# AFS-3 Programmer's Reference: *Volume Server/Volume Location Server* Interface

Edward R. Zayas

Transarc Corporation

Version 1.0 of 29 August 1991 14:48

FS-00-D165

# Contents


**1 Overview** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**
1.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Volumes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2.2 Volume Naming . . . . . . . . . . . . . . . . . . . . . . . 2
1.2.3 Volume Types . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Document Layout . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 *Volume Location Server* Architecture** . . . . . . . . . . . . . . . . **4**
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.2 The Need For Volume Location . . . . . . . . . . . . . . . . . . . 4
2.3 The VLDB . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3.1 Layout . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3.2 Database Replication . . . . . . . . . . . . . . . . . . . . 6
2.4 The *vlserver* Process . . . . . . . . . . . . . . . . . . . . . . . 6

**3 *Volume Location Server* Interface** . . . . . . . . . . . . . . . . . . **8**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.2 Constants . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2.1 Configuration and Boundary Quantities . . . . . . . . . . . . 9
3.2.2 Update Entry Bits . . . . . . . . . . . . . . . . . . . . . . 10
3.2.3 List-By-Attribute Bits . . . . . . . . . . . . . . . . . . . . 11
3.2.4 Volume Type Indices . . . . . . . . . . . . . . . . . . . . . 11
3.2.5 States for `struct vlentry` . . . . . . . . . . . . . . . . . . 12
3.2.6 States for `struct vldbentry` . . . . . . . . . . . . . . . . . 12
3.2.7 `ReleaseType` Argument Values . . . . . . . . . . . . . . . . 13
3.2.8 Miscellaneous . . . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Structures and Typedefs . . . . . . . . . . . . . . . . . . . . . . 14
3.3.1 `struct vldbentry` . . . . . . . . . . . . . . . . . . . . . . 14
3.3.2 `struct vlentry` . . . . . . . . . . . . . . . . . . . . . . . 15
3.3.3 `struct vital_vlheader` . . . . . . . . . . . . . . . . . . . 16

3.3.4 `struct vlheader` . . . . 16
3.3.5 `struct VldbUpdateEntry` . . . . 17
3.3.6 `struct VldbListByAttributes` . . . . 17
3.3.7 `struct single_vldbentry` . . . . 18
3.3.8 `struct vldb_list` . . . . 18
3.3.9 `struct vldstats` . . . . 18
3.3.10 `bulk` . . . . 19
3.3.11 `bulkentries` . . . . 19
3.3.12 `vldblist` . . . . 20
3.3.13 `vlheader` . . . . 20
3.3.14 `vlentry` . . . . 20
3.4 Error Codes . . . . 21
3.5 Macros . . . . 22
3.5.1 *COUNT_REQ()* . . . . 22
3.5.2 *COUNT_ABO()* . . . . 22
3.5.3 *DOFFSET()* . . . . 22
3.6 Functions . . . . 23
3.6.1 VL_CreateEntry . . . . 24
3.6.2 VL_DeleteEntry . . . . 25
3.6.3 VL_GetEntryByID . . . . 26
3.6.4 VL_GetEntryByName . . . . 27
3.6.5 VL_GetNewVolumeId . . . . 28
3.6.6 VL_ReplaceEntry . . . . 29
3.6.7 VL_UpdateEntry . . . . 30
3.6.8 VL_SetLock . . . . 31
3.6.9 VL_ReleaseLock . . . . 32
3.6.10 VL_ListEntry . . . . 33
3.6.11 VL_ListAttributes . . . . 34
3.6.12 VL_LinkedList . . . . 35
3.6.13 VL_GetStats . . . . 36
3.6.14 VL_Probe . . . . 37
3.7 Kernel Interface Subset . . . . 38

**4 *Volume Server* Architecture . . . . 39**
4.1 Introduction . . . . 39
4.2 Disk Representation . . . . 39
4.3 Transactions . . . . 40
4.4 The *volserver* Process . . . . 41
4.5 Log File . . . . 42

**5 *Volume Server* Interface . . . . 44**
5.1 Introduction . . . . 44

5.2 Constants . . . 44
5.2.1 Configuration and Boundary Values . . . 45
5.2.2 Interface Routine Opcodes . . . 45
5.2.3 Transaction Flags . . . 46
5.2.3.1 vflags . . . 46
5.2.3.2 iflags . . . 47
5.2.3.3 tflags . . . 47
5.2.4 Volume Types . . . 47
5.2.5 LWP State . . . 48
5.2.6 States for `struct vldbentry` . . . 48
5.2.7 Validity Checks . . . 48
5.2.8 Miscellaneous . . . 49
5.3 Exported Variables . . . 50
5.4 Structures and Typedefs . . . 51
5.4.1 `struct volser_trans` . . . 51
5.4.2 `struct volDescription` . . . 52
5.4.3 `struct partList` . . . 52
5.4.4 `struct volser_status` . . . 53
5.4.5 `struct destServer` . . . 54
5.4.6 `struct volintInfo` . . . 54
5.4.7 `struct transDebugInfo` . . . 55
5.4.8 `struct pIDs` . . . 56
5.4.9 `struct diskPartition` . . . 56
5.4.10 `struct restoreCookie` . . . 57
5.4.11 `transDebugEntries` . . . 57
5.4.12 `volEntries` . . . 58
5.5 Error Codes . . . 59
5.5.1 `Standard` . . . 59
5.5.2 `Low-Level` . . . 60
5.6 Macros . . . 61
5.6.1 *THOLD()* . . . 61
5.6.2 *ISNAMEVALID()* . . . 61
5.7 Functions . . . 62
5.7.1 AFSVolCreateVolume . . . 64
5.7.2 AFSVolDeleteVolume . . . 66
5.7.3 AFSVolNukeVolume . . . 67
5.7.4 AFSVolDump . . . 68
5.7.5 AFSVolSignalRestore . . . 69
5.7.6 AFSVolRestore . . . 70
5.7.7 AFSVolForward . . . 71
5.7.8 AFSVolClone . . . 72

5.7.9 AFSVolReClone . . . . . . . . . . 73
5.7.10 AFSVolSetForwarding . . . . . . . . . . 74
5.7.11 AFSVolTransCreate . . . . . . . . . . 75
5.7.12 AFSVolEndTrans . . . . . . . . . . 76
5.7.13 AFSVolGetFlags . . . . . . . . . . 77
5.7.14 AFSVolSetFlags . . . . . . . . . . 78
5.7.15 AFSVolGetName . . . . . . . . . . 79
5.7.16 AFSVolGetStatus . . . . . . . . . . 80
5.7.17 AFSVolSetIdsTypes . . . . . . . . . . 81
5.7.18 AFSVolSetDate . . . . . . . . . . 82
5.7.19 AFSVolListPartitions . . . . . . . . . . 83
5.7.20 AFSVolPartitionInfo . . . . . . . . . . 84
5.7.21 AFSVolListVolumes . . . . . . . . . . 85
5.7.22 AFSVolListOneVolume . . . . . . . . . . 86
5.7.23 AFSVolGetNthVolume . . . . . . . . . . 87
5.7.24 AFSVolMonitor . . . . . . . . . . 88

**Index . . . . . . . . . . i**

# Chapter 1

# Overview

## 1.1 Introduction

This document describes the architecture and interfaces for two of the important agents of the AFS distributed file system, the *Volume Server* and the *Volume Location Server*. The *Volume Server* allows operations affecting entire AFS *volumes* to be executed, while the *Volume Location Server* provides a lookup service for volumes, identifying the server or set of servers on which volume instances reside.

## 1.2 Volumes

### 1.2.1 Definition

The underlying concept manipulated by the two AFS servers examined by this document is the *volume*. Volumes are the basic mechanism for organizing the data stored within the file system. They provide the foundation for addressing, storing, and accessing file data, along with serving as the administrative units for replication, backup, quotas, and data motion between *File Server*s.

Specifically, a volume is a container for a hierarchy of files, a connected file system subtree. In this respect, a volume is much like a traditional UNIX file system partition. Like a partition, a volume can be *mounted* in the sense that the *root directory* of the volume can be named within another volume at an AFS mount point. The entire file system hierarchy is built up in this manner, using mount points to glue together the individual subtrees resident within each volume. The root of this hierarchy is then mounted by each AFS client machine using a conventional UNIX mount point within the workstation's local file system. By convention, this entryway into the AFS domain is mounted on the */afs* local directory. From a user's point of view, there is only a single mount point to the system; the internal mount points are generally transparent.

### 1.2.2 Volume Naming

There are two methods by which volumes may be named. The first is via a human-readable string name, and the second is via a 32-bit numerical identifier. Volume identifiers, whether string or numerical, must be unique within any given cell. AFS mount points may use either representation to specify the volume whose root directory is to be accessed at the given position. Internally, however, AFS agents use the numerical form of identification exclusively, having to translate names to the corresponding 32-bit value.

### 1.2.3 Volume Types

There are three basic volume types: read-write, read-only, and backup volumes.

- **Read-write:** The data in this volume may be both read and written by those clients authorized to do so.
- **Read-only:** It is possible to create one or more read-only snapshots of read-write volumes. The read-write volume serving as the source image is referred to as the *parent volume*. Each read-only *clone*, or *child*, instance must reside on a different UNIX disk partition than the other clones. Every clone instance generated from the same parent read-write volume has the identical volume name and numerical volume ID. This is the reason why no two clones may appear on the same disk partition, as there would be no way to differentiate the two. AFS clients are allowed to read files and directories from read-only volumes, but cannot overwrite them individually. However, it *is* possible to make changes to the read-write parent and then *release* the contents of the entire volume to all the read-only replicas. The release operation fails if it does not reach the appropriate replication sites.
- **Backup:** A backup volume is a special instance of a read-only volume. While it is also a read-only snapshot of a given read-write volume, only one instance is allowed to exist at any one time. Also, the backup volume must reside on the same partition as the parent read-write volume from which it was created. It is from a backup volume that the AFS backup system writes file system data to tape. In addition,

backup volumes may be mounted into the file tree just like the other volume types. In fact, by convention, the backup volume for each user's home directory subtree is typically mounted as *OldFiles* in that directory. If a user accidentally deletes a file that resides in the backup snapshot, the user may simply copy it out of the backup directly without the assistance of a system administrator, or any kind of tape restore operation.

Backup volume are implemented in a copy-on-write fashion. Thus, backup volumes may be envisioned as consisting of a set of pointers to the true data objects in the base read-write volume when they are first created. When a file is overwritten in the read-write version for the first time after the backup volume was created, the original data is physically written to the backup volume, breaking the copy-on-write link. With this mechanism, backup volumes maintain the image of the read-write volume at the time the snapshot was taken using the minimum amount of additional disk space.

## 1.3 Scope

This paper is a member of a documentation suite providing specifications of the operation and interfaces offered by the various AFS servers and agents. The scope of this work is to provide readers with a sufficiently detailed description of the *Volume Location Server* and the *Volume Server* so that they may construct client applications which call their RPC interface routines.

## 1.4 Document Layout

After this introductory portion of the document, Chapters 2 and 3 examine the architecture and RPC interface of the *Volume Location Server* and its replicated database. Similarly, Chapters 4 and 5 describe the architecture and RPC interface of the *Volume Server*.

# Chapter 2

# Volume Location Server **Architecture**

## 2.1 Introduction

The *Volume Location Server* allows AFS agents to query the location and basic status of volumes resident within the given cell. *Volume Location Server* functions may be invoked directly from authorized users via the vos utility.

This chapter briefly discusses various aspects of the *Volume Location Server*'s architecture. First, the need for volume location is examined, and the specific parties that call the *Volume Location Server* interface routines are identified. Then, the database maintained to provide volume location service, the Volume Location Database (VLDB), is examined. Finally, the *vlserver* process which implements the *Volume Location Server* is considered.

As with all AFS servers, the *Volume Location Server* uses the *Rx* remote procedure call package for communication with its clients.

## 2.2 The Need For Volume Location

The *Cache Manager* agent is the primary consumer of AFS volume location service, on which it is critically dependent for its own operation. The *Cache Manager* needs to map volume names or numerical identifiers to the set of *File Servers* on which its instances reside in order to satisfy the file system requests it is processing on behalf of it clients. Each time a *Cache Manager* encounters a mount point for which it does not have location information cached, it must acquire this information before the pathname resolution may be successfully completed. Once the *File Server* set is known for a particular volume, the *Cache Manager* may then select the proper site among them (e.g. choosing the single home for a read-write volume, or randomly selecting a site from a read-only volume's replication set) and begin addressing its file manipulation operations to that specific server.

While the *Cache Manager* consults the volume location service, it is not capable of *changing* the location of volumes and hence modifying the information contained therein. This capability to perform acts which change volume location is concentrated within the *Volume Server*. The *Volume Server* process running on each server machine manages all volume operations affecting that platform, including creations, deletions, and movements between servers. It must update the volume location database every time it performs one of these actions.

None of the other AFS system agents has a need to access the volume location database for its site. Surprisingly, this also applies to the *File Server* process. It is only aware of the specific set of volumes that reside on the set of physical disks directly attached to the machine on which they execute. It has no knowlege of the universe of volumes resident on other servers, either within its own cell or in foreign cells.

## 2.3 The VLDB

The Volume Location Database (VLDB) is used to allow AFS application programs to discover the location of any volume within its cell, along with select information about the nature and state of that volume. It is organized in a very straightforward fashion, and uses the *ubik* [4] [5] facility to to provide replication across multiple server sites.

### 2.3.1 Layout

The VLDB itself is a very simple structure, and synchronized copies may be maintained at two or more sites. Basically, each copy consists of header information, followed by a linear (yet unbounded) array of entries. There are several associated hash tables used to perform lookups into the VLDB. The first hash table looks up volume location information based on the volume's name. There are three other hash tables used for lookup, based on volume ID/type pairs, one for each possible volume type.

The VLDB for a large site may grow to contain tens of thousands of entries, so some attempts were made to make each entry as small as possible. For example, server addresses within VLDB entries are represented as single-byte indicies into a table containing the full longword IP addresses.

A free list is kept for deleted VLDB entries. The VLDB will not grow unless all the entries on the free list have been exhausted, keeping it as compact as possible.

### 2.3.2 Database Replication

The VLDB, along with other important AFS databases, may be replicated to multiple sites to improve its availability. The *ubik* replication package is used to implement this functionality for the VLDB. A full description of *ubik* and of the *quorum completion* algorithm it implements may be found in [4] and [5]. The basic abstraction provided by *ubik* is that of a disk file replicated to multiple server locations. One machine is considered to be the *synchronization site*, handling all write operations on the database file. Read operations may be directed to any of the active members of the *quorum*, namely a subset of the replication sites large enough to insure integrity across such failures as individual server crashes and network partitions. All of the quorum members participate in regular *elections* to determine the current synchronization site. The *ubik* algorithms allow server machines to enter and exit the quorum in an orderly and consistent fashion. All operations to one of these replicated "abstract files" are performed as part of a *transaction*. If all the related operations performed under a transaction are successful, then the transaction is *committed*, and the changes are made permanent. Otherwise, the transaction is *aborted*, and all of the operations for that transaction are undone.

