SecureShare®: Guaranteed Multiprotocol File Locking

Secure cross-platform lock management for mixed UNIX®/Windows® networks.

Keith Brown and Andrea Borr, Network Appliance, Inc. | April 2005 | TR 3024

TECHNICAL REPORT

Network Appliance, a pioneer and industry leader in data storage technology, helps organizations understand and meet complex technical challenges with advanced storage solutions and global data management strategies.

[TR3024]

1.0. Introduction

Network Appliance provides a built-in multiprotocol file-sharing technology called SecureShare (Patent Pending). Using SecureShare, Network Appliance filers allow UNIX clients—using the Network File System (NFS) and Network Lock Manager (NLM) protocols—and Microsoft Windows systems—using the Common Internet File System (CIFS) or the (PC)NFS protocols (and FTP in some cases)—to share files with a level of data integrity and performance that has never previously been possible using traditional, uniprotocol-centric file service technologies.

The key features of SecureShare are summarized as follows:

- Kernel-integrated cross-platform lock enforcement
- Performance-optimized cross-protocol oplock management
- Cross-platform data integrity
- Cross-platform Change Notify

SecureShare is a multiprotocol lock management facility that is integrated into the Data ONTAP microkernel. SecureShare enables UNIX and Windows applications to concurrently access and update shared files, with the integrity and cache coherency of the shared data being protected by system-enforced locking and file-open semantics. At the same time, SecureShare implements a multiprotocol extrapolation of the Windows networking performance optimization known as opportunistic locks, or oplocks. With the SecureShare implementation of oplocks, Windows applications can reap the performance benefits of aggressive client-side caching with the assurance that SecureShare will implement the same oplock "break" protocol if a UNIX

application attempts to access an oplocked file, as would occur in an environment that was Windows only if the access were based on Windows.

Previous multiprotocol file service offerings implemented Windows file open and locking functionality in a module that was disjoint from, and had no interaction with, the module that implements UNIX NLM locking functionality. By contrast, SecureShare coordinates and manages—in a unified set of kernel-space data structures—all the lock types used with multiprotocol file-sharing. Multiprotocol lock interoperation is crucial to data integrity in a mixed Windows and UNIX environment. In order to achieve multiprotocol lock interoperation, SecureShare must coordinate, enforce, and manage all locks in a uniform fashion, regardless of which protocol created them.

The types of locks managed by SecureShare include the following:

Windows (CIFS) file locks (whole-file locks acquired at file-open time)
Windows (PC)NFS file locks (whole-file locks acquired at file-open time)
Windows byte-range locks (mandatory locking enforcement model)
Windows (PC)NFS byte-range locks (mandatory locking enforcement model)
UNIX byte-range locks (advisory locking enforcement model)
Oplocks (exclusive file locks optionally acquired at file-open time, "breakable" by file access from any protocol)
FTP file locks using the *ftpd.locking* option (locks on a file while download(s) on that file are in progress; any attempts to delete or rename the file during a download are rejected)
Change monitoring file locks (a special type of file lock used on directories to implement cross-platform Change Notify)

SecureShare manages the interoperation of these locks across the full range of NFS, NLM, and CIFS protocol operations, including

CIFS file open operations (resulting in file lock—maybe oplock—acquisition)
(PC)NFS file open operations (resulting in file lock acquisition)
Data access functions (e.g., read and write operations)
File system manipulation functions (e.g., file creation and deletion operations, directory creation and removal operations, file and directory rename or move operations)
Byte-range locking functions (e.g., CIFS byte-range lock operations and NFS NLM byte-range lock operations)
“Delete-on-close” file manipulation where any file marked as “delete-on-close” by CIFS will not be removed until all locks (from other protocols) are removed

From the basic protocol operations that read and write data to the highly complex management of the Windows NT® CIFS opportunistic locking model in the face of concurrent UNIX NFS access to data, SecureShare coordinates all aspects of the NFS and CIFS protocol interactions. Figure 1 shows a high-level overview of the architecture of SecureShare.

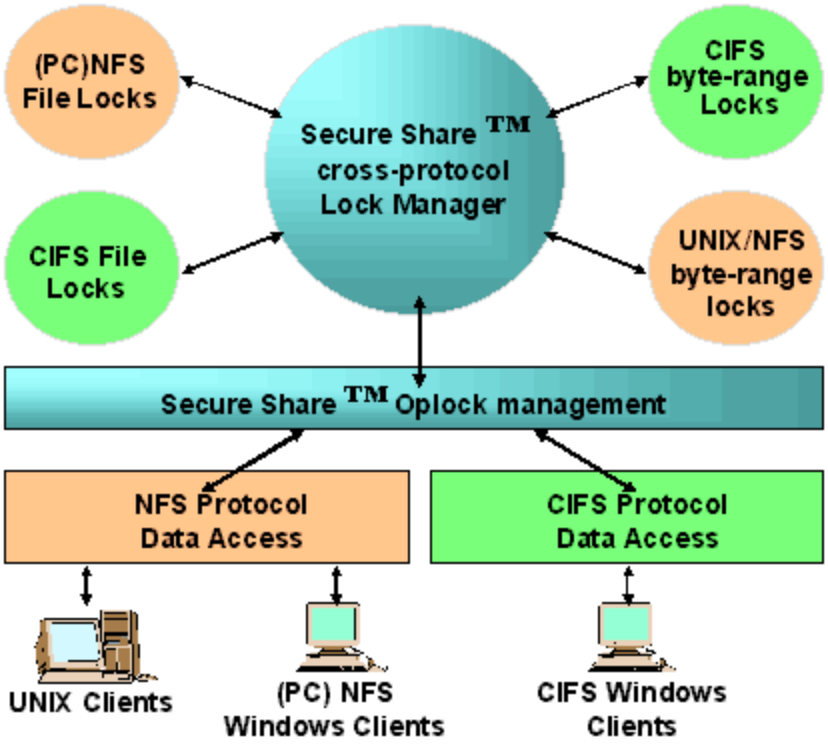

Figure 1) SecureShare high-level architecture.

This paper provides a technical overview of SecureShare technology and its management of NFS and CIFS data access and locking functionality. It also provides a technology overview of the data access and locking functionality found in both the UNIX and Windows operating system environments.

