Examining Semantics In Multi-Protocol Network File Systems

Edward P. A. Hogan[1], Garth A. Gibson[2],
and Gregory R. Ganger[3]

January 2002
CMU-CS-02-103

Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Network file systems provide a robust tool that can be used by many physically dispersed clients. They provide clients with a means of permanent storage and communication. In order to exploit the resources available on a network file system server, a client must use the protocol of the server's file system. Although the goal of any protocol is to guarantee that the client and server can communicate, the introduction of new protocols divides clients into incompatible sets. Soon clients can no longer cooperate and share because they are using different protocols. In addition, each network file system is constructed with a different set of semantics. The result is that it is increasingly difficult to provide a single storage solution that supports all of these clients.

Although difficult, it is extremely desirable to build a multi-protocol network file system, that is, a storage solution that can be used simultaneously by clients of different protocols and semantic sets. A semantic mismatch is a major complexity in building a multi-protocol network file system. These are situations that arise when the normal behavior of a server, expected by a client using a particular semantic set, does not occur because of the effects of a client from a separate semantic set. To achieve the goal of building a multi-protocol file system, the file system semantic sets of the targeted file systems must be carefully examined to determine where semantic mismatches will occur. Next, the possible means of resolving a semantic mismatch can be analyzed for their particular trade-offs. Finally, data from file system traces can be used to determine the frequency of possible semantic mismatches. The data collected from the file system traces, when examined in the context of a cost-benefit analysis, can provide designers of multi-protocol network file systems with important information for examining and resolving semantic differences.

1. Panasas Inc., can be reached via email at *ehogan@panasas.com*
2. School of Computer Science, can be reached via email at *garth@cs.cmu.edu*
3. Department of Electrical and Computer Engineering, can be reached via email at *ganger@ece.cmu.edu*

We are indebted to Randy Appleton, the members of Coda project at Carnegie Mellon University, and the SEER project at the University of California, Los Angeles for providing access to the file system traces used in this research. We thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support.

Keywords: file systems, network file systems, multi-protocol file system, file system semantics, storage devices, operating systems, storage management.

1 Introduction

In the field of network file systems, there is a trend towards larger numbers of heterogeneous file system clients. This trend arose as a result of the existence of different computer platforms, file system client versions, and the differing goals of each file system. As more network file systems become available, it is increasingly difficult for software architects to create a file system server that can interact with these heterogeneous clients.

A typical organization will usually deploy several network file systems in an effort to support each set of file system clients [Hitz99]. There are several common file systems that most organizations will deploy; yet, not all the clients are equipped to communicate natively with every server. One typical solution to this problem is to overload a client system with software so that it can communicate with more and more file servers. This solution works as long as the number of server varieties remains low, but usually results in users maintaining multiple copies of the same files on multiple servers so that they can operate on them. This situation leads to several inefficiencies, it not only wastes the available storage space of the servers, but also increases the administrative costs of the organization for managing each of the separate file servers, and usually results in a version skew in the files that the user needs to manage. In short, it creates a situation that wastes administrative resources and requires users to learn unnecessary details of each particular file system.

To address this problem, a software architect would like to design and build a multi-protocol file system. This system would be capable of responding to requests from a variety of clients. Ideally, this solution would not require changes in the existing client software. From a user's perspective, this is a desirable property because it means the user can get work done without setting up and learning how to operate with the new system. This also serves to reduce administrative overhead since there is only one system that needs to be backed up and maintained. It also lowers resource costs, since there is only one system on which to concentrate hardware resources and reduces the amount of redundantly stored data residing on multiple file servers.

Although a multi-protocol file system is desirable, implementation is very complex. The first obstacle is providing the means of communication between each set of heterogeneous clients. Solving this involves being able to handle a wide variety of client communication protocols and data formats. The next obstacle that must be surmounted is to resolve the differences in the meanings of commands and command arguments used by the clients. These meanings make up the semantic set of the file system. Since each client logically links its set of operations to a set of expected outcomes, the server must be able to correctly respond to each client without breaking the expectations of a concurrent client from a different semantic set. The third obstacle is to provide acceptable performance to each client. Handling each set of client semantics may introduce overhead on the entire system, however this overhead should be minimized in order to make sure the system has usable performance.

This paper will begin by presenting background information that will describe traditional uses and goals of file systems. It explains what semantic mismatches are and why they occur, and describes some of the file system trace research that was used to examine semantic mismatches in further detail. The next section of this paper describes some issues concerning semantic mismatches. It describes some of the design choices selected by some network file systems, and the cost and benefits of these approaches. This analysis is followed by data collected from some of the file system traces, which is used to determine the potential frequency of semantic mismatches in a multi-protocol file system. This semantic mismatch is examined to determine a potential means of resolving the mismatch for some common file systems. Finally, this paper describes some work related to this project and is followed by conclusions.

2 Background

In order to examine multi-protocol file systems, it is first necessary to examine several related issues. This examination will clarify the ideas leading up to and motivating modern multi-protocol filing. The background issues examined in this paper are the traditional uses for file systems, how these uses have changed and been overloaded, some common network file systems that are in use today, and a look at the need for multi-protocol file systems. In addition, this section also describes semantic mismatches, why they occur, and then enumerates and describes the categories of mismatches. Finally, it will look briefly at some file system traces that were used in this project to analyze the semantics of file systems.

2.1 Traditional file system uses

Since the development of the first file systems, several important tasks have been added to their original set of functions. The primary goal of a file system is to provide a means of permanent data storage for applications. Once data has been stored on the file system it could be retrieved later by that application or by another cooperating application. This property is called persistence or durability. However, the file system's ability to provide persistent data storage has become overloaded as a means of sharing data and even used as a tool for communication between concurrent applications.

First, the fact that the primary goal of filesystems is to provide a means of permanent storage means that the storing server and application client have established a contract. In this contract, the server has promised the client that it can access the identical data bytes by using the filename used at creation time.

