OpenAFS for Windows Cache Manager and Kernel Drivers File System Architectures

Contents

[Introduction 2](#_Toc193793844)

[Introduction to the AFS SMB Server 3](#_Toc193793845)

[Introduction to the AFS Cache Manager 6](#_Toc193793846)

[Introduction to AFS Cache Manager Threading 8](#_Toc193793847)

[Introduction to Internal AFS Cache Manager Object Locking 9](#_Toc193793848)

[Introduction to the Kernel Drivers File System (KDFS) 11](#_Toc193793849)

[Overview 11](#_Toc193793850)

[Communication Protocol 11](#_Toc193793851)

[MetaData Cache 11](#_Toc193793852)

[FileData Cache 12](#_Toc193793853)

[KD File System Testing 12](#_Toc193793854)

[Kernel Drivers File System API 12](#_Toc193793855)

[AFS Cache Manager Module Overview 15](#_Toc193793856)

[cm\_memmap: Cache Manager Memory Mapping 15](#_Toc193793857)

[cm\_cell: AFS Cells 15](#_Toc193793858)

[cm\_volume: AFS Volumes 15](#_Toc193793859)

[cm\_scache: AFS Vnode Status Module 16](#_Toc193793860)

[AFS SMB Server Module Overview 16](#_Toc193793861)

[OpenAFS Cache Manager Updates to Support Unicode 17](#_Toc193793862)

[Rewrite SMB/CIFS Functions 17](#_Toc193793863)

[Implement Conditional Parsing 17](#_Toc193793864)

[Extend SMB Virtual Circuit. 17](#_Toc193793865)

[Extend Interfaces 17](#_Toc193793866)

[Character Set Conversion. 17](#_Toc193793867)

[OpenAFS Cache Manager Integration with KDFS 18](#_Toc193793868)

[OpenAFS Network Provider DLL 18](#_Toc193793869)

[OpenAFS Redirector 19](#_Toc193793870)

[OpenAFS Communication Protocol 19](#_Toc193793871)

[Delivery Schedule 20](#_Toc193793872)

[Unicode AFS Cache Manager and AFS SMB Server – July 4 20](#_Toc193793873)

[KDFS Enabled AFS Cache Manager 20](#_Toc193793874)

[Milestone 1 – May 2 20](#_Toc193793875)

[Milestone 2 – May 16 20](#_Toc193793876)

[Milestone 3 - June 13 20](#_Toc193793877)

[Milestone 4 – August 1 21](#_Toc193793878)

[Milestone 5 – August 29 21](#_Toc193793879)

[Milestone 6 – September 26 21](#_Toc193793880)

[Final Delivery – November 28 21](#_Toc193793881)

[Various Implementation Notes 22](#_Toc193793882)

# Introduction

This document describes the architecture of the OpenAFS for Windows (OAFW) cache manager as of the 1.5.53 release, the architecture of the Kerner Drivers File System, and the plan to integrate the two in order to produce an OpenAFS client for the Windows platform that does not rely on the Microsoft SMB Redirector and Netbios.

OAFW derives from IBM AFS for Windows circa October 2000. From a bird’s eye view the architecture of OAFW has not changed dramatically since the initial release. However, there have been significant improvements in the implementation of the architecture including support for persistent caches, 64-bit file sizes, byte range lock management, B+ tree directory search, least recently used object recycling, Unicode object names, and much more.

The OAFW Service (afsd\_service.exe) implements an AFS Cache sandwiched between an implementation of an SMB Server and the AFS Remote Procedure Call (RPC) layer. (see Figure 1)

Figure High Level overview of OAFW Service architecture

This model permits OAFW to be implemented entirely as a user-mode service although at the price of serious tradeoffs in performance and reliability.

## Introduction to the AFS SMB Server

The AFS SMB Server implements a subset of the NTLM0.12 dialect of the Server Message Block (SMB) Protocol Specification[[1]](#footnote-2). The two most important NTLM0.12 features that are not implemented are support for Pipe Transactions (wkssvc, srvsvc, and ipc$) and the Distributed File System extensions.

Communication between the application and the AFS SMB Server follows the path displayed in Figure 2.

Figure : SMB RPC communication path

From the perspective of the application and the SMB redirector, the AFS namespace is simply a very large Windows File Share offered by another computer on the local network. In reality the host “AFS” is actually the AFS SMB Server running on the local machine and accessed via the Microsoft Loopback Adapter network interface. This fiction is the root cause of many of the failings when it comes to the overall AFS user experience.

The AFS SMB Server supports SMB Authentication using a special pass-through authentication mode.[[2]](#footnote-3) SMB authentication became a requirement as part of the Windows security enhancements introduced in XP Service Pack 2 and Windows 2003 Service Pack 1. The AFS SMB Server cannot support any of the Kerberos based authentication methods because there is no CIFS/AFS$@%MACHINE% service principal and all machines on which the AFS SMB Server is installed would have to share the same key. The end result is that the AFS cannot be accessed from Windows accounts that are authenticated with smartcards or even some configurations of password based Kerberos.

The SMB protocol does not include a keep-alive message that can be used to maintain an idle call while the server is processing the request. The SMB Redirector implements a timeout mechanism to determine when the server it is communicating with has failed or can no longer be reached. SMB was designed for small requests sent across a local area network. As a result the timeout period is quite short (30 to 60 seconds.) If the AFS SMB server does not complete the requested operation within the timeout period, the SMB Redirector will disconnect from the server, wait for some period of time, and then attempt to reconnect. All outstanding file system requests from applications are canceled and all open file handles are closed.

There are 87 SMB interfaces implemented by the AFS SMB Server. The following table lists the most frequently called operations.

Figure : Commonly called SMB RPCs

The AFS SMB server implementation is quite small consisting of approximately 22,500 lines of source code in a small number of files.

Figure : SMB source files

The AFS SMB server implementation is responsible for maintaining state information for active SMB virtual circuits (client connections) and implementing SMB RPCs in terms of AFS Cache Manager Operations.

The most important data structures in the AFS SMB Server are listed in Figure 5.