1.1. The SecureShare Advantage

SecureShare technology delivers the following key features to Network Appliance multiprotocol storage appliances:

1.1.1. Kernel-Integrated Cross-Platform Lock Enforcement

Unlike other multiprotocol storage solutions based on general-purpose UNIX systems, SecureShare fully enforces the Windows mandatory file locking and byte-range locking functionality across both Windows and UNIX data access paradigms. Because data access via UNIX does not normally check for lock conflicts, Windows mandatory locks are unenforceable in a CIFS implementation based on UNIX. In such an environment, there is nothing to stop UNIX users and applications from accessing, corrupting, or even removing locked Windows files and data. By contrast, the SecureShare lock manager coordinates multiprotocol lock grant scheduling, lock conflict testing, and lock semantics enforcement at a system level. SecureShare enforces Windows mandatory locking by denying UNIX NFS applications and users access to data locked by Windows.

1.1.2. Performance-Optimized Cross-Protocol Oplock Management

The CIFS Windows Networking protocols feature a performance-enhancing feature known as opportunistic locks, or oplocks. Opportunistic locking is a mechanism that allows a Windows client system to aggressively cache data that is not being concurrently accessed by other systems. SecureShare technology allows Network Appliance filers to fully implement the Windows NT oplock paradigm, even though filer data can potentially be concurrently accessed by UNIX systems via NFS. SecureShare allows Windows systems to take full advantage of the CIFS oplock performance enhancement secure in the knowledge that the oplocks will be correctly "broken" (revoked) if other clients request access to that data, even UNIX clients using NFS.

1.1.3. Cross-Platform Data Integrity

SecureShare has knowledge of which files are open by Windows clients (with which access and deny modes), as well as which files are open with an oplock. At the same time, SecureShare has system level control over NFS accesses to such open files. Thus it can deny NFS operations—such as writes, removes, or renames—that could overwrite and corrupt the file opened by Windows or could delete the file from the Windows application. Such actions could be potentially disastrous to Windows applications if allowed to proceed on files that were open and in active use.

Because SecureShare correctly manages Windows oplocks while taking both UNIX NFS and Windows CIFS data access into consideration, SecureShare ensures that UNIX clients do not receive stale data when accessing files that are concurrently accessed by Windows applications. Other solutions that attempt to implement oplocks without taking UNIX data access into consideration may not deliver the latest version of the data to UNIX NFS clients because Windows clients that hold oplocks do not promptly flush their caches to the server from which the UNIX systems are also sourcing their data.

Similarly, UNIX clients that modify files that are concurrently accessed by Windows systems could potentially have their modifications overwritten by Windows clients that are caching their own writes to those files. Because SecureShare correctly revokes outstanding oplocks before NFS write operations are allowed to a file, the risk of this type of file corruption is eliminated.

1.1.4. Cross-Platform Change Notify

The CIFS file sharing protocol contains a feature called Change Notify. By using this feature, a Windows client can request that the file server notify the client whenever a change occurs in one or more server-based directories that the client wishes to monitor. The implementation depends on the client having the monitored directory open with a special change-monitoring file lock. The notification takes the form of a server-to-client message containing potentially multiple entries, each of which specifies the name of a changed file or subdirectory within the monitored directory, together with the type of change. Examples of the types of changes for which a directory can be monitored are file creation, file deletion, file rename, file move between directories, file attribute change, and file modification time change. When one client changes a directory that is being monitored by other clients, the monitoring clients will be notified of the change so that they can take appropriate action. This Change Notify facility can be observed in the Windows Explorer program. If multiple Windows clients use the Windows Explorer to display a server-based directory to users, any change made to that directory by an individual user will be reflected on the screens of the other Windows clients. Network Appliance SecureShare extends the Change Notify facility to work across platforms. In the SecureShare environment, any change that is made to a server-based directory by either a Windows or a UNIX client (via NFS) will be reflected to Windows clients via the Change Notify feature.

2.0. File- and Data-Locking Paradigms

File-locking functionality is crucial to the provision of data integrity in file-sharing environments. Locking can be used to coordinate concurrent accesses to a file by multiple applications and users. It can prevent concurrent readers and writers of shared data from reading "stale" data (i.e., data currently in the process of being updated by another application) or overwriting each others' updates.

Data-locking paradigms can be divided into two distinct categories:

File locks (also called share locks)
Byte-range locks (also called record locks)

File locks, obtained as a side effect of file open operations, pertain to whole files. The file lock obtained at file open time specifies the type of access the opener intends (read-only, write-only, read-write), and the type of concurrent access on the part of other applications that the opener wants to deny (e.g., deny-read, deny-write, deny-all). In contrast, byte-range locks pertain to sections of a file. They are used to restrict other applications' access to sections of an open file, usually while the holder of the byte-range lock is intending to read or write the locked section. Byte-range locks can be obtained only on already open files.

In addition to these two different types of locking paradigm, there are two distinctly different enforcement semantics commonly implemented by major operating systems. These enforcement semantics are

Mandatory locking
Advisory locking

With mandatory locking, applications and users are "forced" (usually by system code implemented within the operating system) to obey the restrictions imposed by a lock. With mandatory locks, it is usually impossible for a user or application to violate a lock that has been granted to another user or application. By contrast, advisory locking is an enforcement system that relies upon applications and users "volunteering" to comply with lock semantics. With advisory locking, before performing an action (e.g., reading or writing a byte range of the file), an application typically attempts to acquire a lock that would permit the operation. A side effect of the lock acquisition attempt is to test for the presence of another user's conflicting lock that would prevent the operation. Even though the application could technically still perform the operation in violation of the lock, it voluntarily alters its behavior to obey the restriction that the advisory lock imposes.

A major difference between the lock enforcement semantics of UNIX and Windows operating systems is that Windows implements a mandatory locking scheme whereas UNIX systems implement only an advisory locking scheme. The Microsoft Windows and UNIX operating systems also differ considerably in how they allow applications to access files and data while controlling the concurrency issues that arise in multiuser, multiapplication environments. The Windows operating system implements a robust and complete set of file open, data access, and locking paradigms, whereas UNIX systems typically implement only a basic data access facility, ignoring for the most part issues of concurrency and file sharing. These differences in the handling of locking and file-sharing issues carry over into the NFS and CIFS protocols that UNIX and Windows use for remote data access.

2.1. File- Locking in the Windows Environment

The Windows system call interface, also known as the Win32 API, includes a very sophisticated system call that applications use to open files and obtain file locks. This system function is called CreateFile() and is examined below:

CreateFile (LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDistribution, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile)

Although its name suggests that applications use this function only to create files on a Windows system, CreateFile() is used to both create new files and open existing ones. Of particular interest in the context of this paper are the dwDesiredAccess and dwShareMode parameters of the CreateFile() function. These are the parameters used to stipulate the access mode and deny mode that the application wishes to associate with the file. The Windows operating system allows applications to stipulate three types of desired access mode:

Read The application wants permission to read the file.
Write The application wants permission to write the file.
Read/Write The application wants permission to both read and write the file.