## 2.4 The *vlserver* Process

The user-space *vlserver* process is in charge of providing volume location service for AFS clients. This program maintains the VLDB replica at its particular server, and cooperates with all other *vlserver* processes running in the given cell to propagate updates to the database. It implements the RPC interface defined in the *vldbint.xg* definition file for the *rxgen* RPC stub generator program. As part of its startup sequence, it must discover the VLDB version it has on its local disk, move to join the quorum of replication sites for the VLDB, and get the latest version if the one it came up with was out of date. Eventually, it will synchronize with the other VLDB replication sites, and it will begin accepting calls.

The *vlserver* program uses at most three *Rx* worker threads to listen for incoming *Volume Location Server* calls. It has a single, optional command line argument. If the string "`-noauth`" appears when the program is invoked, then *vlserver* will run in an unauthenticated mode where any individual is considered authorized to perform any VLDB operation. This mode is necessary when first bootstrapping an AFS installation.

# Chapter 3

# Volume Location Server **Interface**

## 3.1 Introduction

This chapter documents the API for the *Volume Location Server* facility, as defined by the vldbint.xg *Rxgen* interface file and the vldbint.h include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear here.

It is expected that *Volume Location Server* client programs run in user space, as does the associated vos volume utility. However, the kernel-resident *Cache Manager* agent also needs to call a subset of the *Volume Location Server*'s RPC interface routines. Thus, a second *Volume Location Server* interface is available, built exclusively to satisfy the *Cache Manager*'s limited needs. This subset interface is defined by the *afsvlint.xg Rxgen* interface file, and is examined in the final section of this chapter.

## 3.2 Constants

This section covers the basic constant definitions of interest to the *Volume Location Server* application programmer. These definitions appear in the *vldbint.h* file, automatically generated from the *vldbint.xg Rxgen* interface file, and in *vlserver.h*.

Each subsection is devoted to describing the constants falling into the following categories:

- Configuration and boundary quantities
- Update entry bits
- List-by-attribute bits
- Volume type indices
- States for `struct vlentry`
- States for `struct vldbentry`
- `ReleaseType` argument values
- Miscellaneous items

### 3.2.1 Configuration and Boundary Quantities

These constants define some basic system values, including configuration information.

| *Name* | *Value* | *Description* |
|---|---|---|
| MAXNAMELEN | 65 | Maximum size of various character strings, including volume name fields in structures and host names. |
| MAXNSERVERS | 8 | Maximum number of replication sites for a volume. |
| MAXTYPES | 3 | Maximum number of volume types. |
| VLDBVERSION | 1 | VLDB database version number. |
| HASHSIZE | 8,191 | Size of internal *Volume Location Server* volume name and volume ID hash tables. This must always be a prime number. |
| NULLO | 0 | Specifies a null pointer value. |
| VLDBALLOCCOUNT | 40 | Value used when allocating memory internally for VLDB entry records. |
| BADSERVERID | 255 | Illegal *Volume Location Server* host ID. |
| MAXSERVERID | 30 | Maximum number of servers appearing in the VLDB. |
| MAXSERVERFLAG | 0x80 | First unused flag value in such fields as `serverFlags` in `struct vldbentry` and `RepsitesNewFlags` in `struct VldbUpdateEntry` . |
| MAXPARTITIONID | 126 | Maximum number of AFS disk partitions for any one server. |
| MAXBUMPCOUNT | 0x7fffffff | Maximum interval that the current high-watermark value for a volume ID can be increased in one operation. |
| MAXLOCKTIME | 0x7fffffff | Maximum number of seconds that any VLDB entry can remained locked. |
| SIZE | 1,024 | Maximum size of the `name` field within a `struct VolumeDiskData` inside a `struct Volume`'s `header` field. |

### 3.2.2 Update Entry Bits

These constants define bit values for the `Mask` field in the `struct VldbUpdateEntry`. Specifically, setting these bits is equivalent to declaring that the corresponding field within an object of type `struct VldbUpdateEntry` has been set. For example, setting the `VLUPDATE_VOLUMENAME` flag in `Mask` indicates that the `name` field contains a valid value.

| *Name* | *Value* | *Description* |
|---|---|---|
| VLUPDATE_VOLUMENAME | 0x0001 | If set, indicates that the `name` field is valid. |
| VLUPDATE_VOLUMETYPE | 0x0002 | If set, indicates that the `volumeType` field is valid. |
| VLUPDATE_FLAGS | 0x0004 | If set, indicates that the `flags` field is valid. |
| VLUPDATE_READONLYID | 0x0008 | If set, indicates that the `ReadOnlyId` field is valid. |
| VLUPDATE_BACKUPID | 0x0010 | If set, indicates that the `BackupId` field is valid. |
| VLUPDATE_REPSITES | 0x0020 | If set, indicates that the `nModifiedRepsites` field is valid. |
| VLUPDATE_CLONEID | 0x0080 | If set, indicates that the `cloneId` field is valid. |
| VLUPDATE_REPS_DELETE | 0x0100 | Is the replica being deleted? |
| VLUPDATE_REPS_ADD | 0x0200 | Is the replica being added? |
| VLUPDATE_REPS_MODSERV | 0x0400 | Is the server part of the replica location correct? |
| VLUPDATE_REPS_MODPART | 0x0800 | Is the partition part of the replica location correct? |
| VLUPDATE_REPS_MODFLAG | 0x1000 | Various modification flag values. |

### 3.2.3 List-By-Attribute Bits

These constants define bit values for the `Mask` field in the `struct VldbListByAttributes` is to be used in a match. Specifically, setting these bits is equivalent to declaring that the corresponding field within an object of type `struct VldbListByAttributes` is set. For example, setting the VLLIST_SERVER flag in `Mask` indicates that the `server` field contains a valid value.

| *Name* | *Value* | *Description* |
|---|---|---|
| VLLIST_SERVER | 0x1 | If set, indicates that the `server` field is valid. |
| VLLIST_PARTITION | 0x2 | If set, indicates that the `partition` field is valid. |
| VLLIST_VOLUMETYPE | 0x4 | If set, indicates that the `volumetype` field is valid. |
| VLLIST_VOLUMEID | 0x8 | If set, indicates that the `volumeid` field is valid. |
| VLLIST_FLAG | 0x10 | If set, indicates that the `flag` field is valid. |

### 3.2.4 Volume Type Indices

These constants specify the order of entries in the `volumeid` array in an object of type `struct vldbentry`. They also identify the three different types of volumes in AFS.

| *Name* | *Value* | *Description* |
|---|---|---|
| RWVOL | 0 | Read-write volume. |
| ROVOL | 1 | Read-only volume. |
| BACKVOL | 2 | Backup volume. |

### 3.2.5 States for `struct vlentry`

The following constants appear in the `flags` field in objects of type `struct vlentry`. The first three values listed specify the state of the entry, while all the rest stamp the entry with the type of an ongoing volume operation, such as a move, clone, backup, deletion, and dump. These volume operations are the legal values to provide to the `voloper` parameter of the *VL_SetLock()* interface routine.

| *Name* | *Value* | *Description* |
|---|---|---|
| VLFREE | 0x1 | Entry is in the free list. |
| VLDELETED | 0x2 | Entry is soft-deleted. |
| VLLOCKED | 0x4 | Advisory lock held on the entry. |
| VLOP_MOVE | 0x10 | The associated volume is being moved between servers. |
| VLOP_RELEASE | 0x20 | The associated volume is being cloned to its replication sites. |
| VLOP_BACKUP | 0x40 | A backup volume is being created for the associated volume. |
| VLOP_DELETE | 0x80 | The associated volume is being deleted. |
| VLOP_DUMP | 0x100 | A dump is being taken of the associated volume. |

For convenience, the constant `VLOP_ALLOPERS` is defined as the inclusive `OR` of the above values from `VLOP_MOVE` through `VLOP_DUMP`.

### 3.2.6 States for `struct vldbentry`

Of the following constants, the first three appear in the `flags` field within an object of type `struct vldbentry`, advising of the existence of the basic volume types for the given volume, and hence the validity of the entries in the `volumeId` array field. The rest of the values provided in this table appear in the `serverFlags` array field, and apply to the instances of the volume appearing in the various replication sites.

This structure appears in numerous *Volume Location Server* interface calls, namely *VL_CreateEntry()*, *VL_GetEntryByID()*, *VL_GetEntryByName()*, *VL_ReplaceEntry()* and *VL_ListEntry()*.

| *Name* | *Value* | *Description* |
|---|---|---|
| VLF_RWEXISTS | 0x1000 | The read-write volume ID is valid. |
| VLF_ROEXISTS | 0x2000 | The read-only volume ID is valid. |
| VLF_BACKEXISTS | 0x4000 | The backup volume ID is valid. |
| VLSF_NEWREPSITE | 0x01 | Not used; originally intended to mark an entry as belonging to a partially-created volume instance. |
| VLSF_ROVOL | 0x02 | A read-only version of the volume appears at this server. |
| VLSF_RWVOL | 0x04 | A read-write version of the volume appears at this server. |
| VLSF_BACKVOL | 0x08 | A backup version of the volume appears at this server. |

### 3.2.7 `ReleaseType` Argument Values

The following values are used in the `ReleaseType` argument to various *Volume Location Server* interface routines, namely *VL_ReplaceEntry()*, *VL_UpdateEntry()* and *VL_ReleaseLock()*.

| *Name* | *Value* | *Description* |
|---|---|---|
| LOCKREL_TIMESTAMP | 1 | Is the `LockTimestamp` field valid? |
| LOCKREL_OPCODE | 2 | Are any of the bits valid in the `flags` field? |
| LOCKREL_AFSID | 4 | Is the `LockAfsId` field valid? |

### 3.2.8 Miscellaneous

Miscellaneous values.

| *Name* | *Value* | *Description* |
|---|---|---|
| VLREPSITE_NEW | 1 | Has a replication site gotten a new release of a volume? |

A synonym for this constant is `VLSF_NEWREPSITE`.

## 3.3 Structures and Typedefs

This section describes the major exported *Volume Location Server* data structures of interest to application programmers, along with the typedefs based upon those structures.

### 3.3.1 `struct vldbentry`

This structure represents an entry in the VLDB as made visible to *Volume Location Server* clients. It appears in numerous *Volume Location Server* interface calls, namely *VL_CreateEntry()*, *VL_GetEntryByID()*, *VL_GetEntryByName()*, *VL_ReplaceEntry()* and *VL_ListEntry()*.

**Fields**

**char name**[] - The string name for the volume, with a maximum length of `MAXNAMELEN` (65) characters, including the trailing null.

**long volumeType** - The volume type, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

**long nServers** - The number of servers that have an instance of this volume.

**long serverNumber**[] - An array of indices into the table of servers, identifying the sites holding an instance of this volume. There are at most `MAXNSERVERS` (8) of these server sites allowed by the *Volume Location Server*.

**long serverPartition**[] - An array of partition identifiers, corresponding directly to the `serverNumber` array, specifying the partition on which each of those volume instances is located. As with the `serverNumber` array, `serverPartition` has up to `MAXNSERVERS` (8) entries.

**long serverFlags**[] - This array holds one flag value for each of the servers in the previous arrays. Again, there are `MAXNSERVERS` (8) slots in this array.

**u_long volumeId**[] - An array of volume IDs, one for each volume type. There are `MAXTYPES` slots in this array.

**long cloneId** - This field is used during a cloning operation.

**long flags** - Flags concerning the status of the fields within this structure; see Section 3.2.6 for the bit values that apply.

### 3.3.2 struct vlentry

This structure is used internally by the *Volume Location Server* to fully represent a VLDB entry. The client-visible `struct vldbentry` represents merely a subset of the information contained herein.

**Fields**

- **u_long volumeId**[] - An array of volume IDs, one for each of the `MAXTYPES` of volume types.
- **long flags** - Flags concerning the status of the fields within this structure; see Section 3.2.6 for the bit values that apply.
- **long LockAfsId** - The individual who locked the entry. This feature has not yet been implemented.
- **long LockTimestamp** - Time stamp on the entry lock.
- **long cloneId** - This field is used during a cloning operation.
- **long AssociatedChain** - Pointer to the linked list of associated VLDB entries.
- **long nextIdHash**[] - Array of `MAXTYPES` next pointers for the ID hash table pointer, one for each related volume ID.
- **long nextNameHash** - Next pointer for the volume name hash table.
- **long spares1**[] - Two longword spare fields.
- **char name**[] - The volume's string name, with a maximum of `MAXNAMELEN` (65) characters, including the trailing null.
- **u_char volumeType** - The volume's type, one of `RWVOL`, `ROVOL`, or `BACKVOL`.
- **u_char serverNumber**[] - An array of indices into the table of servers, identifying the sites holding an instance of this volume. There are at most `MAXNSERVERS` (8) of these server sites allowed by the *Volume Location Server*.
- **u_char serverPartition**[] - An array of partition identifiers, corresponding directly to the `serverNumber` array, specifying the partition on which each of those volume instances is located. As with the `serverNumber` array, `serverPartition` has up to `MAXNSERVERS` (8) entries.
- **u_char serverFlags**[] - This array holds one flag value for each of the servers in the previous arrays. Again, there are `MAXNSERVERS` (8) slots in this array.
- **u_char RefCount** - Only valid for read-write volumes, this field serves as a reference count, basically the number of dependent children volumes.
- **char spares2**[] - This field is used for 32-bit alignment.

### 3.3.3 struct vital_vlheader

This structure defines the leading section of the VLDB header, of type `struct vlheader`. It contains frequently-used global variables and general statistics information.

**Fields**

- **long vldbversion** - The VLDB version number. This field must appear first in the structure.
- **long headersize** - The total number of bytes in the header.
- **long freePtr** - Pointer to the first free enry in the free list, if any.
- **long eofPtr** - Pointer to the first free byte in the header file.
- **long allocs** - The total number of calls to the internal *AllocBlock()* function directed at this file.
- **long frees** - The total number of calls to the internal *FreeBlock()* function directed at this file.
- **long MaxVolumeId** - The largest volume ID ever granted for this cell.
- **long totalEntries**[] - The total number of VLDB entries by volume type in the VLDB. This array has `MAXTYPES` slots, one for each volume type.

### 3.3.4 struct vlheader

This is the layout of the information stored in the VLDB header. Notice it includes an object of type `struct vital_vlheader` described above (see Section 3.3.3) as the first field.