Figure : Overview of SMB Data Structures

SMB objects are valid only within the active AFS Client Service session. All SMB objects that maintain references to Cache Manager Objects release those references as part of AFS SMB Server shutdown.

## Introduction to the AFS Cache Manager

The AFS Cache Manager (CM) consists of approximately 38,200 lines of code in 25 modules. The CM is the heart and soul of the AFS Client Service. The CM performs all of the operations necessary to maintain a consistent view of the global AFS namespace while managing AFS users and their authentication credentials; file server and volume location database server connections; synchronizing access to data buffers; and processing AFS file server callbacks.

The most important data types of the cache manager are described in Figure 6.

Figure : Cache Manager Object Types

The Cache Manager stores its configuration data, cell entries, volume entries, status cache entries, ACL entries, directory name lookup cache, and data buffers within a single persisted memory mapped cache file. All other objects including server data, user data, and directory B+ trees are regenerated each time the AFS Cache Manager starts. Cached data is discarded if the previous AFS Cache Manager instance did not shutdown cleanly or if the cache manager configuration was altered in an incompatible manner.[[3]](#footnote-4)

The volume entries, status cache entries, ACL entries, DNLC entries, and data buffers are recyclable objects. For each object class, an LRU queue is maintained. Each time an object is touched by an external request, the object is moved to the front of the queue. Whenever an object must be recycled to satisfy an allocation request they are pulled from the rear of the queue. The allocation of each object type is fixed at service start.

In order to satisfy any external request there must be a matching set of cell object, volume object, status cache object, ACL entry and data buffers. All of these objects are joined by the AFS file identifier. Due to recycling it is possible that one or more of the necessary objects may no longer exist in the cache. As a result the architecture does not permit direct pointer references between these objects. All object references must be satisfied by looking up the required object in a hash table. If an object does not exist in the cache it must be fetched from the volume location database service or the file service.

1->\*

1->\*

1->\*

There exists a one-to-many relationship between the various object classes.

## Introduction to AFS Cache Manager Threading

The AFS Cache Manager is a user-mode multi-threaded service. Threads are created at four layers: SMB, Cache Manager, Rx and other.

The SMB layer directly allocates *smb\_Server* threads. Each instance of the *smb\_Server* thread is a worker thread capable of processing a single SMB request at a time. The requests are accepted by one of the *smb\_Listener* threads. There is a separate instance of the *smb\_*Listener thread for each Lan Adapter on which the AFS Client Service listens. Each *smb\_Listener* thread blocks in a Netbios() call until a request is delivered or an error condition occurs. Once an incoming request has been received, the *smb\_Listener* thread stores the request on a queue, signals that the request is ready for processing, and waits for the next incoming request. The *smb\_Server* threads perform the actual request processing. The *smb\_ClientWaiter* and *smb\_ServerWaiter* threads manage the producer/consumer model. The *smb\_Daemon* thread performs periodic maintenance activities including garbage collection of smb\_vc\_t and smb\_username\_t objects. The *smb\_WaitingLocksDaemon* thread processes outstanding lock requests that could not be issued due to lock conflicts.

The Cache Manager Layer creates a number of daemon threads that perform on-going management operations. These include:

* *cm\_FreelanceChangeNotifier* and *cm\_FreelanceSymlinkChangeNotifier* which monitor the registry for changes to the Freelance (dynroot) root.afs contents.
* *cm\_IpAddrDaemon* blocks in the Windows IpHelper library in order to receive notifications for network stack configuration changes.
* *cm\_Daemon* performs periodic maintenance tasks including probing current server status, expiring callbacks, configuring the windows firewall, forcing new Rx connections on IP address changes, performing volume checks, renewing active file server locks, garbage collecting the token cache, and collecting performance statistics.
* Each instance of *cm\_BkgDaemon* processes requests pulled from the background operations queue. These requests are explicit requests to perform background reads (prefetching) or background writes.
* *buf\_IncrSyncer* periodically walks a list of dirty buffers and writes them to the file server.

The Rx Communication Layer creates the *rx\_Listener* thread pool and the *rx\_eventHandler* thread. The *rx\_Listener* threads process incoming messages. If they are responses to remote procedure calls that were initiated by this client the response is queued and the *rx\_eventHandler* is signaled to associate the response with the thread that is blocked waiting for the response. If the message is an incoming request such as a Callback, a Cache Manager Debug query or TellMeAboutYourself query, the *rx\_Listener* thread becomes a worker and a new *rx\_Listener* thread is allocated from the thread pool[[4]](#footnote-5) to process subsequent incoming messages.

Other threads that exist in the AFS Client Service process include threads created to support the Windows Service Manager, the NETAPI32 library, and other support services not directly related to the AFS Cache Manager architecture.

## Introduction to Internal AFS Cache Manager Object Locking

The AFS Cache Manager uses both mutexes and read/write locks to ensure thread safe access to its data structures and provide for thread synchronization. The locking hierarchy is defined in the header file, src/WINNT/afsd/cm.h. The cm\_scache\_t locks are obtained lowest vnode value first.

In general, for each object class there is either at least one mutex or read/write lock within the structure. These protect access to ‘flags’ and other object specific data. Global read/write locks and mutexes are used to protect object lists, hash tables, and reference counts manipulations.

For reference counts, InterlockedIncrement() and InterlockedDecrement() APIs are used to manipulate the value of the reference count. As a result it is possible to protect the increment and decrement with read-locks and upgrade to a write lock only when decrementing and the resulting value is zero. Write locks are only required if the object is going to be garbage collected as a result of the reference count hitting zero.

The locking objects, osi\_rwlock\_t and osi\_mutex\_t, are defined in src/client\_osi/osibase.[hc]. They are constructed from a pool of 251 CRITICAL\_SECTION objects. There are tens of thousands of locking objects in use within the cache manager which means that multiple objects share the same critical section. While this can result in contention between threads it cannot result in a deadlock.

Adherence to the locking hierarchy can be tested using the Lock Order Validation interface. When activated each thread keeps track of the lock objects that are held. If a lock is obtained out of order a panic is thrown. This permits potential deadlock conditions to be detected even if a deadlock would not actually occur.