In addition to the basic file access mode, Windows also supports the concept of a deny mode, defining what level of access to the file the application wishes to deny other applications while it is holding the file open. Windows supports four basic deny modes that an application can stipulate to the operating system at the time it opens the file:

Deny all

dwShareMode = 0
The application is requesting that all other applications or users be denied all access to the file while it holds the file open. If another application attempts to use CreateFile() to open the file while it is already open with this deny mode, the attempt will fail.

Deny read

dwShareMode = FILE_SHARE_WRITE
The application is requesting that all other applications or users be denied read access to the file while it holds the file open. If another application attempts to open the file for reading (or reading and writing) while it is already open with this deny mode, the attempt will fail.

Deny write

dwShareMode = FILE_SHARE_READ
The application is requesting that all other applications or users be denied write access to the file while it holds the file open. If another application should attempt to open the file for writing (or reading and writing) while it is already open with this deny mode, the attempt will fail.

Deny none

dwShareMode = FILE_SHARE_WRITE | FILE_SHARE_READ
The application is requesting that no other applications or users be denied any access to the file while it holds the file open. Specifying this mode allows other applications to open the file and read or write to it while it is being held open by the calling application.

The combination of the access mode and deny mode used by a Windows application when opening a file constitutes a request for a file lock. File locks can be seen "in action" in many popular Windows applications. Figure 2 shows how Microsoft Word uses file locking to control concurrent access to document files that are located on a network server.

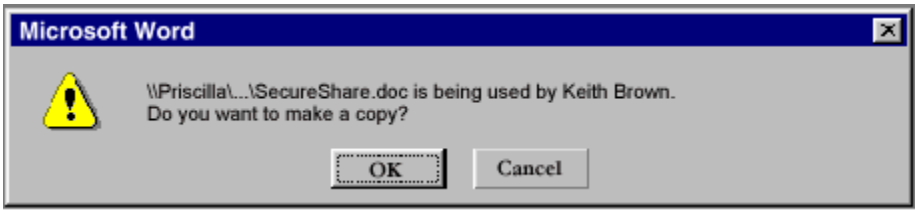

Figure **2) Windows file locks in action.**

2.2. File Locking in the UNIX NFS Environment

In contrast with Windows, UNIX systems do not support the concept of a deny mode when accessing a file. This lack of a file-locking paradigm can be seen when examining the UNIX system call that performs a similar role to that of the CreateFile() function in the Windows environment:

open(char *fname, int mode)

Like the CreateFile() function, the open()system call allows a UNIX application to open a file with an access mode that allows reading, writing, or both. The application stipulates this access mode in the mode parameter of the open() function. However, the UNIX operating system does not allow applications to specify a deny mode when opening a file and, as a consequence, UNIX applications have no true file-locking facility. However, even though UNIX systems do not support or expose a file-locking facility to applications, the NFS Network Lock Manager (NLM) as implemented on most UNIX systems does contain functionality that allows client systems to take out file locks on NFS file servers. This functionality is normally used only by DOS and Windows clients that are using a (PC)NFS implementation to access an NFS server. It was needed to allow (PC)NFS implementations to support the Windows file-locking paradigm on NFS networks. Having no concept of a file lock, UNIX NFS clients do not use this particular aspect of the NLM.

2.3. File- Locking on NFS and CIFS Networks

The CIFS and NFS protocols take considerably different design philosophies when propagating the functionality that is exposed by the CreateFile() and open() system calls back to their respective flavors of file servers. When a Windows application uses CreateFile() to open a file on a network server, the CIFS protocol carries all the information stipulated by the application back to the server. This information includes both the requested access mode and deny mode, allowing the server to permit or deny the open request in accordance with how other applications (if any) have already opened the file. A CIFS protocol server maintains the state of all the client applications that are accessing files located in its local file system, carefully controlling concurrency issues in accordance with how applications have specified that they wish to share the files they have open.

In stark contrast, an NFS server does not maintain any state regarding the activities of client applications and how they are accessing files. The notion of a file being open is inherently stateful, and consequently the NFS protocol contains no function that directly communicates to a server that an application has used the open() system call to access a file. The access-mode stipulated by a UNIX application using the open() system call is usually enforced at the client, not the server.

In the UNIX NFS environment, all supported types of locking are implemented by a separate protocol, residing outside the core NFS protocol. This protocol is called the UNIX Network Lock Manager (NLM) protocol and is usually implemented on UNIX systems as the lockd and statd daemon processes. As previously mentioned, the network lock manager does contain functionality that implements both file- and record-level locking paradigms, although the former is normally used only by DOS and Windows (PC)NFS clients.

2.4. Byte-Range Locking Concepts

Unlike file locking, which is supported only by the Windows environment, both UNIX and Windows systems support the notion of byte-range locking. File locks are imposed by Windows systems when a file is opened, but byte-range locks can be requested only after a file has been opened and is being accessed. There are two major types of byte-range lock used by applications:

Write lock Also called an exclusive byte-range lock, the locked byte-range is read/write for the holder of the lock and deny all for all others.

A write lock on a byte-range is normally acquired by an application that intends to write data into that byte-range, and does not want other applications to be able to read or write to the byte-range while it is accessing that byte-range. A write-lock on a given byte-range is ”exclusive“: it is grantable to only one requester at a time. A write-lock denies other applications the ability to either read or write to the locked byte-range. In environments that implement advisory byte-range locking rather than mandatory byte-range locking, a write lock simply "advises" other applications that they should not read or write to the locked byte-range, even though they are technically able to do so.

Read lock Also called a nonexclusive byte-range lock, the locked byte-range is read-only for the holder of the lock and deny write for all others.

A read-lock on a byte-range is normally acquired by an application that intends to read data from the byte-range, and does not want other applications to be able to write to the byte-range while it is performing the read operations. A read lock on a given byte-range is "sharable": it is grantable to multiple requesters concurrently. However, it is incompatible with a concurrent write lock on the same byte-range. A read lock denies other applications the ability to write to the locked byte-range. In environments that implement advisory record locking rather than mandatory record locking, a read lock simply "advises" other applications that they should not write to the locked byte-range, even though they are technically able to do so.