**Fields**

- **struct vital_vlheader vital_header** - Holds critical VLDB header information.
- **u_long IpMappedAddr**[] - Keeps `MAXSERVERID+1` mappings of IP addresses to relative ones.
- **long VolnameHash**[] - The volume name hash table, with `HASHSIZE` slots.
- **long VolidHash**[][] - The volume ID hash table. The first dimension in this array selects which of the `MAXTYPES` volume types is desired, and the second dimension actually implements the `HASHSIZE` hash table buckets for the given volume type.

### 3.3.5 struct VldbUpdateEntry

This structure is used as an argument to the *VL_UpdateEntry()* routine (see Section 3.6.7). Please note that multiple entries can be updated at once by setting the appropriate `Mask` bits. The bit values for this purpose are defined in Section 3.2.2.

**Fields**

- **u_long Mask** - Bit values determining which fields are to be affected by the update operation.
- **char name**[] - The volume name, up to `MAXNAMELEN` (65) characters including the trailing null.
- **long volumeType** - The volume type.
- **long flags** - This field is used in conjuction with `Mask` (in fact, one of the `Mask` bits determines if this field is valid) to choose the valid fields in this record.
- **u_long ReadOnlyId** - The read-only ID.
- **u_long BackupId** - The backup ID.
- **long cloneId** - The clone ID.
- **long nModifiedRepsites** - Number of replication sites whose entry is to be changed as below.
- **u_long RepsitesMask**[] - Array of bit masks applying to the up to `MAXNSERVERS` (8) replication sites involved.
- **long RepsitesTargetServer**[] - Array of target servers for the operation, at most `MAXNSERVERS` (8) of them.
- **long RepsitesTargetPart**[] - Array of target server partitions for the operation, at most `MAXNSERVERS` (8) of them.
- **long RepsitesNewServer**[] - Array of new server sites, at most `MAXNSERVERS` (8) of them.
- **long RepsitesNewPart**[] - Array of new server partitions for the operation, at most `MAXNSERVERS` (8) of them.
- **long RepsitesNewFlags**[] - Flags applying to each of the new sites, at most `MAXNSERVERS` (8) of them.

### 3.3.6 struct VldbListByAttributes

This structure is used by the *VL_ListAttributes()* routine (see Section 3.6.11).

**Fields**

**u_long Mask** - Bit mask used to select the following attribute fields on which to match.

**long server** - The server address to match.

**long partition** - The partition ID to match.

**long volumetype** - The volume type to match.

**long volumeid** - The volume ID to match.

**long flag** - Flags concerning these values.

### 3.3.7 struct single_vldbentry

This structure is used to construct the `vldblist` object (See Section 3.3.12), which basically generates a queueable (singly-linked) version of `struct vldbentry`.

**Fields**

**vldbentry VldbEntry** - The VLDB entry to be queued.

**vldblist next_vldb** - The next pointer in the list.

### 3.3.8 struct vldb_list

This structure defines the item returned in linked list form from the *VL_LinkedList()* function (see Section 3.6.12). This same object is also returned in bulk form in calls to the *VL_ListAttributes()* routine (see Section 3.6.11).

**Fields**

**vldblist node** - The body of the first object in the linked list.

### 3.3.9 struct vldstats

This structure defines fields to record statistics on opcode hit frequency. The `MAX_NUMBER_OPCODES` constant has been defined as the maximum number of opcodes supported by this structure, and is set to 30.

**Fields**

- **unsigned long start_time** - Clock time when opcode statistics were last cleared.
- **long requests**[] - Number of requests received for each of the `MAX_NUMBER_OPCODES` opcode types.
- **long aborts**[] - Number of aborts experienced for each of the `MAX_NUMBER_OPCODES` opcode types.
- **long reserved**[] - These five longword fields are reserved for future use.

### 3.3.10 bulk

```
typedef opaque bulk<DEFAULTBULK>;
```

This typedef may be used to transfer an uninterpreted set of bytes across the *Volume Location Server* interface. It may carry up to `DEFAULTBULK` (10,000) bytes.

**Fields**

- `bulk_len` - The number of bytes contained within the data pointed to by the next field.
- `bulk_val` - A pointer to a sequence of `bulk_len` bytes.

### 3.3.11 bulkentries

```
typedef vldbentry bulkentries<>;
```

This typedef is used to transfer an unbounded number of `struct vldbentry` objects. It appears in the parameter list for the *VL_ListAttributes()* interface function.

**Fields**

- `bulkentries_len` - The number of `vldbentry` structures contained within the data pointed to by the next field.
- `bulkentries_val` - A pointer to a sequence of `bulkentries_len` `vldbentry` structures.

### 3.3.12 vldblist

```
typedef struct single_vldbentry *vldblist;
```

This typedef defines a queueable `struct vldbentry` object, referenced by the `single_vldbentry` typedef as well as `struct vldb_list`.

### 3.3.13 vlheader

```
typedef struct vlheader vlheader;
```

This typedef provides a short name for objects of type `struct vlheader` (see Section 3.3.4).

### 3.3.14 vlentry

```
typedef struct vlentry vlentry;
```

This typedef provides a short name for objects of type `struct vlentry` (see Section 3.3.2).

## 3.4 Error Codes

This section covers the set of error codes exported by the *Volume Location Server*, displaying the printable phrases with which they are associated.

| *Name* | *Value* | *Description* |
|---|---|---|
| VL_IDEXIST | (363520L) | Volume Id entry exists in vl database. |
| VL_IO | (363521L) | I/O related error. |
| VL_NAMEEXIST | (363522L) | Volume name entry exists in vl database. |
| VL_CREATEFAIL | (363523L) | Internal creation failure. |
| VL_NOENT | (363524L) | No such entry. |
| VL_EMPTY | (363525L) | Vl database is empty. |
| VL_ENTDELETED | (363526L) | Entry is deleted (soft delete). |
| VL_BADNAME | (363527L) | Volume name is illegal. |
| VL_BADINDEX | (363528L) | Index is out of range. |
| VL_BADVOLTYPE | (363529L) | Bad volume type. |
| VL_BADSERVER | (363530L) | Illegal server number (out of range). |
| VL_BADPARTITION | (363531L) | Bad partition number. |
| VL_REPSFULL | (363532L) | Run out of space for Replication sites. |
| VL_NOREPSERVER | (363533L) | No such Replication server site exists. |
| VL_DUPREPSERVER | (363534L) | Replication site already exists. |
| VL_RWNOTFOUND | (363535L) | Parent R/W entry not found. |
| VL_BADREFCOUNT | (363536L) | Illegal Reference Count number. |
| VL_SIZEEXCEEDED | (363537L) | Vl size for attributes exceeded. |
| VL_BADENTRY | (363538L) | Bad incoming vl entry. |
| VL_BADVOLIDBUMP | (363539L) | Illegal max volid increment. |
| VL_IDALREADYHASHED | (363540L) | RO/BACK id already hashed. |
| VL_ENTRYLOCKED | (363541L) | Vl entry is already locked. |
| VL_BADVOLOPER | (363542L) | Bad volume operation code. |
| VL_BADRELLOCKTYPE | (363543L) | Bad release lock type. |
| VL_RERELEASE | (363544L) | Status report: last release was aborted. |
| VL_BADSERVERFLAG | (363545L) | Invalid replication site server flag. |
| VL_PERM | (363546L) | No permission access. |
| VL_NOMEM | (363547L) | malloc(realloc) failed to alloc enough memory. |

## 3.5 Macros

The *Volume Location Server* defines a small number of macros, as described in this section. They are used to update the internal statistics variables and to compute offsets into character strings. All of these macros really refer to internal operations, and strictly speaking should not be exposed in this interface.

### 3.5.1 *COUNT_REQ()*

```
#define COUNT_REQ(op)
    static int this_op = op-VL_LOWEST_OPCODE;
    dynamic_statistics.requests[this_op]++
```

Bump the appropriate entry in the variable maintaining opcode usage statistics for the *Volume Location Server*. Note that a static variable is set up to record `this_op`, namely the index into the opcode monitoring array. This static variable is used by the related *COUNT_ABO()* macro defined below.

### 3.5.2 *COUNT_ABO()*

```
#define COUNT_ABO dynamic_statistics.aborts[this_op]++
```

Bump the appropriate entry in the variable maintaining opcode abort statistics for the *Volume Location Server*. Note that this macro does not take any arguemnts. It expects to find a `this_op` variable in its environment, and thus depends on its related macro, *COUNT_REQ()* to define that variable.

### 3.5.3 *DOFFSET()*

```
#define DOFFSET(abase, astr, aitem)
    ((abase)+(((char *)(aitem)) - ((char *)(astr))))
```

Compute the byte offset of charcter object `aitem` within the enclosing object `astr`, also expressed as a character-based object, then offset the resulting address by `abase`. This macro is used ot compute locations within the VLDB when actually writing out information.

## 3.6 Functions

This section covers the *Volume Location Server* RPC interface routines. The majority of them are generated from the *vldbint.xg Rxgen* file, and are meant to be used by user-space agents. There is also a subset interface definition provided in the *afsvlint.xg Rxgen* file. These routines, described in Section 3.7, are meant to be used by a kernel-space agent when dealing with the *Volume Location Server*; in particular, they are called by the *Cache Manager*.

### 3.6.1 VL_CreateEntry — Create a VLDB entry

```
int VL_CreateEntry(IN struct rx_connection *z_conn,
                   IN vldbentry *newentry)
```

#### Description

This function creates a new entry in the VLDB, as specified in the `newentry` argument. Both the name and numerical ID of the new volume must be unique (e.g., it must not already appear in the VLDB). For non-read-write entries, the read-write parent volume is accessed so that its reference count can be updated, and the new entry is added to the parent's chain of associated entries.

The VLDB is write-locked for the duration of this operation.

#### Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_NAMEEXIST` The volume name already appears in the VLDB.

`VL_CREATEFAIL` Space for the new entry cannot be allocated within the VLDB.

`VL_BADNAME` The volume name is invalid.

`VL_BADVOLTYPE` The volume type is invalid.

`VL_BADSERVER` The indicated server information is invalid.

`VL_BADPARTITION` The indicated partition information is invalid.

`VL_BADSERVERFLAG` The server flag field is invalid.

`VL_IO` An error occurred while writing to the VLDB.

### 3.6.2 VL_DeleteEntry — Delete a VLDB entry

```
int VL_DeleteEntry(IN struct rx_connection *z_conn,
                   IN long Volid,
                   IN long voltype)
```

#### Description

Delete the entry matching the given volume identifier and volume type as specified in the `Volid` and `voltype` arguments. For a read-write entry whose reference count is greater than 1, the entry is not actually deleted, since at least one child (read-only or backup) volume still depends on it. For cases of non-read-write volumes, the parent's reference count and associated chains are updated.

If the associated VLDB entry is already marked as deleted (i.e., its `flags` field has the `VLDELETED` bit set), then no further action is taken, and `VL_ENTDELETED` is returned. The VLDB is write-locked for the duration of this operation.

#### Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_ENTDELETED` The given VLDB entry has already been marked as deleted.

`VL_IO` An error occurred while writing to the VLDB.

### 3.6.3 VL_GetEntryByID — Get VLDB entry by volume ID/type

```
int VL_GetEntryByID(IN struct rx_connection *z_conn,
                    IN long Volid,
                    IN long voltype,
                    OUT vldbentry *entry)
```

#### Description

Given a volume's numerical identifier (`Volid`) and type (`voltype`), return a pointer to the entry in the VLDB describing the given volume instance.

The VLDB is read-locked for the duration of this operation.

#### Error Codes

- `VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.
- `VL_NOENT` This volume instance does not appear in the VLDB.
- `VL_ENTDELETED` The given VLDB entry has already been marked as deleted.

### 3.6.4 VL_GetEntryByName — Get VLDB entry by volume name

```
int VL_GetEntryByName(IN struct rx_connection *z_conn,
                      IN char *volumename,
                      OUT vldbentry *entry)
```

#### Description

Given the volume name in the `volumename` parameter, return a pointer to the entry in the VLDB describing the given volume. The name in `volumename` may be no longer than `MAXNAMELEN` (65) characters, including the trailing null. Note that it is legal to use the volume's numerical identifier (in string form) as the volume name.

The VLDB is read-locked for the duration of this operation.

This function is closely related to the *VL_GetEntryByID()* routine, as might be expected. In fact, the by-ID routine is called if the volume name provided in `volumename` is the string version of the volume's numerical identifier.

#### Error Codes

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_ENTDELETED` The given VLDB entry has already been marked as deleted.

`VL_BADNAME` The volume name is invalid.

### 3.6.5 VL_GetNewVolumeId — Generate a new volume ID

```
int VL_GetNewVolumeId(IN struct rx_connection *z_conn,
                      IN long bumpcount,
                      OUT long *newvolumid)
```

#### Description

Acquire `bumpcount` unused, consecutively-numbered volume identifiers from the *Volume Location Server*. The lowest-numbered of the newly-acquired set is placed in the `newvolumid` argument. The largest number of volume IDs that may be generated with any one call is bounded by the `MAXBUMPCOUNT` constant defined in Section 3.2.1. Currently, there is (effectively) no restriction on the number of volume identifiers that may thus be reserved in a single call.

The VLDB is write-locked for the duration of this operation.

#### Error Codes

- `VL_PERM` The caller is not authorized to execute this function.
- `VL_BADVOLIDBUMP` The value of the `bumpcount` parameter exceeds the system limit of `MAXBUMPCOUNT`.
- `VL_IO` An error occurred while writing to the VLDB.

### 3.6.6 VL_ReplaceEntry — Replace entire contents of VLDB entry

```
int VL_ReplaceEntry(IN struct rx_connection *z_conn,
                    IN long Volid,
                    IN long voltype,
                    IN vldbentry *newentry,
                    IN long ReleaseType)
```

#### Description

Perform a wholesale replacement of the VLDB entry corresponding to the volume instance whose identifier is `Volid` and type `voltype` with the information contained in the `newentry` argument. Individual VLDB entry fields cannot be selectively changed while the others are preserved; *VL_UpdateEntry()* should be used for this objective. The permissible values for the `ReleaseType` parameter are defined in Section 3.2.7.

The VLDB is write-locked for the duration of this operation. All of the hash tables impacted are brought up to date to incorporate the new information.

#### Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_BADRELLOCKTYPE` An illegal release lock has been specified by the `ReleaseType` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_BADENTRY` An attempt was made to change a read-write volume ID.

`VL_IO` An error occurred while writing to the VLDB.

### 3.6.7 VL_UpdateEntry — Update contents of VLDB entry

```
int VL_UpdateEntry(IN struct rx_connection *z_conn,
                   IN long Volid,
                   IN long voltype,
                   IN VldbUpdateEntry *UpdateEntry,
                   IN long ReleaseType)
```

#### Description

Update the VLDB entry corresponding to the volume instance whose identifier is `Volid` and type `voltype` with the information contained in the `UpdateEntry` argument. Most of the entry's fields can be modified in a single call to *VL_UpdateEntry()*. The `Mask` field within the `UpdateEntry` parameter selects the fields to update with the values stored within the other `UpdateEntry` fields. Permissible values for the `ReleaseType` parameter are defined in Section 3.2.7.