## Introduction to Volume Location and File Server Management

Each server, volume location or file, is tracked by a cm\_server\_t object. These objects maintain the per server state information including its IP address, its capabilities, the associated cell, its preference ranking, up/down state, and for a file server the list of volumes known to be served by the server.

Each volume group object (cm\_volume\_t) maintains a list of cm\_serverRef\_t (server reference) objects; one for each of the servers on which a particular volume instance is hosted. The server references maintain the state of the volume on the server.

All servers are probed on a regular basis to determine the current state and to ensure that an open communication path is available for the delivery of callbacks; every three minutes for down servers and every four minutes for up servers. The probes are simultaneously sent to all servers in the designated pool. File servers are probed using RXAFS\_GetCapabiliites and those that do not support that interface are probed with RXAFS\_GetTime. Volume location servers are probed with VL\_ProbeServer. Down servers are avoided when requests must be sent. When a file server up/down state changes the state for all affected volumes is recomputed. This permits a rapid determination of the volume availability without requiring further server communication.

The Windows cache manager supports a volume status notification interface. A DLL written to this interface[[5]](#footnote-6) and registered with the cache manager will receive notifications of status changes for all volumes known to the cache manager.

## Introduction to Callback Processing

Every vnode status object (cm\_scache\_t) maintains callback state information for that vnode. This includes the server the callback is registered with as well as the expiration time. For objects stored within readonly or backup volumes, the expiration time is maintained as part of the volume group object (cm\_volume\_t) as a valid callback registration is refreshed whenever any object stored within the volume has its status information updated.

The cache manager receives two types of callback messages: per-object and per-volume. A volume callback invalidates the callback registrations of all objects within the volume. The object callback invalidates just the specified object. Callback messages are delivered on an unauthenticated connection.

AFS file ids as known to the file server have three components: volume, vnode and a uniqueness value. The windows cache manager augments the file id with a cell identifier to which each file server is bound. The cache manager uses the server IP address to find the appropriate file server object (cm\_server\_t) and from that determines which cell the callback is referring to.

When all of the servers capable of serving a volume are inaccessible, any valid callback registration will continue to be treated as valid. If the callback registration would have expired, the expiration will take place upon the detection of an available file server. This prevents applications that are running out of the AFS cache from crashing due to the executable binaries or data files suddenly becoming unavailable.

## Introduction to Directory Management

The AFS directory format was designed in the 80s when 64 thousand 12 character case sensitive names must have appeared to be a nearly infinite space; then came longer than 12 character names which required the use of multiple sequential directory blocks. The directory structure now averages twelve thousand entries.

The case sensitive design of the directory prevents a case insensitive operating system such as Microsoft Windows from being able to efficiently perform directory searches. A case sensitive hash table lookup when a match is found is quite fast. However, a negative hit does not mean that no case insensitive matches exist. Instead it means that a linear search of the entire directory must be performed to determine if there is a single match, an ambiguous match or if there is no match. The addition of UTF-8 encoding of non-normalized Unicode strings only makes matters worse. Of course, 8.3 short names must be taken into account as well.

In order to provide efficient directory search and manipulation in OAFW a modified B+ tree is constructed that includes both long and short name entries with case insensitive normalized Unicode strings as the keys. Each leaf node instead of being a single data value in the B+ tree is instead a bucket into which all of the potentially matching strings are stored. The end result is a greater than 1000% reduction in the cost associated with a negative match.

The B+ tree is constructed the first time the directory is accessed after a data version change. The tree is updated locally with each modification to the directory in order to reduce the necessary network traffic.

## Introduction to Data Version Management

Data Version management is one of the key concepts behind AFS performance. Each object stored in AFS has an associated data version value that is incremented with each change throughout a 64-bit numeric space. The cache manager stores the metadata for each object as part of the vnode status object (cm\_scache\_t). The data that makes up the file or directory object is broken up into 4K extents. Each extent is linked to a buffer object (cm\_buf\_t) which stores the metadata associated with the extent. The most important of which is the data version.

Once an extent is read from the file server it is stored in the data cache. Subsequent attempt to access the same extent will succeed as long as the data version of the associated buffer object matches the latest data version stored within the vnode status object.

Not so long ago any local change to an object stored in AFS would be written to the file server and then read back from the file server because the data version would change. The current implementation avoids this problem by maintaining a range of consecutive data version values which are to be treated as valid. Anytime a locally originating change is performed the cache manager checks to see if the data version was incremented by exactly one. If so, it is known that no race occurred with another cache manager. As a result less data must be read from the file server.

## Introduction to the Kernel Drivers File System (KDFS)

### Overview

The KD File System was originally implemented as a local file system with an abstracted lower edge. This would allow for easy migration to specific requirements, these generally being user mode service solutions for handling requests from the file system. This type of design would allow for network based solutions for a file system design to be quickly brought to a functional level without the need to implement a full kernel mode sockets layer. The lower edge of the KDFS was extended to work with local media as well as integrating with a kernel mode socket implementation, either through TDI or WSK.

The file system has been extended to include redirector functionality by registering with MUP. This registration also requires support for prefix resolution which is handled within the file system. Integration with a Network Provider dll offers additional support for device mapped resources.

### Communication Protocol

The communication protocol to user mode is implemented through an inverted call model design. In this design, a set of user mode threads allocates resources and passes these to the file system via an IOCtl call. The requests, as they are received by the file system, are queued for later processing and a *pending* status is returned to the caller. As the file system requires information from the user service, it will dequeue a request, populate the request with appropriate context information and complete the request to the calling thread.

These requests can be either synchronous or asynchronous from the file system perspective. Thus it can issue requests to the service that would not require any response, or not an immediate response, such as notification events. Or it can issue requests which are blocking within the file system such as metadata retrieval requests for a specific directory.

The inverted call model, when implemented correctly, offers a robust, system supported cleanup mechanism. Because the user mode service has opened a handle to the file system control device, if for any reason the service were to end the handle open on the control device would be closed. This notification, in the form of an IRP\_MJ\_CLEANUP for the control device, would be an indicator to the file system that the service is going away and all resources would be deallocated and pended IOCtl requests completed.