Once a client receives durable storage and a contract to access its data later, this client can then share this contract with another client. By doing so, both of these clients can cooperate and begin to share this data. At first look, it may appear that sharing data can occur as soon as two clients have access to the contract, which is the filename for the data. In fact, much more must be done to enable the clients to cooperate. The clients must agree on the layout of the data in the file. They must agree on where the file should be located within the organizational structure of the file system to enable it to be found when it is next needed. The clients may want to decide some properties of the file, such as the size, for example, so that they can quickly determine some high-level information about its contents. In addition, these clients need to agree on the security status of the file so that they can share it other clients and protect it from access by undesirable clients.

One consequence of providing clients with the ability to access and share these files is that eventually two or more of these clients will access the same file simultaneously. This could occur either intentionally or accidentally. When it occurs intentionally, it may be the desire of the clients to overload the store and retrieve capabilities of the file system to create a message-passing medium. In this model, the file system capability for *writes* and *reads* are the equivalent to *send* and *receive* communication primitives, respectively [Mann94]. When simultaneous access occurs accidentally, some of the clients may believe that their contract that provided for durable storage has been violated. A simultaneous client could overwrite the data written by another client, causing one of the clients to stop functioning when it later goes back to read the data and finds that it has, from its perspective, been 'corrupted'.

In short, the initial goal of providing for a means of permanent storage has been both enhanced and overloaded in the file systems that we use today. Because of the added responsibilities that have been added to file systems, it is necessary that a multi-protocol network file system be designed carefully and correctly since it will have to honor contracts from several file system clients at the same time.

2.2 Common network file systems

When designing a multi-protocol network file system, it is first necessary to study the current set of network file systems. In this way, a designer can determine the set of clients and the semantic sets that clients are already using. This section will investigate several important network file systems that are discussed in depth throughout this paper, they are: NFS, CIFS, AFS/Coda, and Sprite. In addition, this section will also briefly look into FTP and HTTP, Internet standards that can provide file system-like abstractions.

2.2.1 Sun Network File System (NFS)

Created to serve primarily Unix clients, Sun NFS is a network file system that was built to provide clients with a set of semantics that were as close to the local Unix file system as possible without losing performance [Sandberg85]. A client uses a file handle to access an NFS file just as it does for a file stored on the local file system. The NFS server will grant a file handle to a client when the client presents the server with the name of file that it wants to access along with the client's credentials. In NFS, the client machine is responsible for providing the proper credentials; therefore it is the client machine that provides the information used to enforce the security of the file system. A poorly written or intentionally malicious client may provide incorrect information that results in incorrect security policy decisions. Unfortunately, the fact that the server must trust the client to make correct access decisions creates a security problem since the server cannot guarantee that the client is making the right decision.

Once a client has received a file handle from a server, it is able to reuse the handle to access the file. Since NFS was providing multiple clients the ability to work simultaneously on files over a network, the designers' goal was to avoid the possible error situations that could arise if any of the many clients crashed, the server crashed, or if there was a partition in the network that made the clients or the server temporarily disappear off of the network. To handle these kinds of failures, the designers decided to follow a stateless model, meaning that the server did not keep track of which client machines had open file handles [Pawlowski94]. When an application called open on an NFS file, the client would gather the credentials for the targeted file, and then send a request to the server. Once the server receives the request it responds to the client with a file handle that is generated by looking up the *inode* number of the file. The inode number uniquely identifies the target file on the underlying storage media. The server does not keep any information about this handle. From that point on, the client can reuse the handle to read and write the data in the file. Because the server uses a stateless approach, if the server crashes and restarts, the client's handle may fail, but the client can simply request another handle to continue accessing the file. If the client crashes, the server is unaffected because it does not keep any information about the client. The disadvantage to the stateless approach is that a client maintaining data from a server for a long period of time must communicate often with the server to revalidate the data. A client using a stateful file server would be able to cache the data to use repeatedly until the server informs the client that the data has changed [OSF].

In terms of its caching policy, there are several caching policy implementations depending on the version of NFS. In NFS version 2 (NFSv2), all data modification operations are written through the client's cache and sent directly to the server and must be committed to stable storage before being allowed to responding to the client. Each server implementation uses its own definition of what technically constitutes as being in stable storage, some NFS server implementations commit the data to disk, others write to a non-volatile memory or to a file system log. The client keeps a cache of recently read data, and maintains consistency by periodically verifying its cached data with the server. It verifies directory information frequently to reduce the probability that clients have differing views of the file system hierarchy, and verifies the validity of file data less frequently. The result from the perspective of the file system client is that the client has a slightly delayed view of the file system, and that there is a large amount of traffic between the client and the server used to check the consistency of the client's cache.

Although this is the behavior specified in the NFS version 2 protocol specification, some implementations of this protocol use a differing caching policy. In Linux' implementation of the NFS version 2, when an NFS client writes to a portion of a file, the data written actually modifies a block of data in the client's local data cache. After some period of time, the modified data is sent to the file server and modifies the actual storage media contents of the file. Using this write-behind (delayed write-back) caching scheme means that the Linux NFSv2 file system client reports to the calling application that a write is successful before the write has actually been completed. The future write may fail without an error being reported to the calling application. Another consequence of the caching policy in Linux NFSv2 is that write behavior can appear strange from a concurrent client. Because Linux NFSv2 clients delay the writing back of modified data, they may even delay the writing of the data until after the file has been closed on the client. This semantic choice puts concurrent clients in the situation where reads to a file and attributes for the file will display that no write has occurred, until after the write back has occurred. For the time period between the file system write event and the write back of the data, the client and server are inconsistent, meaning that they have different views of the data stored on the file. The notorious evidence of this cache inconsistency is that code compilation systems, particularly ones such as *make* that uses the modified timestamp on a file to determine whether the file needs to be recompiled, do not always work correctly when not used on the same client machine as the machine modifying the code. In this situation, the build system does always notice that a file has been modified because the client has not yet sent the data to the server. The use of delayed *write back caching* also has important

implications when multiple clients are writing to the same file. In this scenario, the resulting storage media contents of a file are not based on the order that the writes to the file occurred, as is the norm in most file systems, but are based on a combination of both of the clients' writes, the result of when the file system client eventually flushes the write events to the server.