2.5. Byte-Range Locking in the Windows Environment

Both the UNIX and Windows operating systems implement a basic byte-range (record level) locking paradigm within their respective operating system kernels. Windows applications can use the Win32 API LockFile() or LockFileEx() functions to request a byte-range lock on a file that they already hold open. The LockFile() system call is defined as:

LockFile (HANDLE hFile,
DWORD dwFileOffsetLow,
DWORD dwFileOffsetHigh,
DWORD nNumberOfBytesToLockLow,
DWORD nNumberOfBytesToLockHigh)

LockFile() can be used by Windows applications only to request exclusive byte-range locks (write locks), as the function takes no parameters that allow the application to request any other kind of byte-range lock. In addition to the basic LockFile() function, Windows supplies the slightly more advanced LockFileEx() function which is defined as follows:

```
LockFileEx      (HANDLE hFile,
                 DWORD dwFlags,
                 DWORD dwReserved,
                 DWORD nNumberOfBytesToLockLow,
                 DWORD nNumberOfBytesToLockHigh,
                 LPOVERLAPPED lpOverlapped)
```

As can be seen, this function adds some additional parameters to the original LockFile() function, the relevant parameter for this paper being the dwFlags parameter. If this flag is set to the Windows predefined value LOCKFILE_EXCLUSIVE_LOCK, the function obtains an exclusive byte-range lock (write lock) just as the LockFile() function would. However, if this value is omitted from the dwFlags parameter, LockFileEx() obtains a "shared access" lock, which allows other processes to read the locked byte-range, but not to write to it. Although Windows uses slightly different terminology, a shared access byte-range lock is essentially a read lock. An application would normally use a Windows shared access lock to prevent other applications from writing to a byte-range while it is in the process of reading it.

It is worth noting that Windows supports a 64-bit byte-range locking paradigm, where both the offset of the byte-range to be locked and its extent can be stipulated in a 64-bit value. (Actually each is stipulated in two 32-bit values as defined in the LockFile() procedure above.) It is also worth noting that Windows defines both record-locking and file-locking schemes as mandatory. The Windows operating system enforces byte-range locks across the entire system. Under Windows, no application is permitted to violate the semantics of an existing byte-range lock under any circumstances. Consequently, when a Windows application locks a byte-range within a file, it expects the locking semantics to be enforced against all other applications, even in a multiprotocol environment.

2.6. Byte-Range Locking in the UNIX Environment

The UNIX operating system also defines a byte-range locking model, but unlike Windows, the UNIX locking model is advisory in nature. Even if a UNIX application locks a byte-range within a file, it is still possible for other UNIX applications to "violate" the lock and read or write to the locked byte-range. On a UNIX system, the operating system kernel does not enforce locking. Instead, it is up to applications to enforce locking semantics with a mutually accepted application level protocol.

Most UNIX systems make available three different system functions that allow applications to perform byte-range locking on a file, the main one being the fcntl() system call, which is defined as follows:

```
fcntl( int fd, int command, /* arg */) )
```

To accomplish byte-range locking, UNIX applications use the fcntl() function, passing the predefined values of F_SETLK or F_GETLK as the command parameter to the function. When used to perform locking, fcntl() takes a pointer to a data structure as its third argument, which,

among other things, specifies the type of byte-range lock that the application wishes to obtain. The types of byte-range lock an application can request are:

F_WRLCK	An exclusive byte-range lock (write lock)
F_RDLCK	A nonexclusive byte-range lock (read lock)

The F_GETLCK command of the fcntl() function allows an application to test for the presence of a preexisting byte-range lock that would prevent the specified lock from being acquired. Fcntl() will set a new byte-range lock if one does not already exist that would conflict with the lock being requested.

Most UNIX systems also support two alternative functions that can be used by applications to request byte-range locks. The first of these is called lockf(), which can only be used to obtain exclusive byte-range locks (write locks). The second is flock(), which is an older function that is seldom used by newer applications. The flock() function can be used to request both exclusive and nonexclusive byte-range locks, although in some UNIX implementations, flock() locks are not propagated across networks, rendering them ineffective in network environments (e.g., Sun OS 4.X, Solaris 1.X).

2.7. UNIX Advisory Byte-Range Locking on NFS Networks

The NFS protocol itself contains no functionality that implements the UNIX advisory record-locking model. The NFS protocol is stateless in nature and file- and record-locking functionality is inherently stateful. In order to keep the NFS protocol stateless, UNIX record locking is propagated across the network using a separate, adjunct protocol that is usually implemented in the UNIX lockd and statd daemon applications. This protocol is usually referred to as the Network Lock Manager (NLM) protocol.

The NFS NLM contains functionality for propagating information about UNIX byte-range locks across networks, as well as additional functionality for propagating information about file locks in the Windows style across networks. This latter functionality is used only by (PC)NFS client implementations. It is not exposed to or used by UNIX systems.

2.8. Interoperation of UNIX and Windows Byte-Range Locks

It is crucial in a mixed Windows and UNIX environment, that SecureShare coordinate, enforce, and manage—in a unified set of kernel-space data structures—all byte-range locks, regardless of which protocol created them. Thus, existing CIFS byte-range locks, as well as existing NLM byte-range locks, must be tested for conflict when creating a new NLM byte-range lock, and vice versa. CIFS byte-range locks must be enforced with respect to both CIFS and NFS file accesses. NLM byte-range locks, as well as CIFS byte-range locks, must be enforced with respect to CIFS file accesses.

3.0. Windows Locking Paradigms under (PC)NFS

As previously discussed, Windows systems support two types of locks:

File locks (whole-file locks acquired at file open time)
Byte-range locks (mandatory)

On a CIFS network, these types of locks are implemented using functionality that is specifically built into the CIFS protocol. However, in environments that have chosen to use NFS as the file service protocol with which to network Windows systems, file open time file locking and byte-range locking have to be propagated across the network by functionality within the NFS Network Lock Manager (NLM) facility. (PC)NFS uses the following NLM functionality to emulate CIFS file-open time file locking and CIFS byte-range locking:

NLM "share" locks (equivalent to CIFS file locks)
NLM byte-range locks (equivalent to CIFS byte-range locks)

It is relatively straightforward to see how CIFS byte-range locks map directly to NLM byte-range locks. A Windows byte-range lock is simply translated into an equivalent UNIX NFS NLM byte-range lock by the (PC)NFS implementation and then passed across the network to a server system.