The VLDB is write-locked for the duration of this operation.

#### Error Codes

`VL_PERM` The caller is not authorized to execute this function.

`VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.

`VL_BADRELLOCKTYPE` An illegal release lock has been specified by the `ReleaseType` argument.

`VL_NOENT` This volume instance does not appear in the VLDB.

`VL_IO` An error occurred while writing to the VLDB.

### 3.6.8 VL_SetLock — Lock VLDB entry

```
int VL_SetLock(IN struct rx_connection *z_conn,
               IN long Volid,
               IN long voltype,
               IN long voloper)
```

#### Description

Lock the VLDB entry matching the given volume ID (`Volid`) and type (`voltype`) for volume operation `voloper` (e.g., `VLOP_MOVE` and `VLOP_RELEASE`). If the entry is currently unlocked, then its `LockTimestamp` will be zero. If the lock is obtained, the given `voloper` is stamped into the `flags` field, and the `LockTimestamp` is set to the time of the call.

Note: when the caller attempts to lock the entry for a release operation, special care is taken to abort the operation if the entry has already been locked for this operation, and the existing lock has timed out. In this case, `VL_SetLock()` returns `VL_RERELEASE`.

The VLDB is write-locked for the duration of this operation.

#### Error Codes

- `VL_PERM` The caller is not authorized to execute this function.
- `VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.
- `VL_BADVOLOPER` An illegal volume operation was specified in the `voloper` argument. Legal values are defined in the latter part of the table in Section 3.2.5.
- `VL_ENTDELETED` The given VLDB entry has already been marked as deleted.
- `VL_ENTRYLOCKED` The given VLDB entry has already been locked (which has not yet timed out).
- `VL_RERELEASE` A VLDB entry locked for release has timed out, and the caller also wanted to perform a release operation on it.
- `VL_IO` An error was experienced while attempting to write to the VLDB.

### 3.6.9 VL_ReleaseLock — Unlock VLDB entry

```
int VL_ReleaseLock(IN struct rx_connection *z_conn,
                   IN long Volid,
                   IN long voltype,
                   IN long ReleaseType)
```

#### Description

Unlock the VLDB entry matching the given volume ID (`Volid`) and type (`voltype`). The `ReleaseType` argument determines which VLDB entry fields from `flags` and `LockAfsId` will be cleared along with the lock timestamp in `LockTimestamp`. Permissible values for the `ReleaseType` parameter are defined in Section 3.2.7.

The VLDB is write-locked for the duration of this operation.

#### Error Codes

- `VL_PERM` The caller is not authorized to execute this function.
- `VL_BADVOLTYPE` An illegal volume type has been specified by the `voltype` argument.
- `VL_BADRELLOCKTYPE` An illegal release lock has been specified by the `ReleaseType` argument.
- `VL_NOENT` This volume instance does not appear in the VLDB.
- `VL_ENTDELETED` The given VLDB entry has already been marked as deleted.
- `VL_IO` An error was experienced while attempting to write to the VLDB.

### 3.6.10 VL_ListEntry — Get contents of VLDB via index

```
int VL_ListEntry(IN struct rx_connection *z_conn,
                 IN long previous_index,
                 OUT long *count,
                 OUT long *next_index,
                 OUT vldbentry *entry)
```

#### Description

This function assists in the task of enumerating the contents of the VLDB. Given an index into the database, `previous_index`, this call return the single VLDB entry at that offset, placing it in the `entry` argument. The number of VLDB entries left to list is placed in `count`, and the index of the next entry to request is returned in `next_index`. If an illegal index is provided, `count` is set to -1.

The VLDB is read-locked for the duration of this operation.

#### Error Codes

`---` None.

### 3.6.11 VL_ListAttributes — List all VLDB entry matching given attributes, single return object

```
int VL_ListAttributes(IN struct rx_connection *z_conn,
                      IN VldbListByAttributes *attributes,
                      OUT long *nentries,
                      OUT bulkentries *blkentries)
```

#### Description

Retrieve all the VLDB entries that match the attributes listed in the `attributes` parameter, placing them in the `blkentries` object. The number of matching entries is placed in `nentries`. Matching can be done by server number, partition, volume type, flag, or volume ID. The legal values to use in the `attributes` argument are listed in Section 3.2.3. Note that if the `VLLIST_VOLUMEID` bit is set in `attributes`, all other bit values are ignored and the volume ID provided is the sole search criterion.

The VLDB is read-locked for the duration of this operation.

Note that *VL_ListAttributes()* is a potentially expensive function, as sequential search through all of the VLDB entries is performed in most cases.

#### Error Codes

- `VL_NOMEM` Memory for the `blkentries` object could not be allocated.
- `VL_NOENT` This specified volume instance does not appear in the VLDB.
- `VL_SIZEEXCEEDED` Ran out of room in the `blkentries` object.
- `VL_IO` Error while reading from the VLDB.

### 3.6.12 VL_LinkedList — List all VLDB entry matching given attributes, linked list return object

```
int VL_LinkedList(IN struct rx_connection *z_conn,
                  IN VldbListByAttributes *attributes,
                  OUT long *nentries,
                  OUT vldb_list *linkedentries)
```

#### Description

Retrieve all the VLDB entries that match the attributes listed in the `attributes` parameter, creating a linked list of entries based in the `linkedentries` object. The number of matching entries is placed in `nentries`. Matching can be done by server number, partition, volume type, flag, or volume ID. The legal values to use in the `attributes` argument are listed in Section 3.2.3. Note that if the `VLLIST_VOLUMEID` bit is set in `attributes`, all other bit values are ignored and the volume ID provided is the sole search criterion.

The *VL_LinkedList()* function is identical to the *VL_ListAttributes()*, except for the method of delivering the VLDB entries to the caller.

The VLDB is read-locked for the duration of this operation.

#### Error Codes

- `VL_NOMEM` Memory for an entry in the list based at `linkedentries` object could not be allocated.
- `VL_NOENT` This specified volume instance does not appear in the VLDB.
- `VL_SIZEEXCEEDED` Ran out of room in the current list object.
- `VL_IO` Error while reading from the VLDB.

### 3.6.13 VL_GetStats — Get *Volume Location Server* statistics

```
int VL_GetStats(IN struct rx_connection *z_conn,
                OUT vldstats *stats,
                OUT vital_vlheader *vital_header)
```

#### Description

Collect the different types of VLDB statistics. Part of the VLDB header is returned in `vital_header`, which includes such information as the number of allocations and frees performed, and the next volume ID to be allocated. The dynamic per-operation stats are returned in the `stats` argument, reporting the number and types of operations and aborts.

The VLDB is read-locked for the duration of this operation.

#### Error Codes

`VL_PERM` The caller is not authorized to execute this function.

### 3.6.14 **VL_Probe** — Verify *Volume Location Server* connectivity/status

`int` **VL_Probe**`(IN struct rx_connection *z_conn)`

#### Description

This routine serves a "pinging" function to determine whether the *Volume Location Server* is still running. If this call succeeds, then the *Volume Location Server* is shown to be capable of responding to RPCs, thus confirming connectivity and basic operation.

The VLDB is *not* locked for this operation.

#### Error Codes

`---` None.

## 3.7 Kernel Interface Subset

The interface described by this document so far applies to user-level clients, such as the `vos` utility. However, some volume location operations must be performed from within the kernel. Specifically, the *Cache Manager* must find out where volumes reside and otherwise gather information about them in order to conduct its business with the *File Servers* holding them. In order to support *Volume Location Server* interconnection for agents operating within the kernel, the *afsvlint.xg Rxgen* interface was built. It is a minimal subset of the user-level *vldbint.xg* definition. Within *afsvlint.xg*, there are duplicate definitions for such constants as `MAXNAMELEN`, `MAXNSERVERS`, `MAXTYPES`, `VLF_RWEXISTS`, `VLF_ROEXISTS`, `VLF_BACKEXISTS`, `VLSF_NEWREPSITE`, `VLSF_ROVOL`, `VLSF_RWVOL`, and `VLSF_BACKVOL`. Since the only operations the *Cache Manager* must perform are volume location given a specific volume ID or name, and to find out about unresponsive *Volume Location Servers*, the following interface routines are duplicated in *afsvlint.xg*, along with the `struct vldbentry` declaration:

- *VL_GetEntryByID()*
- *VL_GetEntryByName()*
- *VL_Probe()*

# Chapter 4

# Volume Server **Architecture**

## 4.1 Introduction

The *Volume Server* allows administrative tasks and probes to be performed on the set of AFS volumes residing on the machine on which it is running. As described in Chapter 2, a distributed database holding volume location info, the VLDB, is used by client applications to locate these volumes. *Volume Server* functions are typically invoked either directly from authorized users via the `vos` utility or by the AFS backup system.

This chapter briefly discusses various aspects of the *Volume Server*'s architecture. First, the high-level on-disk representation of volumes is covered. Then, the transactions used in conjuction with volume operations are examined. Then, the program implementing the *Volume Server*, *volserver*, is considered. The nature and format of the log file kept by the *Volume Server* rounds out the description.

As with all AFS servers, the *Volume Server* uses the *Rx* remote procedure call package for communication with its clients.

## 4.2 Disk Representation

For each volume on an AFS partition, there exists a file visible in the UNIX name space which describes the contents of that volume. By convention, each of these files is named by concatenating a prefix string, "*V*", the numerical volume ID, and the postfix string "*.vol*". Thus, file *V0536870918.vol* describes the volume whose numerical ID is 0536870918. Internally, each per-volume descriptor file has such fields as a version number, the numerical volume ID, and the numerical parent ID (useful for read-only or backup volumes). It also has a list of related inodes, namely files which are *not* visible from the UNIX name space (i.e., they do not appear as entries in any UNIX directory object). The set of important related inodes are:

- **Volume info inode:** This field identifies the inode which hosts the on-disk representation of the volume's header. It is very similar to the information pointed to by the `volume` field of the `struct volser_trans` defined in Section 5.4.1, recording important status information for the volume.
- **Large vnode index inode:** This field identifies the inode which holds the list of vnode identifiers for all directory objects residing within the volume. These are "large" since they must also hold the Access Control List (ACL) information for the given AFS directory.
- **Small vnode index inode:** This field identifies the inode which holds the list of vnode identifiers for all non-directory objects hosted by the volume.

All of the actual files and directories residing within an AFS volume, as identified by the contents of the large and small vnode index inodes, are also free-floating inodes, not appearing in the conventional UNIX name space. This is the reason the vendor-supplied `fsck` program should **not** be run on partitions containing AFS volumes. Since the inodes making up AFS files and directories, as well as the inodes serving as volume indices for them, are not mapped to any directory, the standard `fsck` program would throw away all of these "unreferenced" inodes. Thus, a special version of `fsck` is provided that recognizes partitions containing AFS volumes as well as standard UNIX partitions.

## 4.3 Transactions

Each individual volume operation is carried out by the *Volume Server* as a *transaction*, but not in the atomic sense of the word. Logically, creating a *Volume Server* transaction can be equated with performing an "exclusive open" on the given volume before beginning the actual work of the desired volume operation. No other *Volume Server* (or *File Server*) operation is allowed on the opened volume until the transaction is terminated. Thus, transactions in the context of the *Volume Server* serve to provide mutual exclusion without any of the normal atomicity guarantees. Volumes maintain enough internal state to enable recovery from interrupted or failed operations via use of the `salvager` program. Whenever volume inconsistencies are detected, this `salvager` program is run, which then attempts to correct the problem.

Volume transactions have timeouts associated with them. This guarantees that the death of the agent performing a given volume operation cannot result in the volume being permanently removed from circulation. There are actually two timeout periods defined for a volume transaction. The first is the *warning time*, defined to be 5 minutes. If a transaction lasts for more than this time period without making progress, the *Volume Server* prints a warning message to its log file (see Section 4.5). The second time value associated with a volume transaction is the *hard timeout*, defined to occur 10 minutes after any progress has been made on the given operation. After this period, the transaction will be unconditionally deleted, and the volume freed for any other operations. Transactions are reference-counted. Progress will be deemed to have occurred for a transaction, and its internal timeclock field will be updated, when:

1. The transaction is first created.
2. A reference is made to the transaction, causing the *Volume Server* to look it up in its internal tables.
3. The transaction's reference count is decremented.

## 4.4 The *volserver* Process

The *volserver* user-level program is run on every AFS server machine, and implements the *Volume Server* agent. It is responsible for providing the *Volume Server* interface as defined by the *volint.xg Rxgen* file.

The *volserver* process defines and launches five threads to perform the bulk of its duties. One thread implements a background daemon whose job it is to garbage-collect timed-out transaction structures. The other four threads are RPC interface listeners, primed to accept remote procedure calls and thus perform the defined set of volume operations.

Certain non-standard configuration settings are made for the RPC subsystem by the *volserver* program. For example, it chooses to extend the length of time that an *Rx* connection may remain idle from the default 12 seconds to 120 seconds. The reasoning here is that certain volume operations may take longer than 12 seconds of processing time on the server, and thus the default setting for the connection timeout value would incorrectly terminate an RPC when in fact it was proceeding normally and correctly.

The *volserver* program takes a single, optional command line argument. If a positive integer value is provided on the command line, then it shall be used to set the debugging level within the *Volume Server*. By default, a value of zero is used, specifying that no special debugging output will be generated and fed to the *Volume Server* log file described below.

## 4.5 Log File

The *Volume Server* keeps a log file, recording the set of events of special interest it has encountered. The file is named `VolserLog`, and is stored in the */usr/afs/logs* directory on the local disk of the server machine on which the *Volume Server* runs. This is a human-readable file, with every entry time-stamped.

Whenever the *volserver* program restarts, it renames the current *VolserLog* file to *VolserLog.old*, and starts up a fresh log. A properly-authorized individual can easily inspect the log file residing on any given server machine. This is made possible by the *BOS Server* AFS agent running on the machine, which allows the contents of this file to be fetched and displayed on the caller's machine via the `bos getlog` command.

An excerpt from a *Volume Server* log file follows below. The numbers appearing in square brackets at the beginning of each line have been inserted so that we may reference the individual lines of the log excerpt in the following paragraph.

```
[1] Wed May  8 06:03:00 1991 AttachVolume: Error attaching volume
        /vicepd/V1969547815.vol; volume needs salvage
[2] Wed May  8 06:03:01 1991 Volser: ListVolumes: Could not attach volume 1969547815
[3] Wed May  8 07:36:13 1991 Volser: Clone: Cloning volume 1969541499 to new
        volume 1969541501
[4] Wed May  8 11:25:05 1991 AttachVolume: Cannot read volume header
        /vicepd/V1969547415.vol
[5] Wed May  8 11:25:06 1991 Volser: CreateVolume: volume 1969547415
        (bld.dce.s3.dv.pmax_ul3) created
```

Line [1] indicates that the volume whose numerical ID is 1969547815 could not be attached on partition `/vicepd`. This error is probably the result of an aborted transaction which left the volume in an inconsistent state, or by actual damage to the volume structure or data. In this case, the *Volume Server* recommends that the `salvager` program be run on this volume to restore its integrity. Line [2] records the operation which revealed this situation, namely the invocation of an *AFSVolListVolumes()* RPC.