### MetaData Cache

The KD File System implements a metadata cache to allow for a seamless user experience while the associated user mode service would retrieve metadata content for a specific directory. The approach uses a proprietary interface to the service to retrieve data for an entire directory, caching the contents in a form that allows for easy translation to any one of the many directory enumeration control structures.

To allow for resource management within the metadata cache, contents of the cache is pruned according to a last access combined with a timeout algorithm. This allows for content which is accessed frequently to remain in cache but the overall resource consumption of the cache is controlled.

### FileData Cache

The KD File System is fully integrated with the Windows Cache and Memory Management subsystems. This allows for support of all IO types including memory mapped IO. This also allows for execution of files within the KDFS name space.

The protocol used for IO to the user mode service is restricted to non-cached or paging IO requests. Thus any data requests destined for the disk under normal local storage semantics is migrated to the user service for processing. The caveat on this is the current cache state of the file. If the file is in a state of NO\_CACHE, then all write requests, regardless of type, are implemented as write through.

### KD File System Testing

The KD File System has been used in several client projects, varying from remote file system, user mode file system and proprietary USB connections. The metadata subsystem has been thoroughly tested through direct testing models as well as black box testing from applications. The test suite used for the metadata interface include specific pathway testing though applications such as the Microsoft Office suite or generalized stability testing over many days via the Winstone test suite.

The file data subsystem, integration with the cache and memory management subsystems, has been tested thoroughly for stability and coherency. The performance aspects of the KDFS for file data have been optimized but are tied to the underlying infrastructure used for handling the data requests.

The KDFS has been tested on the following platforms:

* Windows 2000
* Windows XP
* Windows Server 2003
* Windows Vista
* Windows Server 2008

The testing has been carried out on the 32 bit versions of these platforms only.

### Kernel Drivers File System API

This section covers the API exposed by the Kernel Drivers File System which must be handled by the supporting user mode service. All control structures are located in the AFSUserCommon.h header file along with all IOCtl definitions. This API was the starting point upon which the OpenAFS Cache Manager Communication interface was derived.

#### File System to Service Requests

##### KD\_FSD\_REQUEST\_DIR\_ENUM

This request is issued by the file system when a directory is first accessed any the content of the directory is required. This API, which returns a data payload in the request buffer, is processed using an input control block of type KDFsdDirQueryCB. This control block specifies the last FileIndex of a previous query in the active enumeration. The FileIndex is used by the service as a tracking index indicating where the query is to resume.

The directory contents are passed back to the driver using a series of KDFsdDirEnumEntry control structures. Each of these entries contains information about a single file within the directory.

##### KD\_FSD\_REQUEST\_CREATE\_FILE

This request is issued by the driver when a new file is being created. The driver passes a control structure of type KDFsdFileCreateCB which specifies all the required information about the node to be created.

The result of the operation is passed back to the driver in a control structure of type KDFsdFileCreateResultCB. This information contains identifiers that are used by the driver when performing per file requests as well as the FileIndex value which is used in directory enumeration requests.

##### KD\_FSD\_REQUEST\_READ\_FILE and KD\_FSD\_REQUEST\_WRITE\_FILE

These requests are used for performing IO operations on a given file. The driver sends a control structure of the type KDFsdFileIoCB which contains information about the IO operation as well as a mapped buffer address for the request.

The result of the operation is passed back to the driver in a control structure of type KDFsdFileResultCB. This contains information such as the amount of data actually processed.

##### KD\_FSD\_REQUEST\_UPDATE\_FILE

This request is issued by the driver whenever a metadata update is performed on a file. The driver sends the updated information in a KDFsdFileUpdateCB control structure.

There is no returned information for this request, only the result status of the operation.

##### KD\_FSD\_REQUEST\_DELETE\_FILE

This request is issued by the driver in response to a file delete request. There is no passed data block from the driver, only the FileID is passed within the request buffer for the operation.

There is no returned information other than the result status of the operation.

##### KD\_FSD\_REQUEST\_RENAME\_FILE

This request is issued to the service in response to a rename operation. The driver passes a KDFsdFileRenameCB control structure to the service for processing the request.

The service sends a KDFsdFileRenameResultCB control structure in response to this operation. This returned control block contains information such as an updated FileIndex if the Rename was a cross-directory Rename operation.

#### Service to File System Requests

##### KD\_FSD\_REQUEST\_INVALIDATE\_MD\_CACHE

This request is issued by the user mode service to perform cache invalidation on metadata which has been downloaded on a particular directory. This call forces the KDFS to flush and purge all metadata associated to the directory.

The service sends a KDFsdInvalidateCacheCB to the file system to perform the requested operation.

# AFS Cache Manager Module Overview

## cm\_memmap: Cache Manager Memory Mapping

The cm\_memmap module is responsible for the AFS Cache File. It either configures a reusable dedicate file which is memory mapped into the AFS Client Service memory space, or it allocates a virtual cache out of the existing Windows Paging File. When a persisted cache file is being used the module is responsible for performing consistency checks at startup including attempts to ensure that the AFS cache file has not been cloned to multiple systems. This check is performed by verifying the SID of the machine and the Volume ID of the partition the AFS Cache file is installed on. This check is important because the AFS Cache file maintains the UUID for this AFS cache manager instance. Cloned UUIDs can have a negative impact on the AFS user experience.

## cm\_fid: AFS File Identifiers

The cm\_fid\_t type is defined as part of the AFS Vnode Status package. The File Identifier (FID) ties together loosely coupled data structures related to the same vnode including the cell, the volume, the vnode, the data buffers, lock objects, callbacks, etc. The FID ties together a locally assigned cell identifier with the volume, vnode, and unique identifiers maintained by the AFS File Server. A 32-bit hash value is added to reduce the cost of FID comparison.

The two functions in the API are cm\_SetFid() and cm\_FidCmp() which is an inline function because cm\_FidCmp is the most frequently called function in the AFS cache manager.