However, as the UNIX NFS environment does not normally support the paradigm of file open, with its implied acquisition of a file lock, (PC)NFS implementations for DOS and Windows must emulate that functionality through use of NLM "share" locks. As previously discussed, for the specific purpose of supporting Windows (PC)NFS clients, the NFS NLM implementation found on most UNIX systems does implement a file-locking mechanism. This functionality is not normally exposed to the UNIX application environment, so most UNIX users are unaware of its presence.

Like most UNIX environments, SecureShare implements both NLM "share" locks and NLM byte-range locks. However, it is crucial in a mixed CIFS and (PC)NFS environment, that SecureShare coordinate and manage—in a unified set of kernel-space data structures—all the locks, regardless of which protocol created them. For example, it is crucial that CIFS-generated file opens represented by CIFS file locks and (PC)NFS-generated file opens represented by NLM "share" locks be treated in a unified fashion with regard to lock conflict detection. Otherwise, incompatible opens from the two environments could coexist and lead to data corruption. Likewise, it is crucial that CIFS-generated byte-range locks and (PC)NFS-generated NLM byte-range locks be treated in a unified fashion.

4.0. The Windows CIFS Opportunistic Locking Model

Although the sharing of files and data between multiple Windows client systems is fully supported by the CIFS protocol, such file sharing between multiple client systems is actually quite rare on most networks. CIFS leverages the defacto rarity of file sharing in its implementation of the Windows networking performance optimization known as opportunistic locks, or oplocks. An oplock is an exclusive (i.e., read-write/deny all) file lock that the Windows client system obtains from SecureShare "opportunistically" at file open time if the file being opened is not currently being accessed by any other application. With oplocks, the Windows client system can enable an application to take advantage of the fact that the file opened by the application is not currently being accessed by any other application. The Windows client system can then optimize the network traffic to the file server needed to satisfy the application's file accesses, without compromising data integrity. The client holding an oplock minimizes network traffic to the file server by

Performing aggressive read-ahead on open files. Because the client knows that it is the only system accessing the file, it can be certain that the read-ahead data that it obtains is not being changed by other clients back on the server.
Aggressively caching write operations to files. There is no need to propagate file changes back to a server synchronously if there are no other clients accessing the file concurrently. Write operations can be delayed and aggregated to optimize I/O performance.
Aggressively caching lock requests. The client has no need to inform the server of the various locks an application may have acquired on a file if it can be certain that no other clients are accessing the file concurrently. The locking semantics can be managed entirely on the client side of the connection.

Figure 3 shows how a Windows client application would normally interact with a CIFS file server when oplocks are not in use.

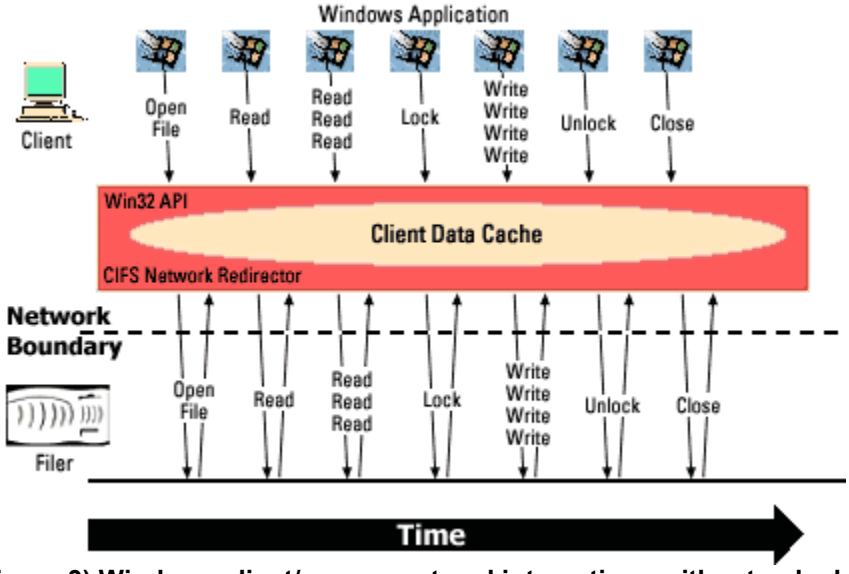

Figure 3) Windows client/server protocol interactions without oplocks.

As can be seen from the diagram, when it does not hold an oplock on the file, the Windows client must synchronously propagate all of the application's interactions with the file back to the network file server. In particular, all of the write operations performed by the Windows client application must be sent immediately back to the server, as must the application requests for byte-range locks on the file. However, Figure 4 shows how the protocol interactions might work when the server has granted the client an oplock on the file.

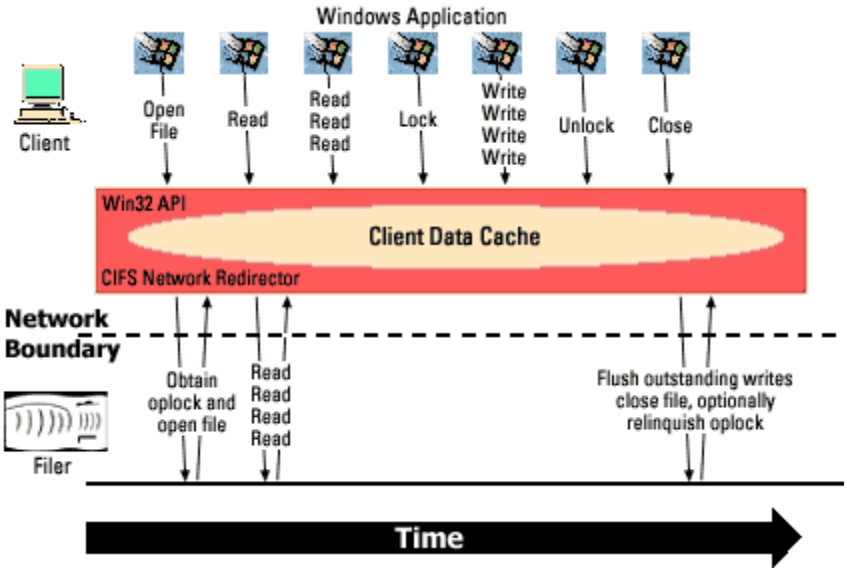

Figure 4) Windows client/server protocol interactions with oplocks.

Oplocks effectively reduce the network traffic that needs to be transmitted between a Windows client and the file server, reducing the burden on both the network and the file server. It is worth noting that oplocks are not exposed to Windows applications through the Win32 API. They are an internal feature of the CIFS protocol and are requested automatically from the CIFS server by the client system at file open time.

