicating shared files. It is optimistic, as files may be updated without any prior consensus or serialization. Therefore, files can be changed by any member without requiring the member to prevent other members from changing the files.

DFS-R is designed to replicate files, attributes, and file metadata. DFS-R is intended to interoperate with the user-level file system semantics: Files are replicated when the applications that modify them close the files. File replication is designed to be performed asynchronously, such that **updates** made on one **member** are processed at the rate at which the receiving **machine** is able to receive the updates, without any real-time restrictions on when the changes must be propagated. DFS-R allows user-level file system operations to continue independent of protocol operations.

## 1.1 Glossary

The following terms are defined in [MS-GLOS]:

**Access Control List (ACL)**
**Active Directory (AD)**
**Authentication Level**
**Authentication Service (AS)**
**Client (DFS-R)**
**Computer**
**Connection (DFS-R)**
**ConnectionId**
**Content Set**
**ContentSetId**
**Database (DFS-R)**
**Distributed File System Replication (DFS-R)**
**Drive**
**Dynamic Endpoint**
**Endpoint**
**Fence**
**File Attribute**
**Filter Max**
**Ghosting**
**Hashes and Checksums**
**Interface Definition Language (IDL)**
**Logical Connection**
**Machine**
**Machine Identifier**
**MD4**
**Member**
**Microsoft Interface Definition Language (MIDL)**
**NTFS**
**Opnum**
**Principal Name**
**Read-Only Replicated Folders**
**Remote Differential Compression (RDC)**
**Remote Procedure Call (RPC)**
**Replica Set**
**ReplicaSetId**

**Replicated Folder**
**Replication Group**
**Replication Session**
**RPC Protocol Sequence**
**RPC Transport**
**Selective Single Master**
**Server**
**SHA1 Hash**
**Slow Sync**
**Tombstone (DFS-R)**
**Topology**
**Unique Identifier (UID)**
**Universally Unique Identifier (UUID)**
**UPDATE**
**Updates**
**Version Chain Vector**
**Version Sequence Number (VSN)**
**Version Vector**
**Volume**
**Windows Server Enterprise**

The following terms are specific to this document:

**File System:** The store where replicated files reside and are changed.

**Global Version Sequence Numbers (GVSN):** A GVSN is a pair: **Machine identifier** and **version sequence number (VSN)**. While two machines might assign the same **VSN**, since they have different **machine identifiers**, the associated GVSNs differ. A GVSN is used to identify a unique version of a unique resource. In other words, no two different resources ever get assigned the same GVSN, and no two different updates to the same resource ever get assigned the same GVSN.

**Persist:** To commit (or save) data to Persistent Storage.

**Persistent Storage:** Non-volatile storage mediums, such as magnetic disks, tapes, and optical disks.

**Signature:** A synonym for hash.

**MAY, SHOULD, MUST, SHOULD NOT, MUST NOT:** These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

## 1.2 References

### 1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[FIPS180-2] Federal Information Processing Standards Publication, "Secure Hash Standard", FIPS PUB 180-2, August 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[MS-ADSC] Microsoft Corporation, "Active Directory Schema Classes", July 2006.

[MS-ADTS] Microsoft Corporation, "Active Directory Technical Specification", July 2006.

[MS-BKUP] Microsoft Corporation, "Microsoft NT Backup File Structure Specification", July 2006.

[MS-DRSR] Microsoft Corporation, "Directory Replication Service (DRS) Remote Protocol Specification", July 2006.

[MS-DTYP] Microsoft Corporation, "Windows Data Types", March 2007.

[MS-FSCC] Microsoft Corporation, "File System Control Codes", July 2006.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-KILE] Microsoft Corporation, "Kerberos Protocol Extensions", July 2006.

[MS-NLMP] Microsoft Corporation, "NT LAN Manager (NTLM) Authentication Protocol Specification", July 2006.

[MS-RDC] Microsoft Corporation, "Remote Differential Compression Protocol Specification", July 2006.

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", July 2006.

[MS-SECO] Microsoft Corporation, "Windows Security Overview", December 2006.

[RFC1320] Rivest, R. "The MD4 Message-Digest Algorithm", RFC 1320, April 1992, http://www.ietf.org/rfc/rfc1320.txt

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC3174] Eastlake III, D. and Jones, P., "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001, http://www.ietf.org/rfc/rfc3174.txt

[RFC3986] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", RFC 3986, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[RFC4122] Leach, P., Mealling, M., and Salz, R., "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005, http://www.ietf.org/rfc/rfc4122.txt

### 1.2.2 Informative References

[DFS-R WMI] Microsoft Corporation, "DFSR WMI Classes", http://msdn2.microsoft.com/en-us/library/aa379508.aspx

**Note** This WMI management interface is exposed as a public API in Windows Server 2003 R2, Windows Vista and Windows Server 2008 operating system. A .mof file publishing the capabilities of the WMI provider associated with DFS-R is included in the DFS-R distribution under %%SYSTEM_ROOT%%\webm\dfsrprov.mof.

[MS-WMI] Microsoft Corporation, "Windows Management Instrumentation Remote Protocol Specification", July 2006.

[MSDN-MIDL] Microsoft Corporation, "Microsoft Interface Definition Language (MIDL)", http://msdn2.microsoft.com/en-us/library/ms950375.aspx

[MSDN-RPC] Microsoft Corporation, "Remote Procedure Call", http://msdn2.microsoft.com/en-us/library/aa378651.aspx

[RFC1321] Rivest, R. "The MD5 Message-Digest Algorithm", RFC 1321, April 1992, http://www.ietf.org/rfc/rfc1321.txt

## 1.3 Protocol Overview (Synopsis)

The Distributed File System: Replication (DFS-R) Protocol is used to implement a multimaster file replication system. In this system, no single **computer** is a master, but rather all computers in the replication system share their knowledge by exchanging **version chain vectors**, updates, and files. A computer may take dual roles as both a **client** and a server. As a client, a computer retrieves replicated metadata and replicated files from a server. Conversely, as a server, a computer serves replicated metadata and replicated files to a client.

DFS-R takes a three-tiered approach to file replication.

1. Version chain vectors are retrieved from a server to determine which file versions are known to the server, but not to the client. The protocol requires that a server ensures that the **global version sequence numbers (GVSNs)** of all replicated files and file metadata that it maintains in **persistent storage** (that is, saved to disk) are eventually included in its version chain vector, such that the state of a server's knowledge can be determined by examining the version chain vectors alone.

2. Updates, which summarize file metadata, are retrieved from a server. The client uses the version chain vector received from the server to limit the set of updates that are retrieved from the server. To retrieve all updates known to the server, but not the client, it is sufficient to request updates with a GVSN range over the version chain vector received from the server less the version chain vector maintained by the client. The updates contain file system information about the replicated files, but not the file data. The information includes the coordinates of the file in terms of a **unique identifier (UID)** identifying the file across different versions of the file, the GVSN (identifying a particular version of the file on a particular machine), a reference to the file's parent directory in terms of a UID for the parent resource (directories are treated as files), and a file name.

3. File data is retrieved if a client determines that the file data corresponding to a received update must be downloaded in order for the client to synchronize with the server.

The process of retrieving updates proceeds alternates with retrieving version chain vectors. A client first registers a callback with the server to retrieve the latest version chain vector from the server. When receiving the server's version chain vector, the client retrieves all updates pertaining to it, using successive calls to the server. Finally, when a client cannot retrieve more updates from the version chain vector, it registers another callback with the server to retrieve the server's version chain vector next time the **version vector** changes relative to the last time the callback was registered.

File data can be downloaded at the same time the client retrieves version chain vectors and updates. File downloads thus proceed as an independent process of synchronizing version chain vectors and updates. The client specifies which file data to download based on the UID in the file metadata.

Clients can update their previously saved version chain vector based on the server's version chain vector after a completed synchronization; that is, when all updates pertaining to a version chain vector have been processed and all file data that a client must in order to synchronize with a server

has been downloaded. A client's version chain vector is updated by taking the union of its version chain vector and the server's version chain vector.

The version chain vectors themselves are an abstract measure of the knowledge of a member. They record the versions of files a member has received, processed, and either discarded or stored in persistent storage. A member can combine its version chain vector with that of a partner by taking the union of the two vectors. The resulting version chain vector will also include the versions of files that the partner, and by transitivity, all its partners, have processed. The difference between the version chain vectors from two members determines a superset of the set of updates required to synchronize one member with the contents from the other member.

To enable replication across multiple **replicated folders**, clients and servers isolate all activity that belongs to one replicated folder in a **replication session**. Thus, DFS-R contains a separate layer for establishing replication activity for each replicated folder.

To summarize DFS-R at the level of detail described so far, the following sequence of activities occur for a client computer.

1. A client establishes a **connection** with a server.
2. For each (in parallel) replicated folder that is shared between the client and server, the client establishes a replication session.
3. For each replication session, the client requests the server version chain vectors.
4. When the client receives a version chain vector from the server, it calculates the versions that are not known to it and requests updates from the server pertaining to these versions.
5. The client processes updates from the server as it receives them. While processing a requested update, the client machine may decide that the server updates correspond to file content that it needs to retrieve. It then requests the file from the server.
6. The client registers a request for updated version chain vectors from the server when the client has received all updates from the previous version chain vector.

At a very high level, this sequence of events can be summarized as follows:

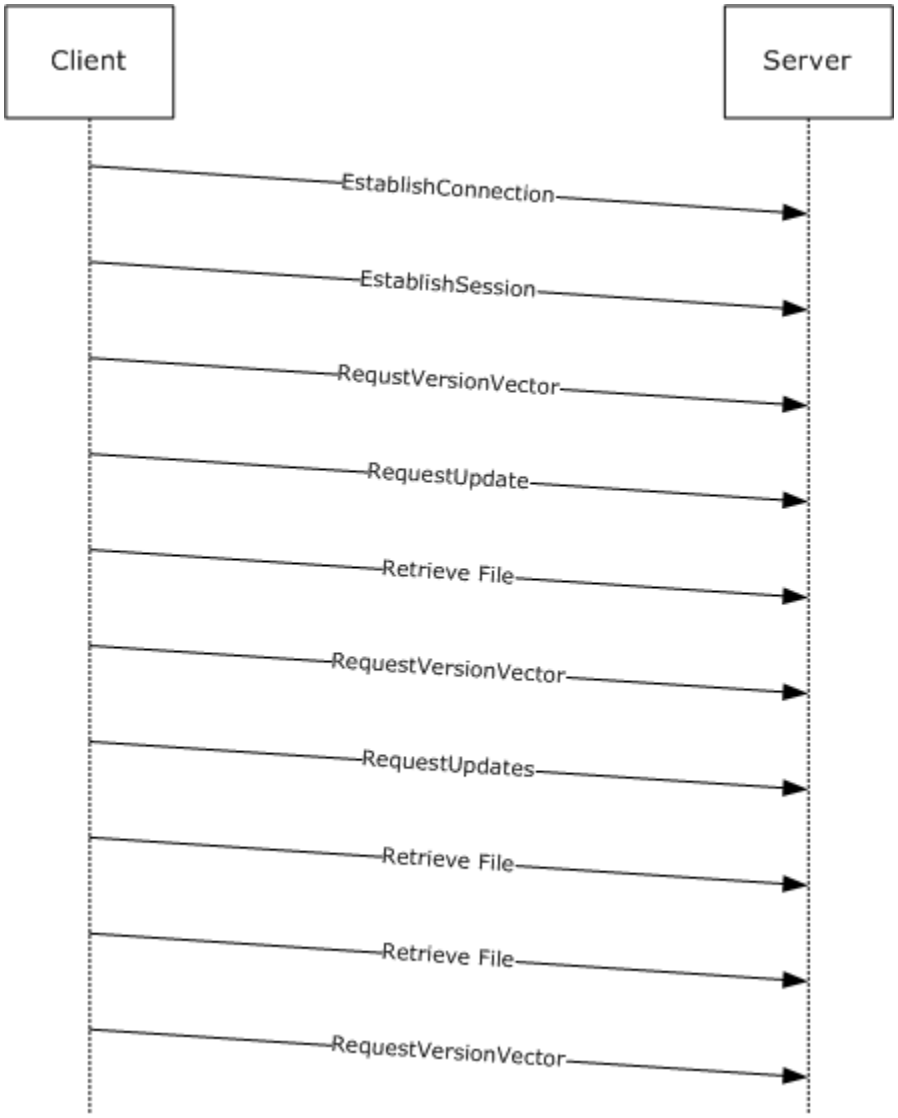



**Figure 1: DFS-R replication sequence**

Sections 2 and 3 specify DFS-R.

The detailed specification introduces several additional messages and layers. Most noteworthy are:

- The **RequestRecords** method is used for retrieving UID and GVSN pairs for each replicated file on the server. This method is used as part of a synchronization protocol (**Slow Sync**) that simply polls the entire content of the server's store of updates in order to synchronize. The Slow Sync protocol acts as an alternate protocol to the main synchronization protocol described in the beginning of this section.
- **RDC** is a file transfer protocol used for efficiently retrieving file data. For more information, see [MS-RDC].
- AsyncPoll is used for polling version chain vectors using a single pending asynchronous RPC call.

## 1.4 Relationship to Other Protocols

The Distributed File System: Replication (DFS-R) Protocol uses RPC, as specified in [C706] and [MS-RPCE], for all synchronization communication. DFS-R relies on authenticated, encrypted RPC traffic and therefore uses the NT LAN Manager (as specified in [MS-NLMP]) and Kerberos (as specified in [MS-KILE]) protocols, which are integral to [MS-RPCE]. It uses **AD** to manage configuration. It uses RDC to retrieve file data. Windows implementations of DFS-R also provide a WMI interface that is used for monitoring the state of a member. The WMI interface serves an additional role in versions of DFS-R on the Windows client, where it is used for injecting configurations.

## 1.5 Prerequisites/Preconditions

DFS-R obtains and uses its configuration from AD. AD supplies the **principal names** of the replication partners and **DFS-R** uses these trusted names for authenticating all replication traffic (which is over **RPC**). The principal names are given by the computer objects in AD. Section 3.1.1 specifies the configuration objects in AD that are used by DFS-R.

## 1.6 Applicability Statement

The Distributed File System: Replication (DFS-R) Protocol is used to replicate files in both AD and non-AD environments. Support for these scenarios differs depending on the operating system in use.

Windows Server 2003 R2 and Windows Server 2008: DFS-R is configured using AD objects. File replication proceeds between computers within the same forest whose principal names are maintained and authenticated by AD. File **ACLs** are replicated fully as participating computers are expected to use AD to identify ACLs.

Windows Vista Client: DFS-R is configured over the WMI management interface independently of AD and therefore works in environments that are not managed by AD. The peer-group authentication layer on RPC can be used for establishing authenticated and encrypted RPC channels. None of these deployments involve server communication and are therefore not expanded upon in this document.

## 1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- Supported Transports: The Distributed File System: Replication (DFS-R) Protocol is implemented on top of RPC over TCP/IP, as specified in section 2.1.
- Protocol Versions: DFS-R negotiates versioning as part of an RPC message; EstablishConnection is specified in section 3.2.5.1.2. This document specifies versions 0x00050000 and 0x00050001.
- Security and authentication methods: DFS-R supports the NT LAN Manager (as specified in [MS-NLMP]) and Kerberos (as specified in [MS-KILE]) authentication methods. These are specified in section 2.1.
- Localization: DFS-R does not expose any functionality that is localization dependent.
- Capability negotiation: DFS-R performs explicit capability negotiation as part of the protocol-version negotiation. Furthermore, on Windows, RDC similarity (as specified in [MS-RDC] section 3.1.5.4) is only enabled for Enterprise SKUs (Windows Server 2003, Enterprise Edition and Windows Server 2008 Enterprise), as specified in sections 2.2.1.2.1 and 3.2.5.1.2.

DFS-R registers itself with RPC using a single **UUID**, as specified in section 2.1. It always uses the same RPC Protocol version 1.0 and negotiates specific extensions using the custom protocol negotiation scheme that uses the method EstablishConnection (section 3.2.5.1.2) to establish the further set of methods that can be used between a DFS-R client and a server. All but two methods can be used in the current two existing protocol versions, 0x00050000 and 0x00050001. The methods applicable to both protocol versions are specified in sections 3.2.5.1.1 through 3.2.5.1.14, inclusive. The two methods that are specific to protocol version 0x00050001 are specified in sections 3.2.5.1.15 and 3.2.5.1.16. The capability of using similarity for speeding up downloads of RDC files can be controlled by using flags specified in section 2.2.1.2.1; the flags are communicated using the EstablishConnection method.

## 1.8 Vendor-Extensible Fields

DFS-R does not expose extensible fields.

## 1.9 Standards Assignments

There are no standards assignments associated with DFS-R.

# 2 Messages

## 2.1 Transport

DFS-R uses authenticated and encrypted RPC for all replication traffic. A client that wants to communicate with a DFS-R server MUST use a format that complies with an RPC marshaled form that uses the following qualifiers.

| UUID of IDL | Version of IDL |
|---|---|
| 897e2e5f-93f3-4376-9c9c-fd2277495c27 | 1.0 |

All traffic MUST be authenticated and encrypted using LAN Manager or Kerberos over TCP/IP, which requires that the client specify to use the **protocol sequence** associated with RPC over TCP/IP, and requires the client to specify packet privacy and authentication negotiation.

Both client and server MUST require authentication and encryption.

The following is a summary of the relevant parameters:

- Protocol sequence: Ncacn_ip_tcp
- DFSR_ENDPOINT_GUID: 5bc1ed07-f5f5-485f-9dfd-6fd0acf9a23c
- Authentication level: RPC_C_AUTHN_LEVEL_PKT_PRIVACY
- Authentication service (one of): RPC_C_AUTHN_GSS_NEGOTIATE, RPC_C_AUTHN_GSS_KERBEROS, or RPC_C_AUTHN_WINNT

A server MAY specify a static port for all DFS-R RPC traffic, or it MAY use **dynamic endpoints** and rely on the **endpoint** mapper to rely inbound requests that use the endpoint GUID to field requests into the DFS-R service.<1>

As part of mutual authentication, a client MUST furthermore specify its principal name when establishing a binding handle to allow a server to authenticate RPC calls. This part of the negotiation is handled opaquely by an RPC runtime that supports principal names, such as the Remote Procedure Call Extensions runtime. Recall that principal names are managed in **AD**.

### 2.1.1 Client Authentication Requirements

An implementation of the Distributed File System: Replication (DFS-R) Protocol MUST require the security provider used by RPC to mutually authenticate against the server.

The DFS-R client specifies that RPC mutually authenticate against the server. The client further specifies to RPC that the server may impersonate but may not delegate. For more information about QOS, see [MSDN-RPC].

It follows from the previous requirements that a client authentication call MUST use the following arguments when setting the authentication information on a binding handle.

- Authn level: RPC_C_AUTHN_LEVEL_PKT_PRIVACY
- Authn service: RPC_C_AUTHN_GSS_NEGOTIATE
- Authn identity: NULL

- Authorization service implemented by server: RPC_C_AUTHZ_NONE

### 2.1.2 Server Side Binding

As specified in section 2.1, the RPC server side of DFS-R uses DFSR_ENDPOINT_GUID with the RPC policy that specifies endpoint flags to "don't fail." The DFSR_ENDPOINT_GUID is used to ensure that the RPC run-time system can delegate incoming RPC calls to the correct executable. Also, as specified in section 2.1, a server MAY bind on a static port or MAY use the endpoint UUID to register a dynamic endpoint.<2>

## 2.2 Message Syntax

All multiple-byte integers represented in this document are in least-significant-byte-first order, called little-endian.

### 2.2.1 Common Data Types

#### 2.2.1.1 Constants

Most of the following constants are used to define the allowable limits of parameters in the structures and function arguments of the interface. In these cases, the RPC library directly enforces these limits.

Some definitions are to provide only a symbolic name to various constants.

##### 2.2.1.1.1 FRS_COMMUNICATION_PROTOCOL_VERSION

```
#define FRS_COMMUNICATION_PROTOCOL_VERSION_W2K3R2  0x00050000L
#define FRS_COMMUNICATION_PROTOCOL_VERSION_VISTA   0x00050001L
#define FRS_COMMUNICATION_PROTOCOL_VERSION_LONGHORN_SERVER 0x00050002L
```

These specify a set of predefined protocol versions.

##### 2.2.1.1.2 CONFIG_RDC_VERSION

```
#define CONFIG_RDC_VERSION              (1)
```

This indicates the major version of RDC. The major version increases when new features or capabilities are added. This version MUST be version 1.

##### 2.2.1.1.3 CONFIG_RDC_VERSION_COMPATIBLE

```
#define CONFIG_RDC_VERSION_COMPATIBLE   (1)
```

This indicates the minimum version of the RDC protocol that can work correctly with this version. The version MUST be 1.

#### 2.2.1.1.4 CONFIG_RDC_MAX_LEVELS

```
#define CONFIG_RDC_MAX_LEVELS           (8)
```

This indicates the maximum depth of **signature** generation and RDC recursion.

#### 2.2.1.1.5 CONFIG_RDC_MAX_NEEDLENGTH

```
#define CONFIG_RDC_MAX_NEEDLENGTH       (65536)
```

This indicates the maximum allowed length of an RDC need. An RDC need is an offset length-pair that prescribes a range of data that a client requests from the server.

#### 2.2.1.1.6 CONFIG_RDC_NEED_QUEUE_SIZE

```
#define CONFIG_RDC_NEED_QUEUE_SIZE      (20)
```

This indicates the maximum number of source needs that can be in a single request.

#### 2.2.1.1.7 CONFIG_RDC_HORIZONSIZE_MIN

```
#define CONFIG_RDC_HORIZONSIZE_MIN      (128)
```

This indicates the minimum allowable RDC horizon parameter used by RDC.

#### 2.2.1.1.8 CONFIG_RDC_HORIZONSIZE_MAX

```
#define CONFIG_RDC_HORIZONSIZE_MAX       (1024*16)
```

This indicates the maximum allowable RDC horizon parameter used by RDC.

#### 2.2.1.1.9 CONFIG_RDC_HASHWINDOWSIZE_MIN

```
#define CONFIG_RDC_HASHWINDOWSIZE_MIN   (2)
```

This indicates the minimum allowable RDC hash window-parameter used by RDC.

#### 2.2.1.1.10 CONFIG_RDC_HASHWINDOWSIZE_MAX

```
#define CONFIG_RDC_HASHWINDOWSIZE_MAX   (96)
```

This indicates the maximum allowable RDC hash window-parameter used by RDC.

#### 2.2.1.1.11 CONFIG_RDC_SIMILARITY_DATASIZE

```
#define CONFIG_RDC_SIMILARITY_DATASIZE  (16)
```

This indicates the size, in bytes, of the similarity data.

#### 2.2.1.1.12 CONFIG_TRANSPORT_MAX_BUFFER_SIZE

```
#define CONFIG_TRANSPORT_MAX_BUFFER_SIZE (262144)
```

This indicates the maximal buffer size allowed by a client for requesting file contents. When a client requests pieces of a file, such as in basic transfer of file contents and when requesting portions of a file or file metadata over RDC, it creates an RPC call with a buffer as an argument. The maximum allocated size of this buffer is bound by this constant.

#### 2.2.1.1.13 CONFIG_FILEHASH_DATASIZE

```
#define CONFIG_FILEHASH_DATASIZE        (20)
```

This indicates the size, in bytes, of the full file hash.

#### 2.2.1.1.14 FRS_UPDATE_FLAG_GHOSTED_HEADER

```
#define FRS_UPDATE_FLAG_GHOSTED_HEADER  (0x04)
```

The update request for **ghosted** header only. A ghosted header consists of the portion of a file data that excludes the main data stream. Section 3.2.5.1.14 specifies the required format of the data stream transmitted by DFS-R. In this context, [MS-BKUP] specifies the format of the backup data stream, which is part of the transmitted data stream. The main data stream comprises of bytes identified by the BACKUP_DATA stream ID, as specified in [MS-BKUP].

Notice that hexadecimal notation for flags is used. Flags can be combined using the bit-wise OR operation.

#### 2.2.1.1.15 FRS_UPDATE_FLAG_DATA

```
#define FRS_UPDATE_FLAG_DATA           (0x08)
```

The update request for file data only.

#### 2.2.1.1.16 TRUE

```
#define TRUE 1
```

The truth value TRUE. In DFS-R Booleans are of type long.

#### 2.2.1.1.17 FALSE

```
#define FALSE 0
```

The truth value FALSE.

#### 2.2.1.1.18 FRS_UPDATE_FLAG_CLOCK_DECREMENTED

```
#define FRS_UPDATE_FLAG_CLOCK_DECREMENTED  (0x10)
```

The update is the result of a dirty shutdown on the remote partner and the clock has been decremented by the remote partner. The client MAY assign a new GVSN when installing an update with that flag.

### 2.2.1.2 Enumerations

#### 2.2.1.2.1 TransportFlags

The **TransportFlags** enumerated type has only one flag defined, TRANSPORT_OS_ENTERPRISE.

```
typedef  enum
{
  TRANSPORT_OS_ENTERPRISE = 1
} TransportFlags;
```

**TRANSPORT_OS_ENTERPRISE:** This flag is set by the server when the server is capable of using similarity features of RDC.

#### 2.2.1.2.2 RDC_FILE_COMPRESSION_TYPES

The **RDC_FILE_COMPRESSION_TYPES** enumerated type identifies the data compression algorithm used for the file transfer.

```
typedef  enum
{
  RDC_UNCOMPRESSED = 1,
  RDC_XPRESS = 2
} RDC_FILE_COMPRESSION_TYPES;
```

**RDC_UNCOMPRESSED:** Data is not compressed. This value MUST be SENT whenever an RDC_FILE_COMPRESSION_TYPES enum value is required.

**RDC_XPRESS:** Not used.

#### 2.2.1.2.3 RDC_CHUNKER_ALGORITHM

The **RDC_CHUNKER_ALGORITHM** enumerated type identifies the RDC chunking algorithm used to generate the signatures for the file to be transferred.

```
typedef  enum
{
  RDC_FILTERGENERIC = 1,
  RDC_FILTERMAX = 2,
  RDC_FILTERPOINT = 3,
  RDC_MAXALGORITHM = 4
} RDC_CHUNKER_ALGORITHM;
```

**RDC_FILTERGENERIC:** Not used. MUST not be sent. If received, call MUST fail.

**RDC_FILTERMAX:** FilterMax chunker algorithm is used. This value MUST be sent whenever an RDC_CHUNKER_ALGORITHM enum value is required.

**RDC_FILTERPOINT:** Not used. MUST not be sent. If received, call MUST fail.

**RDC_MAXALGORITHM:** Not used. MUST not be sent. If received, call MUST fail.

#### 2.2.1.2.4 UPDATE_REQUEST_TYPE

The **UPDATE_REQUEST_TYPE** enumerated type specifies the type of updates for this request.

```
typedef  enum
{
  UPDATE_REQUEST_ALL = 1,
  UPDATE_REQUEST_TOMBSTONES = 2,
  UPDATE_REQUEST_LIVE = 3
} UPDATE_REQUEST_TYPE;
```

**UPDATE_REQUEST_ALL:** Request all updates that pertain to a version chain vector.

**UPDATE_REQUEST_TOMBSTONES:** Request only **tombstone** updates that pertain to a version chain vector.

**UPDATE_REQUEST_LIVE:** Request only non-tombstone updates that pertain to a version chain vector.

#### 2.2.1.2.5 UPDATE_STATUS

In response to a request for updates, a server MUST use a value of the **UPDATE_STATUS** enumerated type to specify whether it was able to send all updates that pertain to an argument version chain vector.

```
typedef  enum
{
  UPDATE_STATUS_WAIT = 1,
  UPDATE_STATUS_LIVE = 2,
  UPDATE_STATUS_DONE = 3,
  UPDATE_STATUS_MORE = 4
} UPDATE_STATUS;
```

**UPDATE_STATUS_WAIT:** To indicate to the client to not request further updates immediately.

**UPDATE_STATUS_LIVE:** To indicate to the client that there are potentially more non-tombstone updates from the argument version chain vector.