## cm\_cell: AFS Cells

The cm\_cell module is responsible for managing all cm\_cell\_t objects and the acquisition of associated volume location database information. Cell objects are allocated as a result of a call to cm\_GetCellGen() which searches for the specified cell name in either the CellServDB file or in AFSDB DNS records. If a matching entry is found, the cm\_cell\_t is constructed, a locally defined cell ID number is allocated, and the object is added to the ID and Name hash tables. The cm\_UpdateCell() function is called to refresh the volume location database server list for the cell. The refresh occurs either after the AFSDB record time-to-live has expired or every two hours.

Each cm\_cell\_t is allocated a mutex which is used to protect the flags and timeout fields of the object.

The hash tables and the list of all cells are protected by the global read write lock, cm\_cellLock.

The server list in each cm\_cell\_t is protected by the global cm\_serverLock from the cm\_server module.

cm\_cell\_t objects are never recycled. The default number of cell objects allocated is 1024, considerably larger than the maximum number of cells in CellServDB.

## cm\_volume: AFS Volumes

The primary responsibility of the cm\_volume module is to maintain the binding between volume identifiers and the list of file servers on which the volumes can be found. Volumes can be located either by name or numeric identifier.

The cm\_volume\_t object originally represented a single volume whereas today it represents an entire volume set. [[6]](#footnote-7)A volume set consists of three volumes: a read/write (or normal) volume, a readonly volume, and a backup volume[[7]](#footnote-8). Not all of the volume types must exist in a given set. The cm\_volume\_t was modified to represent volume sets in order to eliminate duplication of objects and minimize calls to the VLDB. The VL\_GetEntryByName RPC returns all of the data associated with a volume set. Previously a separate cm\_volume\_t object was allocated for each name (“volume”, “volume.readonly”, “volume.backup”, or even numeric string representations of the volume identifiers.) With separate cm\_volume\_t objects for each lookup name, it was possible to issue six VL\_GetEntryByName calls; under the volume set approach a single call is sufficient.

With the cm\_volume\_t object representing a volume set, each cm\_volume\_t object may be listed in up to four hash tables as there are hash tables for base name, rw-id, ro-id, and bk-id. Searching for a volume by ID therefore may require searching all three of the ID hash tables.

The cm\_UpdateVolume() function clears the old server lists for each of the volumes in the set and then proceeds to query the VLDB to update an update set of servers for each volume in the set. The most important side effect of cm\_UpdateVolume() is that the volume ID bindings to the volume name are updated to match the current state. If a volume name no longer exists in the VLDB, a single update will ensure that all references to the name are eliminated.

One of the optimizations in the Windows cache manager is the cm\_VolumeRenewROCallbacks() function. Read only volume callbacks traditionally expire after two hours. If it does, then all of the callbacks on all of the vnode status objects expire and the cache manager is then forced to perform a new FetchStatus RPC for each object in the volume. By periodically performing a FetchStatus operation for one object from the volume, the callback for all objects in the volume is renewed at the same time.

The availability status for any volume can be obtained with the cm\_GetVolumeStatus() API.

## cm\_scache: AFS Vnode Status Module

The AFS Vnode Status module maintains all of the state information associated with a file server vnode entry and acts as the primary source of synchronization for all operations that occur to that object.

Each cm\_scache\_t object is uniquely referenced by its assigned FID. cm\_scache\_t objects are maintained in a list of all cm\_scache\_t objects *cm\_data.allSCachesp* and the *allNextp* field; a LRU queue *cm\_data.scacheLRUFirstp* and *cm\_data.scacheLRULastp* and the *q* field; and a hash table *cm\_data.scacheHashTablep* and the *nextp* field which is keyed using CM\_SCACHE\_HASH(&cm\_fid\_t). The hash table is sufficiently large that the average number of cm\_scache\_t objects in a bucket is 2. This is due to the high frequency with which cm\_scache\_t objects are searched for. These lists and queue along with the *refCount* field are protected by the global osi\_rwlock\_t object *cm\_scacheLock*.

There are several other osi\_rwlock\_t objects that are defined in the cm\_scache\_t object. The *rw* field protects most of the data fields in the cm\_scache\_t object. The *bufCreateLock* is used to prevent buffer creation from taking place at the same time a truncation operation is being performed on the vnode. The *bufCreateLock* must only be held when manipulating the number of buffers associated with the cm\_scache\_t: cm\_SetLength(), buf\_GetNewLocked(), buf\_Truncate(). It must not be held when reading from or writing to allocated buffers. Finally, *dirlock* is used to control access to the associated B+ tree if the cm\_scache\_t represents a directory.

The cm\_scache\_t stores the vnode’s status information. This includes the type (file, directory, mount point, symlink, dfslink, unknown), the last modified time, the length, the UNIX mode bits, the link count, the data version, the owner and group, among others. For symlinks, mount points and dfslinks, the *mountPointString* field stores the target description.

The cm\_scache\_t object is used to track callback information. The status information for a file is only valid to use if a callback is registered with a file server. A callback is obtained by the cache manager successfully issuing a FetchStatus or BulkStatus RPC. The callback is a promise by the file server to a cache manager that it will notify the client if there are any changes to the state of the vnode. These changes can include everything from the deletion of the file, a change to the length or to the content of the data stream, a change to the ACLs, a change to other attributes, a change in the lock state, etc. AFS at the present time only supports a blanket callback notification that simply means that something changed. It is the responsibility of cache manager to fetch the status again and compare the values to determine what changed. The cm\_scache\_t object tracks the callback with both a reference to the file server that the callback is registered on and an expiration time. However, this expiration time does not apply to objects stored in .readonly volumes. For those objects, the cm\_volume\_t tracks the expiration as it applies to all objects in the .readonly volume for which there is a valid callback.

For every object in AFS there is a current data version. Each time a change is made to the object on the file server the data version is incremented and the new data version value is returned to the cache manager requesting the change. If the change resulted in a data version delta of one, the cache manager knows that it was the only party to modify the object.

# AFS SMB Server Module Overview

To Be Completed …

# OpenAFS Cache Manager Updates to Support Unicode

The following changes were implemented in the OpenAFS for Windows Cache Manager and deployed to the community as part of OpenAFS 1.5.50.

## Rewrite SMB/CIFS Functions

There are 71 separate SMB/CIFS functions that must be redesigned to negotiate the use of Unicode on a per-request basis as well as parse and construct the Unicode-aware form of the protocol messages.