4.1. Breaking an Oplock

Although a client can assume that it is the only system accessing a file as long as it holds an oplock for that file, oplocks do not prevent other systems from requesting and gaining access to the same file. When a second client attempts to open a file for which there is an outstanding oplock held by an existing client, the opener is "held off" while the file server attempts to "break" the oplock with its holder (the first opener of the file). When a file server requests that a client "break" an oplock, the client can choose one of two courses of action:

1. Close the file.
2. Flush all outstanding CIFS write and lock operations on the file back to the server and discard any read-ahead data that it may have obtained for the file. The read-ahead data must be discarded because the second client may subsequently write to the file and invalidate the data that the first client originally obtained via read-ahead operations.

Note: An oplock is actually held by a client system, not by an application. Consequently, a client system may choose to hold an oplock (in particular, a type of oplock known as a batch oplock that outlives the application's open) for a file, even after the application that caused that oplock to be obtained has closed the file and exited. If this is the case, a Windows client will usually respond to an oplock break request by simply closing the file.

After an oplock has been successfully "broken," the original holder of the oplock must revert to a mode of operation in which changes made to the file are promptly transmitted to the server, byte-range lock and unlock operations are propagated to the server, and read-ahead is not performed.

5.0. SecureShare Multiprotocol Lock Management

This section describes in detail how SecureShare coordinates the various flavors of locking supported by both the UNIX and Windows operating environments. It also highlights the "risks" to data integrity that are present when SecureShare lock management technology is unavailable in a multiprotocol solution.

This section is divided into two main components. The first describes how Windows locking is enforced in the presence of concurrent data access from other Windows and UNIX client systems. The second section examines how UNIX locking is enforced in the same environment (other Windows and UNIX systems concurrently accessing the same data).

5.1. SecureShare Management of Windows Oplocks

SecureShare supports the complete Windows NT opportunistic locking facility as previously described in this report. Oplocks are managed and coordinated for Windows CIFS access to files, just as they are on a native Windows NT server. However, SecureShare goes further by intelligently managing Windows oplocks in environments where files are being concurrently accessed by UNIX NFS or Windows (PC)NFS client systems.

A Windows CIFS client that requests an oplock for a file being opened on the filer will have that oplock granted if it is the first and only client that is opening the file. Once a Windows CIFS client has obtained an oplock, SecureShare then manages that oplock allocation as other clients subsequently attempt to gain access to the file. The precise semantics of how an allocated oplock is subsequently managed varies according to the type of client that wishes to access the file:

> If a second Windows CIFS client attempts to open the file, SecureShare will attempt to break the oplock held by the first client. To complete the oplock break, the first client either closes the file or keeps the file open but flushes all its outstanding write and lock operations on the file back to the filer. At this point, the second client's open request can be granted—if it is compatible with any file lock remaining after the oplock break. The second client may then access the file.
>
> If a Windows (PC)NFS client should attempt to open the file, SecureShare will attempt to break the oplock held by the first client. To complete the oplock break, the first client either closes the file, or else keeps the file open but flushes all its outstanding write and lock operations on the file back to the filer. At this point, the second client's open request can be granted—if it is compatible with any file lock remaining after the oplock break. The (PC)NFS client may then access the file.
>
> If a UNIX NFS client should attempt to perform a read or write operation to the open file, SecureShare will attempt to break the oplock held by the CIFS client. If the CIFS client completes the oplock break by closing the file, or if the file lock remaining after oplock break completion without a close is compatible with the NFS operation, the NFS request will complete without error. If the NFS request was a read, the read will now access the latest version of the data, as flushed to the filer as a result of the oplock break. If, in the case of NFS write, the oplock break did not result in the file being closed, and the file was opened with deny write or deny all, the NFS request will fail with the UNIX error value EACCESS.
>
> If a UNIX NFS client should attempt to remove or rename the open file, SecureShare will attempt to break the oplock held by the CIFS client. If the CIFS client completes the oplock break by closing the file, the NFS request will complete without error. If the oplock

break did not result in the file being closed, the NFS request will fail with the UNIX error value EACCESS.

It is worth noting the difference in how SecureShare breaks CIFS oplocks in the contexts of UNIX NFS data access and Windows (PC)NFS data access to a file. When a UNIX NFS client attempts to access a file for which an oplock is held by a Windows client, SecureShare will attempt to break the oplock at the time of the first NFS read or write operation to the file. However, when a Windows (PC)NFS client attempts to access such a file, SecureShare will attempt to break the oplock at the time that the file is opened. This difference in behavior is due to the fact that, unlike UNIX NFS clients, Windows (PC)NFS clients request an NLM file lock at the time a file is opened by a Windows application. Thus SecureShare can detect a Windows (PC)NFS client's intent to access a file at an earlier stage than is possible with a UNIX NFS client. As the NFS protocol contains no "file open" operation, SecureShare can only detect a UNIX NFS client's intent to read or write a file at the time that the application actually makes the read/write requests.

Note: Without SecureShare management of Windows oplocks, it is possible for both UNIX NFS and Windows (PC)NFS clients to corrupt data or receive stale data when accessing a multiprotocol file server. NFS clients can receive stale file data from a file server when Windows CIFS clients that hold oplocks for the files being accessed are caching the latest versions of the file data in their local caches. NFS clients can potentially corrupt files by writing into stale areas of server-based files that will later be overwritten when a Windows client flushes its local cache back into the same file. Correct oplock management in a multiprotocol environment is absolutely critical in maintaining data integrity.
It is also necessary for SecureShare to perform oplock allocation management when UNIX NFS clients attempt certain NFS file system manipulation functions. For example, if an NFS client should attempt to remove a file for which a Windows CIFS client holds an oplock, SecureShare will first attempt to break the outstanding oplock with its holder. If the oplock holder does not close the file in response to the oplock break (usually because there is still an application running with the file open on that Windows client), the NFS file removal operation will fail. If the client relinquishes the oplock by closing the file, the NFS removal operation can be performed.

Note: Without SecureShare management of Windows oplocks, it would be possible for UNIX NFS clients to remove or rename files that Windows client systems were actively caching, causing untold problems for Windows users.