In the NFS version 3 protocol (NFSv3), the caching policy has been altered to allow clients to temporarily cache writes before sending them to the server [Pawlowski94]. This was done because analysis of the NFS version 2 protocol showed that there was a bottleneck in the system because writes operations needed to be consist on the server's disk. In this newer version of the protocol, the client can perform asynchronous writes to the server. It can then flush all of the write operations to the server's stable storage through the use of a new commit operation. This operation allows the client to perform true write back caching.

The choices of file system semantics are adequate to emulate a local Unix file system when there is only a single client accessing a particular file. However, when there are concurrent accesses to the file, the semantic choices do not allow the system to resemble the local file system. In fact, a concurrent client reader notices concurrent write events a period of time after the actual event occurs, this is because the reading client must contact the server periodically to check the correctness of the information in its cache. Since it does not contact the server before serving each piece of data, it may serve old data before contacting the server and noticing that the data has been invalidated. This is necessary because there is no means for the server to contact and notify the client of a change

From the point of view of a native Windows client, NFS does not fit in easily with its set of semantics for file systems because in general Windows locks files to prevent applications concurrently writing to files. To use NFS on Windows, most implementations use a separate locking protocol to make the file system appear to be more similar to Windows. Unfortunately, since the Unix clients do not require the locks, the result is that an NFS server that services Windows and Unix clients typically receives lock requests from the Windows clients but not from the Unix clients.

Even though Sun NFS has a set of general characteristics, each version of NFS that exists uses slightly different semantics, these versions are: NFS version 2, NFS version 3, Linux NFS, BSD's Not Quite NFS (NQNFS), and PC NFS.

2.2.2 Common Internet File System (CIFS)

Another common network file system in use today is CIFS, which is the current form of Microsoft's SMB protocol [Leach97]. Microsoft developed this network file system originally for DOS and Windows-based computers. Like Sun NFS, this was a network file system created to give one set of clients a semantic set similar to the one used on a local machine. In CIFS, clients are able to request contracts that are much more restrictive than are available in Sun NFS. For example, since early CIFS clients were designed for systems that ran network-unaware applications, the client required that the server guarantee that a concurrent client would not change the file presented to the client in any way. Therefore, initially CIFS servers only allowed one contract to be issued for a particular file, and denied all other requests for contracts or operations on a file while another contract is issued for that file. Enforcing this behavior means that the server must be dramatically different from the NFS server model. In CIFS, when a client opens a file, the server keeps track of the client and the file. Whenever another client wants to perform an operation on the same file, the server will notice that the file is in use and will deny the operation. This is an example of a stateful approach to file serving; the server has the advantage that it can make decisions based on information it maintains about the activities of other clients. However, the amount of information kept by the server increases as each new client is added, decreasing the scalability of the file system. In addition, the stateful approach that the CIFS server uses is not as easily fault tolerant as the stateless NFS approach. In the event that the CIFS server crashes and restarts, all of the contracts for file use issued to clients are no longer valid. In addition, in the event of a client crash the server will still be keeping information about the files that it believes are still in use by the crashed client. The server will prevent access to the files until it determines that the client has crashed. The server learns that the client crashed either through being informed by the client after restart or after a long timeout.

Later, as the operating system and applications that used the CIFS clients became more network-aware, it was useful to be able to issue multiple contracts on the same file. This required that CIFS change its file system semantics. Older clients would still request contracts preventing concurrent file usage but newer clients could request a less restrictive contract that would permit concurrent accesses. This semantic has interesting ramifications on the caching

policies of the CIFS clients. In older versions of CIFS, the file system client could cache data and delay writing back changes until the file is closed and there would not be any cache consistency issue since no other clients were allowed to access the file concurrently. In newer CIFS implementations, clients use a delayed write back approach when they are the only users of a file but they change their caching policy as soon as another client opens the file to examine the new changes. In this situation, the server will inform the client that another file is interested in the file, the client will then write all of its pending changes and move to a write through approach. The write through approach means that when data is written to a file it is immediately sent to the server. The writing of the data to stable storage on the server is no longer delayed.

Another major difference in CIFS from the model used in Sun NFS is that the CIFS server rather than the CIFS client machine enforces the security of the file system. This policy means that the server has to perform some amount of work verifying that a client has permission to perform an operation, but it allows a better security model than simply trusting the information from the client.

Much like NFS, there have been many versions of CIFS over its long history. Each version and implementation of CIFS has a slightly different semantic set. Some of the major versions of CIFS that exist are: LAN Manager (Lanman) versions 1.0, 1.2, and 2.1, NT LM 0.12, Server Message Block (SMB), and Samba. Perhaps the most interesting version of CIFS from a multi-protocol file system perspective is Samba since it is a CIFS server that runs on a Unix platform. Because of the differences between the underlying Unix file system and the presented CIFS file system view, there are a number of semantic issues that arise.

2.2.3 Andrew File System (AFS)

AFS is a distributed file system that was built at Carnegie Mellon University; its goals were to provide file system access to a large number of primarily Unix clients. In order to achieve its scalability goals, AFS took advantage of research into file system usage that pointed out that sharing data was extremely rare [Howard88, Satya92]. In fact, their research found that most concurrent access to files is concurrent read sharing of binary executable files. And that in general, Unix users write primarily to their home directory and to the temporary directory. Based on this knowledge, Andrew was able to alter the contract that it provided based on the statistical behavior of the clients. Instead of providing clients with a guarantee that no other client would access the file as in CIFS, AFS servers allow simultaneous contracts to be issued on any file, but expected that concurrent write sharing would be very rare. Because of the expected rarity of write sharing, AFS designers wanted to develop a sharing semantic that would not deteriorate as the number of users increased. To achieve this goal they developed a highly scalable sharing semantic called private copy until close. In this semantic, the AFS server provides each client with a contract to access a file, the client uses this contract to transfer a private copy of the entire contents of the file from the server. This private copy of the file resided in the local cache of the client's machine. Any read or write accesses to the file are done to the local private copy. If the file is modified, then when the file is closed the new version of the file is sent back to the server. In the rare case that two clients are writing to the same file, neither client will see the other's changes. When the clients close the file, the last client to close the file will be the client whose version of the file remains on the server. In short, a client makes all changes to its private copy without affecting any other clients that also have contracts on the same file.