**UPDATE_STATUS_DONE:** There are no more updates that pertain to the argument version chain vector. In other words, the server does not have any updates whose versions belong to the version chain vector passed in by the client.

**UPDATE_STATUS_MORE:** There are potentially more updates (tombstone, if the client requested tombstones; live, if the client requested live) from the argument version chain vector.

#### 2.2.1.2.6 RECORDS_STATUS

The **RECORDS_STATUS** enumerated type is used for an output parameter of a Slow Sync request. It indicates whether the server has more records in the scope of the replicated folder over which Slow Sync is performed.

```
typedef  enum
{
  RECORDS_STATUS_DONE = 1,
  RECORDS_STATUS_MORE = 2
} RECORDS_STATUS;
```

**RECORDS_STATUS_DONE:** No more records are waiting to be transmitted on the server.

**RECORDS_STATUS_MORE:** More records are waiting to be transmitted on the server.

#### 2.2.1.2.7 VERSION_REQUEST_TYPE

The **VERSION_REQUEST_TYPE** enumerated value is used to indicate what role the client version vector request has.

```
typedef  enum
{
  REQUEST_NORMAL_SYNC = 1,
  REQUEST_SLOW_SYNC = 2,
  REQUEST_SLAVE_SYNC = 3
} VERSION_REQUEST_TYPE;
```

**REQUEST_NORMAL_SYNC:** Indicates that the client requests a version vector from the server for standard synchronization.

**REQUEST_SLOW_SYNC:** Indicates that the client requests a version vector from the server for Slow Sync.

**REQUEST_SLAVE_SYNC:** Indicates that the client requests a version vector from the server for **selective single master** mode.

#### 2.2.1.2.8 VERSION_CHANGE_TYPE

A clientversion vector request uses a value of **VERSION_CHANGE_TYPE** to indicate whether it is requesting a version chain vector change notification, a reduced increment, or a full version chain vector.

```
typedef  enum
{
  CHANGE_NOTIFY = 1,
  CHANGE_DELTA = 2,
  CHANGE_ALL = 3
} VERSION_CHANGE_TYPE;
```

**CHANGE_NOTIFY:** The client requests notification only for a change of the server'sversion chain vector.

**CHANGE_DELTA:** The client requests receiving only portions of a version vector that have changed relative to time stamp on the server.

**CHANGE_ALL:** The client requests to receive the full version vector of the server.

#### 2.2.1.2.9 FRS_REQUESTED_STAGING_POLICY

The **FRS_REQUESTED_STAGING_POLICY** enumerated type indicates the staging policy for the server to use.

```
typedef  enum
{
  SERVER_DEFAULT = 1,
  STAGING_REQUIRED = 2,
  RESTAGING_REQUIRED = 3
} FRS_REQUESTED_STAGING_POLICY;
```

**SERVER_DEFAULT:** The client indicates to the server that the server is free to use or bypass its cache.

**STAGING_REQUIRED:** The client indicates to the server to store the served content in its cache.

**RESTAGING_REQUIRED:** The client indicates to the server to purge existing content from its cache.

### 2.2.1.3 Simple Type Definitions

In addition to the types defined in this section, DFS-R also uses the DWORDLONG, ULONGLONG, and WCHAR type, as specified in [MS-DTYP].

#### 2.2.1.3.1 FRS_REPLICA_SET_ID

Unique identifier for a **replica set**.

This type is declared as follows:

```
typedef GUID FRS_REPLICA_SET_ID;
```

#### 2.2.1.3.2 FRS_CONTENT_SET_ID

Unique identifier for a **content set**.

This type is declared as follows:

```
typedef GUID FRS_CONTENT_SET_ID;
```

#### 2.2.1.3.3 FRS_DATABASE_ID

Unique identifier for a **DFS-R database**.

This type is declared as follows:

```
typedef GUID FRS_DATABASE_ID;
```

#### 2.2.1.3.4 FRS_MEMBER_ID

Unique identifier for a member.

This type is declared as follows:

```
typedef GUID FRS_MEMBER_ID;
```

#### 2.2.1.3.5 FRS_CONNECTION_ID

Unique identifier for a DFS-R Connection .

This type is declared as follows:

```
typedef GUID FRS_CONNECTION_ID;
```

#### 2.2.1.3.6 EPOQUE

The **EPOQUE** datatype is used only in the **FRS_EPOQUE_VECTOR (section 2.2.1.4.2)**. The **FRS_EPOQUE_VECTOR** is not used in the currently existing protocol versions 0x00050000, 0x00050001, and 0x00050002. However, proper **MIDL** marshaling of the parameters that are passed over the wire depends upon the type information provided by the MIDL. Therefore, these redundant type definitions are included here.

This type is declared as follows:

```
typedef SYSTEMTIME EPOQUE;
```

#### 2.2.1.3.7 BYTE_PIPE

A byte pipe, as defined by RPC.

This type is declared as follows:

```
typedef pipe byte BYTE_PIPE;
```

### 2.2.1.4 Aggregate Definitions

#### 2.2.1.4.1 FRS_VERSION_VECTOR

An entry of a version chain vector.

```
typedef struct _FRS_VERSION_VECTOR {
  GUID dbGuid;
  DWORDLONG low;
  DWORDLONG high;
} FRS_VERSION_VECTOR;
```

**dbGuid:** The **GUID** for the database originating the versions in the interval (low, high).

**low:** Lower bound for **VSN** interval.

**high:** Upper bound for VSN interval.

The number indicated by "low" is excluded from the version chain vector. The number indicated by "high" is included in the version chain vector. Thus, [low, high] indicates a half-open interval of unsigned integers. The GVSNs that are included in this entry are: { (dbGuid, low+1), …, (dbGuid, high) }.

#### 2.2.1.4.2 FRS_EPOQUE_VECTOR

An entry of an epoque vector.

```
typedef struct _FRS_EPOQUE_VECTOR {
  GUID machine;
  EPOQUE epoque;
} FRS_EPOQUE_VECTOR;
```

**machine:** Unused. MUST be 0. MUST be ignored on receipt.

**epoque:** Unused. MUST be 0. MUST be ignored on receipt.

Epoque vectors are attributes of the response payload, as specified in section 2.2.1.4.12. These attributes are not used in the currently existing protocol versions 0x00050000, 0x00050001, and 0x00050002. However, proper MIDL marshaling of the parameters that are passed over the wire depends on the type information provided by the MIDL. Therefore, these redundant type definitions are included here.

#### 2.2.1.4.3 FRS_ID_GVSN

A (UID, GVSN) pair.

```
typedef struct _FRS_ID_GVSN {
  GUID uidDbGuid;
  DWORDLONG uidVersion;
  GUID gvsnDbGuid;
  DWORDLONG gvsnVersion;
} FRS_ID_GVSN;
```

An **FRS_ID_GVSN** encodes a pair that consists of a UID and a GVSN. It is used as part of the messages for Slow Sync.

#### 2.2.1.4.4 FRS_UPDATE

A structure that contains file metadata related to a particular file being processed by DFS-R.

```
typedef struct _FRS_UPDATE {
  long present;
  long nameConflict;
  unsigned long attributes;
  FILETIME fence;
  FILETIME clock;
  FILETIME createTime;
  FRS_CONTENT_SET_ID contentSetId;
  unsigned char hash[CONFIG_FILEHASH_DATASIZE];
  unsigned char rdcSimilarity[CONFIG_RDC_SIMILARITY_DATASIZE];
  GUID uidDbGuid;
  DWORDLONG uidVersion;
  GUID gvsnDbGuid;
  DWORDLONG gvsnVersion;
  GUID parentDbGuid;
  DWORDLONG parentVersion;
  [string] WCHAR name[260+1];
  long flags;
} FRS_UPDATE;
```

**present:** Indicates if the file exists or has been deleted. The value MUST be either 0 or 1.

| Value | Meaning |
|---|---|
| 0x00000000 | File has been deleted. |
| 0x00000001 | File exists. |

**nameConflict:** Set if this update was tombstoned due to a name conflict. The value MUST be either 0 or 1. This field MUST be 0 if **present** is 1.

**attributes:** The file's attributes.

**fence:** The **fence** clock.

**clock:** Logical, last change clock.

**createTime:** File creation time.

**contentSetId:** The content set ID (replicated folder) that this file belongs to.

**hash:** The **SHA-1 hash** of the file.

**rdcSimilarity:** The similarity hash of the file. The value will be all zeros if the similarity data was not computed. See [MS-RDC].

**uidDbGuid:** The **GUID** portion of the file's UID. Same as the database **GUID** of the replicated folder where this file originated.

**uidVersion:** The version sequence number portion of the file's UID. This is assigned when the file is created.

**gvsnDbGuid:** The **GUID** portion of the file's GVSN. Same as the database **GUID** of the replicated folder where this file was last updated.

**gvsnVersion:** The version sequence number portion of the file's GVSN. This is assigned when the file was last updated.

**parentDbGuid:** The **GUID** portion of the UID of the file's parent. Same as the database **GUID** of the replicated folder where this file's parent originated.

**parentVersion:** The version sequence number portion of the UID of the file's parent. This is assigned when the parent of the file was created.

**name:** The file name, in UTF-16 form, of the file.

**flags:** Unused in version 0x00050000. MUST be 0. MUST be ignored on receipt. In version 0x00050001 or later, the value of this field may be some combination of 0 or more of the following flag bits: FRS_UPDATE_FLAG_GHOSTED_HEADER and FRS_UPDATE_FLAG_DATA.

#### 2.2.1.4.5 FRS_UPDATE_CANCEL_DATA

A structure that contains information about updates that were not processed by a client.

```
typedef struct _FRS_UPDATE_CANCEL_DATA {
  FRS_UPDATE blockingUpdate;
  FRS_CONTENT_SET_ID contentSetId;
  FRS_DATABASE_ID gvsnDatabaseId;
  FRS_DATABASE_ID uidDatabaseId;
  FRS_DATABASE_ID parentDatabaseId;
  DWORDLONG gvsnVersion;
  DWORDLONG uidVersion;
  DWORDLONG parentVersion;
  unsigned long cancelType;
  long isUidValid;
  long isParentUidValid;
  long isBlockerValid;
} FRS_UPDATE_CANCEL_DATA;
```

**blockingUpdate:** The update that could not be processed by client.

**contentSetId:** The content set where the blocking update resides.

**gvsnDatabaseId:** The **GUID** part of the GVSN of the update that could not be processed.

**uidDatabaseId:** The **GUID** part of the UID from the update that could not be processed.

**parentDatabaseId:** The **GUID** part of the parent UID from the update that could not be processed.

**gvsnVersion:** The VSN part of the GVSN of the update that could not be processed.

**uidVersion:** The VSN part of the UID from the update that could not be processed.

**parentVersion:** The VSN part of the parent UID from the update that could not be processed.

**cancelType:** The cause for canceling the processing of the update. These MUST be set to one of the following values.

| Value | Meaning |
|---|---|
| UNSPECIFIED<br>0x00000001 | No reason is indicated by the client. The GVSN and UID MUST indicate which update was not processed by the client. |
| PARENT_IS_TOMBSTONE<br>0x00000002 | The **client** holds a tombstone at the location where an update inserts a child. The value of **blockingUpdate** is set to the record associated with the parent; the parent field is ignored. |
| CHILD_IS_LIVE<br>0x00000003 | The update is for a tombstone. The client holds a child, whose version is known by the server. The client cannot proceed until the child has been moved. The **blockingUpdate** field is set to the child record. |
| PARENT_IS_MISSING<br>0x00000004 | The client does not have the parent for the (live) update that was sent. The server should send the parent before proceeding. The **parent** field (**parentGuid** and **parentVersion**) are set to the missing parent. |

**isUidValid:** If set, indicates that the **UID** field is populated.

**isParentUidValid:** If set, indicates that the **parent** field is populated.

**isBlockerValid:** If set, indicates that the **blockingUpdate** field is populated with valid data.

### 2.2.1.4.6 FRS_RDC_SOURCE_NEED

A file range specification for RDC downloads.

```
typedef struct _FRS_RDC_SOURCE_NEED {
  ULONGLONG needOffset;
  ULONGLONG needSize;
} FRS_RDC_SOURCE_NEED;
```

**needOffset:** The offset in the marshaled source file.

**needSize:** The number of bytes (uncompressed) to retrieve.

The client uses this structure to request source data from the server when downloading a file with RDC.

#### 2.2.1.4.7 FRS_RDC_PARAMETERS_FILTERMAX

Configuration parameters for the FilterMax RDC algorithm.

```
typedef struct _FRS_RDC_PARAMETERS_FILTERMAX {
  [range(CONFIG_RDC_HORIZONSIZE_MIN, CONFIG_RDC_HORIZONSIZE_MAX)]
    unsigned short horizonSize;
  [range(CONFIG_RDC_HASHWINDOWSIZE_MIN, CONFIG_RDC_HASHWINDOWSIZE_MAX)]
    unsigned short windowSize;
} FRS_RDC_PARAMETERS_FILTERMAX;
```

**horizonSize:** See [MS-RDC] section 4.3 for the definition of the *horizonSize* parameter of the FilterMax algorithm.

**windowSize:** See [MS-RDC] section 4.3 for the definition of the hash *window size* parameter of the FilterMax algorithm.

#### 2.2.1.4.8 FRS_RDC_PARAMETERS_FILTERPOINT

Configuration for the FilterPoint RDC algorithm. This algorithm and its configuration parameters are not used.

```
typedef struct _FRS_RDC_PARAMETERS_FILTERPOINT {
  unsigned short minChunkSize;
  unsigned short maxChunkSize;
} FRS_RDC_PARAMETERS_FILTERPOINT;
```

**minChunkSize:** Unused. MUST be 0 and MUST be ignored on receipt.

**maxChunkSize:** Unused. MUST be 0 and MUST be ignored on receipt.

#### 2.2.1.4.9 FRS_RDC_PARAMETERS_GENERIC

Blob for alternate RDC algorithms.

```
typedef struct _FRS_RDC_PARAMETERS_GENERIC {
  unsigned short chunkerType;
  byte chunkerParameters[64];
} FRS_RDC_PARAMETERS_GENERIC;
```

**chunkerType:** The **chunkerType** MUST be RDC_FILTERMAX.

**chunkerParameters:** Not used. This is a generic parameter block, which allows for space in future protocol versions.

#### 2.2.1.4.10 FRS_RDC_PARAMETERS

Union of RDC algorithm options.

```
typedef struct {
```

```
  unsigned short rdcChunkerAlgorithm;
  [switch_is(rdcChunkerAlgorithm)]
    union {
    [case(RDC_FILTERGENERIC)]
      FRS_RDC_PARAMETERS_GENERIC filterGeneric;
    [case(RDC_FILTERMAX)]
      FRS_RDC_PARAMETERS_FILTERMAX filterMax;
    [case(RDC_FILTERPOINT)]
      FRS_RDC_PARAMETERS_FILTERPOINT filterPoint;
  } u;
} FRS_RDC_PARAMETERS;
```

**rdcChunkerAlgorithm:** SHOULD be RDC_FILTERMAX, for compatibility with Windows. If the server receives an unrecognized value for this field, the server MUST generate an error.<3>

**filterGeneric:** Placeholder only to fill out the enumeration. Never used, because **rdcChunkerAlgorithm** MUST not have this value.

**filterMax:** The parameters, as specified in [MS-RDC], necessary for the FilterMax algorithm.

**filterPoint:** Never used because **rdcChunkerAlgorithm** MUST not have this value.

The server returns an array of these structures, one each for each level of RDC signatures that are available. The client uses these parameters to ensure that the local signatures match the signatures that will be returned from the server.

#### 2.2.1.4.11 FRS_RDC_FILEINFO

File information specific to RDC downloads.

```
typedef struct _FRS_RDC_FILEINFO {
  DWORDLONG onDiskFileSize;
  DWORDLONG fileSizeEstimate;
  unsigned short rdcVersion;
  unsigned short rdcMinimumCompatibleVersion;
  [range(0, CONFIG_RDC_MAX_LEVELS)]
    byte rdcSignatureLevels;
  RDC_FILE_COMPRESSION_TYPES compressionAlgorithm;
  [size_is(rdcSignatureLevels)] FRS_RDC_PARAMETERS rdcFilterParameters[];
} FRS_RDC_FILEINFO;
```

**onDiskFileSize:** An estimate for the on-disk, compressed, marshaled source file. The server SHOULD make this estimate as accurate as possible, but the protocol does not require that it be exact.

**fileSizeEstimate:** An estimate for the on-disk, uncompressed, unmarshaled source file. The server SHOULD make this estimate as accurate as possible, but the protocol does not require that it be exact.

**rdcVersion:** The current RDC version. It MUST be 1.

**rdcMinimumCompatibleVersion:** The minimum version of the client-side RDC that is compatible with the server-side RDC (rdcVersion). It MUST be 1.

**rdcSignatureLevels:** The depth of the RDC signatures that are available for the client to retrieve. The server MUST allow the client to get signatures at least to this depth (using RdcGetSignatures (section 3.2.5.1.10)).

The server MAY set this value to 0, indicating a non-RDC file transfer is required.

**compressionAlgorithm:** This field MUST be set to RDC_UNCOMPRESSED and MUST be ignored on receipt. Despite the name of this field, data compression is always used as specified in section 3.2.5.1.14.

**rdcFilterParameters:** The array of RDC chunker parameters used, one each for the levels of RDC signatures that are available.<4>

### 2.2.1.4.12 FRS_ASYNC_VERSION_VECTOR_RESPONSE

Version chain vector response payload.

```
typedef struct _FRS_ASYNC_VERSION_VECTOR_RESPONSE {
  ULONGLONG vvGeneration;
  unsigned long versionVectorCount;
  [size_is(versionVectorCount)] FRS_VERSION_VECTOR* versionVector;
  unsigned long epoqueVectorCount;
  [size_is(epoqueVectorCount)] FRS_EPOQUE_VECTOR* epoqueVector;
} FRS_ASYNC_VERSION_VECTOR_RESPONSE;
```

**vvGeneration:** The time stamp associated with the version chain vector on the server. The time stamp is incremented every time a server updates its version chain vector. This gives a way to track whether a client has the newest version of the version chain vector known to the server.

**versionVectorCount:** Number of elements in the **versionVector** array.

**versionVector:** An array of **FRS_VERSION_VECTOR** triples.

**epoqueVectorCount:** Number of elements in the epoque vector array.

**epoqueVector:** An array of FRS_EPOQUE vector pairs.

### 2.2.1.4.13 FRS_ASYNC_RESPONSE_CONTEXT

Version chain vector response payload envelope.

```
typedef struct _FRS_ASYNC_RESPONSE_CONTEXT {
  unsigned long sequenceNumber;
  DWORD status;
  FRS_ASYNC_VERSION_VECTOR_RESPONSE result;
} FRS_ASYNC_RESPONSE_CONTEXT;
```

**sequenceNumber:** Sequence number that associates the response context with a version vector request.

**status:** Error/success status of version vector request. The error code can be any of the values specified in section 2.2.1.5.

**result:** Response payload, comprising a version chain vector.

#### 2.2.1.4.14 PFRS_SERVER_CONTEXT

Context handle that represents the requested file replication operation.

This type is declared as follows:

```
typedef [context_handle] void* PFRS_SERVER_CONTEXT;
```

### 2.2.1.5 Error Codes

| Error code | Meaning |
|---|---|
| ERROR_ACCESS_DENIED<br>0x00000005 | The client does not have rights to access the file. |
| ERROR_SHARING_VIOLATION<br>0x00000020 | The file is already in use by another client. |
| ERROR_INVALID_PARAMETER<br>0x00000057 | The client provided an invalid parameter. |
| ERROR_BUSY<br>0x000000AA | The server is too busy to service the request. |
| FRS_ERROR_CONNECTION_INVALID<br>0x0x00002345 | The connection is invalid. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND<br>0x00002329 | The content set was not found. |
| FRS_ERROR_PARTNER_GUID_NOT_FOUND<br>0x0000232A | The file has changed. |
| FRS_ERROR_WRONG_REPLICA_SET<br>0x0000232B | The session is invalid. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE<br>0x00002340 | Unknown error in RDC. |
| FRS_ERROR_CONNECTION_INVALID<br>0x00002342 | Paused for backup or restore. |
| FRS_ERROR_CONTENTSET_NOT_FOUND<br>0x00002344 | RDC is not used. |
| FRS_ERROR_FILE_HAS_CHANGED<br>0x00002345 | The download is redundant. |
| FRS_ERROR_SESSION_INVALID | The compressed data is invalid. |

| Error code | Meaning |
|---|---|
| 0x00002347 | |
| FRS_ERROR_RDC_GENERIC<br>0x0000234B | The version is incompatible. |
| FRS_ERROR_IN_BACKUP_RESTORE<br>0x0000234C | The content set is not ready. |
| FRS_ERROR_NO_RDC<br>0x00002356 | The content set is disabled. |
| FRS_ERROR_DOWNLOAD_REDUNDANT<br>0x00002357 | The member is invalid. |
| FRS_ERROR_XPRESS_INVALID_DATA<br>0x00002358 | The update does not exist on upstream. |
| FRS_ERROR_INCOMPATIBLE_VERSION<br>0x0000235A | The client's protocol version is incompatible with the server's. |
| FRS_ERROR_RESOURCE_IS_GHOSTED<br>0x00002457 | The server does not possess the full file contents for the requested file. |

### 2.2.2 Configuration Objects in AD

DFS-R relies on global configuration information (stored in Active Directory) for proper functioning.

These objects prescribe configuration information. In particular:

- **replicaSetId:** The **GUID** of replication groups. They are configured as the **GUID** of an object under the path msDFSR-GlobalSettings/msDFSR-ReplicationGroup.
- **connectionId:** The **GUID** of connections. They are configured as the **GUID** of an object under the path msDFSR-GlobalSettings/msDFSR-ReplicationGroup/msDFSR-Member/msDFSR-Connection.
- **contentSetId:** The **GUID** of a replicated folder. They are configured as the **GUID** of an object under msDFSR-GlobalSettings/msDFSR-ReplicationGroup/msDFSR-Content/msDFSR-ContentSet.
- **Principal names:** The principal authenticated computer names. Computer objects form the basis of an AD configuration.

These are used in the RPC messages and MUST be known to both server and client in order for partners to establish trust, communication, and which folders are replicated among them. This section summarizes the set of configuration parameters that are used in AD to configure DFS-R.

An example object layout is illustrated in section 4.3.1.

#### 2.2.2.1 msDFSR-LocalSettings

This object encapsulates all DFS-R local settings on the local machine. The DFS LocalSettings object is found in the context of each computer object:

" CN=DFSR LocalSettings, CN=<computer name>, CN=Computers"

Attributes of this object that are relevant for DFS-R are:

**msDFSR-Version:** Optional, implementation defined.<5>

Exactly one top-level DFS-R LocalSettings object MUST exist for each computer that is configured for replication. Each of these can contain one or more subscriber objects.

ACLs MAY be set on this msDFSR-LocalSettings and inherited to children.

The schema definition for this object is provided by the msDFSR-LocalSettings class definition, as specified in [MS-ADSC].<6>

#### 2.2.2.2 msDFSR-Subscriber

Objects of this class exist under the msDFSR-LocalSettings object and imply that this computer subscribes to a certain **replication group**.

The following attribute of this object is relevant to DFS-R:

- **msDFSR-MemberReference:** Forward link to msDFSR-Member object in msDFSR—Settings tree. MUST exist.

Each msDFSR-Subscriber object can contain a reference to one or more msDFSR-Subscription objects. At most, one msDFSR-Member object MUST be referenced from an msDFSR-Subscriber object.

The schema definition for this object is as specified by the msDFSR-Subscriber class definition in [MS-ADTS].

#### 2.2.2.3 msDFSR-Subscription

Each subscription object represents a replicated folder in the replication group that its parent subscribes to. Objects of this class exist on the msDFSR-Subscriber objects and imply that the computer uses the **topology** specified by the msDFSR-Subscriber object to replicate the folder specified by the attributes of the msDFSR-Subscription object.

The following attributes of this object are relevant to DFS-R:

- **msDFSR-ContentSetGUID:** ContentSet object GUID. MUST exist.
- **msDFSR-RootPath:** Full path of the replicated folder root directory. MUST exist.
- **msDFSR-RootFence:** Time stamp.
- **msDFSR-StagingPath:** Full path of the replicated folder staging directory.
- **msDFSR-StagingSizeInMb:** The maximum size of the staging directory in MB. Optional.
- **msDFSR-ConflictPath:** Full path of the folder used to store files for which there are replication conflicts.
- **msDFSR-ConflictSizeInMB:** The maximum size of the conflicts folder in MB.
- **msDFSR-Enabled:** Enable and disable replicated folder without removing it from the system.
- **msDFSR-Options:** Bit Flags to control optional behavior. The following bits are used:
  - 0x1: Set if the replicated folder is designated as primary.

- All other bits are ignored and SHOULD be set to 0.

- **msDFSR-Extension:** In version 0x00050000 of the Distributed File System: Replication (DFS-R) Protocol, MUST be set to 0 and MUST be ignored on receipt. In version 0x00050001 or later of DFS-R, contains a string that specifies a file name pattern to be used to exclude some files from data compression.<7>

Each msDFSR-Subscription object MUST contain a reference to one msDFSR-Content object. At most, one msDFSR-Content object MUST be referenced from an msDFSR-Subscription object.

The schema definition for this object is as specified by the msDFSR-Subscription class definition in [MS-ADSC].

#### 2.2.2.4 msDFSR-GlobalSettings

Replication topology configurations are grouped under the msDFSR-GlobalSettings object.

The top-level DFS-R global settings object is found in the domain naming context at the following RDN with each domain's AD domainDNS [MS-ADSC] object:

" CN=DFS GlobalSettings, CN=System"

There are no attributes in this container that are significant to DFS-R.

ACLs MAY be set on msDFSR-GlobalSettings and inherited on child objects. There MUST be at exactly one msDFSR-GlobalSettings object for every domain where DFS-R is configured for replication.

The schema definition for this object is provided by the msDFSR-GlobalSettings class definition in [MS-ADSC] section 1.97.

#### 2.2.2.5 msDFSR-ReplicationGroup

Container for content and topology objects. It is found under the msDFSR-GlobalSettings object.

The following attributes of this object are relevant to DFS-R:

- **msDFSR-ReplicationGroupType:** The replication group type; Default is Other=3 (SYSVOL=0, PROTECTION=1, DISTRIBUTION=2, OTHER=3). MUST exist.

- **msDFSR-TombstoneExpiryInMin:** The replication group or replicated folder tombstones expiration in minutes.

- **msDFSR-Schedule:** Replication schedule consisting of time intervals where replication is enabled and which contains the bandwidth throttling settings.

- **objectGUID:** The replication group GUID. This corresponds directly to the ID GUID that appears in the RPC interface. MUST exist.

- **msDFSR-Options:** Bit flags to control certain behavior. The following bits are used:

  - 0x1: Controls how the schedule is interpreted. If 0, the schedule is interpreted in UTC time zone. If 0, the schedule is interpreted in the local time zone.

  - All other bits are ignored and SHOULD be set to 0.

- **nTSecurityDescriptor:** To be used in an implementation-dependent manner.<8>

Each msDFSR-ReplicationGroup MUST appear as a reference under msDFSR-GlobalSettings. It MUST contain an msDFSR-Content and an msDFSR-Topology reference. There MUST be at most one msDFSR-Content and at most one msDFSR-Topology child object under each msDFSR-ReplicationGroup object.

The schema definition for this object is as specified by the msDFSR-ReplicationGroup class definition in [MS-ADSC] section 1.100.

#### 2.2.2.6 msDFSR-Content

Container of replication group replicated folder objects.

There are no attributes in this container that are relevant to DFS-R.

Each msDFSR-Content MAY contain references to one or more msDFSR-ContentSet objects. ACLs MAY be set on msDFSR-Content and inherited on child objects.

The schema definition for this object is as specified by the msDFSR-Content class definition in [MS-ADSC] section 1.95.