### Implement Conditional Parsing

Implement appropriate functions necessary to implement conditional parsing of the SMB request messages based upon the message word count.

## Extend SMB Virtual Circuit.

Extend the SMB Virtual Circuit data structure to maintain the status of client request for Unicode names.

## Extend Interfaces

Extend the following modules to support UTF8 strings: cm\_scache (mount points and symlinks), cm\_dcache, cm\_dnlc, cm\_btree, cm\_dir, cm\_vnodeops.

## Character Set Conversion.

Implement UTF16 to UTF8 conversion functions. On Windows XP and above make use of the Microsoft Internationalized Domain Name Mitigation APIs to obtain access to Unicode Normalization support. Make use of Unicode normalization to permit matches during directory searches of object names even when the names are encoded using different rules (such as when comparing names stored via MacOS X with those generated on Microsoft Windows.) After discussion in the OpenAFS community it was decided that all platforms will store names into AFS in the native encoding for the operating system and locale that generated the name. Normalization will only be performed when reading names out of AFS.

# OpenAFS Cache Manager Integration with KDFS

The AFS Redirector is built using the Kernel Drivers file system implementation which is a generic file system implementation that can support a wide variety of architectures. The core components of the file system comprise the infrastructure required to maintain a name space, integrate with the Windows Cache Manager and Memory Manager, integrate with MUP and a host of other subsystems required for proper function. This document covers the components required for this support along with the interface to the AFS Cache Manager which is used for handling requests from the remote server.

## AFS Redirector Overview

The AFS Redirector along with the AFS CM allows for the full support of a name space represented by a specified server along with a variable size set of cell names, which represent the share names in Windows UNC semantics. The UNC name space support is augmented by the use of a network provider dll which implements the required infrastructure for drive letter mapping of these UNC names. This NP dll is a user mode component which handles requests to map a specified UNC path onto a drive letter for more common interactions with the file system.

All of this support is relying on the user mode AFS CM module which feeds metadata and data information to the AFS Redirector as it is requested by overlying components. This interface is implemented as a standard inverted call model allowing for a stable and moderate performing interface for exchanging data between the two components. Figure 1 depicts the essential components involved in this system with many of the system components left out for clarity.

AFS CM

NP dll

User Mode

Kernel Mode

MUP

AFS Redirector

MM and CM

Figure 1: AFS Overview

In figure 1, system components which are coupled to the AFS Redirector are depicted in green and components implemented through the AFS Redirector support are depicted in blue.

### AFS CM and Redirector Communication

The AFS Redirector and Cache Manager communicate through an inverted call model interface. This interface is driven by the user mode CM which allocates a set of worker threads that call into the redirector through a named control device instance. These threads call into the redirector with a pre-allocated buffer which is used for the generic communication header along with any request specific payload information. This header is the AFSCommRequest control structure. This common header allows for the passing of requests, regardless of the type, between the CM and Redirector for interpretation by the respective module.

Each thread worker pool allocated by the AFSCM calls into the AFS Redirector through an IOCtl interface, the handle which is opened for this call is against a named control device allocated by the redirector. The IOCtl calls are then ‘pended’ by the redirector and each thread blocks on an OVERLAPPED event which is signaled when the redirector completes the request.

In the redirector, it has built a pool of control blocks which represent each of the user mode worker threads along with the buffer used for the common header. As the redirector requires an entry for issuing a request to the AFSCM, it retrieves one of the pool entries, sets context specific information and completes the request. This completion, as previously mentioned, awakens the user mode thread and the request is then processed by the AFSCM.

#### AFS CM Request Types

Following is a listing of the request types which can be issued in either direction. The first section covers the requests initiated by the redirector and this is followed by a section covering requests initiated by the AFS CM module.

##### AFS\_REQUEST\_TYPE\_DIR\_ENUM

Function: This request is made to the AFS CM in response to a directory first being accessed. The request indicates to the AFS CM that all meta data for the specified directory must be retrieved synchronously to this call. This results in the AFS CM enumerating all of the entries in the directory and issuing one or more RXAFS\_InlineBulkStatus RPCs to acquire the meta data for each of the entries.

Request Data: The information passed to the AFS CM is the common header block and the AFSDirQueryCB control block.

Result Data: The returned information comes in the form of one, or more, AFSDirEnum entries.

##### AFS\_REQUEST\_TYPE\_CREATE\_FILE

Function: The request is issued in response to a new file or directory being created within the current volume.

Request Data: This request issues the payload structure AFSFileCreateCB containing the information about the file or directory to create.

Result Data: The AFS CM returns a AFSFileCreateResultCB structure which contains information about the specific node that was created, it successful.

##### AFS\_REQUEST\_TYPE\_FILE\_EXTENTS

##### AFS\_REQUEST\_TYPE\_RELEASE\_FILE\_EXTENTS

Function: These requests are issued to the AFS CM to retrieve and release file buffers. Instead of passing file data back and forth across the Ioctl interface which would prove to be very time consuming, the file system driver and the cache manager both share the same paging file, %windir%\temp\AFSCache. These requests are used to allow the AFS CM to hand off control over sections to the paging file to the file system driver and for the file system driver to notify the AFS CM when they are ready to be released and perhaps written back to the AFS file server.

RequestData: This request issues a ….

##### AFS\_REQUEST\_TYPE\_UPDATE\_FILE

Function: This request is issued to the AFS CM in response to a metadata update on the node.

Request Data: The payload structure for this request is a AFSFileUpdateCB. The structure contains the metadata that is being modified where the common header contains information about which node the request is against.

Result Data: The result status indicating whether the operation was successful is returned to the redirector.

##### AFS\_REQUEST\_TYPE\_DELETE\_FILE

Function: This request is issued against a file or directory which is being deleted. The application of a node that has been deleted is applied when the final handle on the node is closed.

Note: In order to support the distributed locking semantics which are implied with the deletion flag, it may be required to issue these requests at the time the disposition attribute is set or cleared as well as when the deletion is actually applied.

Request Data: The control structure passed to the AFS CM is a AFSFileDeleteCB.