The advantage of caching an entire private copy of a file in the Andrew File System is that part of the contract granted by the server to the client during the file open included a *callback* that had additional benefits above and beyond the use of a file during a single session. As long as the callback on the file remains valid the client is permitted to open, read, write, and close the file. In the event that another client modifies the file, the server will break all of the callbacks that are outstanding on a file by contacting each of the clients that has the file open. The callback can also become invalid as it ages and exceeds its expiration time. The designers sought to increase performance by dramatically reducing the amount of contact a client need with the server. Ideally in AFS, the client only needs to contact the server to receive a file and to renew the callback. This is a large contrast to the near-constant communication necessary in the NFS model. The disadvantage to this approach is that the server must keep state about each of the clients and the files that they are accessing.

As the Andrew File System and its descendants evolved they developed several variations on their semantics. First, with regard to callbacks, the designers decided to allow the creation of whole volume and whole directory callbacks. These are callbacks whose validity is tied to that of an entire set of files. These whole directory callbacks are useful because they reduce the number of callbacks that an AFS server has to manage and it also helps aggregate callbacks

on sets of files that are rarely changed. Later versions of AFS cached 64 kilobyte sections of the file rather than the entire file. This was helpful for an AFS client that needed to utilize a file that was larger than the client's local cache.

Another goal of AFS was to provide strong security guarantees that were not done with network file systems at the time. AFS provided for a means of describing file access permissions that was richer than the traditional Unix permission set. This was done through the use of access control lists. An access control list is a list of access control entries; each entry describes an identity and an associated permission. The identity can describe a single user or a group of users. The permission describes the operations that are explicitly permitted or denied, depending on whether it was a positive or negative access control entry. Through the construction of an access control list, the Andrew File System could allow rich and complex ways of describing permitted operations. The second component to the AFS security model was the use of Kerberos security to make sure that the clients and the server are mutually authenticated. This security measure ensured the server that only properly authenticated clients were able to read and write data [Satya89].

Another benefit of AFS is its ability to provide clients with a globally unique means of naming files. The naming scheme in AFS begins with a directory named "afs". Each subdirectory of this directory corresponds to the name of an Andrew File System Kerberos Cell. The result is that file system users access most of their files in their own cell, but can access files in other cells by traversing into the corresponding directory for that cell. The user will then authenticate to the foreign cell and continue navigating the foreign file system.

A descendant of the Andrew File System is the Coda File System. This file system maintains the same semantics as the original Andrew system; it was targeted towards computers that can become disconnected from the network. This set of clients is predominantly portable computers that can be removed from the network for long periods of time, such as when a user is traveling. This file system has been built to allow the disconnected client to update the server and their local copy of the file system when they reconnect to the network [Kistler92].

There are only a few versions of Andrew that have been created over its history. Besides the versions of the Andrew File System and Coda File System built at Carnegie Mellon University, the Transarc Corporation has built a commercial implementation called AFS, and there also exists a free client implementation called Arla [Westerlund98].

2.2.4 Sprite Network File System

The Sprite Network File System was built at University of California, Berkeley in the late 1980s [Nelson87]. In terms of its semantics, this file system was important because it aimed to follow exact Unix semantics. The designers' goal attempted to provide complete location transparency, meaning that an application should not be able to determine from the behavior of the file system whether it was running on a local Unix file system or on Sprite over a network. To achieve this goal, clients on separate machines needed to be able to read and write to a file concurrently and be able to see the changes done by the other clients. This is the behavior of the local Unix file system, but no other network file system had been able to provide these exact semantics. To provide Unix semantics for this kind of read-write sharing to their clients, Sprite employed a unique caching policy. It permitted clients to cache file data as long as no other client was writing to the file. In the event that another client opened the file for writing, the Sprite server would notify all other clients that had the file open. These clients would flush any pending changes that they had to disk and begin to utilize the file in a *write through* manner, that is writes would be sent to the local cache and immediately to network storage. In short, the caching policies that the Sprite client used were dependent on if the file was being used for write sharing.

Because, the Sprite system allowed clients to cache data, there was a period of vulnerability when newly written data in a file was not also written to stable storage. However, unlike the Andrew File System where these changes were sent to the server upon file close, in Sprite unwritten changes were flushed to disk every 30 seconds. This was a choice of the designers to keep the amount of unwritten data bounded by a time period. This semantic choice also had a benefit in that it didn't write to permanent storage many short-lived temporary files. Much of the research that the Sprite designers examined pointed out that files are opened for a very short time and many are deleted shortly after being created. Because of these two factors, Sprite designers felt that they could eliminate the extra work done to create and delete many files by simply waiting a period of time after the file has been closed.

An important fact about the Sprite Server is that it needs to keep track of all of the open files of all the clients in the system. This means that as the number of clients and open files increases the amount of load on the server increases as well. In addition, it means that the server and client need to be able to handle the possibility that the other party may crash. It is particularly important that in the event of a server crash that the system be able to recover successfully. In Sprite, the clients help a server to recover by informing the server when it restarts of all the files that the client has open. Although this is a useful means of recovering the filesystem state, it could be considered a security risk to trust the file system clients with this server state information.

2.2.5 File system abstractions

In addition to the network file systems described earlier, there are two important file system-like abstractions that need to be addressed for their relevance to network file systems. These are the File Transfer Protocol and the Hypertext Transfer Protocol.