#### 2.2.2.7 msDFSR-ContentSet

Subscriber objects on multiple machines link to the same msDFSR-ContentSet, machine-specific attributes that are stored in the subscriber object as outlined before. The msDFSR-ContentSet object, however, stores the global attributes (policies) that are shared for all subscriber machines.

Attributes of this object are:

- **description:** A string to be used in an implementation-dependent manner.<9>
- **msDFSR-FileFilter:** A comma-separated list of 0 or more wildcard file name filters for the **replica set**. Any file whose name matches any of the filters SHOULD be excluded from replication. The value of this attribute SHOULD contain, at a minimum, "*.tmp,*.bak,~*".<10>
- **msDFSR-DirectoryFilter:** A comma-separated list of 0 or more wildcard folder name filters for the replica set. Any folder whose name matches any of the filters SHOULD be excluded from replication.

The schema definition for this object is as specified by the msDFSR-ContentSet class definition in [MS-ADSC] section 1.96.

#### 2.2.2.8 msDFSR-Topology

Container for all topology objects, namely, members and connections.

There are no relevant attributes of the msDFSR-Topology object, it is only relevant as a container for msDFSR-Member objects.

Each msDFSR-Topology MAY contain references to one or more msDFSR-Member objects.

The schema definition for this object is as specified by the msDFSR-Topology class definition in [MS-ADSC] section 1.103.

#### 2.2.2.9 msDFSR-Member

Member objects represent a computer object in a replication group. Each computer that participates in a replication group MUST have one corresponding member object.

The following attributes of this object are relevant to DFS-R:

- **objectGUID:** The member GUID. This corresponds directly to the member GUID that appears in the RPC interface. MUST exist.
- **msDFSR-ComputerReference:** Distinguished name (DN) of the computer object associated with this member object. MUST exist.
- **msDFSR-Keywords:** A string to be used in an implementation-dependent manner.<11>

Each msDFSR-Member object contains 0 or more msDFSR-Connection objects. An msDFSR-Member object MAY contain more then one msDFSR-Connection object with the same partner.<12>

The schema definition for this object is as specified by the msDFSR-Member class definition in [MS-ADSC] section 1.99.

#### 2.2.2.10 Computer

The following attribute of this object is relevant to DFS-R:

- **DNSHostName:** String that specifies the DNS name of the partner. The format of a DNS name follows the format of DNS names used in URIs. This format is specified in [RFC3986].

#### 2.2.2.11 msDFSR-Connection

Each object of this class represents a directional connection between two machines only.

The following attributes of this object are relevant to DFS-R:

- **FromServer:** DN of the inbound partner (other msDFSR-Member object). MUST exist.
- **msDFSR-Enabled:** Boolean that specifies whether this connection is enabled or disabled.
- **msDFSR-Schedule:** A binary attribute that contains the replication schedule, as defined below.
- **msDFSR-Keywords:** A string to be used in an implementation-dependent manner.<13>
- **msDFSR-RdcEnabled:** Enable or disable RDC transfers on a connection.
- **msDFSR-RdcMinFileSizeInKB:** Minimum size threshold for enabling RDC transfers.
- **msDFSR-Options:** Bit Flags to control certain behavior. The following bits are used:
  - 0x1: Controls how the schedule is interpreted. If 0, the schedule is interpreted in UTC time zone. If 0, the schedule is interpreted in the local time zone.
  - All other bits are ignored and SHOULD be set to 0.

Each msDFSR-Connection object MUST NOT use the same msDFSR-Member object as the client and server.

The schema definition for this object is as specified by the msDFSR-Connection class definition in [MS-ADSC] section 1.94.

A Schedule is a binary attribute of size 336 bytes (2*24*7) that represents a schedule for a week. Each hour is represented as a 16-bit integer. Every hour is divided into four quarters, each of which occupies 4 bits. A schedule starts out specifying the time windows, starting from Sunday midnight in either UTC or system local time.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 30 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sun 00:00 | | | | | | | | | | | | | | | | Sun 01:00 | | | | | | | | | | | | | | | |
| 02:00 | | | | 02:15 | | | | 02:30 | | | | | | 02:45 | | Sun 03:00 | | | | | | | | | | | | | | | |
| Sun 04:00 | | | | | | | | | | | | | | | | Sun 05:00 | | | | | | | | | | | | | | | |
| Sun 06:00 | | | | | | | | | | | | | | | | Sun 07:00 | | | | | | | | | | | | | | | |
| Sun 08:00 | | | | | | | | | | | | | | | | Sun 09:00 | | | | | | | | | | | | | | | |
| Sun 10:00 | | | | | | | | | | | | | | | | Sun 11:00 | | | | | | | | | | | | | | | |
| Sun 12:00 | | | | | | | | | | | | | | | | Sun 13:00 | | | | | | | | | | | | | | | |
| Sun 14:00 | | | | | | | | | | | | | | | | Sun 15:00 | | | | | | | | | | | | | | | |
| Sun 16:00 | | | | | | | | | | | | | | | | Sun 17:00 | | | | | | | | | | | | | | | |
| Sun 18:00 | | | | | | | | | | | | | | | | Sun 19:00 | | | | | | | | | | | | | | | |
| Sun 20:00 | | | | | | | | | | | | | | | | Sun 21:00 | | | | | | | | | | | | | | | |
| Sun 22:00 | | | | | | | | | | | | | | | | Sun 23:00 | | | | | | | | | | | | | | | |
| Mon 00:00 | | | | | | | | | | | | | | | | … | | | | | | | | | | | | | | | |

The 4 bits of a quarter takes 16 values ranges from 0x0 to 0xF. A value of 0 indicates the schedule is off. A value of 0xF indicates an on schedule with full bandwidth. The levels in between are used in an implementation-defined way.<14>

# 3 Protocol Details

## 3.1 Common Details

The following figure illustrates the main components of the Distributed File System: Replication (DFS-R) Protocol synchronization core.

The left side of the figure summarizes the client state machines and the right side illustrates the server actions. The arrows departing the server are matched with corresponding arrows entering client, and the arrows departing the client are matched with the corresponding arrows entering the server.

In the middle, file creations, changes, and deletions are picked up from the file system and inserted into a database. Changes to the database trigger version chain vector change notifications that are managed by a "Join" module depicted on the top right. In response, the client, which maintains an inbound connection (InConnection, for short), engages in a sequence of requests for further version chain vectors, updates, and file contents.

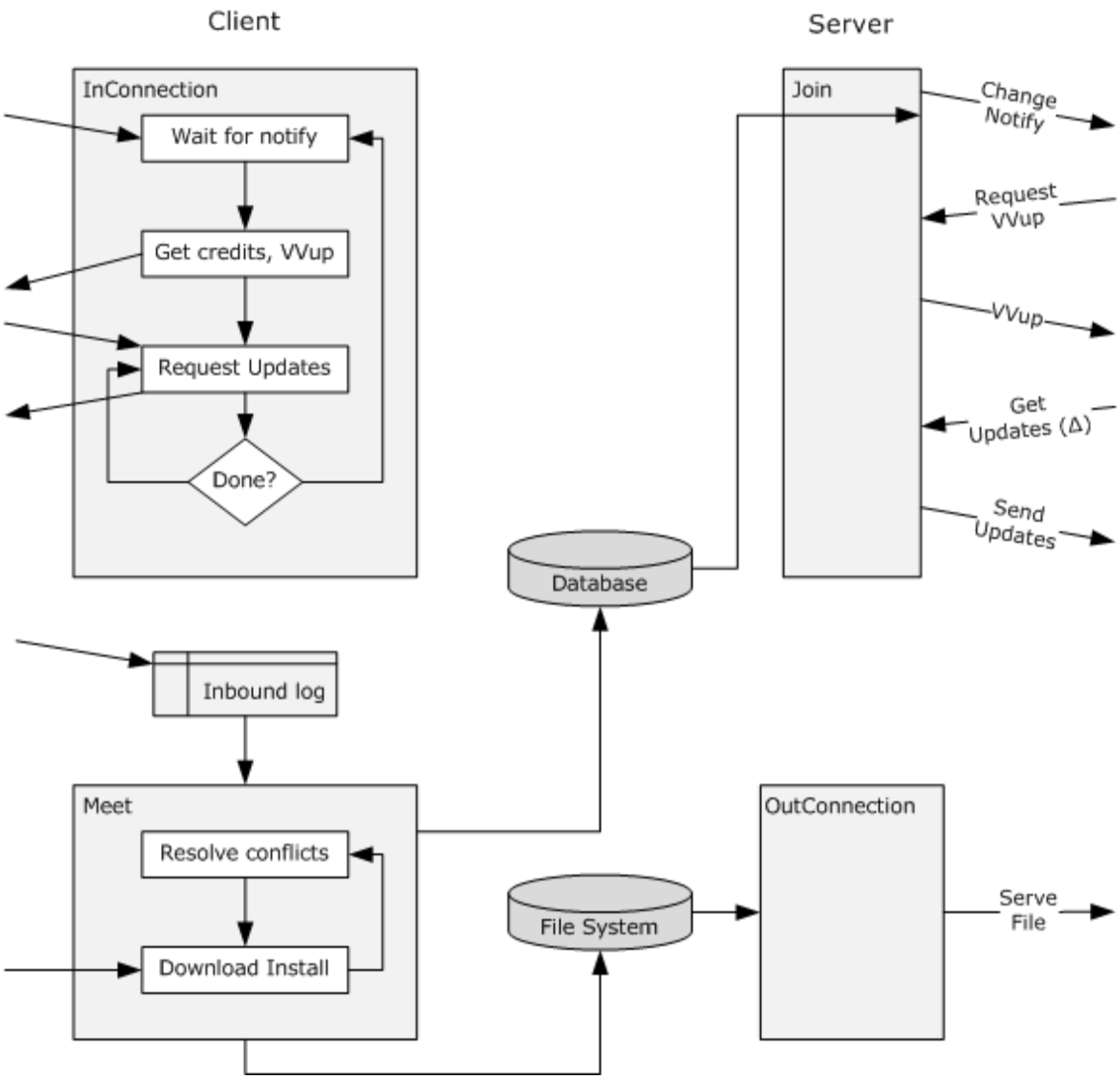



**Figure 2: Main components of the DFS-R synchronized core**

The main protocol of DFS-R is initiated by the client, or client, and is used to transfer metadata and data from the server (upstream partner) to the client. DFS-R may be configured to replicate in both directions, in which case there are two separate instances of the protocol operating between a pair of machines. Each machine, then, is both a client and a server.

### 3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

The overview, as specified in section 3.1, indicates a possible organization. We elaborate slightly on these here as they may help in understanding the Distributed File System: Replication (DFS-R) Protocol in context. As should be clear, an implementation is in no way bound by this organization.

**InConnection:** A logical connection object maintained by **client** to group state pertaining to a configured connection with the **server**.

**OutConnection:** A logical connection object maintained by the **server**.

**File System:** The store where replicated files reside and are changed. The store maintains file data and organizes the data in a way that is specific to the semantics of the store.

**File Data:** Data stream of the replicated content.

**Database:** A store that holds the metadata of replicated files, including updates and version chain vectors.

**Note** The abstract data model can be implemented in a variety of ways. This protocol does not prescribe or advocate any specific implementation technique.

### 3.1.2 Timers

DFS-R is primarily an event-driven protocol. Actions are taken in response to external stimulus, such as changes in the file system. This implementation contains a few implementation-specific timers. These timers are summarized separately for the client and server behaviors in sections 3.3.2 and 3.2.2.

### 3.1.3 Initialization

Both clients and servers initialize by obtaining their configuration. Sections 3.3.3 and 3.2.3 specify client- and server-specific initialization.

### 3.1.4 Higher-Layer Triggered Events

Configuration changes: When a configuration change is injected into DFS-R, clients SHOULD establish or terminate logical connection objects that have changed. Clients may delay establishing a connection based on a configured schedule. Similarly, DFS-R servers MUST terminate logical connections that have been disabled or removed from the updated configuration. If a client performs an RPC call on a connection that has not been configured on a server, a server MUST fail the RPC call with a status ERROR_ACCESS_DENIED.<15>

### 3.1.5 Message Processing Events and Sequencing Rules

Section 3.3.1 summarizes the client state machine for synchronization. Section 3.2.2 summarizes the corresponding server sequencing.

### 3.1.6 Timer Events

Timer events are documented separately for clients and servers.

### 3.1.7 Other Local Events

<16><17>

## 3.2 Server Details

### 3.2.1 Abstract Data Model

**OutConnection:** A logical connection object maintained by the server. The connection object is created when a client issues an EstablishConnection call. All further calls are made in the context of a logical connection object. The DFS-R server MUST maintain at most, one logical connection object per configured connection; therefore asynchronous calls made in the context of old logical connection objects MUST be completed with an error that indicates that the call could not be completed (ERROR_ACCESS_DENIED).

### 3.2.2 Timers

The DFS-R protocol does not rely on timers. However, an implementation, MAY choose to introduce timers to reclaim resources in the context of an unreliable environment. A client that interacts with a Windows implementation of DFS-R MAY take the following timers into account:<18><19>

### 3.2.3 Initialization

None. Servers await connection attempts from configured clients.

### 3.2.4 Higher-Layer Triggered Events

None.

### 3.2.5 Message Processing Events and Sequencing Rules

#### 3.2.5.1 FrsTransport Methods

To receive incoming remote calls for the **FrsTransport** interface, the server MUST implement a DCOM object using the UUID (universally unique identifier) {897e2e5f-93f3-4376-9c9c-fd2277495c27}.

Methods in RPC Opnum Order

| Method | Description |
|---|---|
| CheckConnectivity | Called by a client to check whether the server is reachable and has been configured to replicate with the server.<br>Opnum: 0 |
| EstablishConnection | Establishes a **logical connection** from a client to a server.<br>Opnum: 1 |
| EstablishSession | Establishes a logical relationship on the server for a replicated folder.<br>Opnum: 2 |
| RequestUpdates | Obtains file metadata in the form of updates from a server.<br>Opnum: 3 |
| RequestVersionVector | Obtains the version chain vector persisted on a server.<br>Opnum: 4 |
| AsyncPoll | Registers an asynchronous callback for a server to provide version chain |

| Method | Description |
|---|---|
| | vectors.<br>Opnum: 5 |
| **RequestRecords** | Retrieves UIDs and GVSNs that a server **persists**.<br>Opnum: 6 |
| **UpdateCancel** | Used by a client to indicate to a server that it could not process an update.<br>Opnum: 7 |
| **RawGetFileData** | Transfers successive segments from a file.<br>Opnum: 8 |
| **RdcGetSignatures** | Obtains RDC signature data from a server.<br>Opnum: 9 |
| **RdcPushSourceNeeds** | Registers requests for file ranges on a server.<br>Opnum: 10 |
| **RdcGetFileData** | Obtains file ranges whose requests have previously been registered on a server.<br>Opnum: 11 |
| **RdcClose** | Informs the server that the server context information can be released.<br>Opnum: 12 |
| **InitializeFileTransferAsync** | Used by a client to start a file download.<br>Opnum: 13 |
| **InitializeFileDataTransfer** | Used by a client to initialize a partial file transfer.<br>Opnum: 14 |
| **RawGetFileDataAsync** | Used instead of calling **RawGetFileData** multiple times to obtain file data.<br>Opnum: 15 |
| **RdcGetFileDataAsync** | Used instead of calling **RdcGetFileData** multiple times to obtain file data.<br>Opnum: 16 |

#### 3.2.5.1.1 CheckConnectivity (Opnum 0)

The **CheckConnectivity** method is used for a client to check whether the server is reachable and has been configured to replicate with the server.

```
DWORD CheckConnectivity(
  [in] FRS_REPLICA_SET_ID replicaSetId,
  [in] FRS_CONNECTION_ID connectionId
);
```

**replicaSetId:** From the **GUID** of the replication group. It is configured as the **GUID** of an object under the path msDFSR-GlobalSettings/msDFSR-ReplicationGroup. See section 2.2.2 for a definition of the AD schema. This corresponds to the replication group **GUID** (see section

2.2.2.5). As with all other uses of GUIDs used in DFS-R, the usual assumption that **GUIDs** are globally unique is implied. This means that two **GUIDs** used in different contexts, such as *replicaSetId*s for two different replication groups, or a **GUID** used for a *replicaSetId* and one used for a *connectionId* are different. The DFS-R protocol does not require other constraints on **GUIDs**, such as the specific contents of a **GUID**.

**connectionId:** From the **GUID** of the connection. It is configured as the **GUID** of an object under the path msDFSR-GlobalSettings/msDFSR-ReplicationGroup/msDFSR-Member/msDFSR-Connection.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | Method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | If the *connectionId* or *replicaSetId* are not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server is not ready to accept a connection. |

**Message Handling:** The server checks to see if the *replicaSetId* is a valid configured replication group. If not, then FRS_ERROR_CONNECTION_INVALID MUST be returned.

The server checks if the *connectionId* is a valid connection object (that is, it is a **connectionId** in the server's configuration) of the given replication group, and that this server is a member of that connection (that is, this connection is configured with this machine as the server). If not, FRS_ERROR_CONNECTION_INVALID MUST be returned.

**Actions Triggered:** None.

**Sequencing:** None.

### 3.2.5.1.2 EstablishConnection (Opnum 1)

The **EstablishConnection** method establishes a logical connection from a client to a server. A logical connection to the server is required before most other operations can be performed.

```
DWORD EstablishConnection(
  [in] FRS_REPLICA_SET_ID replicaSetId,
  [in] FRS_CONNECTION_ID connectionId,
  [in] DWORD downstreamProtocolVersion,
  [in] DWORD downstreamFlags,
  [out] DWORD* upstreamProtocolVersion,
  [out] DWORD* upstreamFlags
);
```

**replicaSetId:** From the **GUID** of the configured replication group. Same as in section 3.2.5.1.1.

**connectionId:** From the **GUID** of the configured connection. Same as in CheckConnectivity (section 3.2.5.1.1).

**downstreamProtocolVersion:** Identifies the version of the protocol implemented by the client. The version MUST be one of: 0x00050000, 0x00050001, or 0x00050002, as specified in section 2.2.1.1.1.

**downstreamFlags:** A flags bitmask. The only one bit that is defined is TRANSPORT_OS_ENTERPRISE. This one bit MAY be set by the client, and all other bits MUST be clear (0).

**upstreamProtocolVersion:** Receives the server's protocol version number. Expected values are the same as for *downstreamProtocolVersion*.

**upstreamFlags:** A flags bitmask. The only one bit that is defined is TRANSPORT_OS_ENTERPRISE. This one bit MAY be set by the server, and all other bits MUST be clear.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | Method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | If the connectionId or **replicaSetId** are not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server is not ready to establish a connection. |
| FRS_ERROR_INCOMPATIBLE_VERSION | The client's protocol version is incompatible with the server's. |

The replicaSetId and connectionId uniquely identify a connection and a replication group.<20>

**Message Handling:** The server checks to see whether the replicaSetId is a valid configured replication group. If not, then FRS_ERROR_CONNECTION_INVALID MUST be returned.

The server checks whether the connectionId is a valid connection object of the given replication group, and that this server is a member of that connection. If not, then FRS_ERROR_CONNECTION_INVALID MUST be returned.<21><22>

**Actions Triggered:** Upon successfully checking that the caller is authorized to establish the connection, the server establishes an active connection. The server MUST allow at most one logical connection per unique pair of replicaSetId and connectionId. Previous logical connection objects on the same pair MUST be torn down by the server.

**Sequencing:** At most, one logical connection object MUST be maintained by the DFS-R server; therefore, at most one call to EstablishConnection (per unique connectionId) will succeed if a client issues multiple such requests in parallel. All outstanding asynchronous calls that the server maintains for a previous logical connection MUST be completed with an arbitrary nonzero completion status, when a new call to EstablishConnection arrives.<23><24>

### 3.2.5.1.3 EstablishSession (Opnum 2)

The **EstablishSession** method is used to establish a logical relationship on the server for a replicated folder.

```
DWORD EstablishSession(
```

```
  [in] FRS_CONNECTION_ID connectionId,
  [in] FRS_CONTENT_SET_ID contentSetId
);
```

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the configured connection; same as in the **CheckConnectivity** call.

**contentSetId:** The **GUID** of the configured replicated folder. It is configured as the **GUID** of an object under msDFSR-GlobalSettings/msDFSR-ReplicationGroup/msDFSR-Content/msDFSR-ContentSet. The definition of the AD schema is as specified in section 2.2.2.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | Method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |

**Message Handling:** The server checks if the connectionId is a valid connection object of the given replication group, and that this server is a member of that connection. If not, then FRS_ERROR_CONNECTION_INVALID MUST be returned.

The server checks that there is an established connection associated with this connectionId. If not, FRS_ERROR_CONNECTION_INVALID MUST be returned.

The server checks that the **contentSetId** is a valid replicated folder identifier of the given replication group. If not, FRS_ERROR_CONTENTSET_NOT_FOUND MUST be returned.<25>

**Actions Triggered:** Upon successful validation of the parameters, the server establishes an active session for the specified replicated folder.

**Sequencing:** The client MUST have already created a logical connection, and the server is configured to replicate the indicated folder with the client.

#### 3.2.5.1.4 RequestUpdates (Opnum 3)

The **RequestUpdates** method is used to obtain file metadata in the form of updates from a server.

```
DWORD RequestUpdates(
  [in] FRS_CONNECTION_ID connectionId,
  [in] FRS_CONTENT_SET_ID contentSetId,
```

```
  [in, range(0,256)] DWORD creditsAvailable,
  [in] long hashRequested,
  [in] UPDATE_REQUEST_TYPE updateRequestType,
  [in] unsigned long versionVectorDiffCount,
  [in, size_is(versionVectorDiffCount)]
    FRS_VERSION_VECTOR* versionVectorDiff,
  [out, size_is(creditsAvailable), length_is(*updateCount)]
    FRS_UPDATE* frsUpdate,
  [out] DWORD* updateCount,
  [out] UPDATE_STATUS* updateStatus,
  [out] GUID* gvsnDbGuid,
  [out] DWORDLONG* gvsnVersion
);
```

**connectionId:** The globally unique identifier (GUID) of the connection ID, which represents a specific replication partnership.

**contentSetId:** The GUID of the content set ID (replicated folder) to be processed by this replication connection.

**creditsAvailable:** The number of file replication update slots available for use in filling up the output array of updates (frsUpdate).

**hashRequested:** The value is TRUE if the client requests the **server** to compute hashes for the files indicated by the update records; otherwise, the value is FALSE . The server is not required to compute hashes even if client requests so. Recall that DFS-R represents Boolean values in longs, as specified in sections 2.2.1.1.16 and 2.2.1.1.17.

**updateRequestType:** The value from the **UPDATE_REQUEST_TYPE** enumeration that indicates the type of replication updates requested.

**versionVectorDiffCount:** The number of items specified in the *versionVectorDiff* parameter.

**versionVectorDiff:** The set of **FRS_VERSION_VECTOR** structures that describes version information for files to be replicated.

**frsUpdate:** The set of **FRS_UPDATE** structures that describes the update that occurred to each of the files to be replicated.

**updateCount:** The number of items specified in the *frsUpdate* parameter.

**updateStatus:** The value from the **UPDATE_STATUS** enumeration that describes the status of the requested update.

**gvsnDbGuid:** The global version number (GVSN) GUID (as specified in [MS-DTYP] section 2.3.2) for the last field in the *versionVectorDiff* that was processed.

**gvsnVersion:** The version of the *gvsnDbGuid*.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |

| Return value/code | Description |
|---|---|
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |

The client specifies what information from the server it requires using the *versionVectorDiff* parameter. This parameter specifies the difference between the client's version vector and the client's most recent copy of the server's version vector. This represents the information that the client does not have.

The server returns the **FRS_UPDATE**s that are covered by the *versionVectorDiff*.

Both the client and the server can limit the number of updates that are returned by one invocation of this message. The last two parameters allow the server to tell the client how much of the *versionVectorDiff* has been processed during this call.

**Message Handling:** The server checks whether a logical connection and a session have been established for the replicated folder for which the updates are requested. The server MUST return an error code (as specified in section 3.2.5.1.3) if there is not already an established session for this pair of connectionId and contentSetId.

**Actions Triggered:** The server walks its database to send updates that fall in the range of the version chain vector that the client specifies.

The server selects all known **FRS_UPDATE** records for GVSNs that appear in the given *versionVectorDiff*, and match the *updateRequestType*.

The server SHOULD return as many of the selected **FRS_UPDATE**s as are available, and will fit in the specified limits. The client uses the *creditsAvailable* parameter to limit the number of **FRS_UPDATE**s that may be returned.

The returned GVSN is a cursor into the *versionVectorDiff*. It MUST be the last GVSN covered by *versionVectorDiff* that was considered for populating the returned array **FRS_UPDATE**s. (Note that the server may have more **FRS_UPDATE**s with GVSNs that belong to *versionVectorDiff* than the capacity of the *frsUpdate* array).

A server uses *updateStatus* to indicate to the client whether and how to request for further updates.

- **UPDATE_STATUS_WAIT:** specifies that server was unable to fully process the version chain vector due to an external, but recoverable, condition.
- **UPDATE_STATUS_LIVE:** is used as a response to a client request for UPDATE_REQUEST_ALL, in the case where the server was able to respond with all tombstones pertaining to the input version vector, but not with all non-tombstones. The client MUST field an additional request for non-tombstones to get all **updates** pertaining to the version chain vector.

- **UPDATE_STATUS_DONE:** There are no more updates in the scope of the argument version chain vector.
- **UPDATE_STATUS_MORE:** There are potentially more updates from the argument version chain vector.

**Sequencing:** The client MUST have already created a session on the replicated folder for which the update request is for.

#### 3.2.5.1.5 RequestVersionVector (Opnum 4)

The **RequestVersionVector** method is used to obtain the version chain vector persisted on a server.

```
DWORD RequestVersionVector(
  [in] DWORD sequenceNumber,
  [in] FRS_CONNECTION_ID connectionId,
  [in] FRS_CONTENT_SET_ID contentSetId,
  [in] VERSION_REQUEST_TYPE requestType,
  [in] VERSION_CHANGE_TYPE changeType,
  [in] ULONGLONG vvGeneration
);
```

**sequenceNumber:** The sequence number for this request. The sequence number is used to pair the version vector request with the asynchronous response in **AsyncPoll**. During a given session, the client MUST supply a unique sequence number for each call to this function.

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the connection ID that represents a specific replication partnership.

**contentSetId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the content set ID (replicated folder) to be processed by this replication connection.

**requestType:** The value from the **VERSION_REQUEST_TYPE** enumeration that describes the type of replication sync to perform.

**changeType:** The value from the **VERSION_CHANGE_TYPE** enumeration that indicates whether to notify change only, change delta, or send the entire version chain vector.

**vvGeneration:** The generation of the version chain vector for that replicated folder on the requesting protocol.

This requests that the server return the specified version vector information in the outstanding **AsyncPoll** request.

The *vvGeneration* parameter is used to calibrate what incarnation of the server's version chain vector is known to the client. The client supplies the last generation number that it received from the server.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |

| Return value/code | Description |
|---|---|
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |

**Message Handling:** The server checks whether a logical connection and a session have been established for the replicated folder for which the version chain vector is requested. The server MUST keep a time stamp on its own version vector. When the server modifies its version vector (in a way visible to clients), the time stamp is incremented. The server communicates its version vector time stamp information to the client when it responds to **AsyncPoll** requests.

**Actions Triggered:** The *vvGeneration* supplied by the client is used to control when an **AsyncPoll** request can be fulfilled. The request can be filled when the server's version vector time stamp supersedes the time stamp passed in as the *vvGeneration* parameter of the version vector request. The actions associated with the request type passed in by the client are:

- **CHANGE_NOTIFY:** The client requests that the server complete the asynchronous RPC call associated with a version vector request when the server has a time stamp for its local version chain vector that supersedes the time stamp passed in by the client. The server MUST NOT provide any version vector with the callback.
- **CHANGE_DELTA:** The client requests that the server send only the difference between the local version chain vector on the server from the time stamp provided by the version chain vector request and the newest version chain vector on the server. The server MAY ignore this value and instead treat it as CHANGE_ALL.
- **CHANGE_ALL:** The client requests to receive the full version vector of the server.