Result Data: The AFS CM returns information about the parent in a AFSFileDeleteResultCB control structure.

##### AFS\_REQUEST\_TYPE\_RENAME\_FILE

Function: This request is issued when a file or directory is renamed.

Request Data: The control structure passed to the AFS CM is a AFSFileRenameCB. This structure specifies information about the source and target parent nodes of the operation. The common header specifies the specific node.

Result Data: The AFS CM returns a AFSFileRenameResultCB control block.

##### AFS\_REQUEST\_TYPE\_FLUSH\_FILE

Function: This request is issued when ….

Request Data: The control structure passed to the AFS CM is a

Result Data: The AFS CM returns a

##### AFS\_REQUEST\_TYPE\_OPEN\_FILE

Function: This request is issued when a file or directory is opened in order to perform the necessary permissions check.

Request Data: The control structure passed to the AFS CM is a

Result Data: The AFS CM returns a

##### AFS\_REQUEST\_TYPE\_EVAL\_TARGET\_BY\_ID

Function: This request is issued when ….

Request Data: The control structure passed to the AFS CM is a

Result Data: The AFS CM returns a

##### AFS\_REQUEST\_TYPE\_EVAL\_TARGET\_BY\_NAME

Function: This request is issued when ….

Request Data: The control structure passed to the AFS CM is a

Result Data: The AFS CM returns a

##### AFS\_REQUEST\_TYPE\_PIOCTL\_READ AFS\_REQUEST\_TYPE\_PIOCTL\_WRITE AFS\_REQUEST\_TYPE\_PIOCTL\_OPEN AFS\_REQUEST\_TYPE\_PIOCTL\_CLOSE

Function: This request is issued when ….

Request Data: The control structure passed to the AFS CM is a

Result Data: The AFS CM returns a

##### AFS\_REQUEST\_TYPE\_BYTE\_RANGE\_LOCK AFS\_REQUEST\_TYPE\_BYTE\_RANGE\_UNLOCK AFS\_REQUEST\_TYPE\_BYTE\_RANGE\_UNLOCK\_ALL

Function: This request is issued when ….

Request Data: The control structure passed to the AFS CM is a

Result Data: The AFS CM returns a

#### Ioctl Operations

The following Ioctl operations may be executed against the file system driver.

##### IOCTL\_AFS\_INITIALIZE\_CONTROL\_DEVICE

##### IOCTL\_AFS\_INITIALIZE\_REDIRECTOR\_DEVICE

##### IOCTL\_AFS\_PROCESS\_IRP\_REQUEST

##### IOCTL\_AFS\_PROCESS\_IRP\_RESULT

##### IOCTL\_AFS\_SET\_FILE\_EXTENTS

##### IOCTL\_AFS\_RELEASE\_FILE\_EXTENTS

##### IOCTL\_AFS\_INVALIDATE\_CACHE

##### IOCTL\_AFS\_RELEASE\_FILE\_EXTENTS\_DONE

##### IOCTL\_AFS\_NETWORK\_STATUS

##### IOCTL\_AFS\_VOLUME\_STATUS

##### IOCTL\_AFS\_SHUTDOWN

##### IOCTL\_AFS\_SYSNAME\_NOTIFICATION

##### IOCTL\_AFS\_STATUS\_REQUEST

##### IOCTL\_AFS\_SET\_BYTE\_RANGE\_LOCKS

## AFS Redirector Name Space Support

The AFS Redirector supports accesses to its name space via UNC or drive letter based requests. The redirector registers with MUP as a UNC provider and hence is queried about specific server and share names by the MUP driver. These requests allow MUP to determine which redirector handles requests for the different server and share name combinations on the system. These results are then cached by MUP, for up to 15 minutes currently, in order to facilitate the issuing of the request to the correct redirector. Once a redirector has ‘accepted’ a server/share/path combination, MUP will not query other redirectors until the cache entry has been flushed.

For the redirector to support drive letter based access, an accompanying network provider dll is registered. This user mode library interacts with MPR to direct which drive letter is associated to which redirector. This way a UNC path can be mapped to a drive letter to offer more standardized Windows access.

Once the path has been parsed by the IO Manager with the help of other system components, the name passed into the redirector is a format indicating to which server/share name combination the request is directed as well as the root qualified name for the access. This way the redirector can properly discern to which path the request is being issued.

### Name Resolution

The redirector will handle all opens requests by passing them to the AFS CM if internal consistency checks are passed.

# Initial Delivery Schedule

## Unicode AFS Cache Manager and AFS SMB Server – July 4

1.5.50 released to the general public on July 16.

## KDFS Enabled AFS Cache Manager

### Milestone 1 – May 2

* Establishment of DeviceIoControl communication path between KDFS and AFS CM
* Metadata Browsing of Directories from the AFS root
* Network Provider complete
* pioctl interface permitting use of “aklog, tokens, fs, etc”
* AFS CM extensions
  + Definition of a “user” based upon Process Handle provided with each request
  + Implementation of Open/Cleanup/Close transaction tracking

At this milestone of the implementation, the file system will be loadable and accessible via a mapped driver letter (via the Network Provider) or through UNC name access. Basic functionality will be supported for maintaining the underlying name space for browsing through tools such as Explorer.   
  
No metadata modifications will be supported.   
  
No Data processing will be supported.

### Milestone 2 – May 16

* Metadata operations
  + Create
  + Open
  + Delete
  + Rename

The modifications of the name space will be supported for both files and directories at this point of the implementation.

### Milestone 3 - June 13

* I/O
  + Extent based read operations
  + Extent based write operations
  + Pass-through write operations

This version of the file system and AFSCM will support the enhanced design for IO processing. This method will allow the file system to directly access the cache file by retrieving the file data extents from the AFSCM. The file system will then have pinned pages for data extents  within the cache file allowing for higher performance of large buffer transfers without the necessity of attaching to the AFSCM process.