The File Transfer Protocol was developed to enable machines connected by a TCP/IP network to transfer files [Postel85]. An FTP server presents its clients with an image of the file system hierarchy through which the client can navigate to find files. The image that the server presents resembles that of a generic Unix file system. Since the FTP server is usually an application executing in user space rather than inside the server's kernel, the FTP server can only forward to its clients contracts that it has been issued by the underlying operating system. In addition, the FTP server cannot make any guarantees about the network that exists between the client and the server. This network issue arises because, unlike other network file systems that are designed for local area networks, FTP was designed to enable file transferring over wide area networks. For this reason, FTP clients tend to be very resistant to and expectant of failures. Because of the near anticipation of failures during FTP transfers, FTP servers are not generally used as repositories for executable files but rather for archival purposes. Client interaction with FTP servers tends to be for whole file transfer, rather than to use the file while being connected to the FTP server. FTP transfers data over a connection stream, therefore the FTP clients are not able to access pieces of target files at the block level. Despite these issues, FTP servers are presented as file systems by many applications.

Similar to FTP, the HyperText Transfer Protocol is another Internet protocol designed to enable sharing over wide area networks [Berners-Lee96, Fielding97]. Because it is used primarily to share documents to clients, and because each of these documents could contain references or links to other documents, it was very desirable to build a global naming system that could be used by HTTP clients to reference other files and documents. In this way, two separate clients on different network could use the same HTTP address to resolve the same file. In essence, the result is that, like the Andrew File System, clients are presented with globally unique read-only file system [Satya96]. Clients do have a means of writing by posting data in forms, however, this is very different from traditional file system writes. Another difference from traditional file systems is that HTTP servers do not normally present clients with a full hierarchical directory listing information. HTTP servers usually present clients with a single default document from a directory and may present user-readable directory information if no default document is specified. This means that unlike normal file systems, a client cannot normally browse the directory structure of an HTTP server. Despite these differences from a traditional network file system, there is enough similarity to allow the construction of a basic file system using HTTP primitives [Kiselyov99].

2.3 Multi-protocol file systems

As an organization increases in size and age, it is frequently forced to address the issue that many users use a very diverse set of client computers. The diversity of users' computer platforms can be attributed to many reasons; for example, the personal tastes of each of the many users, the specific requirements of a user's tasks, the level of training of each user, and the aging of the computer equipment and software platform. From the perspective of a multi-protocol file system designer, the users can be grouped into client sets based on the semantics of the computer platform they are using.

Since information sharing is a vital necessity for many organizations, each client set typically demands priority access to the shared storage resources. Furthermore, as more and more users interact, eventually it is desirable to centralize the storage of shared information so clients from each client set can access the shared resources. Unfortunately, in many cases there is no common network file system that each client can use to access the shared data repository. An administrator could select a network file system that most clients have access to, however, this will usually create a situation where some clients have second-class access to the data and others do not have any access at all. The users with second-class clients program may not be productive because of several issues. They

may require administrative assistance in setting up access to the non-native network file system, they may require training on the new file system, and may not be able to take full advantage of the shared data because of limitations of their client programs.

A desirable solution is to allow each client set to access the shared data on the centralized storage server through their own preferred network file system client. A naïve approach to allow this level of sharing is to enable a file server to display the same file system structure using several file-serving interfaces simultaneously. Although this may seem to be a simple solution to this problem it leaves some major issues unresolved. Primarily, since each of the file servers was not developed with each of the other servers and its set of clients in mind, many undesirable side effects result from allowing the servers simultaneous control over the same file system structures. These side effects and the reasons why they arise can be traced to the semantic differences in the various file systems. These semantic differences are also called semantic mismatches between file systems.

In short, constructing a multi-protocol file system server is the proper means of providing access to a repository of data simultaneously to several client sets. Optimally, this multi-protocol server would be able to control the accesses of the various client sets in such a way that each client set would not be affected by or able to detect the presence of incompatible clients. However, the task of the multi-protocol server is complicated because it must be able to appear to client sets as if it is their own native file server, therefore it may not be acceptable if it cannot provide all of the functionality of the native server.

2.3.1 Multi-protocol file system products

Because of the desirability and demand by users for access to data through multiple file system interfaces, many storage vendors have products that provide some form of multi-protocol storage. The most common multi-protocol solutions are the NetApp Filer products from Network Appliance. Other products include NAS products from Auspex Systems Inc., Network Storage Solutions' NASengine, Quantum Corporation's Snap! Appliances (formerly owned by Meridian Data Inc.), and Sun Microsystems' Cobalt RaQ (formerly owned by Cobalt Networks). Each of these products provides a solution that is intended to give clients access to a common set of files through multiple file system interfaces, usually through an NFS and an CIFS interface. In general, providing access through these two file system protocols will satisfy the majority of all of network file system users, since it gives an interface to both the Unix and Windows sets of clients.

Although it is fairly straightforward to construct a dedicated file server that hosts both a CIFS and NFS interface, namely through Unix machine running a Samba server and an NFS server, the solution becomes complex when vendors aim to provide products that solve some of the multi-protocol issues that arise. Of the multi-protocol storage vendors listed above, Network Appliance has gone out of its way to contribute to the storage research community by submitting some of their white papers on multi-protocol semantics to various special interest groups. Because of this insight provided by Network Appliance into their own products and design, this paper will, where possible, contain descriptions of the solution chosen by Network Appliance's designers as an example of a possible solution. In other cases, descriptions of the behavior of some multi-protocol products will be used to gain insight into their solutions.

2.4 Semantic mismatches

A multi-protocol file serving solution must be able to resolve the semantic mismatches that occur between the various file systems of the client sets it is serving. Simply defined, a semantic mismatch is the situation that can occur when a multi-protocol server cannot provide a client with a semantic that is expected by a native file server. These situations occur because file systems have different sets of valid arguments, different interpretations of the meanings of file system commands and arguments, additional sets of file system commands, and different sets of possible errors that can occur during each command. In short, artifacts that one client set can create and use in a file system may be completely inaccessible to clients from another client set. Furthermore, the goal of many file system commands is to change the state of the file system structure in order to affect the result of future operations. For example, changing the access permissions on a file to prevent access by undesirable clients. If another client set does not understand the new state of the file system structure, it may not be able to use the structure in the intended manner.