Line [4] reveals that the volume header file for a specific volume could not be read. Line [5], as with line [2] in the above paragraph, indicates why this is true. Someone had called the *AFSVolCreateVolume()* interface function, and as a precaution, the *Volume Server* first checked to see if such a volume was already present by attempting to read its header.

Thus verifying that the volume did not previously exist, the *Volume Server* allowed the *AFSVolCreateVolume()* call to continue its processing, creating and initializing the proper volume file, *V1969547415.vol*, and the associated header and index inodes.

# Chapter 5

# Volume Server **Interface**

## 5.1 Introduction

This chapter documents the API for the *Volume Server* facility, as defined by the *volint.xg Rxgen* interface file and the *volser.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear here.

## 5.2 Constants

This section covers the basic constant definitions of interest to the *Volume Server* application programmer. These definitions appear in the *volint.h* file, automatically generated from the *volint.xg Rxgen* interface file, and in *volser.h*.

Each subsection is devoted to describing the constants falling into the following categories:

- Configuration and boundary values
- Interface routine opcodes
- Transaction Flags
- Volume Types
- LWP State

- States for `struct vldbentry`
- Validity Checks
- Miscellaneous

### 5.2.1 Configuration and Boundary Values

These constants define some basic system configuration values, along with such things as maximum sizes of important arrays.

| *Name* | *Value* | *Description* |
|---|---|---|
| `MyPort` | 5,003 | The *Rx* UDP port on which the *Volume Server* service may be found. |
| `NameLen` | 80 | Used by the `vos` utility to define maximum lengths for internal filename variables. |
| `VLDB_MAXSERVERS` | 10 | Maximum number of server agents implementing the AFS Volume Location Database (VLDB) for the cell. |
| `VOLSERVICE_ID` | 4 | The *Rx* service number on the given UDP port (`MyPort`) above. |
| `INVALID_BID` | 0 | Used as an invalid read-only or backup volume ID. |
| `VOLSER_MAXVOLNAME` | 65 | The number of characters in the longest possible volume name, including the trailing null. **Note:** this is only used by the `vos` utility; the *Volume Server* uses the "old" value below. |
| `VOLSER_OLDMAXVOLNAME` | 32 | The "old" maximum number of characters in an AFS volume name, including the trailing null. In reality, it is also the current maximum. |
| `VOLSER_MAX_REPSITES` | 7 | The maximum number of replication sites for a volume. |
| `VNAMESIZE` | 32 | Size in bytes of the `name` field in `struct volintInfo` (see Section 5.4.6). |

### 5.2.2 Interface Routine Opcodes

These constants, appearing in the *volint.xg Rxgen* interface file for the *Volume Server*, define the opcodes for the RPC routines. Every *Rx* call on this interface contains this opcode, and the dispatcher uses it to select the proper code at the server site to carry out the call.

| *Name* | *Value* | *Description* |
|---|---|---|
| VOLCREATEVOLUME | 100 | Opcode for *AFSVolCreateVolume()* |
| VOLDELETEVOLUME | 101 | Opcode for *AFSVolDeleteVolume()* |
| VOLRESTORE | 102 | Opcode for *AFSVolRestoreVolume()* |
| VOLFORWARD | 103 | Opcode for *AFSVolForward()* |
| VOLENDTRANS | 104 | Opcode for *AFSVolEndTrans()* |
| VOLCLONE | 105 | Opcode for *AFSVolClone()* . |
| VOLSETFLAGS | 106 | Opcode for *AFSVolSetFlags()* |
| VOLGETFLAGS | 107 | Opcode for *AFSVolGetFlags()* |
| VOLTRANSCREATE | 108 | Opcode for *AFSVolTransCreate()* |
| VOLDUMP | 109 | Opcode for *AFSVolDump()* |
| VOLGETNTHVOLUME | 110 | Opcode for *AFSVolGetNthVolume()* |
| VOLSETFORWARDING | 111 | Opcode for *AFSVolSetForwarding()* |
| VOLGETNAME | 112 | Opcode for *AFSVolGetName()* |
| VOLGETSTATUS | 113 | Opcode for *AFSVolGetStatus()* |
| VOLSIGRESTORE | 114 | Opcode for *AFSVolSignalRestore()* |
| VOLLISTPARTITIONS | 115 | Opcode for *AFSVolListPartitions()* |
| VOLLISTVOLS | 116 | Opcode for *AFSVolListVolumes()* |
| VOLSETIDSTYPES | 117 | Opcode for *AFSVolSetIdsTypes()* |
| VOLMONITOR | 118 | Opcode for *AFSVolMonitor()* |
| VOLDISKPART | 119 | Opcode for *AFSVolPartitionInfo()* |
| VOLRECLONE | 120 | Opcode for *AFSVolReClone()* |
| VOLLISTONEVOL | 121 | Opcode for *AFSVolListOneVolume()* |
| VOLNUKE | 122 | Opcode for *AFSVolNukeVolume()* |
| VOLSETDATE | 123 | Opcode for *AFSVolSetDate()* |

### 5.2.3 Transaction Flags

These constants define the various flags the *Volume Server* uses in assocation with volume transactions, keeping track of volumes upon which operations are currently proceeding. There are three sets of flag values, stored in three different fields within a `struct volser_trans`: general volume state, attachment modes, and specific transaction states.

#### 5.2.3.1 vflags

These values are used to represent the general state of the associated volume. They appear in the `vflags` field within a `struct volser_trans`.

| *Name* | *Value* | *Description* |
|---|---|---|
| `VTDeleteOnSalvage` | 1 | The volume should be deleted on next salvage. |
| `VTOutOfService` | 2 | This volume should never be put online. |
| `VTDeleted` | 4 | This volume has been deleted (via *AFSVolDeleteVolume()* ), and thus should not be manipulated. |

#### 5.2.3.2 iflags

These constants represent the desired *attachment mode* for a volume at the start of a transaction. Once attached, the volume header is marked to reflect this mode. Attachment modes are useful in salvaging partitions, as they indicate whether the operations being performed on individual volumes at the time the crash occured could have introduced inconsistencies in their metadata descriptors. If a volume was attached in a read-only fashion, then the salvager may decide (taking other factors into consideration) that the volume doesn't need attention as a result of the crash.

These values appear in the `iflags` field within a `struct volser_trans`.

| *Name* | *Value* | *Description* |
|---|---|---|
| `ITOffline` | 0x1 | Volume offline on server (returns `VOFFLINE`). |
| `ITBusy` | 0x2 | Volume busy on server (returns `VBUSY`). |
| `ITReadOnly` | 0x8 | Volume is read-only on client, read-write on server - DO NOT USE. |
| `ITCreate` | 0x10 | Volume does not exist correctly yet. |
| `ITCreateVolID` | 0x1000 | Create volid. |

#### 5.2.3.3 tflags

This value is used to represent the transaction state of the associated volume, and appears in the `tflags` field within a `struct volser_trans`.

| *Name* | *Value* | *Description* |
|---|---|---|
| `TTDeleted` | 1 | Delete transaction not yet freed due to high reference count. |

### 5.2.4 Volume Types

The following constants may be supplied as values for the `type` argument to the *AFSVolCreateVolume()* interface call. They are just synonyms for the three values `RWVOL`, `ROVOL`, and `BACKVOL`.

| *Name* | *Value* | *Description* |
|---|---|---|
| `volser_RW` | 0 | Specifies a read-write volume type. |
| `volser_RO` | 1 | Specifies a read-only volume type. |
| `volser_BACK` | 2 | Specifies a backup volume type. |

### 5.2.5 LWP State

This set of exported definitions refers to objects internal to the *Volume Server*, and strictly speaking should not be visible to other agents. Specifically, a `busyFlags` array keeps a set of flags referenced by the set of lightweight threads running within the *Volume Server*. These flags reflect and drive the state of each of these worker LWPs.

| *Name* | *Value* | *Description* |
|---|---|---|
| `VHIdle` | 1 | *Volume Server* LWP is idle, waiting for new work. |
| `VHRequest` | 2 | A work item has been queued. |

### 5.2.6 States for `struct vldbentry`

The *Volume Server* defines a collection of synonyms for certain values defined by the *Volume Location Server*. These particular constants are used within the `flags` field in objects of type `struct vldbentry`. The equivalent *Volume Location Server* values are described in Section 3.2.6.

| *Name* | *Value* | *Description* |
|---|---|---|
| `RW_EXISTS` | 0x1000 | Synonym for `VLF_RWEXISTS`. |
| `RO_EXISTS` | 0x2000 | Synonym for `VLF_ROEXISTS`. |
| `BACK_EXISTS` | 0x4000 | Synonym for `VLF_BACKEXISTS`. |
| `NEW_REPSITE` | 0x01 | Synonym for `VLSF_NEWREPSITE`. |
| `ITSROVOL` | 0x02 | Synonym for `VLFS_ROVOL`. |
| `ITSRWVOL` | 0x04 | Synonym for `VLSF_RWVOL`. |
| `ITSBACKVOL` | 0x08 | Synonym for `VLSF_BACKVOL`. |

### 5.2.7 Validity Checks

These values are used for performing validity checks. The first one appears only within the `partFlags` field within objects of type `partList` (see Section 5.4.3). The rest (except VOK and VBUSY) appear in the volFlags field within an object of type struct volDescription. These latter defintions are used within the volFlags field to mark whether the rest of the fields within the struct volDescription are valid. Note that while several constants are defined, only some are actually used internally by the *Volume Server* code.

| *Name* | *Value* | *Description* |
|---|---|---|
| PARTVALID | 0x01 | The indicated partition is valid. |
| CLONEVALID | 0x02 | The indicated clone (field volCloneId) is a valid one. |
| CLONEZAPPED | 0x04 | The indicated clone volume (field volCloneId) has been deleted. |
| IDVALID | 0x08 | The indicated volume ID (field volId) is valid. |
| NAMEVALID | 0x10 | The indicted volume name (field volName) is valid. Not used internally by the *Volume Server*. |
| SIZEVALID | 0x20 | The indicated volume size (field volSize) is valid. Not used internally by the *Volume Server*. |
| ENTRYVALID | 0x40 | The struct volDescription refers to a valid volume. |
| REUSECLONEID | 0x80 | The indicated clone ID (field volCloneId) should be reused. |
| VOK | 0x02 | Used in the status field of struct volintInfo to show that everything is OK. |
| VBUSY | 110 | Used in the status field of struct volintInfo to show that the volume is currently busy. |

### 5.2.8 Miscellaneous

This section covers the set of exported *Volume Server* definitions that don't easily fall into the above categories.

| *Name* | *Value* | *Description* |
|---|---|---|
| SIZE | 1,024 | Not used internally by the *Volume Server*; used as a maximum size for internal character arrays. |
| MAXHELPERS | 10 | Size of an internal *Volume Server* character array (busyFlags), it marks the maximum number of threads within the server. |
| STDERR | stderr | Synonym for the UNIX standard input file descriptor. |
| STDOUT | stdout | Synonym for the UNIX standard output file descriptor. |

## 5.3 Exported Variables

This section describes the single variable that the *Volume Server* exports to its applications.

The `QI_GlobalWriteTrans` exported variable represents a pointer to the head of the global queue of transaction structures for operations being handled by a *Volume Server*. Each object in this list is of type `struct volser_trans` (see Section 5.4.1 below).

## 5.4 Structures and Typedefs

This section describes the major exported *Volume Server* data structures of interest to application programmers, along with some of the the typedefs based on those structures. Please note that typedefs in shose definitions angle brackets appear are those fed through the *Rxgen* RPC stub generator. *Rxgen* uses these angle brackets to specify an array of indefinite size.

### 5.4.1 struct volser_trans

This structure defines the transaction record for all volumes upon which an active operation is proceeding.

**Fields**

**struct volser_trans *next** - Pointer to the next transaction structure in the queue.

**long tid** - Transaction ID.

**long time** - The time this transaction was last active, for timeout purposes. This is the standard UNIX time format.

**long creationTime** - The time a which this transaction started.

**long returnCode** - The overall transaction error code.

**struct Volume *volume** - Pointer to the low-level object describing the associated volume. This is included here for the use of lower-level support code.

**long volid** - The associated volume's numerical ID.

**long partition** - The partition on which the given volume resides.

**long dumpTransId** - Not used.

**long dumpSeq** - Not used.

**short refCount** - Reference count on this structure.

**short iflags** - Initial attach mode flags.

**char vflags** - Current volume status flags.

**char tflags** - Transaction flags.

**char incremental** - If non-zero, indicates that an incremental restore operation should be performed.

**char lastProcName**[] - Name of the last internal *Volume Server* procedure that used this transaction. This field may be up to 30 characters long, including the trailing null, and is intended for debugging purposes only.

**struct rx_call *rxCallPtr** - Pointer to latest associated rx_call. This field is intended for debugging purposes only.

### 5.4.2 `struct volDescription`

This structure is used by the AFS backup system to group certain key fields of volume information.

**Fields**

**char volName**[] - The name of the given volume; maximum length of this string is `VOLSER_MAXVOLNAME` characters, including the trailing null.

**long volId** - The volume's numerical ID.

**int volSize** - The size of the volume, in bytes.

**long volFlags** - Keeps validity information on the given volume and its clones. This field takes on values from the set defined in Section 5.2.7

**long volCloneId** - The volume's current clone ID.

### 5.4.3 `struct partList`

This structure is used by the backup system and the `vos` tool to keep track of the state of the AFS disk partitions on a given server.

**Fields**

**long partId**[] - Set of 26 partition IDs.

**long partFlags**[] - Set to `PARTVALID` if the associated partition slot corresponds to a valid partition. There are 26 entries in this array.

### 5.4.4 struct volser_status

This structure holds the status of a volume as it is known to the *Volume Server*, and is passed to clients through the *AFSVolGetStatus()* interface call.

Two fields appearing in this structure, `accessDate` and `updateDate`, deserve a special note. In particular, it is important to observe that these fields are **not** kept in full synchrony with reality. When a *File Server* provides one of its client *Cache Manager*s with a chunk of a file on which to operate, it is incapable of determining *exactly* when the data in that chunk is accessed, or *exactly* when it is updated. This is because the manipulations occur on the client machine, without any information on these accesses or updates passed back to the server. The only time these fields can be modified is when the chunk of a file resident within the given volume is delivered to a client (in the case of `accessDate`), or when a client writes back a dirty chunk to the *File Server* (in the case of `updateDate`).