**Sequencing:** The client MUST have already created a session for the replicated folder of the version vector request. The client SHOULD have an outstanding call to AsyncPoll. If the client does not have an outstanding AsyncPoll request, then the server MUST queue up any response until an AsyncPoll is received such that a response can be sent.

Sequence number for this request: The sequence number is used to pair the version vector request with the asynchronous response in **AsyncPoll**. The asynchronous response from the server that corresponds to a version vector request MUST contain the same sequence number that was created by the client. A client SHOULD therefore not have two outstanding asynchronous requests with the same sequence number.<26>

#### 3.2.5.1.6 AsyncPoll (Opnum 5)

The **AsyncPoll** method used to register an asynchronous callback for a server to provide version chain vectors.

```
DWORD AsyncPoll(
  [in] FRS_CONNECTION_ID connectionId,
  [out] FRS_ASYNC_RESPONSE_CONTEXT* response
);
```

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the connection ID that represents a specific replication partnership.

**response:** The **FRS_ASYNC_RESPONSE_CONTEXT** structure that contains the context for the requested poll.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |

**Message Handling:** The server registers an asynchronous callback with the logical connection on which **AsyncPoll** is made.

**Actions Triggered:** The server responds to this call when requested, using **RequestVersionVector**, or when its version chain vector has changed.

**Sequencing:** None.

### 3.2.5.1.7 RequestRecords (Opnum 6)

The **RequestRecords** method used to retrieve UIDs and GVSNs that a server persists.

```
DWORD RequestRecords(
  [in] FRS_CONNECTION_ID connectionId,
  [in] FRS_CONTENT_SET_ID contentSetId,
  [in] FRS_DATABASE_ID uidDbGuid,
  [in] DWORDLONG uidVersion,
  [in, out] DWORD* maxRecords,
  [out] DWORD* numRecords,
  [out] DWORD* numBytes,
  [out, size_is(,*numBytes)] byte** compressedRecords,
  [out] RECORDS_STATUS* recordsStatus
```

```
);
```

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the connection ID that represents a specific replication partnership.

**contentSetId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the content set ID (replicated folder) to be processed by this replication connection.

**uidDbGuid:** The GVSN that corresponds to the first instance (creation) of this file within the FRS replicated folder and version of the uidDatabaseId from the array of **FRS_ID_GVSN**, respectively. On the first call to this function, the client MUST initialize these fields to all zeros. On subsequent calls, they MUST be the last uidDatabaseId and *uidVersion* processed from the returned results in *compressedRecords*. This provides a cursor, so that it can iterate through all records on the server.

**uidVersion:** The GVSN that corresponds to the first instance (creation) of this file within the FRS replicated folder and version of the *uidDatabaseId* from the array of **FRS_ID_GVSN**, respectively. On the first call to this function, the client MUST initialize them to all zeros. On subsequent calls, they MUST be the last *uidDatabaseId* and *uidVersion* processed from the returned results in *compressedRecords*. This provides a with a cursor, so that it can iterate through all records on the server.

**maxRecords:** The maximum number of records that can be returned to the client.

**numRecords:** The number of items in the *compressedRecords* parameter.

**numBytes:** The size, in bytes, of the *compressedRecords* parameter.

**compressedRecords:** The data records compressed, according to the format specified in [MS-DRSR].

DFS-R uses custom marshaling in this RPC call to compress the set of transmitted records into a blob that conforms to the compressed format specified by the decompression algorithm specified in [MS-DRSR]. Note that the decompression algorithm specified in [MS-DRSR] gives uniquely the set of compressed representations of a source string. Any algorithm that computes one of the set of possible pre-images to the decompression algorithm is an admissible compression algorithm.

The *compressedRecords* bytes correspond to an array of **FRS_ID_GVSN** entries. The size of the array is given by *numRecords*. Thus, the uncompressed **FRS_ID_GVSN** array can be obtained by using a compression engine that implements the format (as specified in [MS-DRSR]) to have it decode *numBytes* from *compressedRecords* into sizeof(**FRS_ID_GVSN**)**numRecords* bytes, which can be re-interpreted as an array of **FRS_ID_GVSN** entries.

**recordsStatus:** The value from the **RECORDS_STATUS** enumeration that indicates whether more update records are available.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |

| Return value/code | Description |
|---|---|
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |

**Message Handling:** The server checks that a session has been established for the contentSetId and that connectionId passed as parameter to the call. It MUST complete the call with an error code if no session has been established.

**Actions Triggered:** The server fills in updates from its own persistent store into the output buffer.

**Sequencing:** None.

Provided that the persistent store on a server is unchanged between calls to RequestRecords, a server MUST be able to fill in all updates present in its persistent store in the course of repeated calls to **RequestRecords**. When all updates have been supplied in the **RequestRecords** call, the server MUST be able to resend all updates again if another round of **RequestRecords** arrive.

### 3.2.5.1.8 UpdateCancel (Opnum 7)

The **UpdateCancel** method is used by a client to indicate to a server that it could not process an update. Recall that "client"—as used in DFS-R—means the downstream partner, and does not indicate the operating system version being run.

```
DWORD UpdateCancel(
  [in] FRS_CONNECTION_ID connectionId,
  [in] FRS_UPDATE_CANCEL_DATA cancelData
);
```

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the connection ID that represents a specific replication partnership.

**cancelData:** The **FRS_UPDATE_CANCEL_DATA** structure that describes an update to cancel.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |

| Return value/code | Description |
|---|---|
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |

**Message Handling:** The server identifies the replicated folder where the **UpdateCancel** call is directed.

**Actions Triggered:** The server MUST record the GVSN from the call; and optionally the other reasons why an update was canceled. When a change to the server's version chain vector, or one of the reasons indicated by the **UpdateCancel** parameters do not hold on the server; and the server has received an outstanding **RequestVersionVector** call for the replicated folder, the server MUST complete the version chain vector request and include the GVSNs included in the payload of the received **UpdateCancel** calls.

**Sequencing:** A session MUST have been established with the server for the replicated folder used in the **UpdateCancel** call.

**Error Handling:** If the specified bufferSize is too small, the server MUST return ERROR_INVALID_PARAMETER.

### 3.2.5.1.9 RawGetFileData (Opnum 8)

The **RawGetFileData** method is used for to transfer successive segments from a file.

```
DWORD RawGetFileData(
  [in, out] PFRS_SERVER_CONTEXT* serverContext,
  [out, size_is(bufferSize), length_is(*sizeRead)]
    byte* dataBuffer,
  [in, range(0, CONFIG_TRANSPORT_MAX_BUFFER_SIZE)]
    DWORD bufferSize,
  [out] DWORD* sizeRead,
  [out] long* isEndOfFile
);
```

**serverContext:** The context handle that represents the requested file replication operation. Use the **InitializeFileTransferAsync** method of the FrsTransport interface to create a *serverContext* prior to using this method.

**dataBuffer:** The file data received from the server.

**bufferSize:** The size, in bytes, of *dataBuffer*.

**sizeRead:** The size, in bytes, of the file data returned in *dataBuffer*.

**isEndOfFile:** The value is TRUE if the end of the specified file has been reached and there is no more file data to replicate to the client; otherwise, the value is FALSE.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

**Message Handling:** The context handle established by the **InitializeFileTransferAsync** determines the context in which this call appears.

**Actions Triggered:** The server retrieves file contents allowed by the call.

**Sequencing:** Calls associated with a context handle are serialized by the RPC runtime.

### 3.2.5.1.10 RdcGetSignatures (Opnum 9)

The **RdcGetSignatures** method is used to obtain RDC signature data from a server.

```
DWORD RdcGetSignatures(
```

```
  [in] PFRS_SERVER_CONTEXT serverContext,
  [in, range(0, CONFIG_RDC_MAX_LEVELS)]
    byte level,
  [in] DWORDLONG offset,
  [out, size_is(length), length_is(*sizeRead)]
    byte* buffer,
  [in, range(1, CONFIG_RDC_MAX_NEEDLENGTH)]
    DWORD length,
  [out] DWORD* sizeRead
);
```

**serverContext:** The context handle that represents the requested file replication operation. Section 3.3.1.5, which gives the sequencing assumptions associated with this method, specifies using the **InitializeFileTransferAsync** method of the FrsTransport interface to create a *serverContext* prior to using this method.

**level:** The recursion level. A client MUST specify a number in the range 1 to CONFIG_RDC_MAX_LEVELS.

**offset:** The zero-based offset, in bytes, at which to retrieve data from the file.

**buffer:** The file signature data received from the server.

**length:** The size, in bytes, of *buffer*.

**sizeRead:** The size, in bytes, of the file data returned in *buffer*.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |

| Return value/code | Description |
|---|---|
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

This method uses the Remote Differential Compression Protocol, as specified in [MS-RDC], when replicating a changed file.

**Message Handling:** The context handle established by the **InitializeFileTransferAsync** determines the context in which this call appears.

**Actions Triggered:** The server returns a buffer of Remote Differential Compression (RDC) signature information for the specified file. The server MUST return as many bytes as requested, except when the end of file is reached. In this case, fewer bytes may be returned, including 0 bytes. The server MUST return an error if *offset* is beyond the end of the file.

**Sequencing:** This method is used after a server context is established with the **InitializeFileTransferAsync** method of the FrsTransport interface. The server MUST allow the client to read randomly from all available signature streams.

**Error Handling:** If *level* is set to 0, the server MUST return ERROR_INVALID_PARAMETER.

The format of the signature data is as is specified in [MS-RDC].

#### 3.2.5.1.11 RdcPushSourceNeeds (Opnum 10)

The **RdcPushSourceNeeds** method is used to register requests for file ranges on a server.

```
DWORD RdcPushSourceNeeds(
  [in] PFRS_SERVER_CONTEXT serverContext,
  [in, size_is(needCount)] FRS_RDC_SOURCE_NEED* sourceNeeds,
  [in, range(0, CONFIG_RDC_NEED_QUEUE_SIZE)]
    DWORD needCount
);
```

**serverContext:** The context handle that represents the requested file replication operation. Section 3.3.1.5, which gives the sequencing assumptions associated with this method, specifies using the **InitializeFileTransferAsync** method of the FrsTransport interface to create a *serverContext* prior to using this method.

**sourceNeeds:** The pointer to a set of **FRS_RDC_SOURCE_NEED** structures that indicate the offsets and lengths of file data that must be sent from the server to the client.

**needCount:** The number of **FRS_RDC_SOURCE_NEED** structures pointed to by *sourceNeeds*.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

**Message Handling:** The context handle established by the **InitializeFileTransferAsync** determines the context in which this call appears.

**Actions Triggered:** The server queues up the requested file offset/length pairs.

**Sequencing:** Calls associated with a context handle are serialized by the RPC runtime.

The server uses these needs to form a stream of data from the marshaled source file. The format of this stream of data is specified in section 3.2.5.1.12, and is returned in the buffer supplied in **RdcGetFileData**.

### 3.2.5.1.12 RdcGetFileData (Opnum 11)

The **RdcGetFileData** method is used to obtain file ranges whose requests have previously been registered on a server.

```
DWORD RdcGetFileData(
  [in] PFRS_SERVER_CONTEXT serverContext,
  [out, size_is(bufferSize), length_is(*sizeReturned)]
    byte* dataBuffer,
  [in, range(0, CONFIG_TRANSPORT_MAX_BUFFER_SIZE)]
    DWORD bufferSize,
  [out] DWORD* sizeReturned
);
```

**serverContext:** The context handle that represents the requested file replication operation. Section 3.3.1.5, which gives the sequencing assumptions associated with this method, specifies using the **InitializeFileTransferAsync** method of the FrsTransport interface to create a serverContext prior to using this method.

**dataBuffer:** The file data received from the server. The server MUST return at least 4 bytes, the data stream header.

**bufferSize:** The size, in bytes, of *dataBuffer*. The *bufferSize* MUST be at least XPRESS_RDC_MIN_GET_DATA_BUFFER_SIZE_WITH_FILE_HEADER bytes.

**sizeReturned:** The size, in bytes, of the file data returned in *dataBuffer*.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |

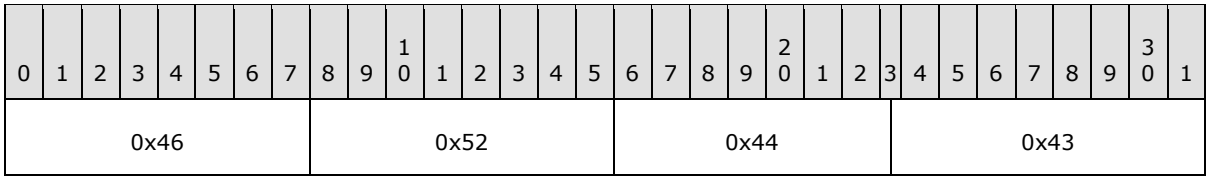
| Return value/code | Description |
|---|---|
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

The format of *dataBuffer* is of the form:

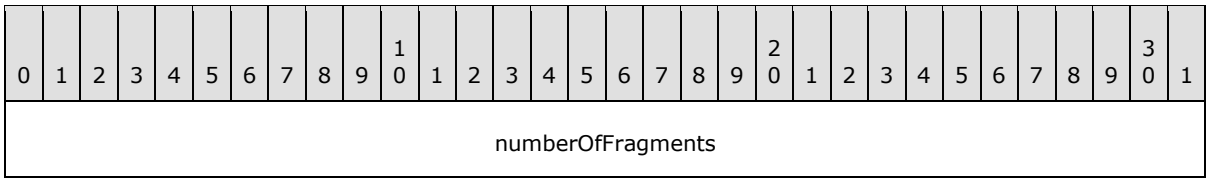
FileHeader
Fragment Header 1
List of fragments 1 <optional>
Data 1
Fragment Header 2
List of fragments 2 <optional>
Data 2

The Fragment header, List of fragments and Data sections are repeated for each fragment that appears in the buffer, as specified above.

**FileHeader:** Consists of the four bytes 0x46, 0x52, 0x44, and 0x43 (in ASCII, that is 'F', 'R', 'D', and 'C').




**Fragment Header:** Consists of a 4-byte number, numberOfFragments, which is number of data fragments to follow. This value MUST be no greater than XPRESS_RDC_MAX_NB_NEEDS_FOR_COMPRESSION.




**Fragment:** The list of fragments consists of 0 or more instances of this structure.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 30 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Block Offset | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Fragment Size | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Data (Variable) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

**BlockOffset:** A starting offset into the uncompressed bytes from the Data portion. The valid range of this field is 0 to X_CONFIG_XPRESS_BLOCK_SIZE-1.

**FragmentSize:** Refers to a number of uncompressed bytes that the server instructs the client to extract. The uncompressed bytes start from BlockOffset and include at most, X_CONFIG_XPRESS_BLOCK_SIZE-blockOffset-1 bytes. The valid range of this field is 1 to X_CONFIG_XPRESS_BLOCK_SIZE.

**Data:** This portion is variable size and is formatted in the same way as the data returned from RawGetFileData (section 3.2.5.1.9), beginning with the block header, which indicates its size. It represents one block.

Additionally, for each fragment, the sum of BlockOffset and FragmentSize MUST be less or equal to X_CONFIG_XPRESS_BLOCK_SIZE.

**Data:** Following the optional list of fragments is a variable size data block that is formatted in the same way as the data segment in a Fragment.

All the data is tightly packed – no padding bytes are added for alignment purposes.

**Message Handling:** The context handle established by the **InitializeFileTransferAsync** determines the context in which this call appears.

**Actions Triggered:** The server serves file data from the source needs that were queued up. The server MUST NOT return more data (from the marshaled source file) than the client, as specified in section 3.2.5.1.11.

**Sequencing:** Calls associated with a context handle are serialized by the RPC runtime.

### 3.2.5.1.13 RdcClose (Opnum 12)

The **RdcClose** method informs the server that the server context information can be released.

```
DWORD RdcClose(
  [in, out] PFRS_SERVER_CONTEXT* serverContext
);
```

**serverContext:** The File Replication Service server context that represents the file replication operation to be ended.

**Return Values:**

| Return value | Description |
|---|---|
| 0 | The method completed successfully. |

**Message Handling:** The context handle established by the **InitializeFileTransferAsync** determines the context in which this call appears.

**Actions Triggered:** The server collects the state associated with the file transfer context handle.

**Sequencing:** Calls associated with a context handle are serialized by the RPC runtime.

### 3.2.5.1.14 InitializeFileTransferAsync (Opnum 13)

The **InitializeFileTransferAsync** method is used by a client to start a file download. The client supplies an update to specify which file to download. The server provides its latest version of the update and initial file contents.

```
DWORD InitializeFileTransferAsync(
  [in] FRS_CONNECTION_ID connectionId,
  [in, out] FRS_UPDATE* frsUpdate,
  [in] long rdcDesired,
  [in, out] FRS_REQUESTED_STAGING_POLICY* stagingPolicy,
  [out] PFRS_SERVER_CONTEXT* serverContext,
  [out] FRS_RDC_FILEINFO** rdcFileInfo,
  [out, size_is(bufferSize), length_is(*sizeRead)]
    byte* dataBuffer,
  [in, range(0, CONFIG_TRANSPORT_MAX_BUFFER_SIZE)]
    DWORD bufferSize,
  [out] DWORD* sizeRead,
  [out] long* isEndOfFile
);
```

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the connection ID that represents a specific replication partnership.

**frsUpdate:** The **FRS_UPDATE** structure that contains information about the file being replicated. The fields for the UID in *frsUpdate* must be set to the UID of the file to be downloaded. All other fields MAY have arbitrary values. On return, all fields of *frsUpdate* MUST contain the values that are held by the server. <27>

**rdcDesired:** The value is TRUE if Remote Differential Compression (RDC) should be used when replicating this file; otherwise, the value is FALSE.

**stagingPolicy:** The **FRS_REQUESTED_STAGING_POLICY** enumeration value that indicates the type of staging requested.

**serverContext:** The context handle that represents the requested file replication operation.

**rdcFileInfo:** The **FRS_RDC_FILEINFO** structure that describes the file whose replication is in progress.

**dataBuffer:** The file data received from the server.

**bufferSize:** The size, in bytes, of *dataBuffer*. CONFIG_TRANSPORT_MAX_BUFFER_SIZE is 262144.

**sizeRead:** The size, in bytes, of the file data returned in *dataBuffer*.

**isEndOfFile:** The value is TRUE if the end of the specified file has been reached and there is no more file data to replicate to the client; otherwise, the value is FALSE.

The **server** returns information about the file currently being replicated, and the first buffer of data from that file (if any).

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

File data in *dataBuffer* is transferred over the wire in a format that is composed of a layering:

1. A stream of file data that consists of a custom marshaled format. The custom marshaled format encapsulates file data compatible with [MS-BKUP] and file metadata compatible with formats specified in [MS-FSCC].

2. An encapsulation of the marshaled stream using the format generated by the compression utility, as specified in [MS-DRSR].

The format of the backup stream is as specified in [MS-BKUP], and the format of the compressed stream is as specified in [MS-DRSR].

**Message Handling:** The server MUST check that a logical connection has been established for the *connectionId* supplied in the call, and fail the call with a nonzero completion status if not.

**Actions Triggered:** The server MUST retrieve the file data associated with the requested UID in the supplied update. It MUST then send the file data in the way that the client has specified (using RDC or not using RDC). The server sends file data by providing as much marshaled and compressed data as fits into the output buffer provided in the **InitializeFileTransferAsync** call. The remaining marshaled and compressed file data is sent in response to subsequent client calls to retrieve file contents. The server MUST provide the file metadata that is associated with the file that it serves. It can do so by providing its own view of the update associated with the requested UID in the return value of *frsUpdate*.

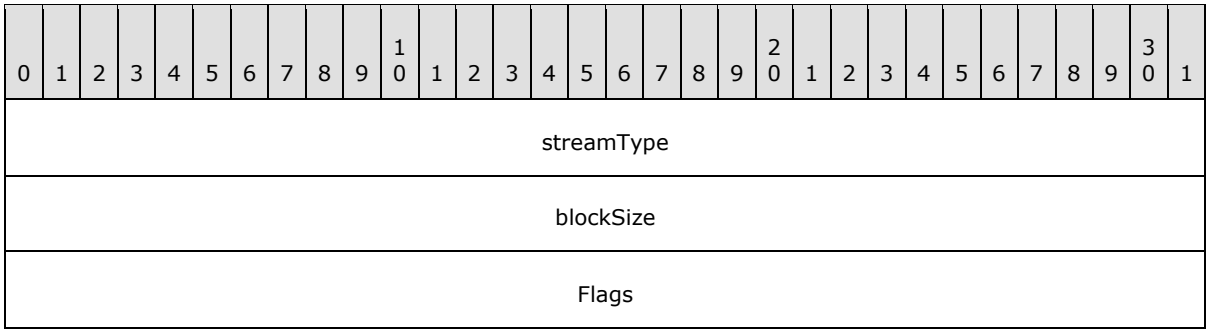
The server must complete the call with FRS_ERROR_FILE_HAS_CHANGED if the file on the server has been deleted, and the corresponding file metadata has been updated with the present flag set to 0.

**Sequencing:** None.

In the case where the client requests an RDC transfer, the server informs the client of the RDC parameters that were used for the signatures for the file being transferred. Typically the parameters are different for the first recursion level and for all other levels.<28><29>

#### 3.2.5.1.14.1 Custom Marshaling Format

The encapsulated marshaled format is a byte stream that encodes a sequence of headers that describe the type and length of the marshaled data that is encoded between the headers. A header is a structure, MARSHAL_BLOCK_HEADER.




**streamType:** An enumeration, which MUST be one of the following values.

| Value | Meaning |
|---|---|
| MS_TYPE_META_DATA | 0x00000001 |

| Value | Meaning |
|---|---|
| MS_TYPE_COMPRESSION_DATA | 0x00000002 |
| MS_TYPE_REPARSE_DATA | 0x00000003 |
| MS_TYPE_FLAT_DATA | 0x00000004 |
| MS_TYPE_SECURITY_DATA | 0x00000006 |
| MS_TYPE_SPARSE_DATA | 0x00000007 |

Stream type 0x00000005 is not used.

**blockSize:** The number of bytes of data in the chunk following the header.

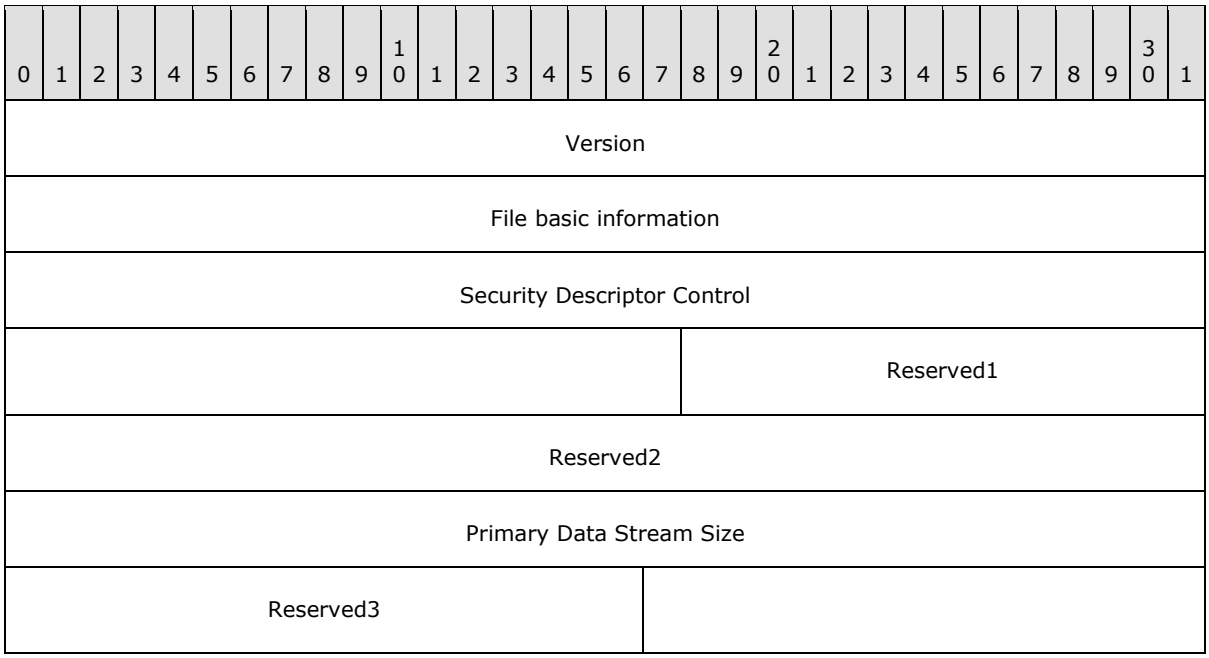
**Flags:** The bitmask, with the HEADER_FLAGS_END_OF_STREAM bit (0x00000001) set if the end of the stream being marshaled has been reached. All other bits MUST be set to 0.

The HEADER_FLAGS_END_OF_STREAM indicates that an end of the stream that is being marshaled has been reached. For instance, if sending two chunks (each of which should be interpreted as belonging to a file's security data), the marshaled stream should contain two headers (each with the MS_TYPE_SECURITY_DATA tag and the HEADER_FLAGS_END_OF_STREAM bit set). On the other hand, if a stream requires multiple chunks, only the last header from that stream MUST have the HEADER_FLAGS_END_OF_STREAM bit set; all other headers MUST have the flags set to 0.

The data MUST be tightly packed. There MUST NOT be any additional bytes of padding.

The format of the data between the headers depends on the value of streamType. These formats are described as follows:

**MS_TYPE_META_DATA (1):** The metadata is written to the top of the marshaled file. The metadata contains information used by the marshaler and other processing code. It consists of data of the following format.




**version:** The marshaler version. It MUST be 3.

**basicInfo:** The FILE_BASIC_INFORMATION structure, as specified in [MS-FSCC] section 2.2.4.5. It contains file times that are stamped on replicated files.

**sdControl:** The original SD control bits of the file being transferred. The format conforms to the format for SECURITY_DESCRIPTOR_CONTROL, as specified in [MS-SECO] section 2.1.2.

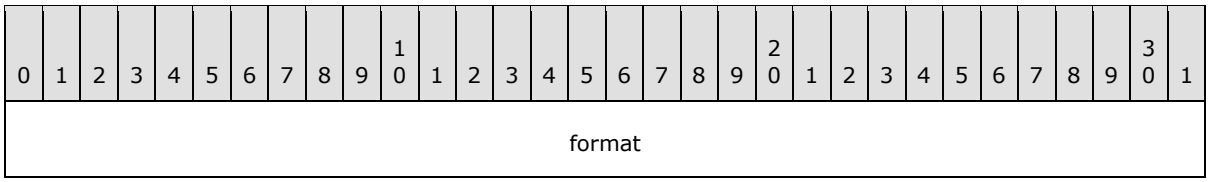
**Reserved1:** Unused, 2 bytes for alignment. MUST be 0. MUST be ignored on receipt.

**Reserved2:** Unused, pads 4 bytes for alignment. MUST be 0. MUST be ignored on receipt.

**primaryDataStreamSize:** A 64-bit unsigned integer. Only used in version 0x00050001 or later. It contains the absolute new end-of-file position as a byte offset from the start of the file, as specified in [MS-FSCC] section 2.2.4.29 (FileStandardInformation).

**Reserved3:** Unused, pads 2 bytes. MUST be 0. MUST be ignored on receipt.

**MS_TYPE_COMPRESSION_DATA (2):** Defines which compression algorithm is used to store the file in compressed format on disk. DFS-R replicates the compression attribute as well as compression algorithm to allow that files get compressed uniformly among replication partners.




**format:** This SHOULD be the value of the **CompressionFormat** field of the FILE_COMPRESSION_INFORMATION structure, as specified in [MS-FSCC] section 2.2.4.7.