In essence, the problem of providing client sets with the semantic sets that they can use to accomplish work requires that a multi-protocol file system designer reexamine the notion and implications of the contracts for service that it is issuing to the clients. After extensive examination of many file system semantics, it may become apparent to the

designer that some semantics in various file systems cannot be resolved. This means that if a server intends to provide a semantic for this functionality to one client set, it would be breaking the contract it has already issued to another client set. There would be no way for the server to simultaneously issue contracts to clients from different client sets. This seems to imply that there is no way for a multi-protocol server to operate, but in practice this is not the case. It turns out that a server can sometimes alter the details of the contracts it is issuing to clients without affecting the intended results of the client. This is possible because in many cases applications are not built to rely on the details and nuances of their file system activity, but instead are built to complete a logical task. The designer of the application may not be concerned about the particular approach used so long as the end result is identical. In short, by leveraging knowledge about the clients' file system activity, the server may be able to break or alter the contracts that it has issued to clients without affecting the clients operation. From the perspective of the clients, as long as its operations achieve the desired effects and do not fail unexpectedly, they may not be able to detect that the contract that they were issued has been broken.

Building a multi-protocol file server demands that the designer resolve many semantic mismatches so that concurrent users of different file systems do not interfere with or corrupt the data of other users. For example, consider a multi-protocol file server that provided streams multimedia data to clients. If this server provided a client with access to a multimedia file on a CIFS interface, many times the client's application will request a contract denying other clients the ability to write the file. Now, imagine that a concurrent NFS client decides to overwrite this multimedia file; there is no way that the multi-protocol server can honor the exact semantics of both clients. The CIFS client requested a contract that locked the file as read-only while in-use, but the NFS client has no notion that the file can be locked. In this scenario, the file system designer has many possible options that it can use, but none will exactly match both of the file system's intended effect. The designer can choose to force one of the clients to succeed and the other fail, but from the perspective of each client their operation should succeed. The designer could also choose to delay one of the requests until it is able to service the request without breaking a semantic. This would get around the problem of having to choose which file system's semantic should be broken. However, delaying one of client's requests could result in the file system appearing to have halted or might give the user the sense that the file system is having performance problems. The file system designer may determine that it is acceptable to delay the operation for a bounded period of time, if the request can not be fulfilled after the time has passed, it could then fail the request. Any choice to allow or deny a request in this situation would seem to be arbitrarily favoring one interface over another. In this scenario, if the designer chooses to permit the NFS write operation there would be repercussions for the CIFS client. First, the file lock guarantee will have been compromised. This is the logical foundation for many of the assumptions that the CIFS client makes. Next, it would allow the NFS client to possibly overwrite and even corrupt the data being read and processed by the CIFS client. This could result in unexpected behavior by the CIFS client who was not expecting and may not be able to recover from a corrupted data stream. Another option that the multi-protocol designer may choose is to permit the NFS client to write to an alternate copy of the target file. In this way the CIFS client would be protected from writes from other file system interfaces. However, the result may be two versions of the same file that either the server or the client would be required to resolve.

In short, semantic mismatches put designers in a position where they must in many cases choose which of the supported file system interfaces will receive a better quality of service, that is a semantic closer to their expected semantic. Before the designer selects what policy should be used to handle a specific semantic mismatch it would be desirable to understand the effects that the clients were intending when performing the operation. Unfortunately, that is not easy to determine from the arriving file system events.

2.5 Categories of semantic mismatches

To fully understand semantic mismatches, it is necessary to categorize them in a way that allows the file system designer to analyze them. In this way, the designer may be able to determine a policy that could resolve each category of mismatches. In order to categorize the mismatches, the goals of a traditional file system must be reexamined in the context of multi-protocol filing to determine whether it is feasible to provide all of the desired and overloaded functionality of file systems.

The primary goal of a network file system must be to provide its clients with a means of permanent or durable storage. This permanent storage must allow clients to reference and access the stored data through the use of a file name. In addition to being able to reference the file by name, it is necessary that clients be able to share information

about the location of particular pieces of data within the file. To enable these goals, the designer of a multi-protocol file system must resolve the **definition of files** that each of the file system client sets is using.

Along with providing a means of accessing data files through file names, it useful for clients to be able to organize collections of their files so that they can easily found and retrieved. It is also useful so that policies can be applied to groups of related files at a time. This means that a designer of a multi-protocol file system must be able to resolve each of the client sets' **definition of a view (directory)**. By resolving this definition, clients can agree on how to navigate or browse through a directory structure to find the intended file.

As in traditional file systems, once a contract for permanent storage has been granted to clients, it is then desirable for these clients to share these contracts with other clients so that both can provide more functionality to a client. However, deciding the degree of sharing that is allowed in a multi-protocol file system is a serious decision that must be made by a designer. Some network file systems support client sharing so rigorously that it can be a means of inter-process communication [Devarakonda96, Mann94, Nelson87]. Other file systems recognize that file sharing for inter-process communication is quite rare and do not allow this level of sharing [Gronvall99, Satya92]. The multi-protocol file system designer must decide on a **definition of sharing** that is acceptable to all of its client sets. To decide the designer's sharing policy, the designer must evaluate the trade-offs of the desire for network transparency and the desire for file system performance. The trade-off is that the more tightly integrated clients can share, the more state must be kept about the clients by the servers and consequently the less scalable the entire system.

Next, as the number of users increases so does the need to provide accurate, robust, and reliable security. The designer of a multi-protocol file system should certainly recognize the marketability of a secure file system. Increasingly, security has become a requirement of file systems, especially as more systems become connected to the Internet. Therefore, the designer of a multi-protocol file system must decide on a rigorous **definition of security** to determine which users are allowed to access and reference a file.

Another definition that a designer must decide on is the **definition of properties**, specifically the kinds and meanings of properties of files and directories that are provided to the client sets. These properties are statistics and high-level data that enable a file system user and administrator to determine useful information about the file system structure.

Finally, a multi-protocol file system designer must determine the **definition of an error** that the system will use to represent unusual conditions to its clients. Because of the nature of these complex systems, there can be errors in the server, clients, and with the underlying network. In addition to these errors, it may be necessary for a server to respond with an error condition to a client in order to influence the actions of the client. By responding with an error condition, the server may be able to influence the client to retry a request later or to change a parameter of a command. In this manner, rather than issue a contract to a client that a server cannot honor, the server can respond with an unusual error condition and try to influence the client to request a more acceptable contract.