**Fields**

**long volID** - The volume's numerical ID, unique within the cell.

**long nextUnique** - Next value to use for a vnode uniquifier within this volume.

**int type** - Basic volume class, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

**long parentID** - Volume ID of the parent, if this volume is of `type` `ROVOL` or `BACKVOL`.

**long cloneID** - ID of the latest read-only clone, valid iff the `type` field is set to `RWVOL`.

**long backupID** - Volume ID of the latest backup of this read-write volume.

**long restoredFromID** - The volume ID contained in the dump from which this volume was restored. This field is used to simply make sure that an incremental dump is not restored on top of something inappropriate. Note that this field itself is not dumped.

**long maxQuota** - The volume's maximum quota, in 1Kbyte blocks.

**long minQuota** - The volume's minimum quota, in 1Kbyte blocks.

**long owner** - The user ID of the person responsible for this volume.

**long creationDate** - For a volume of type `RWVOL`, this field marks its creation date. For the original copy of a clone, this field represents the cloning date.

**long accessDate** - Last access time by a user for this volume. This value is expressed as a standard UNIX longword date quantity.

**long updateDate** - Last modification time by a user for this volume. This value is expressed as a standard UNIX longword date quantity.

**long expirationDate** - Expiration date for this volume. If the volume never expires, then this field is set to zero.

**long backupDate** - The last time a backup clone was created for this volume.

**long copyDate** - The time that this copy of this volume was created.

### 5.4.5 `struct destServer`

Used to specify the destination server in an *AFSVolForward()* invocation (see Section 5.7.7).

**Fields**

**long destHost** - The IP address of the destination server.

**long destPort** - The UDP port for the *Volume Server Rx* service there.

**long destSSID** - Currently, this field is always set to 1.

### 5.4.6 `struct volintInfo`

This structure is used to communicate volume information to the *Volume Server*'s RPC clients. It is used to build the `volEntries` object, which appears as a parameter to the *AFSVolListVolumes()* call.

The comments in Section 5.4.4 concerning the `accessDate` and `updateDate` fields are equally valid for the analogue fields in this structure.

**Fields**

**char name**[] - The null-terminated name for the volume, which can be no longer than `VNAMESIZE` (32) characters, including the trailing null.

**long volid** - The volume's numerical ID.

**long type** - The volume's basic class, one of `RWVOL`, `ROVOL`, or `BACKVOL`.

**long backupID** - The latest backup volume's ID.

**long parentID** - The parent volume's ID.

**long cloneID** - The latest clone volume's ID.

**long status** - Status of the volume; may be one of `VOK` or `VBUSY`.

**long copyDate** - The time that this copy of this volume was created.

**unsigned char inUse** - If non-zero, an indication that this volume is online.

**unsigned char needsSalvaged** - If non-zero, an indication that this volume needs to be salvaged.

**unsigned char destroyMe** - If non-zero, an indication that this volume should be destroyed.

**long creationDate** - Creation date for a read/write volume; cloning date for the original copy of a read-only volume.

**long accessDate** - Last access time by a user for this volume.

**long updateDate** - Last modification time by a user for this volume.

**long backupDate** - Last time a backup copy was made of this volume.

**int dayUse** - Number of times this volume was accessed since midnight of the current day.

**int filecount** - the number of file system objects contained within the volume.

**int maxquota** - The upper limit on the number of 1-Kbyte disk blocks of storage that this volume may obtain.

**int size** - Not known.

**long flags** - Values used by the backup system are stored here.

**long spare1 - spare3** - Spare fields, reserved for future use.

### 5.4.7 `struct transDebugInfo`

This structure is provided for monitoring and debugging purposes. It is used to compose the `transDebugEntries` variable-sized object, which in turn appears as a parameter to the *AFSVolMonitor()* interface call.

**Fields**

**long tid** - The transaction ID.

**long time** - The time when the transaction was last active, for timeout purposes.

**long creationTime** - The time the transaction started.

**long returnCode** - The overall transaction error code.

**long volid** - The open volume's ID.

**long partition** - The open volume's partition.

**short iflags** - Initial attach mode flags (IT*).

**char vflags** - Current volume status flags (VT*).

**char tflags** - Transaction flags (TT*).

**char lastProcName**[] - The string name of the last procedure which used transaction. This field may be up to 30 characters long, including the trailing null, and is intended for debugging purposes only.

**int callValid** - Flag which determines if the following fields are valid.

**long readNext** - Sequence number of the next *Rx* packet to be read.

**long transmitNext** - Sequence number of the next *Rx* packet to be transmitted.

**int lastSendTime** - The last time anything was sent over the wire for this transaction.

**int lastReceiveTime** - The last time anything was received over the wire for this transaction.

### 5.4.8 `struct pIDs`

Used by the *AFSVolListPartitions()* interface call, this structure is used to store information on all of the partitions on a given *Volume Server*.

**Fields**

**long partIds**[] - One per letter of the alphabet (/vicepa through /vicepz). Filled with 0 for '/vicepa', 25 for '/vicepz'. Invalid partition slots are filled in with a -1.

### 5.4.9 `struct diskPartition`

This structure contains information regarding an individual AFS disk partition. It is returned as a parameter to the *AFSVolPartitionInfo()* call.

**Fields**

**char name**[] - Mounted partition name, up to 32 characters long including the trailing null.

**char devName**[] - Device name on which the partition lives, up to 32 characters long including the trailing null.

**int lock_fd** - A lock used for mutual exclusion to the named partition. A value of -1 indicates the lock is not currently being held. Otherwise, it has the file descriptor resulting from the UNIX *open()* call on the file specified in the `name` field used to "acquire" the lock.

**int totalUsable** - The number of blocks within the partition which are available.

**int free** - The number of free blocks in the partition.

**int minFree** - The minimum number of blocks that must remain free regardless of allocation requests.

### 5.4.10 `struct restoreCookie`

Used as a parameter to both *AFSVolRestore()* and *AFSVolForward()*, a `restoreCookie` keeps information that must be preserved between various *Volume Server* operations.

**Fields**

**char name**[] - The volume name, up to 32 characters long including the trailing null.

**long type** - The volume type, one of `RWVOL`, `ROVOL`, and `BACKVOL`.

**long clone** - The current read-only clone ID for this volume.

**long parent** - The parent ID for this volume.

### 5.4.11 `transDebugEntries`

```
typedef transDebugInfo transDebugEntries<>;
```

This typedef is used to generate a variable-length object which is passed as a parameter to the *AFSVolMonitor()* interface function. Thus, it may carry any number of descriptors for active transactions on the given *Volume Server*. Specifically, it causes a C structure of the same name to be defined with the following fields:

**Fields**

- **u_int transDebugEntries_len** - The number of `struct transDebugInfo` (see Section 5.4.7) objects appearing at the memory location pointed to by the `transDebugEntries_val` field.
- **transDebugInfo *transDebugEntries_val** - A pointer to a region of memory containing an array of `transDebugEntries_len` objects of type `struct transDebugInfo`.

### 5.4.12 volEntries

```
typedef volintInfo volEntries<>;
```

This typedef is used to generate a variable-length object which is passed as a parameter to *AFSVolListVolumes()*. Thus, it may carry any number of descriptors for volumes on the given *Volume Server*. Specifically, it causes a C structure of the same name to be defined with the following fields:

**Fields**

- **u_int volEntries_len** - The number of `struct volintInfo` (see Section 5.4.6) objects appearing at the memory location pointed to by the `volEntries_val` field.
- **volintInfo *volEntries_val** - A pointer to a region of memory containing an array of `volEntries_len` objects of type `struct volintInfo`.

## 5.5 Error Codes

The *Volume Server* advertises two groups of error codes. The first set consists of the standard error codes defined by the package. The second is a collection of lower-level return values which are exported here for convenience.

### 5.5.1 Standard

The error codes described in this section were defined by the *Volume Server* to describe exceptional conditions arising in the course of RPC call handling.

| *Name* | *Value* | *Description* |
|---|---|---|
| VOLSERTRELE_ERROR | 1492325120L | internal error releasing transaction. |
| VOLSERNO_OP | 1492325121L | unknown internal error. |
| VOLSERREAD_DUMPERROR | 1492325122L | badly formatted dump. |
| VOLSERDUMPERROR | 1492325123L | badly formatted dump(2). |
| VOLSERATTACH_ERROR | 1492325124L | could not attach volume. |
| VOLSERILLEGAL_PARTITION | 1492325125L | illegal partition. |
| VOLSERDETACH_ERROR | 1492325126L | could not detach volume. |
| VOLSERBAD_ACCESS | 1492325127L | insufficient privilege for volume operation. |
| VOLSERVLDB_ERROR | 1492325128L | error from volume location database. |
| VOLSERBADNAME | 1492325129L | bad volume name. |
| VOLSERVOLMOVED | 1492325130L | volume moved. |
| VOLSERBADOP | 1492325131L | illegal volume operation. |
| VOLSERBADRELEASE | 1492325132L | volume release failed. |
| VOLSERVOLBUSY | 1492325133L | volume still in use by volserver. |
| VOLSERNO_MEMORY | 1492325134L | out of virtual memory in volserver. |
| VOLSERNOVOL | 1492325135L | no such volume. |
| VOLSERMULTIRWVOL | 1492325136L | more than one read/write volume. |
| VOLSERFAILEDOP | 1492325137L | failed volume server operation. |

### 5.5.2 Low-Level

These error codes are duplicates of those defined from a package which is internal to the *Volume Server*. They are re-defined here to make them visible to *Volume Server* clients.

| *Name* | *Value* | *Description* |
|---|---|---|
| VSALVAGE | 101 | Volume needs to be salvaged. |
| VNOVNODE | 102 | Bad vnode number encountered. |
| VNOVOL | 103 | The given volume is either not attached, doesn't exist, or is not online. |
| VVOLEXISTS | 104 | The given volume already exists. |
| VNOSERVICE | 105 | The volume is currently not in service. |
| VOFFLINE | 106 | The specified volume is offline, for the reason given in the offline message field (a subfield within the `volume` field in `struct volser_trans`). |
| VONLINE | 107 | Volume is already online. |
| VDISKFULL | 108 | The disk partition is full. |
| VOVERQUOTA | 109 | The given volume's maximum quota, as expressed in the `maxQuota` field of the `struct volintInfo`, has been exceeded. |
| VBUSY | 110 | The named volume is temporarily unavailable, and the client is encouraged to retry the operation shortly. |
| VMOVED | 111 | The given volume has moved to a new server. |

The VICE_SPECIAL_ERRORS constant is defined to be the lowest of these error codes.

## 5.6 Macros

The *Volume Server* defines a small number of macros, as described in this section.

### 5.6.1 *THOLD()*

```
#define THOLD(tt)  ((tt)->refCount++)
```

This macro is used to increment the reference count field, `refCount`, in an object of type `struct volser_trans`. Thus, the associated transaction is effectively "held", insuring it won't be garbage-collected. The counterpart to this operation, `TRELE()`, is implemented by the *Volume Server* as a function.

### 5.6.2 *ISNAMEVALID()*

```
#define ISNAMEVALID(name) (strlen(name) < (VOLSER_OLDMAXVOLNAME - 9))
```

This macro checks to see if the given `name` argument is of legal length. It must be no more than the size of the container, which is at most `VOLSER_OLDMAXVOLNAME` characters, minus the length of the longest standardized volume name postfix known to the system. That postfix is the 9-character `.restored` string, which is tacked on to the name of a volume that has been restored from a dump.

## 5.7 Functions

This section covers the *Volume Server* RPC interface routines, defined by and generated from the *volint.xg Rxgen* file. The following is a summary of the interface functions and their purpose:

| *Fcn Name* | *Description* |
|---|---|
| `AFSVolCreateVolume` | Create a volume. |
| `AFSVolDeleteVolume` | Delete a volume. |
| `AFSVolNukeVolume` | Obliterate a volume completely. |
| `AFSVolDump` | Dump (i.e., save) the contents of a volume. |
| `AFSVolSignalRestore` | Show intention to call *AFSVolRestore()*. |
| `AFSVolRestore` | Recreate a volume from a dump. |
| `AFSVolForward` | Dump a volume, then restore to a given server and volume. |
| `AFSVolClone` | Clone (and optionally purge) a volume. |
| `AFSVolReClone` | Re-clone a volume. |
| `AFSVolSetForwarding` | Set forwarding info for a moved volume. |
| `AFSVolTransCreate` | Create transaction for a [volume, partition]. |
| `AFSVolEndTrans` | End a transaction. |
| `AFSVolGetFlags` | Get volume flags for a transaction. |
| `AFSVolSetFlags` | Set volume flags for a transaction. |
| `AFSVolGetName` | Get the volume name associated with a transaction. |
| `AFSVolGetStatus` | Get status of a transaction/volume. |
| `AFSVolSetIdsTypes` | Set header info for a volume. |
| `AFSVolSetDate` | Set creation date in a volume. |
| `AFSVolListPartitions` | Return a list of AFS partitions on a server. |
| `AFSVolPartitionInfo` | Get partition information. |
| `AFSVolListVolumes` | Return a list of volumes on the server. |
| `AFSVolListOneVolume` | Return header info for a single volume. |
| `AFSVolGetNthVolume` | Get volume header given its index. |
| `AFSVolMonitor` | Collect server transaction state. |

There are two general comments that apply to most of the *Volume Server* interface routines:

1. AFS partitions are identified by integers ranging from 0 to 25, corresponding to the letters 'a' through 'z'. By convention, AFS partitions are named `/vicepx`, where $x$ is any lower-case letter.

2. Legal volume types to pass as parameters are RWVOL, ROVOL, and BACKVOL, as defined in Section 3.2.4.

### 5.7.1 AFSVolCreateVolume — Create a volume

```
int AFSVolCreateVolume(IN struct rx_connection *z_conn,
                       IN long partition,
                       IN char *name,
                       IN long type,
                       IN long parent,
                       INOUT long *volid,
                       OUT long *trans)
```

#### Description

Create a volume named `name`, with numerical identifier `volid`, and of type `type`. The new volume is to be placed in the specified `partition` for the server machine as identified by the *Rx* connection information pointed to by `z_conn`. If a value of 0 is provided for the `parent` argument, it will be set by the *Volume Server* to the value of `volid` itself. The `trans` parameter is set to the *Volume Location Server* transaction ID corresponding to the volume created by this call, if successful.

The numerical volume identifier supplied in the `volid` parameter must be generated beforehand by calling *VL_GetNewVolumeID()* (see Section 3.6.5). After *AFSVolCreateVolume()* completes correctly, the new volume is marked as offline. It must be explicitly brought online through a call to *AFSVolSetFlags()* (see Section 5.7.14) while passing the `trans` transaction ID generated by `AFSVolCreateVolume()`. The "hold" on the new volume guaranteed by the `trans` transaction may be "released" by calling *AFSVolEndTrans()*. Until then, no other process may operate on the volume.