**MS_TYPE_REPARSE_DATA (3):** Reparse point data from reparse points that are replicated by DFS-R. The data that follows a header tagged by MS_TYPE_REPARSE_DATA MUST be of a format compatible with the reply format of FSCTL_GET_REPARSE_POINT, as specified in [MS-FSCC] section 2.2.3.16.

**MS_TYPE_FLAT_DATA (4):** A sequence of bytes that complies with the format of [MS-BKUP].

**MS_TYPE_SECURITY_DATA (6):** A sequence of bytes that complies with the format of a SECURITY_DESCRIPTOR. For more information, see [MS-SECO] section 2.1.2.

**MS_TYPE_SPARSE_DATA (7):** A sequence of bytes that complies with the reply format of FSCTL_QUERY_ALLOCATED_RANGES, as specified by [MS-FSCC] section 2.2.3.26.

**File hash:** DFS-R defines the hash of a marshaled file to be the SHA-1 hash of only the chunks associated with:

- MS_TYPE_FLAT_DATA
- MS_TYPE_SECURITY_DATA

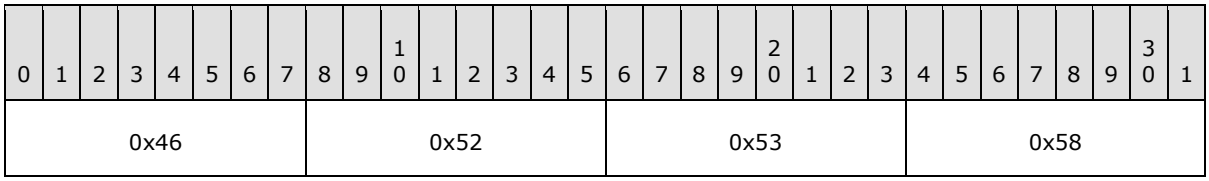
Changes to, for instance, the time stamps that are transferred in chunks associated with MS_TYPE_META_DATA, do not incur a change of the hash. Also, notice that the hash is only computed for the chunks and does not include the headers.

The file hash is included in the hash field of FRS_UPDATE.

#### 3.2.5.1.14.2 Compressed Data Format

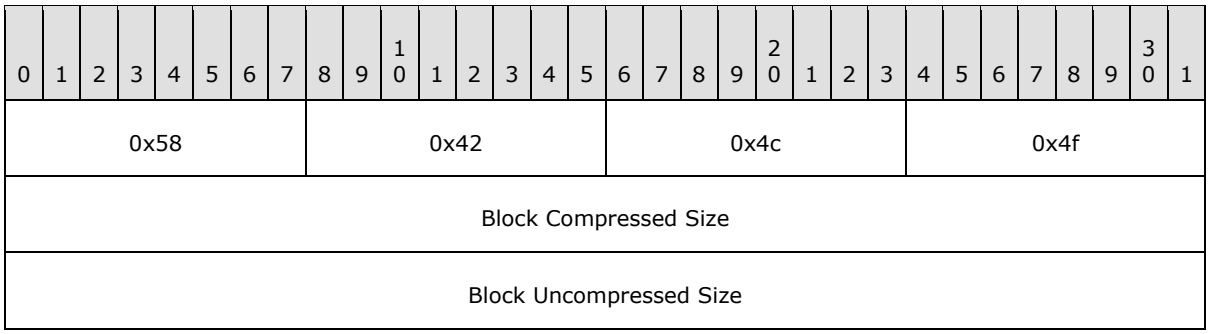
The compressed data stream starts with a header, then blocks of data. Each block of data MUST have a data block header. The format of this data MUST be as follows:

The data stream header MUST consist of a 4-byte signature.




Where "Signature" is always composed of the four bytes 0x46, 0x52, 0x53, and 0x58 (in ASCII, that is 'F', 'R', 'S', and 'X').

Each block header is formatted in the following way.




**Block Signature:** MUST be composed of the four bytes 0x58, 0x42, 0x4c, and 0x4f (in ASCII, that is 'X', 'B', 'L', and 'O').

**Block Compressed Size:** 32-bit unsigned integer size of the data as it appears in this data stream whether it is compressed or not. The compressed size MUST NOT be equal to 0.

**Block Uncompressed Size:** 32-bit unsigned integer size of the data as it would be uncompressed. The uncompressed size MUST be less than or equal to 8192 bytes. The compressed size MUST be less than or equal to the uncompressed size.

If the compressed size and the uncompressed size are equal, the data has not been compressed. That is, the uncompressed data is obtained by copying bytes from the data block without modification.

If the compressed size is less than the uncompressed size, the data has been compressed. For more information about the compressed format, see [MS-DRSR].

The uncompressed size MUST be equal to 8192 bytes, except for the last block of a file transfer, which may be smaller, and except when this data is from RdcGetFileData (see section 3.2.5.1.12 or 3.2.5.1.17).

### 3.2.5.1.15 InitializeFileDataTransfer (Opnum 14)

The **InitializeFileDataTransfer** call is used by a client to initialize a partial file transfer.

The file data is transferred over the wire in a format that is composed of a layering:

1. The main data stream of a file, starting from the offset specified in the argument and extending to the length requested by the caller.
2. An encapsulation of the file stream using the format generated by the compression utility, as specified in [MS-DRSR].

```
DWORD InitializeFileDataTransfer(
  [in] FRS_CONNECTION_ID connectionId,
  [in, out] FRS_UPDATE* frsUpdate,
  [out] PFRS_SERVER_CONTEXT* serverContext,
  [in] DWORDLONG offset,
  [in] DWORDLONG length,
  [out, size_is(bufferSize), length_is(*sizeRead)]
    byte* dataBuffer,
  [in, range(0, 262144)] DWORD bufferSize,
  [out] DWORD* sizeRead,
  [out] long* isEndOfFile
```

```
);
```

**connectionId:** The GUID (as specified in [MS-DTYP] section 2.3.2) of the connection ID that represents a specific replication partnership.

**frsUpdate:** The **FRS_UPDATE** structure containing information about the file being replicated.

**serverContext:** The context handle that represents the requested file replication operation.

**offset:** The file offset of requested data.

**length:** The length of requested data.

**dataBuffer:** The file data received from the server.

**bufferSize:** The size, in bytes, of *dataBuffer*.

**sizeRead:** The size, in bytes, of the file data returned in *dataBuffer*.

**isEndOfFile:** The value is TRUE if the end of the specified file has been reached and there is no more file data to replicate to the client; otherwise, the value is FALSE.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |

| Return value/code | Description |
|---|---|
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

**Message Handling:** The server MUST check that a logical connection has been established for the *connectionId* supplied in the call, and fail the call with a nonzero completion status if not.

**Actions Triggered:** The server MUST retrieve the file data associated with the requested UID in the supplied update. The server MUST send file data by providing as much marshaled and compressed data as fits into the output buffer provided in the call. The remaining marshaled and compressed file data is sent in response to subsequent client calls to retrieve file contents. If the GVSN specified in the *frsUpdate* argument of the client does not correspond to the GVSN persisted on the server, the server MUST fail the call with FRS_ERROR_FILE_HAS_CHANGED.

**Sequencing:** None.

**Note** This method is available only in protocol version 0x00050001, or later.

### 3.2.5.1.16 RawGetFileDataAsync (Opnum 15)

The **RawGetFileDataAsync** method is used instead of calling **RawGetFileData** multiple times to obtain file data. As specified in [MS-RPCE], the specification for asynchronous RPC, an RPC client pulls file data from the byte pipe until receiving an end-of-file notification from the pipe.

```
DWORD RawGetFileDataAsync(
  [in] PFRS_SERVER_CONTEXT serverContext,
  [out] BYTE_PIPE* bytePipe
);
```

**serverContext:** The context handle that represents the requested file replication operation.

**bytePipe:** The asynchronous RPC byte pipe that contains returned file data.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |

| Return value/code | Description |
|---|---|
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

The data stream returned by **RawGetFileDataAsync** is identical to format of the data received by a single call to **RawGetFileData**, should **RawGetFileData** be passed a buffer large enough to hold all the data returned by the pipe.

#### 3.2.5.1.17 RdcGetFileDataAsync (Opnum 16)

The **RdcGetFileDataAsync** method is used instead of calling **RawGetFileData** multiple times to obtain file data. As specified in [MS-RPCE], the specification for asynchronous RPC, an RPCclient pulls file data from the byte pipe until receiving an end-of-file notification from the pipe.

```
DWORD RdcGetFileDataAsync(
  [in] PFRS_SERVER_CONTEXT serverContext,
  [out] BYTE_PIPE* bytePipe
);
```

**serverContext:** The context handle that represents the requested file replication operation.

**bytePipe:** The asynchronous RPC byte pipe that contains returned file data.

**Return Values:**

| Return value/code | Description |
|---|---|
| 0 | The method completed successfully. |
| FRS_ERROR_CONNECTION_INVALID | The connectionId or replicaSetId is not recognized on the server. |
| FRS_ERROR_IN_BACKUP_RESTORE | The server cannot service the connection. |
| FRS_ERROR_CONNECTION_SHUTDOWN | The server cannot service the connection. |
| ERROR_ACCESS_DENIED | The server cannot service the connection. |
| FRS_ERROR_CONTENTSET_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_GUID_NOT_FOUND | The server cannot service the replicated folder. |
| FRS_ERROR_CONTENTSET_NOT_READY | The server cannot service the replicated folder. |
| ERROR_BUSY | The server cannot service the requested file download. |
| FRS_ERROR_FILE_HAS_CHANGED | The file on the server has been deleted. |
| FRS_ERROR_RESOURCE_IS_GHOSTED | The server does not possess the full file contents for the requested file. |
| FRS_ERROR_FILE_INFORMATION_IS_STALE | The server does not have coherent information about the resource. |
| FRS_ERROR_UPDATE_NOT_EXISTS | A client requested a resource that does not exist on the server. |
| FRS_ERROR_RDC_GENERIC | Transferring the file over RDC failed for an unspecified reason. |
| FRS_ERROR_NO_RDC | The server refuses to serve the file over RDC. |
| FRS_ERROR_XPRESS_INVALID_DATA | The cached file on the server is corrupt with respect to the compressed data format. |
| ERROR_SHARING_VIOLATION | A server is unable to read the file from the replicated folder due to an external condition. |

The data stream returned by **RdcGetFileDataAsync** is identical to the data received by a single call to **RdcGetFileData**, should **RdcGetFileData** be passed a buffer large enough to hold all the data returned by the pipe. There is only one file header as described in **RdcGetFileData** call.

### 3.2.6 Timer Events

<30><31>

### 3.2.7 Other Local Events

There are no server-specific local events.

## 3.3 Client Details

In the Distributed File System: Replication (DFS-R) Protocol , the client is responsible for driving the state machine to receive and process version chain vectors, updates, and file transfers. A high-level view of the main state machine maintained by the client is depicted in the following figure. It is intended as an example of a DFS-R client. In this instance, the client first requests a notification of a version vector change, then after receiving the notification, it requests the actual version vector.

<32>

A DFS-R server MUST be agnostic to the specific way that a client chooses to throttle processing updates.

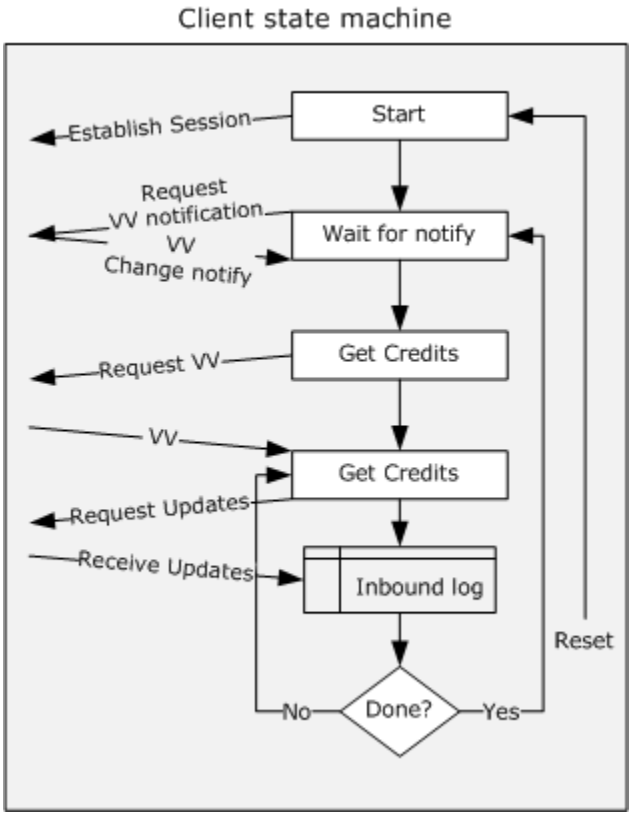



**Figure 3: Main state machine maintained by the client**

To prepare the ground for the detailed walk-through, we first present a refinement of the high-level sequence diagram, as specified in section 1.3. We have added the additional call **AsyncPoll**, which carries the version chain vector response payload and illustrated a file transfer call sequence. Furthermore, we have given asynchronous RPC replies explicitly for the asynchronous RPC messages.

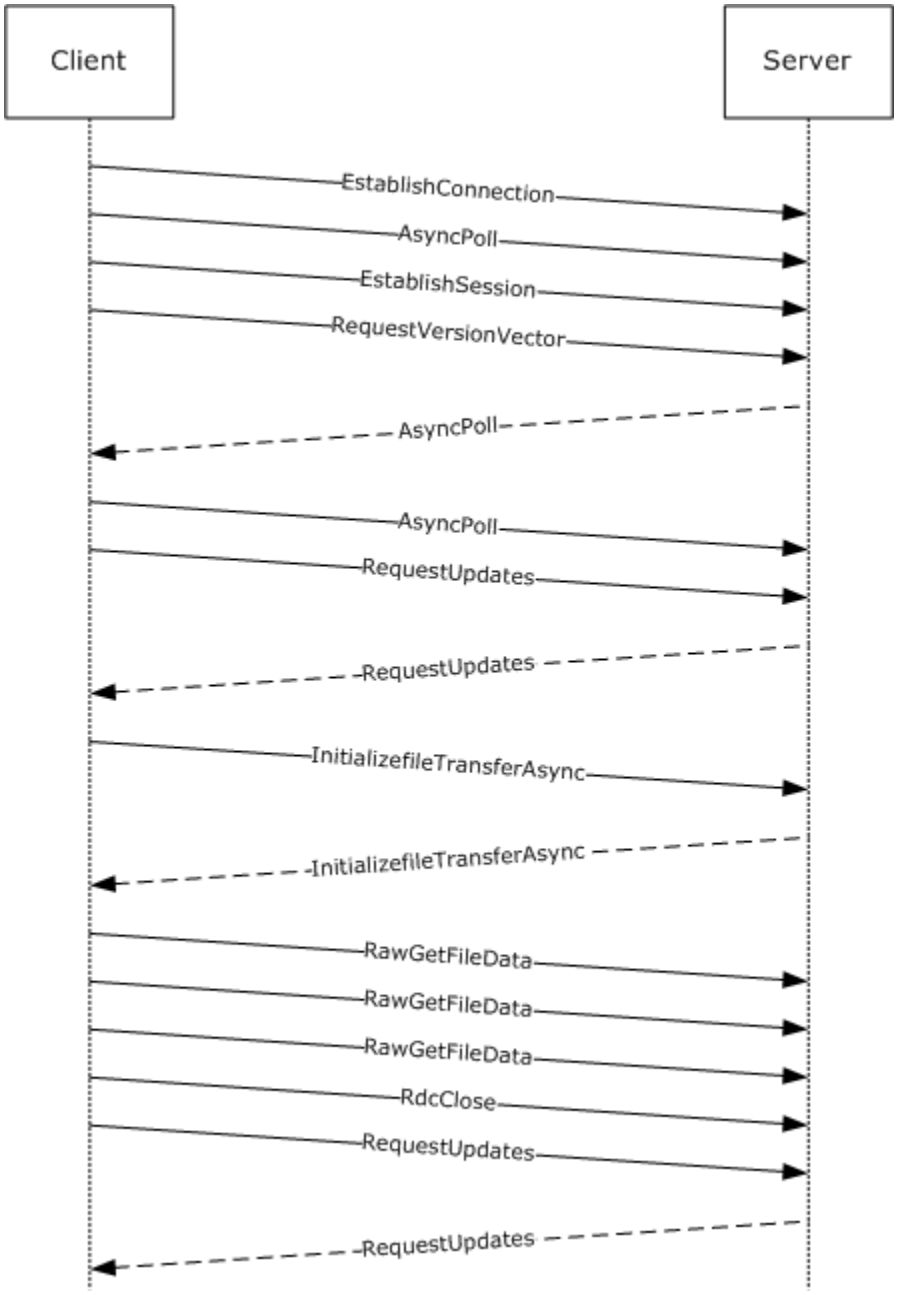



**Figure 4: File transfer call sequence**

The refined synopsis proceeds as a client.

1. Establishes a logical connection with a server.
2. Registers one asynchronous poll with the server for each logical connection.
3. For each replicated folder that is shared between the client and the server, the client establishes a replication session.

4. For each replication session, the client requests the server version chain vectors.

5. The server completes an asynchronous poll request when it is ready with the version chain vector payload in response to the version chain vector request.

6. When receiving an asynchronous poll response, a client SHOULD field a new asynchronous poll request to handle other or later-version vector requests.

7. When the client receives a version chain vector from the server, it calculates the versions that are not known to it and requests updates from the server pertaining to these versions.

8. When the client receives updates from the server, it processes these. While processing a requested update, the client machine may decide that the server updates correspond to file content that it needs to replicate in. It then requests the server to send the file.

9. A file transfer starts with an initialization of file transfer (**InitializeFileTransferAsync**). This establishes a context handle for the file transfer.

10. A raw file transfer proceeds when the client requests chunks of a file by using the context handle.

11. When the file transfer has completed by reaching the end of file or as a result of cancellation, a client MUST close the context handle by using RdcClose.

12. The client registers a request for updated version chain vectors from the server when it has received all updates from the previous version chain vector.

For detailed summaries of the set of state machines that clarify their relative dependencies, see diagrams in sections 3.3.1.1, 3.3.1.2, 3.3.1.3, 3.3.1.4, and 3.3.1.5.

### 3.3.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

#### 3.3.1.1 Establishing Connections

The following figure specifies the state machine to be referred to when DFS-R establishes a connection. It introduces the following states.

- **Disconnected:** A client is not connected.
- **Connected:** A client has established a connection successfully.
- **Polling:** A client has registered an asynchronous poll.

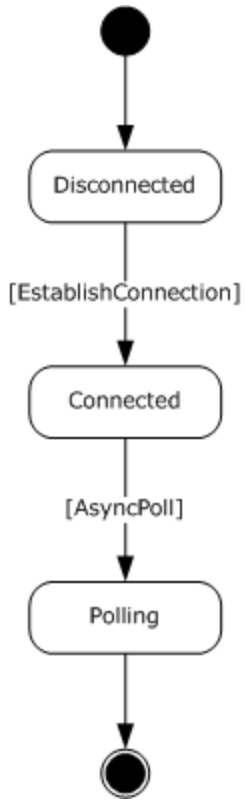



**Figure 5: State machine referred to when DFS-R establishes a connection**

### 3.3.1.2 Main Update Request State Machine

The Distributed File System: Replication (DFS-R) Protocol requires that a client establishes a session for each **replicated folder** that is configured on a connection. All replication activity takes place within one such session. The following figure specifies the state machine for replicating content on replicated folder. In this initial state, it is assumed that the state machine that establishes a connection has reached its terminal state. It introduces the following states.

- **InSession:** A client has established a replication session for a replicated folder.
- **Requesting Version Vector:** A client has registered a request for a version chain vector with its **server**.
- **Poll Again:** A client has received a response for a version chain vector request.
- **Requesting Updates:** A client is requesting updates based on a server's version chain vector.

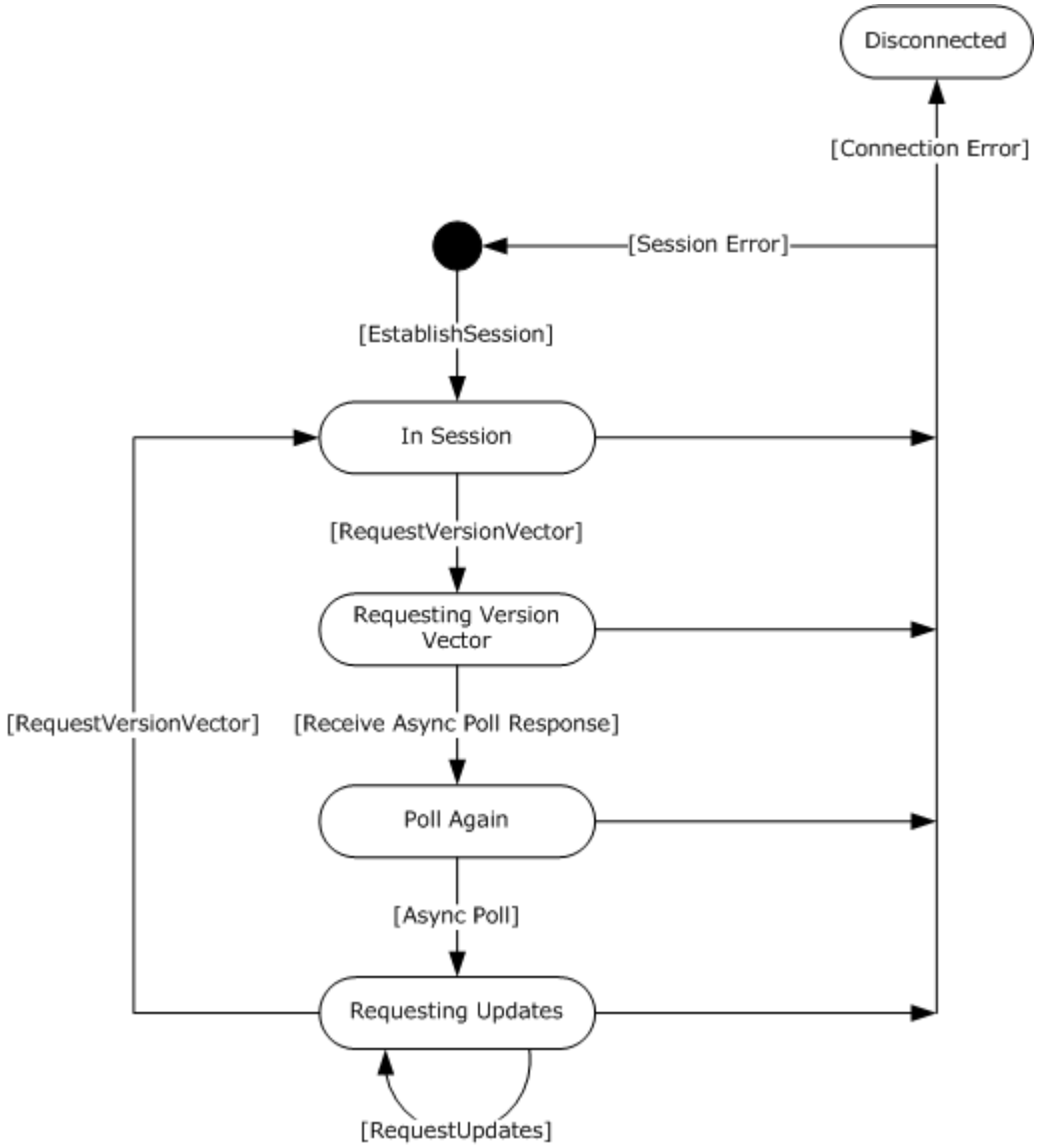



**Figure 6: Main update request state machine**

### 3.3.1.3 Slow Sync

Slow Sync is a secondary means to ensure the consistency of the data between each pair of machines. The main part of the Distributed File System: Replication (DFS-R) Protocol of exchanging version vectors may in some circumstances, such as garbage collection of tombstone updates, leave inconsistencies between machines undetected.

When specifying the Slow Sync protocol, the following states are referred to.

- **SlowSyncInit:** The Slow Sync protocol is to be initialized.
- **SlowSync Requesting Version Vector:** The Slow Sync protocol is requesting server's version chain vector.

- **SlowSync Poll Again:** The client registers an additional poll with the server.
- **SlowSync Request Records:** The client is to request records from the server.

The state transitions associated with SlowSync follow.

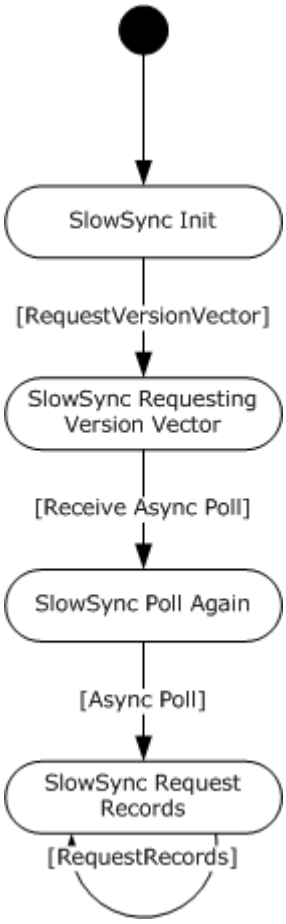



**Figure 7: Slow Sync state machine**

### 3.3.1.4 Raw File Transfer

A direct file transfer (one that does not use RDC) starts by a call to **InitializeFileTransferAsync** with rdcDesired set to 0. The transfer may be fully completed when this call completes, or may have to be finished by either using **RawGetFileData** or **RawGetFileDataAsync**. When the server signals end of file, the client uses **RdcClose** to dispose of the context handle associated with the file download. The state transitions associated with a raw file download are illustrated as follows:

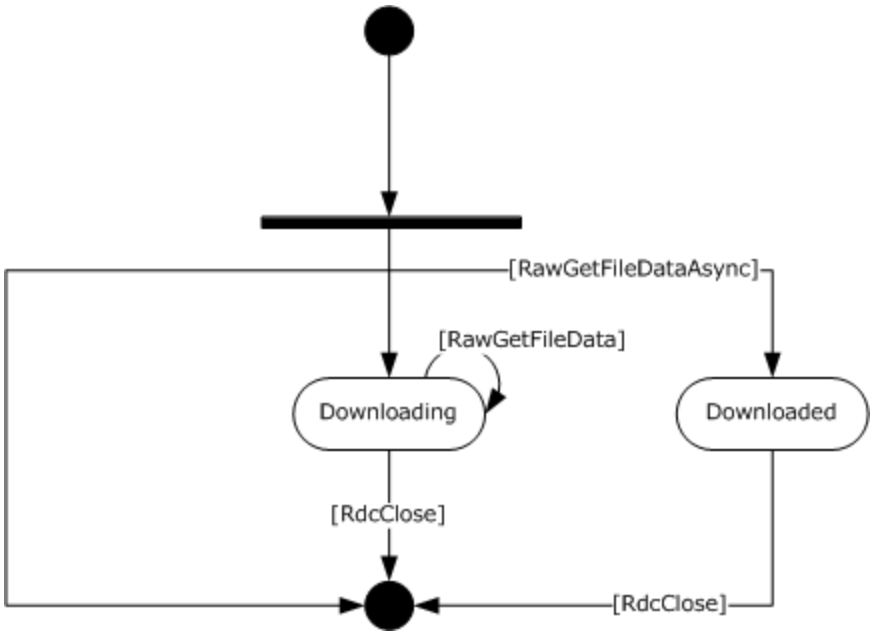



**Figure 8: Raw file transfer state machine**

### 3.3.1.5 RDC File Transfer

A transfer over the RDC protocol starts and finishes similarly to a direct file transfer. The client first uses an **InitializeFileTransferAsync** to establish a context handle for the file, and then calls **RdcGetSignatures** to retrieve the signatures (possibly at recursive levels) that pertain to the file being transferred. To retrieve file data at given offsets and lengths, the client uses **RdcPushSourceNeeds** to indicate which data ranges are requested from the file to be transferred. Notice that several ranges may be coalesced in a single call to **RdcPushSourceNeeds**. To retrieve actual file data, the client uses separate calls, **RdcGetFileData**, or a single call to **RdcGetFileDataAsync**. When the file transfer is done, the client is required to use **RdcClose** to context handle gracefully.