### Milestone 4 – August 1

* Byte range locks
* Callback Invalidation
* Buffer Cache Page Recycling
* Application Change Notifications
* DFS redirection

This release of the file system will complete the implementation and feature set for the redirector. It will complete the metadata cache page invalidation mechanism for distributed updates to a file or directory. Support for the IO Buffer Cache page recycling by the AFSCM will be handled to allow for the AFSCM to indicate to the file system that it must release buffer cache pages pinned by IO requests.

### Milestone 5 – August 29

* Other

At the conclusion of this phase of development a beta ready redirector will be delivered. The functionality incorporated into the redirector will be near completion but as the driver enters beta testing, additional functionality may be added for support of enhanced features.

### Milestone 6 – September 26

* 64-bit Port
* WOW64 request tagging for @sys processing

### Final Delivery – November 28

Notifications will be handled entirely in KDFS.

# Various Implementation Notes

The FID 0.0.0.0 is a magic value indicating a query for the root.afs for the workstation cell regardless of whether it is a Freelance volume or a real volume.

KDFS functionality falls into four categories:

* Browsing
* I/O
* Metadata Operations
* Change Notifications

Callback processing:

* All AFS callbacks are delivered to KDFS via an Ioctl(). Invalidation messages will be of two types:
  + Single FID
  + Whole Volume (Cell, Volume) only
* Directory Invalidation
  + KDFS will query new content and perform a diff with the previous content to determine if any relevant changes took place
* File Invalidation
  + KDFS will query new file info
* Callbacks are delivered for any of the following events:
  + AFS File Server Lock state changes
  + Attributes
  + Data
  + Metadata

I/O Processing:

In order to optimize performance reads and writes will be performed by KDFS directly out of the AFS Cache file. Communication Port exchanges will be limited to command and control.

Three Stage Process:

1. Call to get extents from Cache Manager. (Performed on first I/O to the specified range. Extents are not referenced counted. cm\_buf\_t objects will be flagged as being held by KDFS. When held no changes may be performed on the buffer.)
2. Read or write data direct from/to the AFS Cache file
3. Call to release extents. (Extents may not be released until “Cleanup” is performed on the file.)

A message will be defined permitting the Cache Manager to request the release of extents for buffer recycling.

The KDFS holds extents for 1K blocks at offsets 4K, 6K and 8K. If the KDFS requests extents for the range 0K to 12K, the Cache Manager will return extents for the full range regardless of the fact that three of the extents were already locked by KDFS.

We will implement a write-through mode. That will permit KDFS to request the cache manager write more data to the file server than would fit in the AFS Cache file. This mode will be used for any file that is opened in PassThrough mode and for any writes in which a request for extents fails due to lack of availability of sufficient free cm\_buf\_t objects.

When using write through mode, if the range overlaps extents already held by KDFS, KDFS must update the content of the extents before they are released. The AFS CM will manage the data version number on the buffers assuming that they were updated.

Changes since November:

* 32-bit Hash value added to FID
* All open/cleanup/close operations need to be passed to the AFS CM.
* In response to each Open, the AFS CM will return an instance ID to be used in subsequent Cleanup and Close operations.
* Cleanup operations will result in the discarding of all locks.
* Between the cleanup and the Close (if there is one), paging write may occur from the SYSTEM context. Must remember the cm\_user\_t that is associated with the Open Instance.
* All requests must in the Process Handle. The AFS CM will use the process handle to obtain the security token which in turn will be used to define the user. (UAC processing will be interesting.)
* All I/O operations on <path>\\_\_AFS\_IOCTL.\_\_ must be passed directly to the cache manager. This is how pioctl() operations will continue to be implemented.

Open:

1. Share and Access control will first be checked by KDFS. If a definitive answer is known, it will be used. If any lock requests are required, the request must be sent to the AFS CM.
2. Otherwise, the AFS CM will be asked.
3. The KDFS will update the share and access data for the FID.
4. Delivering the file open to the CM also permits the CM to pin the file and its data to the cache and reduce the possibility of either the vnode or the associated data buffers from being recycled while the object is in use.
5. The CM should re-use/update/clone the smb\_fid\_t structure for this purpose.
6. Open Issue: When a file is opened for memory mapped I/O the process is Open, Cleanup and then paged reads/writes are issued up until the Close. All locks are dropped at the Cleanup stage. So what happens when a memory mapped file is opened by another process for an exclusive open? AFS will not care
7. Need to associate Process Handle to Security Token.

File Locking:

* OpLocks will not be supported
* Byte Range Lock requests will be passed to the AFS CM.
  + KDFS will check locally
  + Pass a request to the AFS CM
  + Update local lock info
* KDFS will enforce locks at the top-level.
* Locks obtained by KDFS are valid until either KDFS drops the lock or the CM has been unable to renew the lock and the data version on the file has changed

# Known Bugs (as of 9 Oct 2008)

1. Memory leak at shutdown of file system driver
2. UNC path access from Explorer is unreliable. Works from Start->Run but not yet from Explorer bar.
3. AMD64 builds not fully tested.
4. @sys not fully tested.

1. http://msdn2.microsoft.com/en-us/library/cc212363.aspx [↑](#footnote-ref-2)
2. The HKLM\SYSTEM\CurrentControlSet\Control\Lsa\MSV1\_0 “BackConnectionHostNames” registry value specifies a list of Netbios names for which local loopback authentication should be used. [↑](#footnote-ref-3)
3. The AFSCache paging file should be broken up into separate files for each type of data structure (cells, volumes, status objects, buffers, and data blocks). This would permit larger cache sizes on 32-bit Windows and allow room for increasing the number of objects without losing the already cached data. This will become increasingly important when disconnected operations are supported and cache sizes reach into the hundreds of gigabytes. [↑](#footnote-ref-4)
4. The ability to transition between rx listener and rx worker thread states is known as “Rx Hot Threading”. This functionality borrowed from the AFS servers and is not present in the UNIX cache manager. [↑](#footnote-ref-5)
5. /src/WINNT/afsd/cm\_volstat.[hc] [↑](#footnote-ref-6)
6. In theory it could be renamed to cm\_volumeSet\_t and the cm\_volstate\_t object be renamed to cm\_volume\_t. [↑](#footnote-ref-7)
7. Additional volume types may be added to the volume set in the future. [↑](#footnote-ref-8)