Upon creation, a volume's maximum quota (as specified in the `maxquota` field of a `struct volintInfo`) is set to 5,000 1-Kbyte blocks.

Note that the *AFSVolCreateVolume()* routine is the only *Volume Server* function that manufactures its own transaction. All others must have already acquired a transaction ID via either a previous call to *AFSVolCreateVolume()* or *AFSVolTransCreate()*.

#### Error Codes

`VOLSERBADNAME` The volume `name` parameter was longer than 31 characters plus the trailing null.

`VOLSERBAD_ACCESS` The caller is not authorized to create a volume.

- `EINVAL` The `type` parameter was illegal.
- `E2BIG` A value of `0` was provided in the `volid` parameter.
- `VOLSERVOLBUSY` A transaction could not be created, thus the given volume was busy.
- `EIO` The new volume entry could not be created.
- `VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.
- `<misc>` If the `partition` parameter is unintelligible, this routine will return a low-level UNIX error.

### 5.7.2 AFSVolDeleteVolume — Delete a volume

```
int AFSVolDeleteVolume(IN struct rx_connection *z_conn,
                       IN long trans)
```

#### Description

Delete the volume associated with the open transaction ID specified within `trans`. All of the file system objects contained within the given volume are destroyed, and the on-disk volume metadata structures are reclaimed. In addition, the in-memory volume descriptor's `vflags` field is set to `VTDeleted`, indicating that it has been deleted.

Under some circumstances, a volume should be deleted by calling *AFSVolNukeVolume()* instead of this routine. See Section 5.7.3 for more details.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to delete a volume.

`ENOENT` The `trans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

### 5.7.3 AFSVolNukeVolume — Obliterate a volume completely

```
int AFSVolNukeVolume(IN struct rx_connection *z_conn,
                     IN long partID,
                     IN long volID)
```

#### Description

Completely obliterate the volume living on partition `partID` whose ID is `volID`. This involves scanning all inodes on the given partition and removing those marked with the specified `volID`. If the volume is a read-only clone, only the header inodes are removed, since they are the only ones stamped with the read-only ID. To reclaim the space taken up by the actual data referenced through a read-only clone, this routine should be called on the read-write master. Note that calling *AFSVolNukeVolume()* on a read-write volume effectively destroys all the read-only volumes cloned from it, since everything except for their indicies to the (now-deleted) data will be gone.

Under normal circumstances, it is preferable to use *AFSVolDeleteVolume()* instead of *AFSVolNukeVolume()* to delete a volume. The former is much more efficient, as it only touches those objects in the partition that belong to the named volume, walking the on-disk volume metadata structures. However, *AFSVolNukeVolume()* **must** be used in situations where the volume metadata structures are known to be damaged. Since a complete scan of all inodes in the partition is performed, all disconnected or unreferenced portions of the given volume will be reclaimed.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERNOVOL` The partition specified by the `partID` argument is illegal.

### 5.7.4 AFSVolDump — Dump (i.e., save) the contents of a volume

```
int AFSVolDump(IN struct rx_connection *z_conn,
               IN long fromTrans,
               IN long fromDate)
```

#### Description

Generate a canonical dump of the contents of the volume associated with transaction `fromTrans` as of calendar time `fromDate`. If the given `fromDate` is zero, then a full dump will be carried out. Otherwise, the resulting dump will be an incremental one.

This is specified as a *split* function within the *volint.xg Rxgen* interface file. This specifies that *two* routines are generated, namely *StartAFSVolDump()* and *EndAFSVolDump()*. The former is used to marshall the `IN` arguments, and the latter is used to unmarshall the return value of the overall operation. The actual dump data appears in the *Rx* stream for the call (see the section entitled *Example Server and Client* in the companion AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility document).

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to dump a volume.

`ENOENT` The `fromTrans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

### 5.7.5 AFSVolSignalRestore — Show intention to call *AFSVolRestore()*

```
int AFSVolSignalRestore(IN struct rx_connection *z_conn,
                        IN char *name,
                        IN int type,
                        IN long pid,
                        IN long cloneid)
```

#### Description

Show an intention to the *Volume Server* that the client will soon call *AFSVolRestore()*. The parameters, namely the volume `name`, `type`, parent ID `pid` and clone ID `cloneid` are stored in a well-known set of global variables. These values are used to set the restored volume's header, overriding those values present in the dump from which the volume will be resurrected.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERBADNAME` The volume name contained in `name` was longer than 31 characters plus the trailing null.

### 5.7.6 AFSVolRestore — Recreate a volume from a dump

```
int AFSVolRestore(IN struct rx_connection *z_conn,
                  IN long toTrans,
                  IN long flags,
                  IN struct restoreCookie *cookie)
```

#### Description

Interpret a canonical volume dump (generated as the result of calling *AFSVolDumpVolume()*), passing it to the volume specified by the `toTrans` transaction. Only the low bit in the `flags` argument is inspected. If this low bit is turned on, the dump will be restored as incremental; otherwise, a full restore will be carried out.

All callbacks to the restored volume are broken.

This is specified as a *split* function within the *volint.xg Rxgen* interface file. This specifies that *two* routines are generated, namely *StartAFSVolRestore()* and *EndAFSVolRestore()* . The former is used to marshall the `IN` arguments, and the latter is used to unmarshall the return value of the overall operation. The actual dump data flows over the *Rx* stream for the call (see the section entitled *Example Server and Client* in the companion AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility document).

The *AFSVolSignalRestore()* routine (see Section 5.7.5) should be called before invoking this function in order to signal the intention to restore a particular volume.

#### Error Codes

`VOLSERREAD_DUMPERROR` Dump data being restored is corrupt.

`VOLSERBAD_ACCESS` The caller is not authorized to restore a volume.

`ENOENT` The `fromTrans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

### 5.7.7 AFSVolForward — Dump a volume, then restore to given server and volume

```
int AFSVolForward(IN struct rx_connection *z_conn,
                  IN long fromTrans,
                  IN long fromDate,
                  IN struct destServer *destination,
                  IN long destTrans,
                  IN struct restoreCookie *cookie)
```

#### Description

Dumps the volume associated with transaction `fromTrans` from the given `fromDate`. The dump itself is sent to the server described by `destination`, where it is restored as the volume associated with transaction `destTrans`. In reality, an *Rx* connection is set up to the `destServer`, *StartAFSVolRestore()* directs writing to the *Rx* call's stream, and then *EndAFSVolRestore()* is used to deliver the dump for the volume corresponding to `fromTrans`. If a non-zero `fromDate` is provided, then the dump will be incremental from that date. Otherwise, a full dump will be delivered.

The *Rx* connection set up for this task is always destroyed before the function returns. The destination volume should exist before carrying out this operation, and the invoking process should have started transactions on both participating volumes.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to forward a volume.

`ENOENT` The `fromTrans` transaction was not found.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

`ENOTCONN` An *Rx* connection to the destination server could not be established.

### 5.7.8 **AFSVolClone** — Clone (and optionally purge) a volume

```
int AFSVolClone(IN struct rx_connection *z_conn,
                IN long trans,
                IN long purgeVol,
                IN long newType,
                IN char *newName,
                INOUT long *newVol)
```

#### Description

Make a clone of the read-write volume associated with transaction `trans`, giving the cloned volume a name of `newName`. The `newType` parameter specifies the type for the new clone, and may be either `ROVOL` or `BACKVOL`. If `purgeVol` is set to a non-zero value, then that volume will be purged during the clone operation. This may be more efficient that separate clone and purge calls when making backup volumes. The `newVol` parameter sets the new clone's ID. It is illegal to pass a zero in `newVol`.

#### Error Codes

- `VOLSERBADNAME` The volume name contained in `newName` was longer than 31 characters plus the trailing null.
- `VOLSERBAD_ACCESS` The caller is not authorized to clone a volume.
- `ENOENT` The `fromTrans` transaction was not found.
- `VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.
- `VBUSY` The given transaction was already in use; indicating that someone else is currently manipulating the specified clone.
- `EROFS` The volume associated with the given `trans` is read-only (either `ROVOL` or `BACKVOL`).
- `EXDEV` The volume associated with the `trans` transaction and the one specified by `purgeVol` must be on the same disk device, and they must be cloned from the same parent volume.
- `EINVAL` The `purgeVol` must be read-only, i.e. either type `ROVOL` or `BACKVOL`.

### 5.7.9 AFSVolReClone — Re-clone a volume

```
int AFSVolReClone(IN struct rx_connection *z_conn,
                  IN long tid,
                  IN long cloneID)
```

#### Description

Recreate an existing clone, with identifier `cloneID`, from the volume associated with transaction `tid`.

#### Error Codes

- `VOLSERBAD_ACCESS` The caller is not authorized to clone a volume.
- `ENOENT` The `tid` transaction was not found.
- `VOLSERTRELE_ERROR` The `tid` transaction's reference count could not be dropped to the proper level.
- `VBUSY` The given transaction was already in use; indicating that someone else is currently manipulating the specified clone.
- `EROFS` The volume to be cloned must be read-write (of type `RWVOL`).
- `EXDEV` The volume to be cloned and the named clone itself must be on the same device. Also, `cloneID` must have been cloned from the volume associated with transaction `tid`.
- `EINVAL` The target clone must be a read-only volume (i.e., of type `ROVOL` or `BACKVOL`).

### 5.7.10 AFSVolSetForwarding — Set forwarding info for a moved volume

```
int AFSVolSetForwarding(IN struct rx_connection *z_conn,
                        IN long tid,
                        IN long newsite)
```

#### Description

Record the IP address specified within `newsite` as the location of the host which now hosts the volume associated with transaction `tid`, formerly resident on the current host. This is intended to gently guide *Cache Manager*s who have stale volume location cached to the volume's new site, ensuring the move is transparent to clients using that volume.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to create a forwarding address.

`ENOENT` The `trans` transaction was not found.

### 5.7.11 AFSVolTransCreate — Create transaction for a [volume, partition]

```
int AFSVolTransCreate(IN struct rx_connection *z_conn,
                      IN long volume,
                      IN long partition,
                      IN long flags,
                      OUT long *trans)
```

#### Description

Create a new *Volume Server* transaction associated with volume ID `volume` on partition `partition`. The type of volume transaction is specified by the `flags` parameter. The values in `flags` specify whether the volume should be treated as busy (`ITBusy`), offline (`ITOffline`), or in shared read-only mode (`ITReadOnly`). The identifier for the new transaction built by this function is returned in `trans`.

Creating a transaction serves as a signal to other agents that may be interested in accessing a volume that it is unavailable while the Volume Server is manipulating it. This prevents the corruption that could result from multiple simultaneous operations on a volume.

#### Error Codes

- `EINVAL` Illegal value encountered in `flags`.
- `VOLSERVOLBUSY` A transaction could not be created, thus the given [volume, partition] pair was busy.
- `VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level after creation.

### 5.7.12 AFSVolEndTrans — End a transaction

```
int AFSVolEndTrans(IN struct rx_connection *z_conn,
                   IN long trans,
                   OUT long *rcode)
```

#### Description

End the transaction identified by `trans`, returning its final error code into `rcode`. This makes the associated [volume, partition] pair eligible for further *Volume Server* operations.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to create a transaction.

`ENOENT` The `trans` transaction was not found.

### 5.7.13 AFSVolGetFlags — Get volume flags for a transaction

```
int AFSVolGetFlags(IN struct rx_connection *z_conn,
                   IN long trans,
                   OUT long *flags)
```

#### Description

Return the value of the vflags field of the struct volser_trans object describing the transaction identified as trans. The set of values placed in the flags parameter is described in Section 5.2.3.1. Briefly, they indicate whether the volume has been deleted (VTDeleted), out of service (VTOutOfService), or marked delete-on-salvage (VTDeleteOnSalvage).

#### Error Codes

ENOENT The trans transaction was not found.

VOLSERTRELE_ERROR The trans transaction's reference count could not be dropped to the proper level.

### 5.7.14 AFSVolSetFlags — Set volume flags for a transaction

```
int AFSVolSetFlags(IN struct rx_connection *z_conn,
                   IN long trans,
                   IN long flags)
```

#### Description

Set the value of the `vflags` field of the `struct volser_trans` object describing the transaction identified as `trans` to the contents of `flags`. The set of legal values for the `flags` parameter is described in Section 5.2.3.1. Briefly, they indicate whether the volume has been deleted (`VTDeleted`), out of service (`VTOutOfService`), or marked delete-on-salvage (`VTDeleteOnSalvage`).

#### Error Codes

`ENOENT` The `trans` transaction was not found.

`EROFS` Updates to this volume are not allowed.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

### 5.7.15 AFSVolGetName — Get the volume name associated with a transaction

```
int AFSVolGetName(IN struct rx_connection *z_conn,
                  IN long tid,
                  OUT char **tname)
```

#### Description

Given a transaction identifier `tid`, return the name of the volume associated with the given transaction. The `tname` parameter is set to point to the address of a string buffer of at most 256 chars containing the desired information, which is created for this purpose. **Note:** the caller is responsible for freeing the buffer pointed to by `tname` when its information is no longer needed.

#### Error Codes

`ENOENT` The `tid` transaction was not found, or a volume was not associated with it (`VSrv` internal error).

`E2BIG` The volume name was too big (greater than or equal to `SIZE` (1,024) characters.

`VOLSERTRELE_ERROR` The `trans` transaction's reference count could not be dropped to the proper level.

### 5.7.16 AFSVolGetStatus — Get status of a transaction/volume

```
int AFSVolGetStatus(IN struct rx_connection *z_conn,
                    IN long tid,
                    OUT struct volser_status *status)
```

#### Description

This routine fills the `status` structure passed as a parameter with status information for the volume identified by the transaction identified by `tid`, if it exists. Included in this status information are the volume's ID, its type, disk quotas, the IDs of its clones and backup volumes, and several other administrative details.

#### Error Codes

`ENOENT` The `tid` transaction was not found.

`VOLSERTRELE_ERROR` The `tid` transaction's reference count could not be dropped to the proper level.

### 5.7.17 AFSVolSetIdsTypes — Set header info for a volume

```
int AFSVolSetIdsTypes(IN struct rx_connection *z_conn,
                      IN long tId
                      IN char *name,
                      IN long type,
                      IN long pId,
                      IN long cloneId,
                      IN long backupId)
```

#### Description

The transaction identifed by `tId` is located, and the values supplied for the volume `name`, volume `type`, parent ID `pId`, clone ID `cloneId` and backup ID `backupId` are recorded into the given transaction.

#### Error Codes

ENOENT The `tId` transaction was not found.

VOLSERBAD_ACCESS The caller is not authorized to call this routine.

VOLSERBADNAME The volume name contained in `name` was longer than 31 characters plus the trailing null.

VOLSERTRELE_ERROR The `tId` transaction's reference count could not be dropped to the proper level.