RDC allows the application to choose the recursion depth arbitrarily. Choosing an optimum recursion depth is difficult because many factors are involved, including the available network bandwidth and network latency, the speed and load of the disk storage systems on the server and client machines, as well as available CPU time for computing Signatures, if necessary.<33>

There are few constraints on the order of which a client uses these APIs. Signature data and file data MAY be downloaded concurrently.

The DFS-R server requires that the client MUST follow the following rules:

1. A client initializes RDC transfer using **InitializeFileTransferAsync**, specifying that it wants to perform a download using RDC.

2. If a server determines that the file to be served is not suitable for the RDC protocol, it sets rdcSignatureLevels to 0 in the **FRS_RDC_FILEINFO** structure. The client MUST then complete the download by using a direct transfer, as if it had not requested the RDC protocol.

3. A client serializes its calls to **RdcGetSignatures**, **RdcPushSourceNeeds**, **RdcGetFileData**, or **RdcGetFileDataAsync**.

4. The accumulated amount of data requested using **RdcGetFileData** or **RdcGetFileDataAsync** never exceeds the length of the intervals pushed by **RdcPushSourceNeeds**. The DFS-R server is otherwise free to fail all subsequent requests on that context handle.

5. **RdcClose** terminates a context handle and no further requests can be made on the context handle following this call. The RPC runtime furthermore enforces this constraint.

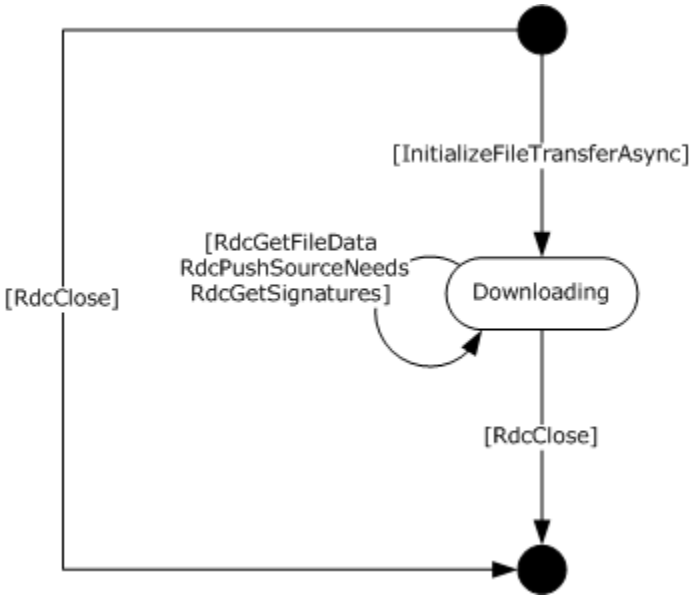



**Figure 9: RDC file transfer state machine**

**Note** The client MUST call **RdcGetFileData** or **RdcGetFileDataAsync**, but not both.

### 3.3.2 Timers

<34>

### 3.3.3 Initialization

For each configured connection, a client creates a separate state machine, as specified in the figure in section 3.3.1.1, and enters the DisConnected state in each state machine.

### 3.3.4 Higher-Layer Triggered Events

None.

### 3.3.5 Message Processing Events and Sequencing Rules

In DFS-R, the client initiates all communication with the server. Only in a few cases does the server trigger a callback into the client when completing asynchronous RPC messages. For uniformity, the message processing events are presented as triggered by a successful call to **EstablishConnection**. All subsequent actions and messages are triggered by replies by the server. The initial client state is DisConnected.

#### 3.3.5.1 DisConnected

**Message Handling:** None.

**Actions Triggered:** A client MUST, possibly with a delay induced by a schedule, call **EstablishConnection** to establish connection objects with each configured inbound connection.

**Sequencing:** All RPC traffic, other than **CheckConnectivity**, with the server must have been preceded by an **EstablishConnection**.

**Error Handling:** None.

#### 3.3.5.2 EstablishConnection Completes

**Message Handling:** Upon a successful call to **EstablishConnection**, the client MUST enter the Connected state from the DisConnected state. For a summary of the state machine used, see section 3.3.1.1.

**Actions Triggered:** A client MUST call **AsyncPoll** to register a callback with the server. If the call to **AsyncPoll** succeeds, the client MUST enter the Polling state. A client MUST call **EstablishConnection** for each replicated folder that the server replicates to the client, as listed in the configuration.

**Sequencing:** None. **EstablishConnection** can be invoked by the client at any time.

**Error Handling:** If the call to **EstablishConnection** returns a nonzero error code, a client MUST retry **EstablishConnection** based on a time-out of its choice. If the call to **AsyncPoll** returns a nonzero error code, the client MUST enter the DisConnected state.

#### 3.3.5.3 EstablishSession Completes

**Message Handling:** For each replicated folder, the client MUST enter an InSession state when **EstablishSession** completes with a nonzero error code for the corresponding replicated folder.

**Actions Triggered:** A client MUST register a request for a version chain vector for each established session by calling **RequestVersionVector** and enter the Requesting Version Vector state.

**Sequencing:** A client MUST not have more than one simultaneous pending asynchronous RPC call to **RequestVersionVector** or **RequestUpdates**.

**Error Handling:** A client must change its state to either DisConnected or remain in Polling based on whether the returned status of **EstablishSession** and **RequestVersionVector** is only session fatal or connection fatal. A list of the session-fatal error codes is specified in section 3.2.5.1.2.

#### 3.3.5.4 RequestVersionVector Completes

**Message Handling:** None. There is no payload associated with the completion of **RequestVersionVector**.

**Actions Triggered:** None.

**Sequencing:** None.

**Error Handling:** A client MUST enter the DisConnected state on a nonzero completion status from **RequestVersionVector**.

#### 3.3.5.5 AsyncPoll Completes

**Message Handling:** Upon an asynchronous callback of **AsyncPoll** with the payload of a requested version chain vector, the client MUST enter the Poll Again state.

**Actions Triggered:** The client MUST register another asynchronous poll, by calling **AsyncPoll**, with the server. If the version chain vector returned by the server includes versions not already persisted on the client, the client MUST enter the Requesting Updates state. The client MUST then request updates pertaining to the version chain vector received from the server. It uses calls to **RequestUpdates** to retrieve updates from the server.

**Sequencing:** None.

**Error Handling:** A client MUST enter the DisConnected state on a nonzero completion status from **AsyncPoll**. A client MUST enter the DisConnected state if a **RequestUpdates** call fails.

#### 3.3.5.6 RequestUpdates Completes

**Message Handling:** Upon an asynchronous callback from **RequestUpdates** in the Requesting Updates state, the client MUST queue up received updates so that it can download file content associated with files that it MUST replicate in order to synchronize with the server.

**Actions Triggered:** The client MUST follow the state transitions (as specified in section 3.3.1.2) to ensure that it receives all updates held by the server.

**Sequencing:** None.

**Error Handling:** A client MUST enter the Polling state on session fatal completion status of **RequestUpdates**; otherwise, enter the DisConnected state.

##### 3.3.5.6.1 Requesting Updates (State Transitions)

As specified in section 3.2.5.1.4, among the parameters used when requesting updates is an updateRequestType and a version chain vector.

If a client specifies UPDATE_REQUEST_ALL, the server is required to send all updates whose GVSN belong to the version chain vector that is part of the request. If the server is unable to send all updates with an UPDATE_REQUEST_ALL request, it MUST specify to the client where to pick up when requesting further updates. It also MUST specify, using the UPDATE_STATUS, whether the client should request tombstones or live updates using the output value of the update request. If a client specifies UPDATE_REQUEST_TOMBSTONES, the server SHOULD send only tombstones that belong to the requested version chain vector. If the client specifies UPDATE_REQUEST_LIVE, the server SHOULD send only updates that are not tombstones.

In summary, to obtain all updates whose GVSNs are contained in a version chain vector VV, the client is expected to maintain a state machine of the form:

Initially; set updateRequestType = UPDATE_REQUEST_ALL and VV = VV.

In response to calls to **RequestUpdates**, the client updates updateRequestType and VV according to state transitions.

Table: State Transitions for Requesting Updates

| in updateRequestType | out updateStatus | State transition |
|---|---|---|
| UPDATE_REQUEST_ALL | UPDATE_STATUS_DONE | Done. All updates from VV will have been received if the client followed this protocol. |
| UPDATE_REQUEST_ALL | UPDATE_STATUS_MORE<br>UPDATE_STATUS_WAIT | Set updateRequestType to UPDATE_REQUEST_TOMBSTONES. |
| UPDATE_REQUEST_TOMBSTONES | UPDATE_STATUS_DONE<br>UPDATE_STATUS_LIVE | Set updateRequestType to UPDATE_REQUEST_LIVE and set VV = VV. |
| UPDATE_REQUEST_TOMBSTONES | UPDATE_STATUS_MORE, UPDATE_STATUS_WAIT | Remove all entries from VV that are lexicographically less than or equal to (gvsnDbGuid, gvsnVersion). |
| UPDATE_REQUEST_LIVE | UPDATE_STATUS_DONE | Done. All updates from VV will have been received if the client followed this protocol. |
| UPDATE_REQUEST_LIVE | UPDATE_STATUS_MORE, UPDATE_STATUS_WAIT, UPDATE_STATUS_LIVE | Remove all entries from VV that are lexicographically less than or equal to (gvsnDbGuid, gvsnVersion). |

Recall that GVSNs are ordered by the lexicographic extension of the byte-wise ordering on GUIDs and unsigned comparison of version sequence numbers. The ordering on GVSNs is used to prune VV in the above protocol by treating version chain vectors as sets of GVSNs, and then removing elements that are lexicographically less than or equal to a given GVSN.

#### 3.3.5.6.2 Processing Updates

DFS-R ensures convergence by imposing a total ordering on updates. A total ordering is obtained from the fields (**fence**, **attributes**, **createTime**, **clock**, **uidDbGuid**, **uidVersion**, **gvsnDbGuid**, and **gvsnVersion**).

An update with a higher value of **fence** supersedes updates with lower **fence** values, otherwise, if the **fence** values are equal.

An update with the directory attribute set in the **attributes** field supersedes updates that do not have the directory attribute set, otherwise, if these attributes coincide.

An update with a lower value of the **createTime** supersedes updates with higher values, otherwise, if the create times are the same.

An update with a higher value of the **clock** field supersedes updates with a lower value, otherwise, if the **clock** fields are the same.

An update with the lexicographically highest **uidDbGuid** supersedes one with a lower value. GUIDs are compared using a lexicographic left to right comparison of each byte. Where each byte is treated as an unsigned 8-bit number. The C-standard routine, memcmp, may for instance be used to realize this ordering as a positive return value from this routine stipulates that a GUID is lexicographically largest. If the **uidDbGuid** coincide, comparison proceeds to version numbers.

An update with the largest value of **uidVersion** supersedes an update with a lower value of a **uidVersion**. Recall that version sequence numbers are unsigned 64-bit numbers. Otherwise, if the versions are the same.

An update with the lexicographically largest **gvsnDbGuid** supersedes one with a lower value; otherwise, if the GUIDs are the same.

An update with the largest **gvsnVersion** supersedes an update with a lower **gvsnVersion**; otherwise, the two updates have the same GVSN, which a well-behaved implementation of DFS-R only would allow if the updates are in fact identical. That is, a well-behaved implementation of DFS-R treats the fields, except for the file hashes, of an update as immutable after it is created. Furthermore, at most one machine should create an update with a given GVSN.

To ensure convergence, a replicating member MUST store one update per UID that is maximal with respect to the above lexicographic ordering. A replicating member MUST implement a conflict resolution strategy according to standard file system semantics of updates. The minimal set of file system conflicts follow.

**Dangling child conflict:** An update is a dangling child if its present field is nonzero, it is not a replicated folder root, and its parent present field is 0.

A well-formed set of updates does not contain any of the above conflicts and such that whenever update1.parentDbGuid = update2.uidDbGuid, update1.parentVersion = update2.uidVersion and update1.present, then update2.attributes contains the 0x00000010 bit. That is, the bit-wise-and with the attribute and the mask 0x00000010 equals 0x00000010. A client MUST maintain a well-formed set of updates.

A dangling-child conflict MUST be resolved by ensuring that parents are saved in persistent storage prior to their children. Absence of dangling children is then enforced as a protocol invariant.

In addition to dangling-child conflicts, a client MAY also resolve cycle and name conflicts:

**Cycle conflict:** An update update1 introduces a directory cycle if there is a sequence update1, update2 ,…, . updatek such update1.present is nonzero and updatei.parentDbGuid = update(i+1).uidDbGuid, updatei.parentVersion = update(i+1).uidVersion, for I = 1…k, and update1 = updatek.

A cycle conflict MAY be resolved by creating an update with a fresh GVSN and a higher clock value that retains the old parent. To ensure convergence using this scheme in the presence of cycle conflicts, a client MUST process received updates in ancestral order—parents before children.<35>

**Name conflict:** Two updates update1 and update2 are in name conflict if their **parentDbGuid** and **parentVersion** fields are the same, they both have the present field set to a nonzero value, and their **name** fields are equal according to an implementation specific string comparison, but their **uidDbGuid** or **uidDbVersion** are different.

If a client decides to resolve a name conflict, it MUST generate a new update for the conflict loser with a fresh GVSN, a clock value that is higher than the name conflict loser, set the **present** field set to 0 and the **nameConflict** field set to 1. To guarantee convergence, a client MUST not supersede any update with **nameConflict** set to 1 by any update that sets **present** to 1.<36>

**Reserved UIDs:** DFS-R reserves a number of UIDs for designated resources:

**The UID of replicated folder roots:** The UID of replicated folder roots is fixed by using version 1 and the GUID of the replicated folder, that is:

- uidDbGuid = {GUID Replicated Folder};
- uidVersion = 1;

The GUID of the replicated folder is present in the configuration for DFS-R.

**The UID of version vector tombstones:** DFS-R allows members to garbage collect entries in their version vectors. When a member determines that a GUID m1, which is in the domain of a version chain vector, is stale, it MAY generate an update whose UID consists of

- uidDbGuid = bit-wise-XOR({GUID of Replicated Folder}, m1)
- uidVersion = 2

The update's **present** field is set to 0 and the update is broadcast to replication partners. If a replication partner determines that m1 is not stale, it MAY generate an update that subsumes the broadcasted update with the **present** field set to 1 (a nonzero value).<37>

Version sequence numbers 0-8 are reserved; therefore, the versions of all other UIDs that correspond to replicated files MUST start with at least version 9.

### 3.3.5.7 File Downloads

If a client receives an update whose weight is larger than any corresponding update that it already has for the same UID, and the received update has the **present** field set to a nonzero value, the client MUST download and persist file contents pertaining to the file. The client MUST either use raw file transfer or use RDC file transfer to download the file. A file transfer with either protocol is initiated by a call to **InitializeFileTransferAsync**.

#### 3.3.5.7.1 stagingPolicy Parameter

A staging area refers to a cache containing serialized replicated files. The cache need not be kept consistent with the file system. It is up to the client to detect possible inconsistencies and instruct the server how to recover using FRS_REQUESTED_STAGING_POLICY. In other cases, a client may find itself unable to download a file directly from the replicated folder because network latency and bandwidth limitations repeatedly cause the download to fail. In such cases, a client MAY use STAGING_REQUIRED to instruct the server to use this more suitable strategy.

This does not affect the format of the data sent. The server SHOULD honor the request.<38>

### 3.3.5.8 InitializeFileTransferAsync Completes

**Message Handling:** Upon an asynchronous callback of **InitializeFileTransferAsync**, a client checks the completion status. Upon a nonzero completion status, the client MUST process the completion status using the error handling logic. Upon successful completion, the client MUST proceed to downloading the full file contents.

**Actions Triggered:** If the context handle returned by **InitializeFileTransferAsync** is set to 0, the entire contents of the downloaded file fits in the buffer provided as part of the output parameters of **InitializeFileTransferAsync**. The client MUST assume that the returned value of *frsUpdate* holds the authoritative metadata for the file contents that correspond to the time that the file download took place.

If the returned context handle is nonzero, the client MUST proceed to download the file contents fully.

1. If the call to **InitializeFileTransferAsync** had set *rdcDesired* to a nonzero value and the server sets rdcSignatureLevels in the **FRS_RDC_FILEINFO** structure to a positive number, the client MUST proceed by downloading the file contents over the RDC Protocol. It MUST use **FRS_RDC_FILEINFO** to obtain signatures at the level dictated by the rdcFileInfo.

2. The client MUST ensure that the parameters of the remote signatures match the parameters of the local signatures. If the parameters do not match, the client MUST either generate signatures with the correct parameters, or opt for a non-RDC file transfer.

3. If the call to **InitializeFileTransferAsync** had set rdcDesired to 0 or if the server sets rdcSignatureLevels in the **FRS_RDC_FILEINFO** structure to 0, the client MUST proceed by downloading the file contents using **RawGetFileData** or **RawGetFileDataAsync**.

Recall that **RawGetFileDataAsync** is supported only in protocol version 0x00050001, or later.

**Sequencing:** None.

**Error Handling:** The following table summarizes the set of errors indicated in the completion status of **InitializeFileTransferAsync** that cause a client to behave in specific ways. A client MUST enter the DisConnected state on all other error codes.

| Error code | Meaning |
|---|---|
| 0 | Success |
| FRS_ERROR_CONNECTION_INVALID<br>FRS_ERROR_IN_BACKUP_RESTORE<br>FRS_ERROR_CONNECTION_SHUTDOWN<br>ERROR_ACCESS_DENIED | The client MUST enter DisConnected. |
| FRS_ERROR_CONTENTSET_NOT_FOUND<br>FRS_ERROR_CONTENTSET_GUID_NOT_FOUND<br>FRS_ERROR_CONTENTSET_NOT_READY | The client MUST enter Polling. |
| FRS_ERROR_UPDATE_NOT_EXISTS | The client MUST discard processing the update. |
| FRS_ERROR_FILE_HAS_CHANGED<br>FRS_ERROR_RESOURCE_IS_GHOSTED<br>FRS_ERROR_FILE_INFORMATION_IS_STALE<br>ERROR_SHARING_VIOLATION<br>ERROR_BUSY | The client MUST retry the download. |
| FRS_ERROR_NO_RDC<br>FRS_ERROR_RDC_GENERIC | A client MUST attempt to download the file, setting rdcDesired to 0. |
| FRS_ERROR_XPRESS_INVALID_DATA | The client MUST retry the download, setting stagingPolicy to RESTAGING_REQUIRED |

### 3.3.5.9 RawGetFileData Completes

**Message Handling:** Upon successful completion, the server MUST return the next buffer of marshaled data (as specified in section 3.2.5.1.14) from the file identified by the specified server context. This method transfers the marshaled file data, and does not use the Remote Differential Compression Protocol (as specified in [MS-RDC]) when replicating a changed file.

**Actions Triggered:** In order to receive the full file contents, the client MUST create another call to **RawGetFileData** if the output value of the *isEndOfFile* parameter is 0. If the output value of *isEndOfFile* is 1, the client MUST call **RdcClose** on the context handle associated with the file download.

**Sequencing:** The client MUST not issue another call to **RawGetFileData** on the same file before the previous call has completed.

**Error Handling:** Same as for **InitializeFileTransferAsync**.

#### 3.3.5.10 RdcClose Completes

**Message Handling:** The error status is checked.

**Actions Triggered:** None.

**Sequencing:** A client MUST not use the context handle after it has been closed using **RdcClose**.

**Error Handling:** The client MUST enter the DisConnected state if the call to **RdcClose** returns a nonzero value.

#### 3.3.5.11 RawGetFileDataAsync Completes

**Message Handling:** Upon successful completion, the client has received the entire file contents in the pipe passed in to the asynchronous call.

**Actions Triggered:** The client MUST persist the file data received in the call and MUST call **RdcClose** to dispose of the context handle.

**Sequencing:** None.

**Error Handling:** Same as for **InitializeFileTransferAsync**.

#### 3.3.5.12 RdcGetSignatures Completes

This method uses the Remote Differential Compression Protocol, as specified in [MS-RDC], when replicating a changed file.

**Message Handling:** Upon successful completion, the client received the requested signature data.

**Actions Triggered:** The client may use the requested signature data to reconstruct the file under download by using the Remote Differential Compression Protocol. After comparing the source signatures to the seed signatures (as specified in [MS-RDC]), the client produces a list of needs (list of file ranges that the client needs in order to reconstruct the file). The client separates the seed and source needs, and sends the source needs to the server with the RdcPushSourceNeeds function.

Thus, file reconstruction proceeds by submitting further calls to RdcGetSignatures or to call **RdcPushSourceNeeds** with data ranges of bytes that the client requests to be downloaded.

**Sequencing:** None.

**Error Handling:** Same as for **InitializeFileTransferAsync**.

#### 3.3.5.13 RdcPushSourceNeeds Completes

The source needs specify byte ranges from the marshaled source file being transferred.

**Message Handling:** Upon successful completion, the client has successfully requested a set of data ranges from the source file being transferred.

**Actions Triggered:** A client MUST call **RdcGetFileData** or **RdcGetFileDataAsync** in order to obtain the file data specified by the **RdcPushSourceNeeds** calls.

**Sequencing:** This method is used after a server context is established with the **InitializeFileTransferAsync** method of the FrsTransport interface. The client may call this function multiple times but MUST not exceed more than 20 source needs outstanding before retrieving them with **RdcGetFileData** or **RdcGetFileDataAsync**.

**Error Handling:** Same as for **InitializeFileTransferAsync**.

The client MUST send only valid needs. A valid need is one that specifies only a range of data that is part of the marshaled source file – as indicated by the source signatures.

### 3.3.5.14 RdcGetFileData Completes

This method uses the Remote Differential Compression Protocol (as specified in [MS-RDC]) when replicating a changed file. The data stream returned by **RdcGetFileData** is composed of ranges from the marshaled source file. The returned ranges are specified by the source needs.

**Message Handling:** Upon successful completion, the client has successfully received file data as specified by previous **RdcPushSourceNeeds** calls.

**Actions Triggered:** The data stream is broken into blocks of compressed and uncompressed data. The client MUST track the number of data bytes returned to know when to stop asking for more data. Therefore, a client MUST call **RdcGetFileData**, possibly repeatedly, to receive all the source data requested by **RdcPushSourceNeeds**.

**Sequencing:** This method is used after a server context is established with the **InitializeFileTransferAsync** method of the FrsTransport interface and after the set of source needs for this file has been sent to the server with the **RdcPushSourceNeeds** method of the FrsTransport interface.

**Error Handling:** Same as for **InitializeFileTransferAsync**.

### 3.3.5.15 RdcGetFileDataAsync Completes

Recall that **RdcGetFileDataAsync** is supported only in protocol version 0x00050002.

**Message Handling:** Upon successful completion, the client has successfully received file data, as specified by previous **RdcPushSourceNeeds** calls.

**Actions Triggered:** The client MUST persist the data received in the call and MUST call **RdcClose** to dispose of the context handle.

**Sequencing:** None.

**Error Handling:** Same as for **InitializeFileTransferAsync**.

### 3.3.5.16 Request Records (Slow Sync)

In contrast to exchanging updates that are identified by exchanging version vectors, Slow Sync will have a client validate the contents of its database by walking these with the server.

The Slow Sync protocol is as follows:

**SlowSync Init:** The client requests a version chain vector from the server. It MUST use the type REQUEST_SLOW_SYNC in the **RequestVersionVector** call not to interfere with calls that are outstanding to normal synchronization.

**SlowSync Requesting Version Vector:** The client waits for the asynchronous callback to **AsyncPoll** with the version chain vector.

**SlowSync Poll Again:** Upon receiving the asynchronous callback with the server's version chain vector, the client MUST issue another asynchronous poll using **AsyncPoll**, and enter the SlowSync Request Records state.

**SlowSync Request Records:** The client requests all (UID, GVSN) pairs that correspond to records on the server that are not **tombstones**. It uses **RequestRecords** to retrieve these records.

For each record received from the server that is contained in both the server and client version vectors, the client determines whether there is a mismatch:

If the UID of the record is not saved in persistent storage locally, this record is a mismatch.

Otherwise, if the GVSN of the record does not match the GVSN of the corresponding record in the local database, this record is a mismatch.

If the record was found to be a mismatch, the client engages in a resolution process. The policy for this resolution is implementation dependent.<39><40>

#### 3.3.5.17 UpdateCancel

A client MAY signal that it is unable to process an update at any time by submitting **UpdateCancel**.

**Error Handling:** The client MUST reestablish a logical connection on any nonzero error code.

#### 3.3.5.18 AsyncPoll Completes for REQUEST_SLAVE_SYNC

A client MAY specify REQUEST_SLAVE_SYNC in the requestType associated with a version chain vector request at any time. When such a version chain vector request is issued and how this version chain vector is used are both implementation specific.<41>

#### 3.3.5.19 InitializeFileDataTransfer

A client may request ranges of the main data-stream using this method. When it returns:

**Message Handling:** The client persists the returned file data and uses **RawGetFileData** to retrieve remaining file data not returned in the buffer provided by the server in this call.

**Actions Triggered:** None.

**Sequencing:** None.

**Error Handling:** Same as for **InitializeFileTransferAsync**.<42>

### 3.3.6 Timer Events

<43><44>

### 3.3.7 Other Local Events

None.

# 4 Protocol Examples

## 4.1 Abstract Protocol Examples

In the following examples, we will examine how two machines—A and B—synchronize a common replicated folder using the DFS-R synchronization protocol. The examples are intended to illustrate the use of the basic DFS-R components and data structures.

We assume that A starts out with a database that has a designated record for the replicated folder root. We introduce a designated machine R, the replicated folder root, to own this resource. The fields of the root record in the database maintained by DFS-R include gvsn, the global version sequence number, **uid**, the unique identifier, fid, a file reference number to identify the replicated folder root directory on the file system, the name of the replicated folder directory, and a pointer to a parent record, which for the root is null:

{gvsn = (R,0), uid = (R,0), fid = 57, name = "share", parent = null}.

Note that the GVSN and UID are the same here. This is a general rule: UID coincides with the gvsn value that corresponds to the first occurrence (creation) of resource. We have reused the gvsn representation to generate unique identifiers, by taking advantage of that machine, version number combinations are globally unique.The version chain vector of machine A consists of the map: {}, and its local version sequence number count is initially at 0. The state of machine B is similar, except that the fid associated with the share is most likely different.

### 4.1.1 Basic Content Distribution

Suppose machine A creates two files, a and b, under the replicated folder root. DFS-R updates the database accordingly. In particular, the global version of A is incremented and this gvsn value is assigned to the newly created resource. During the next communication, session machine A realizes by comparing version chain vectors that something happened on A that B does not know. Consequently, A sends all the updates that correspond to unknown gvsn values. Thus, the file creation (edition, renaming, or deletion) operation is propagated to B.

In more detail, when the local updates are processed by FRS, A's database will look like the following figure.

| uid | gvsn | fid | name | parent |
|---|---|---|---|---|
| (R,0) | (R,0) | 57 | *share* | null |
| (A,1) | (A,1) | 69 | *a* | (R,0) |
| (A,2) | (A,2) | 77 | *b* | (R,0) |

vv: {A → 2}

**Figure 10: Database contents of machine A**

The mapping {R 0} is implicit in the version chain vector. In general, version chain vectors map to 0 on machines that are not part of the provided domain. When A joins with B, it sends {A 2} to B.