5.2. SecureShare Enforcement of Windows File Locks

SecureShare fully enforces Windows mandatory file locking across the Windows CIFS , Windows/(PC)NFS, and UNIX NFS data access environments. When a Windows client attempts to open a file located on a Network Appliance filer, SecureShare tests whether the open request can be granted based on the following criteria:

If other Windows CIFS or Windows (PC)NFS clients already have the file open, a test is made to see whether the access or deny mode of the new open conflicts with access or deny modes already granted to another opener. Table 1 indicates which access and deny modes conflict with existing access and deny modes in the Windows file-locking environment. If the access and deny modes of the new open request conflict with an existing deny mode, the request is rejected.

If a UNIX NFS client holds a byte-range lock within the file being opened, a test is made to determine if the deny mode of the new open conflicts with the access mode (read or read/write) of the NLM byte-range lock. For example, a Windows application can successfully open a file with preexisting NLM byte-range locks only if the open specifies a deny mode of deny none.

New mode being requested	Pre-existing file lock										
	NULL	A:R D:DN	A:R D:DR	A:R D:DW	A:W D:DN	A:W D:DR	A:W D:DW	A:RW D:DN	A:RW D:DR	A:RW D:DW	A:Any D:DA
NULL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A:R D:DN	✓	✓	X	✓	✓	X	✓	✓	X	✓	X
A:R D:DR	✓	X	X	X	✓	X	✓	✓	X	✓	X
A:R D:DW	✓	✓	X	✓	X	X	X	X	X	X	X
A:W D:DN	✓	✓	✓	X	✓	✓	X	✓	✓	X	X
A:W D:DR	✓	X	X	X	✓	✓	X	X	X	X	X
A:W D:DW	✓	✓	✓	X	X	X	X	X	X	X	X
A:RW D:DN	✓	✓	X	X	✓	X	X	✓	X	X	X
A:RW D:DR	✓	X	X	X	✓	X	X	X	X	X	X
A:RW D:DW	✓	✓	X	X	X	X	X	X	X	X	X
A:Any D:DA	✓	X	X	X	X	X	X	X	X	X	X

Table 1
File Lock Enforcement across Windows clients
A = Access Mode, D = Deny mode
✓ = New open request will be granted. X = New open request will be denied.

When a Windows client successfully opens a file on a Network Appliance filer, SecureShare will coordinate NFS access to the open file in accordance with the deny modes that have been granted to the Windows CIFS clients as follows:

> Any attempt by NFS client systems (UNIX or Windows) to delete, rename, or move a file that is being held open by Windows client systems in any access or deny mode on a mixed or NTFS-defined security qtree or volume will be rejected.
>
> Any attempt by NFS client systems (UNIX or Windows) to delete, rename, or move a file that is being held open by Windows client systems in any access or deny mode on a security qtree or volume defined by UNIX will be allowed.
>
> Any attempt by an NFS client to write to a file that is being held open by a Windows client with a deny mode of deny write or deny all will be rejected.
>
> Other NFS operations will normally be allowed.

Note: Unlike most other multiprotocol network storage solutions based on the UNIX operating system, SecureShare is able to manage NFS file system manipulation functions at the system level so that they do not violate or interfere with files and directories that are being accessed by CIFS-based Windows clients. On most UNIX multiprotocol solutions, it is possible for UNIX users to remove, rename. or move files that Windows systems are holding open and actively accessing. **Network Appliance filers with SecureShare technology provide the industry-leading solution in protecting user data from this potentially disastrous type of violation.**

It is worth noting that the SecureShare design does allow UNIX NFS client systems to read files that are being held open by Windows CIFS or Windows (PC)NFS clients, even if those files have been opened with deny modes of deny read or deny all. This design decision was consciously

made with consideration to practicality over technical "purity.” The ability of a UNIX NFS client to read files that have been opened by Windows systems cannot harm or damage data. However, if SecureShare were to deny a UNIX system NFS read operations on all files that were actively being accessed by Windows systems, this could cause an unacceptably large number of seemingly inexplicable errors on UNIX client systems reading data from a filer file system.

Note: Even though the SecureShare design allows NFS clients based on UNIX to read files that have been opened in deny read or deny all mode by Windows clients, Windows (PC)NFS clients are not granted this ability, even though they are using the same NFS client protocol as UNIX systems. All Windows deny modes, including deny read and deny none are enforced across all Windows client systems, whether they are using CIFS or (PC)NFS to access a Network Appliance filer.

5.3. SecureShare Enforcement of Windows Byte-Range Locks

As previously noted, Windows applications usually request byte-range locks by calling the Win32 LockFile() or LockFileEx() system functions. When used on server-based files, these calls result in CIFS byte-range lock requests being sent to the filer. Enforcing Windows byte-range locking is a relatively straightforward process that occurs as follows:

> If another Windows CIFS client system is already holding a byte-range lock that is in conflict with the CIFS byte-range lock being requested, the lock request will be denied.
>
> If another UNIX NFS or Windows (PC)NFS system is already holding an NLM byte-range lock that conflicts with the CIFS byte-range lock that is being requested, the lock request will be denied.
>
> If neither of the above situations is true, the lock request will be granted.

Note: Unlike other multiprotocol network storage solutions based on the UNIX operating system, SecureShare is able to manage NFS file access functions at the system level so that they do not violate or interfere with files in which byte-ranges have been mandatory-locked by Windows systems. On most multiprotocol solutions based on UNIX, it is still possible for UNIX users and applications to write data to the Windows CIFS locked byte-ranges of files due to the advisory nature of UNIX NFS locking. **Network Appliance filers with SecureShare technology provide the industry-leading solution in protecting user data from this potentially disastrous type of violation.**

5.4. SecureShare Enforcement of UNIX Byte-Range Locks

As previously discussed, UNIX byte-range locking is performed across networks using the NFS Network Lock Manager (NLM) facility, usually implemented as the lockd daemon process on most UNIX systems. SecureShare handles a new UNIX NLM byte-range lock request as follows:

> SecureShare first breaks any preexisting oplock that may be held on the file by a Windows CIFS client. This ensures that the SecureShare lock manager has all the information on preexisting Windows CIFS byte-range locks for the file. Windows CIFS clients that hold oplocks do not propagate lock information back to the filer.
>
> SecureShare next tests whether the UNIX NLM byte-range lock request conflicts with any file lock (deny mode) that has already been granted to a Windows application:
>
> - If the UNIX NLM byte-range lock request is for an exclusive lock (write lock), it is incompatible with a preexisting CIFS open having a deny mode of deny write or deny all.

- If the UNIX NLM byte-range lock request is for a nonexclusive lock (read lock), it is incompatible with a preexisting CIFS open having a deny mode of deny read or deny all. This logic is expressed in Table 2.