### 5.7.18 AFSVolSetDate — Set creation date in a volume

```
int AFSVolSetDate(IN struct rx_connection *z_conn,
                  IN long tid,
                  IN long newDate)
```

#### Description

Set the `creationDate` of the `struct volintInfo` describing the volume associated with transaction `tid` to `newDate`.

#### Error Codes

VOLSERBAD_ACCESS The caller is not authorized to call this routine.

ENOENT The `tId` transaction was not found.

VOLSERTRELE_ERROR The `tid` transaction's reference count could not be dropped to the proper level.

#### 5.7.19 AFSVolListPartitions — Return a list of AFS partitions on a server

```
int AFSVolListPartitions(IN struct rx_connection *z_conn,
                         OUT struct pIDs *partIDs)
```

**Description**

Return a list of AFS partitions in use by the server processing this call. The output parameter is the fixed-length `partIDs` array, with one slot for each of 26 possible partitions. By convention, AFS partitions are named `/vicepx`, where $x$ is any letter. The `/vicepa` partition is represented by a zero in this array, `/vicepa` by a 1, and so on. Unused partitions are represented by slots filled with a -1.

**Error Codes**

--- None.

### 5.7.20 AFSVolPartitionInfo — Get partition information

```
int AFSVolPartitionInfo(IN struct rx_connection *z_conn,
                        IN char *name,
                        OUT struct diskPartition *partition)
```

#### Description

Collect information regarding the partition with the given character string `name`, and place it into the `partition` object provided.

#### Error Codes

`VOLSERBAD_ACCESS` The caller is not authorized to call this routine.

`VOLSERILLEGAL_PARTITION` An illegal partition was specified by `name`

### 5.7.21 AFSVolListVolumes — Return a list of volumes on the server

```
int AFSVolListVolumes(IN struct rx_connection *z_conn,
                      IN long partID,
                      IN long flags,
                      OUT volEntries *resultEntries)
```

#### Description

Sweep through all the volumes on the partition identified by `partid`, filling in consecutive records in the `resultEntries` object. If the `flags` parameter is set to a non-zero value, then full status information is gathered. Otherwise, just the volume ID field is written for each record. The fields for a `volEntries` object like the one pointed to by `resultEntries` are described in Section 5.4.6, which covers the `struct volintInfo` definition.

#### Error Codes

`VOLSERILLEGAL_PARTITION` An illegal partition was specified by `partID`

`VOLSERNO_MEMORY` Not enough memory was available to hold all the required entries within `resultEntries`.

### 5.7.22 AFSVolListOneVolume — Return header info for a single volume

```
int AFSVolListOneVolume(IN struct rx_connection *z_conn,
                        IN long partID,
                        IN long volid,
                        OUT volEntries *resultEntries)
```

#### Description

Find the information for the volume living on partition `partID` whose ID is `volid`, and place a single `struct volintInfo` entry within the variable-size `resultEntries` object.

This is similar to the *AFSVolListVolumes()* call, which returns information on *all* volumes on the specified partition. The full volume information is always written into the returned entry (equivalent to setting the `flags` argument to *AFSVolListVolumes()* to a non-zero value).

#### Error Codes

`VOLSERILLEGAL_PARTITION` An illegal partition was specified by `partID`

`ENODEV` The given volume was not found on the given partition.

### 5.7.23 AFSVolGetNthVolume — Get volume header given its index

```
int AFSVolGetNthVolume(IN struct rx_connection *z_conn,
                       IN long index,
                       OUT long *volume,
                       OUT long *partition)
```

#### Description

Using `index` as a zero-based index into the set of volumes hosted by the server chosen by the `z_conn` argument, return the `volume` ID and `partition` of residence for the given index.

**Note: This functionality has not yet been implemented.**

#### Error Codes

| | |
|---|---|
| VOLSERNO_OP | Not implemented. |

### 5.7.24 AFSVolMonitor — Collect server transaction state

```
int AFSVolMonitor(IN struct rx_connection *z_conn,
                  OUT transDebugEntries *result)
```

#### Description

This call allows the transaction state of a *Volume Server* to be monitored for debugging purposes. Anyone wishing to supervise this *Volume Server* state may call this routine, causing all active transactions to be recorded in the given `result` object.

#### Error Codes

--- None.

# Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[4] Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[5] Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

# Index

VOLSER_OLDMAXVOLNAME, 61
*AFSVolForward()*, 57

const BACK_EXISTS, 48
const BACKVOL, 12
const BADSERVERID, 10
const CLONEZAPPED, 49
const DEFAULTBULK, 19
const ENTRYVALID, 49
const HASHSIZE, 10
const IDVALID, 49
const INVALID_BID, 45
const ITBusy, 47
const ITCreateVolID, 47
const ITCreate, 47
const ITOffline, 47
const ITReadOnly, 47
const ITSBACKVOL, 48
const ITSROVOL, 48
const ITSRWVOL, 48
const LOCKREL_AFSID, 13
const LOCKREL_OPCODE, 13
const LOCKREL_TIMESTAMP, 13
const MAX_NUMBER_OPCODES, 18
const MAXBUMPCOUNT, 10
const MAXHELPERS, 49
const MAXLOCKTIME, 10
const MAXNAMELEN, 10
const MAXNSERVERS, 10
const MAXPARTITIONID, 10
const MAXSERVERFLAG, 10
const MAXSERVERID, 10
const MAXTYPES, 10
const MyPort, 45
const NameLen, 45
const NAMEVALID, 49
const NEW_REPSITE, 48
const NULLO, 10
const REUSECLONEID, 49
const RO_EXISTS, 48
const ROVOL, 12
const RW_EXISTS, 48
const RWVOL, 12
const SIZEVALID, 49
const SIZE, 10, 49
const STDERR, 49
const STDOUT, 49
const TTDeleted, 47
const VBUSY, 49, 60
const VDISKFULL, 60
const VHIdle, 48
const VHRequest, 48
const VICE_SPECIAL_ERRORS, 60
const VL_BADENTRY, 21
const VL_BADINDEX, 21
const VL_BADNAME, 21
const VL_BADPARTITION, 21
const VL_BADREFCOUNT, 21
const VL_BADRELLOCKTYPE, 21
const VL_BADSERVERFLAG, 21
const VL_BADSERVER, 21
const VL_BADVOLIDBUMP, 21
const VL_BADVOLOPER, 21
const VL_BADVOLTYPE, 21
const VL_CREATEFAIL, 21
const VL_DUPREPSERVER, 21
const VL_EMPTY, 21
const VL_ENTDELETED, 21

const VL_ENTRYLOCKED, 21
const VL_IDALREADYHASHED, 21
const VL_IDEXIST, 21
const VL_IO, 21
const VL_NAMEEXIST, 21
const VL_NOENT, 21
const VL_NOMEM, 21
const VL_NOREPSERVER, 21
const VL_PERM, 21
const VL_REPSFULL, 21
const VL_RERELEASE, 21
const VL_RWNOTFOUND, 21
const VL_SIZEEXCEEDED, 21
const VLDB_MAXSERVERS, 45
const VLDBALLOCCOUNT, 10
const VLDBVERSION, 10
const VLDELETED, 12
const VLF_BACKEXISTS, 13
const VLF_ROEXISTS, 13
const VLF_RWEXISTS, 13
const VLFREE, 12
const VLLIST_FLAG, 11
const VLLIST_PARTITION, 11
const VLLIST_SERVER, 11
const VLLIST_VOLUMEID, 11
const VLLIST_VOLUMETYPE, 11
const VLLOCKED, 12
const VLOP_ALLOPERS, 12
const VLOP_BACKUP, 12
const VLOP_DELETE, 12
const VLOP_DUMP, 12
const VLOP_MOVE, 12
const VLOP_RELEASE, 12
const VLREPSITE_NEW, 13
const VLSF_BACKVOL, 13
const VLSF_NEWREPSITE, 13
const VLSF_ROVOL, 13
const VLSF_RWVOL, 13
const VLUPDATE_BACKUPID, 11
const VLUPDATE_CLONEID, 11
const VLUPDATE_FLAGS, 11
const VLUPDATE_READONLYID, 11
const VLUPDATE_REPS_ADD, 11
const VLUPDATE_REPS_DELETE, 11
const VLUPDATE_REPS_MODFLAG, 11
const VLUPDATE_REPS_MODPART, 11
const VLUPDATE_REPS_MODSERV, 11
const VLUPDATE_REPSITES, 11
const VLUPDATE_VOLUMENAME, 11
const VLUPDATE_VOLUMETYPE, 11
const VMOVED, 60
const VNAMESIZE, 45
const VNOSERVICE, 60
const VNOVNODE, 60
const VNOVOL, 60
const VOFFLINE, 60
const VOK, 49
const VOLCLONE, 46
const VOLCREATEVOLUME, 46
const VOLDELETEVOLUME, 46
const VOLDISKPART, 46
const VOLDUMP, 46
const VOLENDTRANS, 46
const VOLFORWARD, 46
const VOLGETFLAGS, 46
const VOLGETNAME, 46
const VOLGETNTHVOLUME, 46
const VOLGETSTATUS, 46
const VOLLISTONEVOL, 46
const VOLLISTPARTITIONS, 46
const VOLLISTVOLS, 46
const VOLMONITOR, 46
const VOLNUKE, 46
const VOLRECLONE, 46
const VOLRESTORE, 46
const volser_BACK, 48
const VOLSER_MAX_REPSITES, 45
const VOLSER_MAXVOLNAME, 45
const VOLSER_OLDMAXVOLNAME, 45
const volser_RO, 48
const volser_RW, 48
const VOLSERATTACH_ERROR, 59

const VOLSERBAD_ACCESS, 59
const VOLSERBADNAME, 59
const VOLSERBADOP, 59
const VOLSERBADRELEASE, 59
const VOLSERDETACH_ERROR, 59
const VOLSERDUMPERROR, 59
const VOLSERFAILEDOP, 59
const VOLSERILLEGAL_PARTITION, 59
const VOLSERMULTIRWVOL, 59
const VOLSERNO_MEMORY, 59
const VOLSERNO_OP, 59
const VOLSERNOVOL, 59
const VOLSERREAD_DUMPERROR, 59
const VOLSERTRELE_ERROR, 59
const VOLSERVICE_ID, 45
const VOLSERVLDB_ERROR, 59
const VOLSERVOLBUSY, 59
const VOLSERVOLMOVED, 59
const VOLSETDATE, 46
const VOLSETFLAGS, 46
const VOLSETFORWARDING, 46
const VOLSETIDSTYPES, 46
const VOLSIGRESTORE, 46
const VOLTRANSCREATE, 46
const VONLINE, 60
const VOVERQUOTA, 60
const VSALVAGE, 60
const VTDeleted, 47
const VTDeleteOnSalvage, 47
const VTOutOfService, 47
const VVOLEXISTS, 60
const , 49

file *afsvlint.xg*, 23, 38
file *vldbint.xg*, 6, 38
file *volint.h*, 44
file *volint.xg*, 41, 44, 45, 62, 68, 70
file *volser.h*, 44
fsck, 40
function *AFSVolClone()*, 46, 62, 72
function *AFSVolCreateVolume()*, 42, 43, 46, 47, 62, 64
function *AFSVolDeleteVolume()*, 46, 47, 62, 66
function *AFSVolDump()*, 46, 62, 68
function *AFSVolDumpVolume()*, 70
function *AFSVolEndTrans()*, 46, 62, 64, 76
function *AFSVolForward()*, 46, 54, 62, 71
function *AFSVolGetFlags()*, 46, 62, 77
function *AFSVolGetName()*, 46, 62, 79
function *AFSVolGetNthVolume()*, 46, 62, 87
function *AFSVolGetStatus()*, 46, 53, 62, 80
function *AFSVolListOneVolume()*, 46, 62, 86
function *AFSVolListPartitions()*, 46, 56, 62, 83
function *AFSVolListVolumes()*, 42, 46, 54, 58, 62, 85, 86
function *AFSVolMonitor()*, 46, 55, 57, 62, 88
function *AFSVolNukeVolume()*, 46, 62, 67
function *AFSVolPartitionInfo()*, 46, 62, 84
function *AFSVolReClone()*, 46, 62, 73
function *AFSVolRestore()*, 57, 62, 69, 70
function *AFSVolRestoreVolume()*, 46
function *AFSVolSetDate()*, 46, 62, 82
function *AFSVolSetFlags()*, 46, 62, 78
function *AFSVolSetForwarding()*, 46, 62, 74
function *AFSVolSetIdsTypes()*, 46, 62, 81
function *AFSVolSignalRestore()*, 46, 62, 69, 70
function *AFSVolTransCreate()*, 46, 62, 64, 75

function *EndAFSVolDump()*, 68
function *EndAFSVolRestore()*, 70, 71
function *StartAFSVolDump()*, 68
function *StartAFSVolRestore()*, 70, 71
function *VL_CreateEntry()*, 12, 14, 24
function *VL_DeleteEntry()*, 25
function *VL_GetEntryByID()*, 12, 14, 26, 27, 38
function *VL_GetEntryByName()*, 12, 14, 27, 38
function *VL_GetNewVolumeID()*, 64
function *VL_GetNewVolumeId()*, 28
function *VL_GetStats()*, 36
function *VL_LinkedList()*, 18, 35
function *VL_ListAttributes()*, 17, 18, 34, 35
function *VL_ListByAttributes()*, 19
function *VL_ListEntry()*, 12, 14, 33
function *VL_Probe()*, 37, 38
function *VL_ReleaseLock()*, 13, 32
function *VL_ReplaceEntry()*, 12--14, 29
function *VL_SetLock()*, 12, 31
function *VL_UpdateEntry()*, 13, 17, 29, 30

macro *COUNT_ABO()*, 22
macro *COUNT_REQ()*, 22
macro *DOFFSET()*, 22
macro *ISNAMEVALID()*, 61
macro *THOLD()*, 61

salvager, 40, 42
struct destServer, 54
struct diskPartition, 56
struct partList, 52
struct pIDs, 56
struct restoreCookie, 57
struct single_vldbentry, 18
struct transDebugInfo, 55
struct vital_vlheader, 16
struct vldb_list, 18, 20
struct vldbentry, 10--12, 14, 15, 18, 38
struct VldbListByAttributes, 11, 17
struct VldbUpdateEntry, 10, 17
struct vldstats, 18
struct vlentry, 12, 15
struct vlheader, 16
struct volDescription, 52
struct volintInfo, 54
struct volser_status, 53
struct volser_trans, 40, 51
struct VolumeDiskData, 10
struct Volume, 10

typedef bulkentries, 19
typedef bulk, 19
typedef single_vldbentry, 20
typedef transDebugEntries, 57
typedef vldblist, 20
typedef vlentry, 20
typedef vlheader, 20
typedef volEntries, 58

var QI_GlobalWriteTrans, 50
VLDB, 45