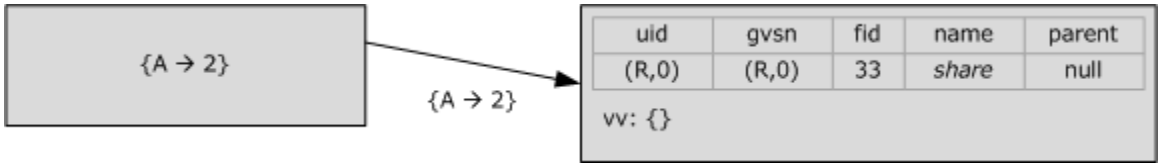



**Figure 11: Sending the version vector to machine B**

Based on this information, B computes the set difference between its version chain vector, which is empty, and the version chain vector {A 2} received from A. The set difference is {A 2}. B then requests updates that come from the set difference. Responding to the update request, A sends the two records for a and b to B. B determines that it must ask A to also send the contents of the files a and b to it. After receiving the content, it can insert those files into its local version of the share, and update its database.

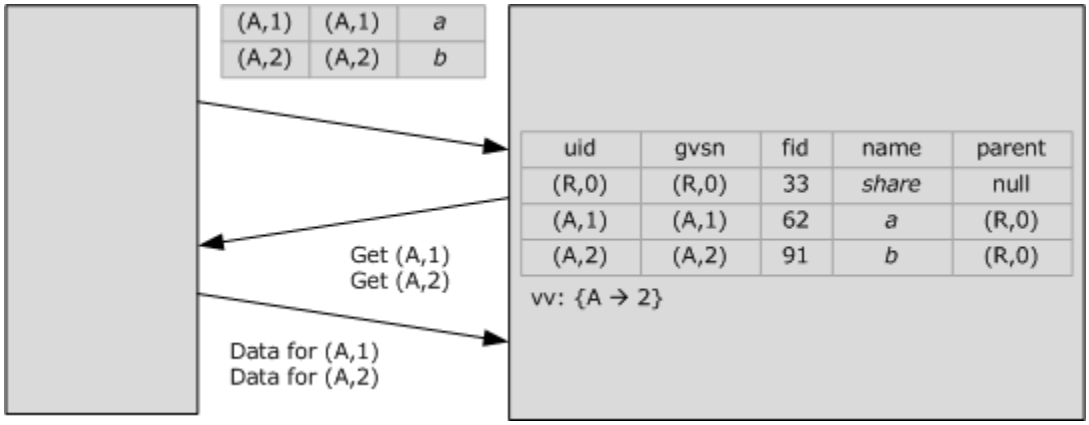



**Figure 12: Database contents of machine B after file replication**

### 4.1.2 Version Chain Vector Logic – Two Machines

Machines A and B are initially synchronized, and have the same version chain vectors (these are intended to be synonyms), for example, {A20, B30}. When a replicated file is edited on A, the version of A is incremented. Therefore, A's version chain vector now is {A21, B30}. Also, the global version sequence number (A21) is assigned to the resource. During the next communication session, the difference between the version chain vectors is computed first. In this case, it is {A21}, exactly the gvsn index of the edited resource. In the end of the session, after the new content is propagated to B, B's version chain vector is updated to {A21, B30} to reflect new state.

### 4.1.3 Version Chain Vector Logic – Three Machines

Assume now that the network has three machines, A, B, and C, and they are configured such that machine B receives updates from machine A, machine C from machine B, and machine A from machine C.

Initially the replicated content on the machines is synchronized, and the version chain vector on all the machines is {A20, B30, C50}. Then, two files are created on A, and at the same time, another file is edited on B. Therefore, the version chain vectors become the values in the following table.

| A: {A22, B30, C50} | B: {A20, B31, C50} | C: {A20, B30, C50} |
|---|---|---|

B meets A session: The difference of the version chain vectors is {A21, A22}, so B requests these resources from A, and, when completed, updates own version chain vector. Note that by difference, we mean the set of gvsn values in the other machines vector that are not covered by my version chain vector. Values, like B31 in this case, are not processed because the protocol is one way directed (not symmetric).

| A: {A22, B30, C50} | B: {A22, B31, C50} | C: {A20, B30, C50} |
|---|---|---|

C meets B session: In this case, the difference is {A21, A22, B31}, so resources marked with these **gvsn** values are communicated to C. The version chain vectors are now the values in the following table.

| A: {A22, B30, C50} | B: {A22, B31, C50} | C: {A22, B31, C50} |
|---|---|---|

A meets C session: The computed difference is {B31}. Note that since the last session between A and C, machine C has also gotten resources corresponding to A21 and A22 values. Yet, the version chain vectors turn out to be sufficient to recognize that these two resources do not need to be sent to A. Finally, the version chain vectors are identical again, and the content is synchronized on the machines.

| A: {A22, B31, C50} | B: {A22, B31, C50} | C: {A22, B31, C50} |
|---|---|---|

### 4.1.4 Concurrent Updates and Tombstones

Machines can update different content concurrently, and synchronization propagates such changes seamlessly when there are no conflicts. For instance, A could modify a, and B could delete b.

| uid | gvsn | fid | name | parent |
|---|---|---|---|---|
| (R,0) | (R,0) | 57 | *share* | null |
| (A,1) | (A,3) | 69 | *a* | (R,0) |
| (A,2) | (A,2) | 77 | *b* | (R,0) |

vv: {A → 3}

| uid | gvsn | fid | name | parent |
|---|---|---|---|---|
| (R,0) | (R,0) | 33 | *share* | null |
| (A,1) | (A,1) | 62 | *a* | (R,0) |
| (A,2) | (B,1) | -1 | *b* | (R,0) |

vv: {A → 2, B → 1}

**Figure 13: A and B database contents, showing tombstone on B**

As usual, a session with A as server starts by A sending the updated version chain vector to B. A session with B as server starts with B sending its version chain vector to A. B may take advantage of knowing that A knows at least {A 2} by sending a smaller version chain vector. In general, B sends its entire version chain vector.

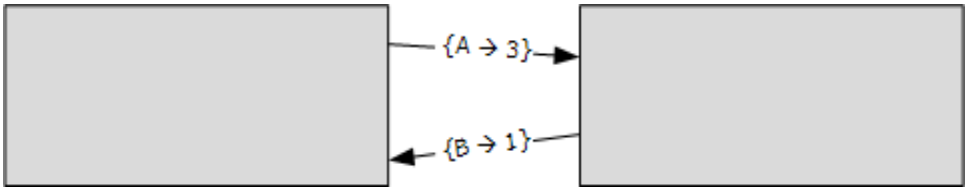



**Figure 14: Version chain vector exchange between A and B**

The machines now exchange updates based on the version chain vectors.

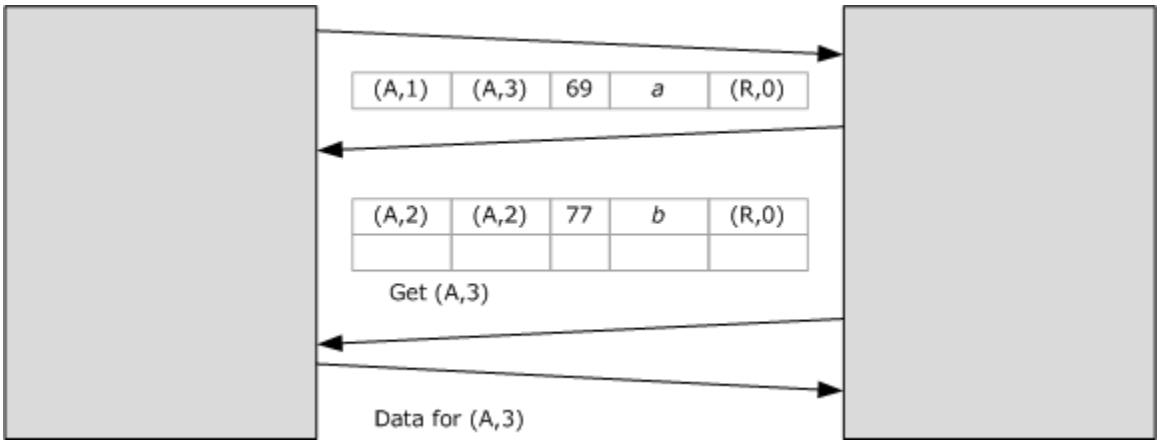
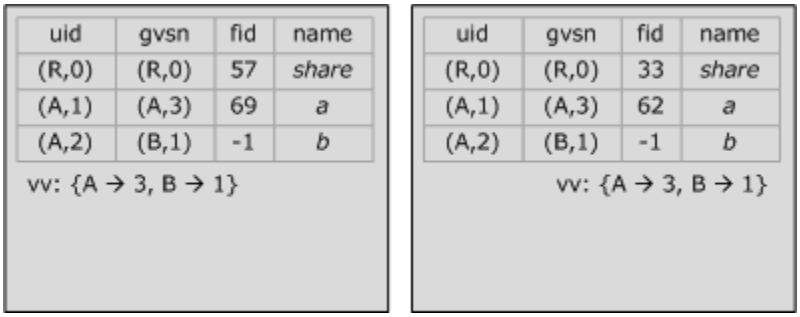



**Figure 15: File replication sequence, showing tombstone replication**

Data for a tombstone is naturally not transmitted. The content on the two machines is then reconciled after synchronization.



| uid | gvsn | fid | name |
|---|---|---|---|
| (R,0) | (R,0) | 57 | *share* |
| (A,1) | (A,3) | 69 | *a* |
| (A,2) | (B,1) | -1 | *b* |

vv: {A → 3, B → 1}

| uid | gvsn | fid | name |
|---|---|---|---|
| (R,0) | (R,0) | 33 | *share* |
| (A,1) | (A,3) | 62 | *a* |
| (A,2) | (B,1) | -1 | *b* |

vv: {A → 3, B → 1}

**Figure 16: A and B database contents after replication, showing tombstone on A**

### 4.1.5 Directory Moves

FRS associates unique identifiers (UIDs) with resources. They identify resources irrespective of their names or location in the file system. A replicated directory p may, for instance, be renamed to q while a child gets created under p on a different machine without encountering a conflict.

FRS furthermore associates a hash with each record to summarize the content on disk. File transmission is redundant if the hash is unchanged between different versions. The hash obviously changes in unpredictable ways when small changes are made to files. The implementation of FRS-2 uses RDC (remote differential compression) to minimize network overhead on small file changes. In

the following example, the content of directory p did not change when it was renamed to q. The database record, therefore, still has the old hash for p, namely, 42. Consequently, only A needs to request content for b from B.

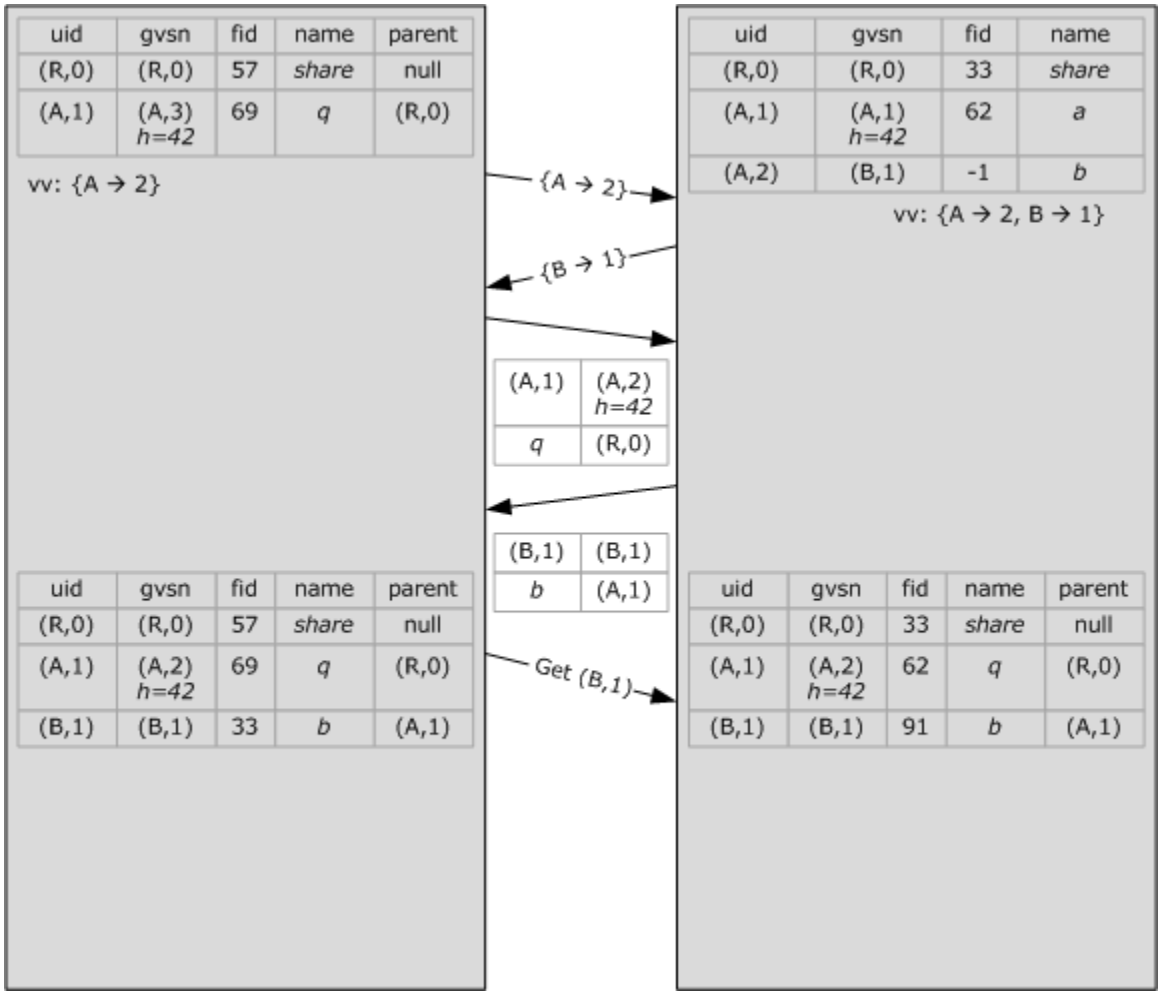



**Figure 17: Example "directory move" file replication sequence between A and B**

### 4.1.6 Name Conflicts

If two machines create files with the same names, they will have a different UID to distinguish them as name conflicting files. Machines handle name conflicts by creating a tombstone for the name conflict loser. Whether the tombstone also requires deleting content immediately depends on where the name conflict is detected. The same name conflict may also be detected on multiple machines. Each machine then generates a tombstone for the loser but in the end, only one tombstone prevails in the final database.

In this example, we examine the case where the name conflict is first resolved by a machine that has the name conflict winner. It then sends the conflict winner before the tombstone to the machine that owns the loser. This causes another tombstone to be created before the original name conflict tombstone arrives. The example illustrates how a single tombstone eventually prevails, and how FRS deals with races that may cause it to perform distributed decisions.

Initially both A and B have a file named a under the share root.

| uid | gvsn | fid | name | parent |
|---|---|---|---|---|
| (R,0) | (R,0) | 57 | *share* | null |
| (A,1) | (A,1) | 69 | *a* | (R,0) |

vv: {A → 1}

| uid | gvsn | fid | name | parent |
|---|---|---|---|---|
| (R,0) | (R,0) | 33 | *share* | null |
| (B,1) | (B,1) | 62 | *a* | (R,0) |

vv: {B → 1}

**Figure 18: A and B database contents**

A starts by sending updates to B. B decides that its version supersedes the version of *a* received from A, so it generates a tombstone for A's update.

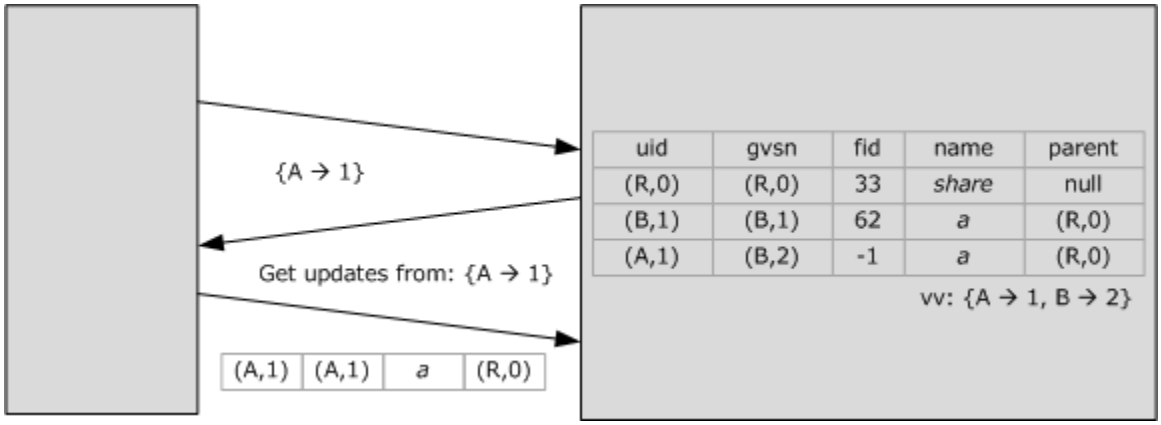



**Figure 19: Machine B database contents after receipt of superseded version of file from machine A**

Now, it is B's turn to send updates to A. This time, B sends the name conflict winner before the tombstone. Machine A will resolve the name conflict similarly to B, but will have to generate its own tombstone. (It may not know the fate of its version on B).

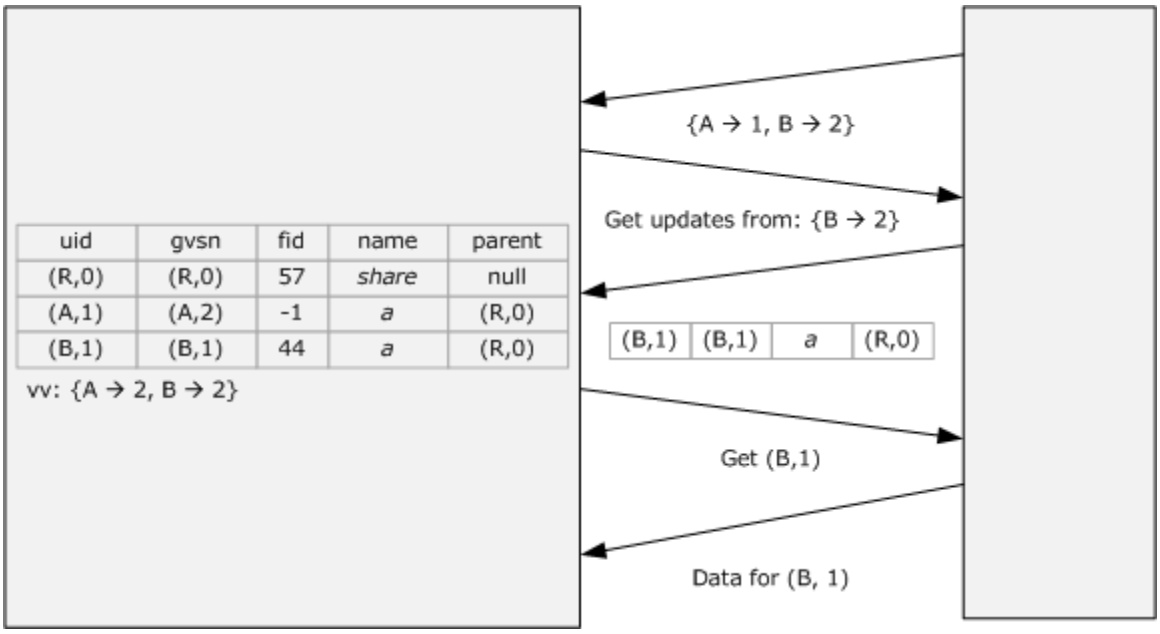



**Figure 20: Machine A database contents after generation of tombstone for old version of file A**

When B sends its own tombstone for *a*—the tombstone with version (B,2)—this tombstone will be compared against the tombstone created by A and conflict resolution chooses a winner. The following figure illustrates the case where A's tombstone wins, and B has not yet synced again with A to update the version of the tombstone.

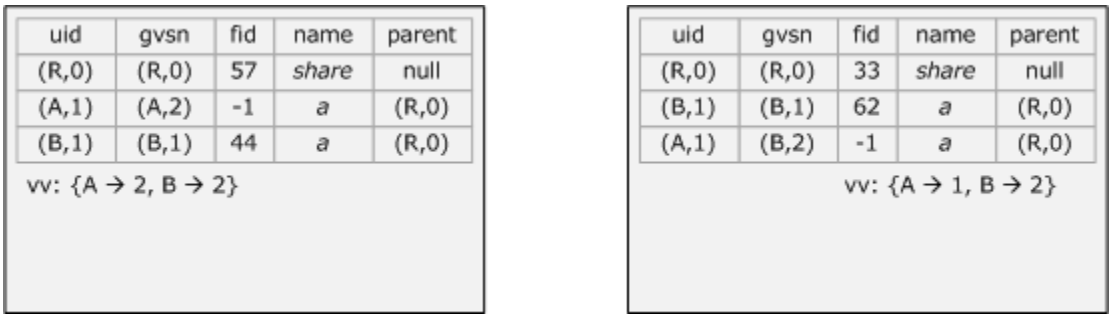



**Figure 21: A and B database contents, showing tombstone conflict**

## 4.2 Examples with Wire-Format Arguments

### 4.2.1 RequestVersionVector

The client may engage in a sequence of calls.

The following table shows the call **RequestVersionVector** with arguments.

| | |
|---|---|
| **sequenceNumber** | 23 |
| **requestType** | REQUEST_NORMAL_SYNC |
| **changeType** | CHANGE_ALL |
| **vvGeneration** | 0 |

The server then responds with a callback to **AsyncPoll** where the *return* parameter of **AsyncPoll** is populated.

| | |
|---|---|
| **sequenceNumber** | 23 |
| **status** | status |
| **vvResponse** | |

To obtain the next server version vector, the client may then call the following.

| | | |
|---|---|---|
| **sequenceNumber** | 103 | Fresh sequence number. |
| **requestType** | REQUEST_NORMAL_SYNC | |
| **changeType** | CHANGE_ALL | |
| **vvGeneration** | 45 | The generation returned by the previous call. |

Note that the client uses the value of *vvGeneration* returned by the server to ensure that the server does not immediately invoke the **AsyncPoll** callback with the same version chain vector, but waits until the server has made a change to its version chain vector that supersedes the version chain vector returned in the first response.

### 4.2.2 Requesting Updates

Suppose versionVectorDiff in a request for updates consists of:

{ { guid1, 10, 200 }, { guid1, 203, 300 }, { guid2, 12, 203} }

The server returns gvsnDbGuid=guid1; gvsnVersion=272, (together with a full frsUpdate array), then the client may retrieve the remaining updates in a second call using *versionVectorDiff* consisting of:

{ { guid1, 272, 300 }, { guid2, 12, 203 } }

All FRS_UPDATEs whose versions belong to the delta:

{ {guid1, 10, 200 }, { guid1, 203, 272 } }

MUST be included in the return value of the initial call.

To further exemplify the functionality of **RequestUpdates**, a client may receive all updates whose GVSN belong to a version chain vector given above by using a sequence of calls.

The following table shows **RequestUpdates**, with parameters.

| | |
|---|---|
| **creditsAvailable** | 256 |
| **hashRequested** | FALSE |
| **updateRequestType** | UPDATE_REQUEST_ALL |
| **versionVectorDiffCount** | 3 |
| **versionVectorDiff (VV)** | { { guid1, 10, 200 }, { guid1, 203, 300 }, { guid2, 12, 203} } |

In one possible scenario, the server supplies the output parameters.

| | |
|---|---|
| **frsUpdate** | An array of *updateCount*updates. |
| **updateCount** | The number of valid entries in the *frsUpdate* array. |
| **updateState** | UPDATE_STATUS_LIVE. The server sent all tombstones whose versions lie within VV and possibly some, but not all, live updates whose versions lie within VV. |
| **gvsnDbGuid** | guid1 |
| **gvsnVersion** | 272 |

The client can then make another call to get more updates from VV by issuing another call.

| | |
|---|---|
| **creditsAvailable** | 256 |
| **hashRequested** | FALSE |
| **updateRequestType** | UPDATE_REQUEST_LIVE |
| **versionVectorDiffCount** | 2 |
| **versionVectorDiff** | { { guid1, 272, 300 }, { guid2, 12, 203} } |

### 4.2.3 Marshaled Data Format

As an illustration, a marshaled data stream may look like the following table.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 20 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 30 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00000001 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x00000024 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x00000001 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x24 bytes of marshaled metadata begin … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| … continued … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Metadata ends. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x00000004 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x0FFFA144 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x00000000 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0x0FFFA144 bytes of data … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| … in the format compatible with the output … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| … of the Win32 API BackupRead … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| … data ends … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... more headers and data continue … | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

### 4.2.4 Ordering on UIDs and GVSNs

Suppose the first byte in guid1 is 0xFA and the first byte in guid2 is 0xFB, then:

(guid1, 0x0000000000000001) < (guid1, 0x0000000000000002)

and

(guid1, 0x0000000000000005) < (guid2, 0x0000000000000004)

## 4.3 Configuration

### 4.3.1 Example Objects in the DFS-R Object Hierarchy

The following figure illustrates the object hierarchy required in AD for storing configuration parameters for Windows implementations of DFS-R.

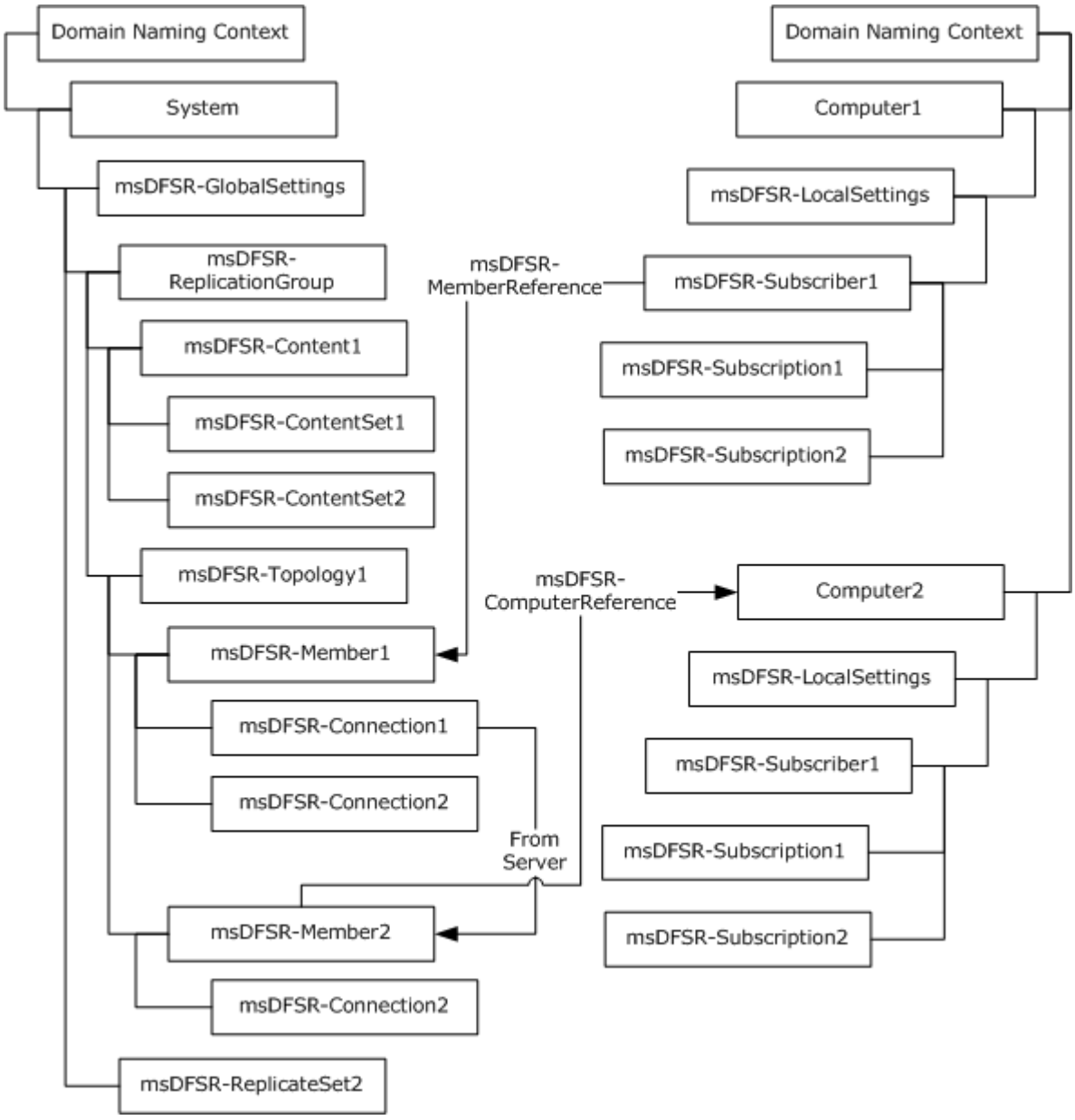



**Figure 22: DFS-R object hierarchy in Active Directory**

# 5 Security

## 5.1 Security Considerations for Implementers

Chunk hashes used in the RDC sub-protocol are computed using a cryptographically weak hash. To check the integrity of a file transfer using RDC, DFS-R furthermore uses a stronger hash, a SHA-1 (160 bit) for checking that the assembled file coincides with the source file on the server. A client that manages content from multiple replicated folders with different access rights should take into account the scope of these integrity checks. For instance, if seed files are permitted across replicated folders, an attack scenario, however constructed, is to inject a seed file, which is different from, but whose chunk hashes and file hash coincide with, a particular plaintext.