2.6 Resolving mismatches

When a multi-protocol designer is confronted with a semantic mismatch there may be many possible methods of resolving the differences between the two file systems without affecting the clients. The designer may choose one particular semantic or accepting different semantics. For example, a designer may be faced with the issue of resolving the differences between different file property sets of several file systems. Rather than attempt to support the union of both property sets, the designer may decide that a particular subset of all of the properties is most appropriate and may choose to represent the values for all of the other properties with default values. Another designer may be forced to accept the different semantics in an issue such as the communication method of the file systems. For example, if the designer were building a system that supported CIFS and HTTP, resolving the differences between the communication protocols would seem unreasonable. It would not be possible to get all of the HTTP clients to change to use the CIFS protocol. However, the designer could build the multi-protocol file system so that it is able to understand communication from both protocols. In this way the multi-protocol file system supports both client sets fully.

2.7 Ignoring mismatches

Although it is desirable for a multi-protocol server to be able to choose definitions for each category of semantic mismatch and develop policies to handle mismatches that enable it to serve all of its intended client sets, many times this extra work may not be necessary. Some semantic mismatches may be acceptable to clients and servers. For instance, two client sets may disagree on the effects of changing a file's permissions while that file is in use. One client set may have a semantic where the permission change takes effect immediately and affects all future accesses to the file, the other client set may have a semantic whereby the permission does not affect users that already have the file opened. Rather than attempt to resolve these mismatched semantics, it is possible for a multi-protocol server implementation to ignore this mismatch entirely. In this situation, the designer may feel that choosing the semantic of one file system would be a hindrance to the other. The designer could ignore this mismatch if they believed that the chance that this mismatch would be detected in normal usage is significantly low. Furthermore, the designer may feel that the benefit gained by detecting the mismatch is very low, and would be willing to accept the risk rather than provide a solution. An extreme result of this policy might be that the file system designer would be willing to accept the risk that one of the clients may halt or crash provided that the chances of this severe event were extremely low.

The designer may consider ignoring a semantic mismatch an acceptable policy if the probability of the mismatch is low or if the cost of a mismatch is deemed acceptable. The designer may be able to leverage knowledge about the usage patterns of the users of the file system and knowledge about the intended deployment of the file system. This knowledge can be gained through examining data from actual file system usage or through a better understanding of the access patterns of users or the applications used in the target environment. If the designer knows information about the behavior of the file system users or their applications they may be able to determine that the probability of a semantic mismatch is reduced. In essence the file system designer is basing their semantic decision on a predicted model of the *mean time between semantic failures* for the targeted client set. This measure reflects the estimated time between semantic mismatches events that the user or application has the ability to detect. The file system designer may recognize that the actual *mean time between semantic mismatch* may be a much shorter period of time than the mean time between semantic failure, but if the mismatch does not result in a noticeable failure it may be acceptable to the file system designer. Based on an analysis of the file system traces, the designer would be able to determine the frequency of a semantic mismatch and from this data better choose a policy to deal with common and uncommon mismatches.

2.8 Fault severity in semantic mismatches

When the file system designer decides to ignore a semantic mismatch because the risk seems acceptable, it is important to realize that in the event that a mismatch occurs this will appear as a fault in the system. There are several classes of faults of which the file system designer needs to be aware when making decisions about file system semantics. Faults can be the result of an operation performing the incorrect task, an operation's unintentional side effects, an operation failing to achieve its objective, or an intentional failure designed to evoke an action from the client or application. The designer must consider that the fault is not necessarily a failure in the system, but may manifest itself as a failure in the system at some period of time after the fault event. The designer should try to recognize the impact that a failure will have on the end user and the applications that the user will run on the file system. The failure from the mismatch may cause a range of effects on the machine, the application, and the file. In addition, it may cause some activity that affects the user's experience with the system. Understanding the possible failure scenarios that can occur because of a semantic mismatch is an essential ingredient for the file system designer to use when determining the best strategy to approach a semantic mismatch.

First, a failure resulting from a semantic mismatch can affect the operating system's file system client that is communicating with the multi-protocol server. If the file system client does not expect a certain result from the server, or is expecting activity from the server that does not occur, the file system client may perform unexpectedly. The result might be that the file system client would crash or halt, possibly shutting down the machine in the process. In another case, the file system client may interpret the meaning of the server's response incorrectly, possibly resulting in the client keeping the wrong values of data.

Next, a failure could affect the application that is using the multi-protocol file system. In the event that the application performs an action that results in a semantic mismatch, the result may be that the application crashes or halts. An example of this behavior is the CIFS clients changing ACL security permission on older Samba filesystems, since the Samba system did not implement a means of handling security, the result would sometimes be

that the ACL manipulating application would exit unexpectedly. Another result may be that the application does not detect the semantic mismatch. In this case, the application may believe that it is performing correctly and will continue its operations. An example of this behavior is the Windows Find Files tool that searches through file system cycles in a Samba directory tree indefinitely (or at least until it runs out of memory).

A fault from a semantic mismatch can also affect the stored data in the file system. For example, the mismatch may affect some metadata of the file such as the security information or file attributes. In other cases, the file may be corrupted or even destroyed. This can occur for example when the mismatch affects the sharing semantics of the file. This is an extremely important set of faults to recognize because it affects the durability property of the file system.

All of the faults affecting the file system client, application, and stored data can eventually force interaction from the user. In some instances of mismatches users may be affected by a message informing them that an error has occurred. An example of this is a dialog box informing the user that a file is unavailable because of the sharing state of the file. Depending on the user and the environment, this may or may not be considered acceptable behavior. Other behaviors such as the application or file system crashing or halting can also affect the user in that they may need to restart the application or even the machine. Finally, a fault corrupting or destroying the stored data can force the user to perform some action to replace or recreate the missing data values.