SecureShare then tests whether the new UNIX NLM byte-range lock request conflicts with any preexisting NLM or CIFS byte-range lock that has already been granted on the same file. If a conflict exists, the lock request is rejected.

If none of the above violations is detected, the NLM byte-range lock request can be granted.

Pre-existing file lock (CIFS open)											
	None	A:R D:DN	A:R D:DR	A:R D:DW	A:W D:DN	A:W D:DR	A:W D:DW	A:RW D:DN	A:RW D:DR	A:RW D:DW	A:Any D:DA
Write Lock	✓	✓	✓	X	✓	✓	X	✓	✓	X	X
Read Lock	✓	✓	X	✓	✓	X	✓	✓	X	✓	X

Table 2
Compatibility of NLM byte-range locks with Windows/CIFS file locks
A = Access Mode, D = Deny mode
✓ = New NLM byte-range lock request will be granted
X = New NLM byte-range lock request will be denied

Note: SecureShare maintains the standard advisory record-locking model for data that is being accessed via NFS. As is the norm in the UNIX NFS environment, applications must be instrumented to the NLM locking system to properly control concurrency issues when accessing data. NFS requests that violate the semantic meaning of an NLM lock will be permitted by SecureShare, just as they would be permitted by an NFS server and NLM implementation based on UNIX.

6.0. The Windows NT Change Notify Facility

The CIFS networking protocol version implemented by Windows NT supports an advanced file system monitoring facility called Change Notify. This facility allows Windows applications to efficiently monitor file system directories located on a Windows NT server for modifications that may be being made outside the scope of the monitoring application.

Users can observe the functionality provided by Change Notify by using the Windows NT Explorer program. This program makes extensive use of the Windows change notification API to keep users' on-screen views of directory contents up to date. If two separate Windows clients use the Explorer to view the contents of the same server-based directory, changes made to that directory by either client will immediately be reflected on the other's screen. For example, if the first client creates a new file in the common directory, the Explorer program running on the second client will be asynchronously notified of the change to the directory contents and will refresh its view pane to show the newly created file.

Without an efficient change notification mechanism built into the CIFS protocol, client applications would be forced to repeatedly scan directories across the network to monitor for changes. Such activity would be extremely inefficient from the perspective of the load that would be incurred on the client and server systems, as well as the network between them.

Although Change Notify is arguably most visible to users through the Windows Explorer program, the facility is actually much broader in application. Using the Windows system API, any application can register its interest in receiving change notifications for any given directory on a

file system. The Win32 system calls that are used by Windows applications to take advantage of the Change Notify facility are

```
HANDLE
FindFirstChangeNotification     (LPCTSTR lpPathName,
                                 BOOL bWatchSubtree,
                                 DWORD dwNotifyFilter)
```

This call is used by an application to register its interest in receiving change notifications for the directory specified in the lpPathName string. The second parameter specifies whether the application wishes to receive change notifications for just the directory specified or for the directory and all its underlying subdirectories. Finally, the third parameter specifies the kinds of changes the application is interested in being notified about. Applications can watch for any combination of many different types of directory changes, including file creations, deletions, and renames; directory creations, deletions, and renames; file size changes; and file attribute changes. The handle returned by this function is later used by the application to wait for and collect change events (see below).

```
DWORD
WaitForSingleObject             (HANDLE hHandle, DWORD
                                 dwMilliseconds)
```

This call is used by an application to wait for a change notification event that it has previously registered an interest in receiving. The function blocks in an efficient wait state until the event arrives. The first parameter is the handle returned in a previous call to FindFirstChangeNotification(), the second parameter is a timeout value specifying the maximum time the function will wait for a change notification. The Windows WaitForSingleObjectEx() or WaitForMultipleObjects() API calls can also be used to monitor for change notification events

```
DWORD
FindNextChangeNotification      (HANDLE hHandle )
```

This call is used by an application to register its interest in receiving subsequent change notifications after the first notification has been received (using the same event mask and directory as before).

```
BOOL
FindCloseChangeNotification     (HANDLE hHandle)
```

This call is used by an application to notify Windows that it is no longer interested in receiving events on this change handle.

6.1. SecureShare Cross-Platform Change Notify Technology

The Data ONTAP microkernel fully implements the Windows NT change notification mechanism. Windows applications can use the Win 32 change notification APIs to efficiently monitor filer-based directories for changes, just as they could with directories that were located on Windows NT server systems. A Network Appliance filer will asynchronously notify Windows clients of changes to monitored directories in the same way that a Windows NT server would.

SecureShare also extends the standard Windows NT Change Notify facility into the cross-

platform, multiprotocol realm. With SecureShare, modifications that are made to Network Appliance filer-based file directories by UNIX NFS client systems or Windows (PC)NFS client systems will trigger asynchronous CIFS change notification events that inform Windows CIFS clients of the changes being made. Independent software vendors can utilize Network Appliance filers to develop cooperating, cross-platform applications that communicate through the filer's file system. UNIX applications can process files and then hand them off for further processing by Windows NT applications, simply by copying the files into directories that are being monitored by the cross-platform Change Notify mechanism.

Note: The Change Notify mechanism is actually implemented using a special case of a Windows file lock. An application that uses the FindFirstChangeNotification() call to register its interest in receiving asynchronous change notifications takes out a change-monitoring file lock on the directory in which it is interested (by opening the directory). Change Notify is thus an integral part the SecureShare integrated lock management functionality.

7.0. Summary

In delivering SecureShare to the marketplace, Network Appliance is the leading vendor of multiprotocol network storage solutions which fully addresses the data integrity and data safety issues that are inherent in cross-platform file-sharing environments. SecureShare protects both Windows and UNIX data from all the major data corruption problems that exist when the same files are being accessed concurrently by UNIX and Windows platforms. A multiprotocol environment without SecureShare is like a ticking bomb waiting to explode in the user's data center. Integrating the many different flavors of locking paradigms that are prevalent in the UNIX and Windows environments is a complex task. Most vendors have historically chosen to either ignore the problem completely or implement just the simplest forms of integrated lock coordination. Only Network Appliance SecureShare encompasses the management of all the locking paradigms across the entire NFS and CIFS protocol suites.

8.0. Revision History

Date	Name	Description
03/31/2005	Network Appliance, Inc.	Revision
01/01/1999	Network Appliance, Inc.	Creation

NetApp®

Network Appliance, Inc.
495 East Java Drive
Sunnyvale, CA 94089
www.netapp.com