## 5.2 Index of Security Parameters

DFS-R uses authenticated encrypted RPC for all replication communication. The relevant security parameters in this context are shown in the following table.

| Security parameter | Section |
|---|---|
| **Authentication level** | 2.1 |
| **Authentication service** | 2.1 |

# 6 Appendix A: Full IDL

The Distributed File System: Replication (DFS-R) Protocol contains one interface, whose **IDL** definition is listed in this section. The IDL definition for this interface imports the "ms-dtyp.idl" file, as specified in [MS-DTYP] section 7.

```
import "ms-dtyp.idl";


#define FRS_COMMUNICATION_PROTOCOL_VERSION_W2K3R20x00050000L
#define FRS_COMMUNICATION_PROTOCOL_VERSION_VISTA 0x00050001L
#define FRS_COMMUNICATION_PROTOCOL_VERSION_LONGHORN_SERVER 0x00050002L

#define CONFIG RDC VERSION(1)
#define CONFIG_RDC_VERSION_COMPATIBLE(1)

#define CONFIG_FILEHASH_DATASIZE (20)
#define CONFIG RDC SIMILARITY DATASIZE (16)
#define CONFIG_RDC_HORIZONSIZE_MIN      (128)
#define CONFIG RDC HORIZONSIZE MAX       (1024*16)
#define CONFIG_RDC_HASHWINDOWSIZE_MIN   (2)
#define CONFIG_RDC_HASHWINDOWSIZE_MAX   (96)
#define CONFIG_RDC_MAX_LEVELS           (8)
#define CONFIG RDC MAX NEEDLENGTH       (65536)
#define CONFIG_TRANSPORT_MAX_BUFFER_SIZE (262144)
#define CONFIG RDC NEED QUEUE SIZE      (20)

#define TRUE 1
#define FALSE 0

typedef GUID FRS_REPLICA_SET_ID;
typedef GUID FRS CONTENT SET ID;
typedef GUID FRS_DATABASE_ID;
typedef GUID FRS_MEMBER_ID;
typedef GUID FRS_CONNECTION_ID;

typedef SYSTEMTIME EPOQUE;

typedef struct _FRS_VERSION_VECTOR {

    GUID dbGuid;
    DWORDLONG low;
    DWORDLONG high;
} FRS_VERSION_VECTOR;

typedef struct _FRS_EPOQUE_VECTOR {

    GUID machine;
    EPOQUE epoque;

} FRS EPOQUE VECTOR;


typedef struct _FRS_ID_GVSN {
   GUID uidDbGuid;
   DWORDLONG uidVersion;
   GUID gvsnDbGuid;
   DWORDLONG gvsnVersion;
} FRS_ID_GVSN;
```

```
typedef struct _FRS_UPDATE {

    long            present;
    long            nameConflict;

    unsigned long   attributes;
    FILETIME        fence;
    FILETIME        clock;
    FILETIME        createTime;


    FRS_CONTENT_SET_ID  contentSetId;
    unsigned char           hash[CONFIG_FILEHASH_DATASIZE];
    unsigned char           rdcSimilarity[CONFIG_RDC_SIMILARITY_DATASIZE];

    GUID            uidDbGuid;
    DWORDLONG       uidVersion;

    GUID            gvsnDbGuid;
    DWORDLONG       gvsnVersion;

    GUID            parentDbGuid;
    DWORDLONG       parentVersion;

    [string] WCHAR  name[260+1];

    long            flags;

} FRS UPDATE;


typedef struct _FRS_UPDATE_CANCEL_DATA {

    FRS UPDATE          blockingUpdate;

    FRS CONTENT SET ID  contentSetId;

    FRS_DATABASE_ID     gvsnDatabaseId;
    FRS_DATABASE_ID     uidDatabaseId;
    FRS DATABASE ID     parentDatabaseId;

    DWORDLONG           gvsnVersion;
    DWORDLONG           uidVersion;
    DWORDLONG           parentVersion;

    unsigned long       cancelType;

    long                isUidValid;
    long                isParentUidValid;
    long                isBlockerValid;

} FRS UPDATE CANCEL DATA;


typedef struct _FRS_RDC_SOURCE_NEED {
    ULONGLONG           needOffset;
    ULONGLONG           needSize;
} FRS_RDC_SOURCE_NEED;


typedef enum
{
    TRANSPORT OS ENTERPRISE = 1
} TransportFlags;
```

```
typedef enum
{
    RDC_UNCOMPRESSED = 1,
    RDC_XPRESS = 2
} RDC_FILE_COMPRESSION_TYPES;

typedef enum
{
    RDC_FILTERGENERIC = 1,
    RDC_FILTERMAX = 2,
    RDC_FILTERPOINT = 3,
    RDC_MAXALGORITHM = 4
} RDC_CHUNKER_ALGORITHM;

typedef enum
{
   UPDATE_REQUEST_ALL = 1,
   UPDATE_REQUEST_TOMBSTONES = 2,
   UPDATE_REQUEST_LIVE = 3
} UPDATE_REQUEST_TYPE;

typedef enum
{
    UPDATE_STATUS_WAIT = 1,
    UPDATE_STATUS_LIVE = 2,
    UPDATE_STATUS_DONE = 3,
    UPDATE_STATUS_MORE = 4
} UPDATE_STATUS;

typedef enum
{
    RECORDS_STATUS_DONE = 1,
    RECORDS_STATUS_MORE = 2
} RECORDS_STATUS;

typedef enum
{
    REQUEST_NORMAL_SYNC = 1,
    REQUEST_SLOW_SYNC = 2,
    REQUEST_SLAVE_SYNC = 3
} VERSION_REQUEST_TYPE;

typedef enum
{
  CHANGE_NOTIFY = 1,
  CHANGE_DELTA = 2,
  CHANGE_ALL = 3
} VERSION_CHANGE_TYPE;

typedef enum
{
    SERVER_DEFAULT = 1,
    STAGING_REQUIRED = 2,
    RESTAGING_REQUIRED = 3
} FRS_REQUESTED_STAGING_POLICY;

typedef struct  FRS_RDC_PARAMETERS_FILTERMAX
{
    [range(CONFIG_RDC_HORIZONSIZE_MIN, CONFIG_RDC_HORIZONSIZE_MAX)]
    unsigned short horizonSize;

    [range(CONFIG_RDC_HASHWINDOWSIZE_MIN, CONFIG_RDC_HASHWINDOWSIZE_MAX)]
    unsigned short windowSize;
} FRS_RDC_PARAMETERS_FILTERMAX;
```

```
typedef struct _FRS_RDC_PARAMETERS_FILTERPOINT
{
    unsigned short minChunkSize;
    unsigned short maxChunkSize;
} FRS_RDC_PARAMETERS_FILTERPOINT;

typedef struct _FRS_RDC_PARAMETERS_GENERIC
{
    unsigned short    chunkerType;
    byte    chunkerParameters[64];
} FRS RDC PARAMETERS GENERIC;


typedef struct
{
    unsigned short      rdcChunkerAlgorithm;
    [switch is(rdcChunkerAlgorithm)] union
    {
        [case(RDC FILTERGENERIC)] FRS RDC PARAMETERS GENERIC filterGeneric;
        [case(RDC_FILTERMAX)]     FRS_RDC_PARAMETERS_FILTERMAX filterMax;
        [case(RDC_FILTERPOINT)]   FRS_RDC_PARAMETERS_FILTERPOINT filterPoint;
    } u;
} FRS RDC PARAMETERS;


typedef struct _FRS_RDC_FILEINFO
{
    DWORDLONG onDiskFileSize;
    DWORDLONG fileSizeEstimate;
    unsigned short      rdcVersion;
    unsigned short      rdcMinimumCompatibleVersion;
    [range(0, CONFIG_RDC_MAX_LEVELS)]
    byte      rdcSignatureLevels;
    RDC FILE COMPRESSION TYPES  compressionAlgorithm;

    [size is(rdcSignatureLevels)]
        FRS_RDC_PARAMETERS rdcFilterParameters[*];
} FRS_RDC_FILEINFO;


typedef struct _FRS_ASYNC_VERSION_VECTOR_RESPONSE {
  ULONGLONG            vvGeneration;
  unsigned long               versionVectorCount;
  [size_is(versionVectorCount)]
  FRS_VERSION_VECTOR * versionVector;
  unsigned long               epoqueVectorCount;
  [size_is(epoqueVectorCount)]
  FRS EPOQUE VECTOR *  epoqueVector;
} FRS_ASYNC_VERSION_VECTOR_RESPONSE;


typedef struct  FRS ASYNC RESPONSE CONTEXT {
    unsigned long sequenceNumber;
    DWORD status;

    FRS_ASYNC_VERSION_VECTOR_RESPONSE result;

} FRS_ASYNC_RESPONSE_CONTEXT;

// The following two defines are needed only for Windows Server 2008
#define FRS_UPDATE_FLAG_GHOSTED_HEADER = 0x04;      // Update request for ghosted header
only
#define FRS UPDATE FLAG DATA           = 0x08;      // Update request for file data only
#define FRS UPDATE FLAG CLOCK DECREMENTED = 0x10;      // Update request
```

```
typedef pipe byte BYTE_PIPE;

[
    uuid(897e2e5f-93f3-4376-9c9c-fd2277495c27),
    version(1.0)
]
interface FrsTransport
{
    DWORD
    CheckConnectivity(
        [in] FRS REPLICA SET ID replicaSetId,
        [in] FRS_CONNECTION_ID connectionId
        );

    DWORD
    EstablishConnection(
        [in] FRS REPLICA SET ID replicaSetId,
        [in] FRS_CONNECTION_ID connectionId,
        [in] DWORD downstreamProtocolVersion,
        [in] DWORD downstreamFlags,
        [out] DWORD *upstreamProtocolVersion,
        [out] DWORD *upstreamFlags
        );

    DWORD
    EstablishSession(
        [in] FRS_CONNECTION_ID connectionId,
        [in] FRS_CONTENT_SET_ID contentSetId
        );


    DWORD
    RequestUpdates(
        [in] FRS CONNECTION ID connectionId,
        [in] FRS_CONTENT_SET_ID contentSetId,
        [in, range(0,256)] DWORD creditsAvailable,
        [in] long hashRequested,
        [in] UPDATE_REQUEST_TYPE updateRequestType,
        [in] unsigned long versionVectorDiffCount,
        [in, size is(versionVectorDiffCount)]
            FRS_VERSION_VECTOR *versionVectorDiff,
        [out, size is(creditsAvailable), length is(*updateCount)] FRS UPDATE *frsUpdate,
        [out] DWORD *updateCount,
        [out] UPDATE_STATUS *updateStatus,
        [out] GUID *gvsnDbGuid,
        [out] DWORDLONG *gvsnVersion
        );

    DWORD
    RequestVersionVector(
        [in] DWORD sequenceNumber,
        [in] FRS CONNECTION ID connectionId,
        [in] FRS_CONTENT_SET_ID contentSetId,
        [in] VERSION REQUEST TYPE requestType,
        [in] VERSION_CHANGE_TYPE changeType,
        [in] ULONGLONG vvGeneration
        );

    DWORD
    AsyncPoll(
        [in]  FRS_CONNECTION_ID connectionId,
        [out] FRS_ASYNC_RESPONSE_CONTEXT* response
        );

    DWORD
```

```
RequestRecords(
    [in] FRS_CONNECTION_ID connectionId,
    [in] FRS_CONTENT_SET_ID contentSetId,
    [in] FRS_DATABASE_ID uidDbGuid,
    [in] DWORDLONG uidVersion,
    [in, out] DWORD *maxRecords,
    [out] DWORD *numRecords,
    [out] DWORD *numBytes,
    [out, size_is(,*numBytes)] byte **compressedRecords,
    [out] RECORDS_STATUS *recordsStatus
    );

DWORD
UpdateCancel(
    [in] FRS_CONNECTION_ID connectionId,
    [in] FRS_UPDATE_CANCEL_DATA cancelData
    );

typedef [context handle] void * PFRS SERVER CONTEXT;

DWORD
RawGetFileData(
    [in, out] PFRS SERVER CONTEXT *serverContext,
    [out, size_is(bufferSize), length_is(*sizeRead)] byte *dataBuffer,
    [in, range(0, CONFIG_TRANSPORT_MAX_BUFFER_SIZE)] DWORD bufferSize,
    [out] DWORD *sizeRead,
    [out] long *isEndOfFile
    );

DWORD
RdcGetSignatures(
    [in] PFRS_SERVER_CONTEXT serverContext,
    [in, range(0, CONFIG_RDC_MAX_LEVELS)] byte level,
    [in] DWORDLONG offset,
    [out, size_is(length), length_is(*sizeRead)] byte *buffer,
    [in, range(1, CONFIG RDC MAX NEEDLENGTH)] DWORD length,
    [out] DWORD *sizeRead
   );

DWORD
RdcPushSourceNeeds(
    [in] PFRS SERVER CONTEXT serverContext,
    [in, size_is(needCount)] FRS_RDC_SOURCE_NEED *sourceNeeds,
    [in, range(0, CONFIG_RDC_NEED_QUEUE_SIZE)] DWORD needCount
    );

DWORD
RdcGetFileData(
    [in] PFRS_SERVER_CONTEXT serverContext,
    [out, size_is(bufferSize), length_is(*sizeReturned)] byte *dataBuffer,
    [in, range(0, CONFIG_TRANSPORT_MAX_BUFFER_SIZE)] DWORD bufferSize,
    [out] DWORD *sizeReturned
   );

DWORD
RdcClose(
   [in, out] PFRS SERVER CONTEXT *serverContext
   );

DWORD
InitializeFileTransferAsync(
    [in] FRS_CONNECTION_ID  connectionId,
    [in, out] FRS UPDATE *frsUpdate,
    [in] long rdcDesired,
    [in,out] FRS_REQUESTED_STAGING_POLICY *stagingPolicy,
```

```
        [out] PFRS_SERVER_CONTEXT *serverContext,
        [out] FRS_RDC_FILEINFO **rdcFileInfo,
        [out, size_is(bufferSize), length_is(*sizeRead)] byte *dataBuffer,
        [in, range(0, CONFIG_TRANSPORT_MAX_BUFFER_SIZE)] DWORD bufferSize,
        [out] DWORD *sizeRead,
        [out] long *isEndOfFile
        );

// This method exists only in Windows Server 2008 and later releases
    DWORD
    InitializeFileDataTransfer(
        [in] FRS_CONNECTION_ID  connectionId,
        [in, out] FRS_UPDATE *frsUpdate,
        [out] PFRS_SERVER_CONTEXT *serverContext,
        [in] DWORDLONG offset,
        [in] DWORDLONG length,
        [out, size is(bufferSize), length is(*sizeRead)] byte *dataBuffer,
        [in, range(0, 262144)] DWORD bufferSize,
        [out] DWORD *sizeRead,
        [out] long *isEndOfFile
        );

// This method exists only in Windows Server 2008 and later releases
    DWORD
    RawGetFileDataAsync(
        [in]  PFRS_SERVER_CONTEXT serverContext,
        [out] BYTE_PIPE* bytePipe
        );

// This method exists only in Windows Server 2008 and later releases
    DWORD
    RdcGetFileDataAsync(
        [in]  PFRS_SERVER_CONTEXT serverContext,
        [out] BYTE PIPE* bytePipe
        );
}
```

# 7 Appendix B: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows Server 2003 R2
- Windows Vista
- Windows Server 2008

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> Section 2.1: The default behavior of a server is to use dynamic endpoints. Static ports can be specified on a connection using the attribute DNSHostName, as specified in section 2.2.2.10.

<2> Section 2.1.2: The default behavior of a Windows-based server is to use dynamic endpoints. Static ports can be specified on a connection using the attribute DNSHostName specified in section 2.2.2.10.

<3> Section 2.2.1.4.10: Windows Server 2003 R2 and Windows Server 2008 use an **rdcChunkerAlgorithm** value of RDC_FILTERMAX. This is the only value of **rdcChunkerAlgorithm** recognized by Windows Server 2003 R2 and Windows Server 2008.

<4> Section 2.2.1.4.11: The parameter **onDiskFileSize** is computed from the size of a cached version of the marshaled, compressed file. The parameter **fileSizeEstimate** is computed based on the allocated byte ranges for the main data stream of a file. The way that Windows computes **rdcSignatureLevels** is specified in [MS-RDC].

<5> Section 2.2.2.1: If not present or not equal to "1.0.0.0", Windows replaces it with "1.0.0.0"".

<6> Section 2.2.2.1: ACLs are set on msDFSR-LocalSettings to protect changing or disclosing configuration information.

<7> Section 2.2.2.3: In version 0x00050001 or later of the Distributed File System: Replication (DFS-R) Protocol, it contains a comma-separated list of 0 or more strings that specify which files should not be compressed. Each string can be a file name, or can be a file name with the initial portion replaced by the wild card character "*". There is no escape character; therefore, it is not possible to specify a file name with a comma.

<8> Section 2.2.2.5: Windows uses this security descriptor as a template for setting the security descriptor on the DFS-R WMI provider.

<9> Section 2.2.2.7: Used only for informative purposes.

<10> Section 2.2.2.7: If no value is set for this attribute, Windows uses a value of "*.tmp,*.bak,~*".

<11> Section 2.2.2.9: Windows configuration tools allow an arbitrary string to be stored here; otherwise ignore this field.

<12> Section 2.2.2.9: DFS-R picks only one such connection by sorting connections alphabetically (by converting the connection GUID to a string) and picks the first one.

<13> Section 2.2.2.11: Windows configuration tools allow an arbitrary string to be stored here; otherwise ignore this field.

<14> Section 2.2.2.11: Windows approximates its bandwidth usage and attempts to limit it based on the setting in the schedule. If the setting in the schedule is 0xF, Windows does not attempt to limit its bandwidth usage. Windows attempts to limit its bandwidth usage for the connection for all values from 0x1 to 0xE, according to the following table.

| Value | Limit in KBps |
|---|---|
| 1 | 16 |
| 2 | 64 |
| 3 | 128 |
| 4 | 256 |
| 5 | 512 |
| 6 | 1024 |
| 7 | 2048 |
| 8 | 4096 |
| 9 | 8192 |
| 10 | 16384 |
| 11 | 32768 |
| 12 | 65536 |
| 13 | 131072 |
| 14 | 262144 |

<15> Section 3.1.4: Windows injects configuration changes into DFS-R over Active Directory in Windows Server 2003 R2 and over WMI in Windows Vista.

<16> Section 3.1.7: Volume dismounts and errors. Per-replicated folder state gets reset when encountering errors that prevent processing files on the volume where they reside. At the protocol level, such events result in discontinuing per-replicated folder replication activity. On the server side, this amounts to failing RPC calls that are specific to the particular replicated folders. Locally, volume state is re-checked based on a periodic timer. Remotely, a client attempts to reestablish per-replicated folder sessions by using the EstablishSession RPC call. The call is retried using an exponential back-off scheme with a maximal time-out of 5 minutes. That is, retries are attempted with a delay of first 1 second, then 2 seconds, then 4, 8, 16, 32, 64, 128, 256, followed by 300-second delays.

<17> Section 3.1.7: Sharing violations. DFS-R in Windows Server 2003 R2, Windows Vista, and Windows Server 2008 accesses files based on **NTFS** semantics. This implies respecting NTFS sharing semantics, which means that if other applications have files open denying shared read access, DFS-R cannot read these files from disk. Similarly, if applications have files open that deny shared delete access, DFS-R cannot update these files (by renaming the old version of these files to a temporary name and renaming a new version of the file). DFS-R in Windows Server 2003 R2 relies on internal

timers (that use an exponential back-off scheme with a maximal time-out of 5 minutes) to re-attempt the file system operations it requires.

<18> Section 3.2.2: Version vector request time-out: The server tears down connection associations when a client establishes a session, but does not request version vectors in a timely fashion. Logically, there is no requirement that the client requests version vectors within a time limit, so the default timeout is 12 hours.

<19> Section 3.2.2: Time-outs on open file handles: When a client requests to initialize a file transfer, the server opens a context handle, which indirectly causes resources, such as file handles to be kept open on the server. The server maintains a 2-minute time-out for inactivity on the open context handles before closing these.

<20> Section 3.2.5.1.2: As a client, Windows Server 2003, Enterprise Edition and Windows Server 2008 Enterprise set the TRANSPORT_OS_ENTERPRISE flag in *downstreamFlags*. As a server, Windows Server 2003, Enterprise Edition and Windows Server 2008 Enterprise set the TRANSPORT_OS_ENTERPRISE flag in *upstreamFlags*.

<21> Section 3.2.5.1.2: The server validates the binding handle and returns ERROR_ACCESS_DENIED if the client cannot be authenticated according to its principal name that is maintained by AD.

<22> Section 3.2.5.1.2: For the server and *downstreamFlags*, the bit TRANSPORT_OS_ENTERPRISE, set when the server or client, respectively, is running an Enterprise SKU (Windows Server 2003, Enterprise Edition or Windows Server 2008 Enterprise). This is informative only, and does not otherwise affect the protocol.

<23> Section 3.2.5.1.2: Check whether the high 16 bits of the version match; if not, the version check is failed with FRS_ERROR_INCOMPATIBLE_VERSION. There are symbolic constants defined in the IDL that refer to the protocol versions currently used by Windows implementations of DFS-R.

<24> Section 3.2.5.1.2: The server refuses the connection with the error FRS_ERROR_IN_BACKUP_RESTORE if a backup operation is in progress on the server.

<25> Section 3.2.5.1.3: Even if the parameters are acceptable, the server can return FRS_ERROR_IN_BACKUP_RESTORE if the server is not ready to accept connections for any reason. Windowsclients retry the operation at some later time ("later" is not defined by the protocol).

<26> Section 3.2.5.1.5: CHANGE_DELTA is treated as CHANGE_ALL in Windows Server 2003 R2, Windows Vista, and Windows Server 2008.

<27> Section 3.2.5.1.14: *frsUpdate* comprises either of the values provided by a server in the response to a **RequestUpdates** call, or all fields are cleared (zeroed out).

<28> Section 3.2.5.1.14: DFS-R in Windows Server 2003 R2, Windows Vista, and Windows Server 2008 always use the values:

```
HorizonSize, level 11024
Hash Window Size, level 148
HorizonSize, level 2+ 128
Hash Window Size, level 2+2
```

<29> Section 3.2.5.1.14: The **server** limits the number of simultaneous outstanding file downloads and returns ERROR_BUSY when a file download is attempted while a configured threshold of simultaneous downloads has been reached.

<30> Section 3.2.6: Version vector request time-out. Upon a version vector request time-out, the server tears down its state associated with the logical connection. Pending asynchronous calls on the connection are completed with an error (ERROR_ACCESS_DENIED) and new calls that assume presence of the connection object fail with the same error code as well.

<31> Section 3.2.6: Time-outs on open file handles. When a context handle is determined stale by a server, it closes all state associated with the context handle. Subsequent calls by a client on the context handle fail with ERROR_INVALID_PARAMETER.

<32> Section 3.3: The client uses a credit system to throttle the number of synchronization instances that it is performing at any given time. A state, GetCredits, is included in the following figure to indicate that the client throttles the number of version vectors that it receives at any given time. The client also uses a credit system to throttle the number of updates that are received, but not yet processed at any given time. The use of credits is internal to the Microsoft implementation of DFS-R. There are no on-the-wire dependencies on how or whether a client manages credits, therefore credits will not be covered beyond this point.

<33> Section 3.3.1.5: DFS-R uses the following algorithm to compute RDC recursion depth:

```
h := 18 / horizonSize0
depth := 0
minSizeLevel := 65536
WHILE (depth < 8) AND (size > minSizeLevel)
    size := size * h + 24
    h := 18 / (2 * horizonSize1)
    minSizeLevel := 32768
    depth := depth + 1
ENDWHILE
```

The various constants used are:

- 18 is the size, in bytes, of an RDC signature (16-byte hash and 2-byte length).
- 65536 is the minimum size, in bytes, of a file that will be considered for RDC transfer.
- 32768 is the minimum size, in bytes, of a signature file that allows increasing the recursion level.
- 24 is the size, in bytes, of the RDC signature file header.
- horizonSize0 is 1024. Used for calculating RDC signatures of the file data.
- horizonSize1 is 256. Used for calculating all recursive levels of RDC signatures.

<34> Section 3.3.2: Connection Schedules. Clients establish and terminate connections based on configured schedules. The only observed effect on the server is that clients may periodically re-connect to it. Connection schedules can be configured externally to Windows with a 15-minute granularity.

<35> Section 3.3.5.6.2: Cycle conflicts are resolved on Windows implementations of DFS-R.

<36> Section 3.3.5.6.2: Name conflicts are resolved. File names are compared using case-insensitive string comparison. Language-specific collation policies are not used when comparing file names.

<37> Section 3.3.5.6.2: For each GUID (m1), Windows implementations of DFS-R generate a version vector tombstone when the last refresh within a set of updates, whose GVSN share the same GUID (m1), is beyond 60 days. Version vector tombstones are subsumed if the machine originating the GUID (m1) on the version vector tombstone receives the version vector tombstone.

<38> Section 3.3.5.7.1: Clients set STAGING_REQUIRED when encountering return errors ERROR_SHARING_VIOLATION when attempting to download files from a **server**.

<39> Section 3.3.5.16: In Windows, the resolution is to reanimate missing records. That is, updates whose UIDs are not in the client's store are downloaded from the server.

<40> Section 3.3.5.16: In Windows, SlowSync is initiated by the client once a week.

<41> Section 3.3.5.18: DFS-R clients with version 0x00050001 or later use REQUEST_SLAVE_SYNC to retrieve the server's version chain vector in response to changes on the client's file system. The server's version chain vector is used to synchronize the clients back to a state where they are a mirror image of the server, thus deleting possibly new files on the clients, as opposed to replicating these out. This behavior is also known as **read-only replicated folders**.

<42> Section 3.3.5.19: DFS-R in the Windows Server 2008 (protocol version 0x00050001 or later) uses this method as part of a mode where DFS-R clients download only file placeholders during synchronization. Selected file data is then downloaded lazily by calling this method before using **RawGetFileData** to retrieve further file data.

<43> Section 3.3.6: Connection Schedules: When a client enters a connection schedule, it uses EstablishConnection to create a fresh logical connection with the server.

<44> Section 3.3.6: Update throttle time-outs: Clients maintain a throttle in the frequency for requesting new updates. This is entirely for the purpose of ensuring that frequently changing files do not overwhelm the network.

# 8 Index

## G



## H



## I



## L



## M



## N



## O



## P



## R

## S



## T



## U



## V



## W