In conclusion, a file system semantic mismatch may cause instances where the application or file system may halt or even crash, some failures can also result in lost or corrupted data. While these are undesirable outcomes, they may in fact be more acceptable in some environments than having the system continue execution with the incorrect values. The benefit of the system halting or crashing is that the user and the system immediately notice the semantic mismatch and associated fault. In short, for usage environments where only the correct result is acceptable, the fail-fast properties of the system crash or halt may be a better outcome than continuing incorrectly. In fact, the worst scenario for the user of the system to deal with may be a fault that has a long period of latency before revealing itself. In this situation, the system may begin to execute erratically long after the actual fault has occurred [Brown00]. By not failing fast on a semantic mismatch, the user and application would be presented with a system delusion, that is the file system would appear to be executing as expected but in fact is not performing the intended side effects [Gray96]. The decision-making on the failure policy of the multi-protocol system should come after examining the requirements of the system users for the environment. Once the designers determine from the users what level of severity is associated with each kind of failure, they can then make better decisions on regarding resolutions of the semantic differences.

2.9 File system traces

In order to study the behavior of clients and their applications, this project analyzed file system traces from several research projects by building a file system trace parser. The raw data from the file system trace sessions was parsed and converted by a trace-parsing client into a single, uniform representation and then sent to a trace-parsing server. The trace-parsing server then analyzed these common event representations to determine whether a potential semantic mismatch event had occurred. From the information collected, frequency data regarding semantic mismatches could be examined.

The following sections describe briefly the sets of file system traces that were used for semantic analysis in this project.

2.9.1 DFS traces

A large amount of the traces used in this project came from the Coda Project Trace project at Carnegie Mellon University, which was part of the research collected for the development of the Coda File System [Mummert96a, Mummert96b]. These traces were collected over a 26-month period on 33 workstation machines using the DFSTrace tool. This set of traces was extremely useful from the perspective of a semantic analysis because it contained traces of clients accessing files through the local Unix file system as well as through NFS, AFS, and Coda. This allowed the behavior of clients accessing each file system to be analyzed for locations of possible semantic mismatches.

This set of traces was extremely useful for observing the behavior of file system clients; however, it was not useful for discovering how a file system server perceived the file system accesses. In these traces, the machines were clients of the AFS and Coda file systems. It would be possible to combine the entire set of file system events from

each of the separate traces and extract a combined trace that would approximate a lower bound for the actual concurrency perceived by file server. However, because of the enormous amount of data, approximately 150 Gigabytes, and timing issues that could lead to inaccurate results, this process was not undertaken.

2.9.2 Appleton traces

The second set of traces came from Randy Appleton, the leader of a project conducted at the University of Kentucky [Appleton]. This set of traces was collected on two Sun NFS servers over a two-week period. This was an extremely useful collection of traces because it included the details of each read and write operation, information that was not provided in the DFS trace set. This allowed for the analysis of concurrent file accesses to be analyzed. These sets of traces also included the start and finish times for read and write operations and allowed for better understanding of the behavior of a file server while serving a file concurrently.

2.9.3 Seer Project traces

The final set of traces used in this project came from the SEER Project, developed at UCLA [Kuenning, Kuenning94]. These traces tracked the usage patterns of nine users and the file system activity on their Linux laptops. Users operated these laptops while connected and disconnected to a network. These traces were collected over periods ranging from one to six months. These traces were useful because they were able to double-check results from the other traces and added insight into the behavior of mobile computers; a perspective that was not available in other the file system traces [Lee99].

2.10 File system trace analysis tool

A file system trace parser was built as part of this project to analyze the set of file system traces that was available. This analysis tool consisted of two components: a file system parsing client, and a semantic analyzer server. The file system trace-parsing client is a stand-alone Java application that converts each record in a file system trace into Java objects that are serialized and sent over a network. These objects were sent over the network to the semantic analyzer server that examined the records as if they were events occurring to a file system and searched for instances where a semantic mismatch might arise. To aid this process, the semantic analyzer server maintained a table of currently running processes that it analyzed as well as an open file table for each of these running processes. In addition, this server kept track of changes to the open file table and the currently running process table as file system events occurred. In this way, the server could determine if any interactions between concurrent processes would have led to a situation in which a semantic mismatch may have occurred on a multi-protocol file system.

There were several limitations on the analysis of file system traces in this project. First, since the analysis tool searched primarily for interactions between currently running processes, when each of the file system traces began there was no information recording the current state of the system. Therefore, some semantic mismatch interactions may not have been recorded because part of the interaction occurred before the trace began. Next, many of the traces came from similar file system sources. The DFS traces came from the most diverse set of clients who used Local Unix, NFS, AFS, and Coda file systems. Recently there has been research conducted on Windows file system traces that focused on usage patterns [Vogels99]. However, none of the file system traces analyzed in this report came from a Windows or CIFS source, because at the time none were available for study.

3 Architecting a multi-protocol file system

The first thing that a designer needs to consider when building a multi-protocol file system is how to organize the architecture of the system. A well-designed architecture can make it easier for the designer to resolve semantic mismatches. This software architecture issue arises when building a network file system because of the heterogeneous nature of clients and servers and the large variety in the kinds of network file systems. Essentially, both the clients and the server speak different languages and it is necessary to try to resolve this in order for them to interact. Mary Shaw outlines nine techniques that are used in an analogous problem faced by software engineers in dealing with mismatched architectural components [Shaw95]. This same line of reasoning can be applied to semantics of distributed file systems.

The nine ways to resolve architectural mismatches are as follows:

1. Change Server's semantics to Client's semantics.
2. Publish an abstraction of Client's semantics.
3. Transform from Server's semantics to Client's semantics on the fly.
4. Negotiate to find a common semantics for Server and Client.
5. Make Client multilingual.
6. Provide Client with import/export converters.
7. Introduce an intermediate semantics.
8. Attach an adapter or wrapper to Server.
9. Maintain parallel consistent versions of Server and Client.

Figure 1 below depicts the nine techniques that Shaw describes can be used to resolve an architectural mismatch and the components involved in the resolution.

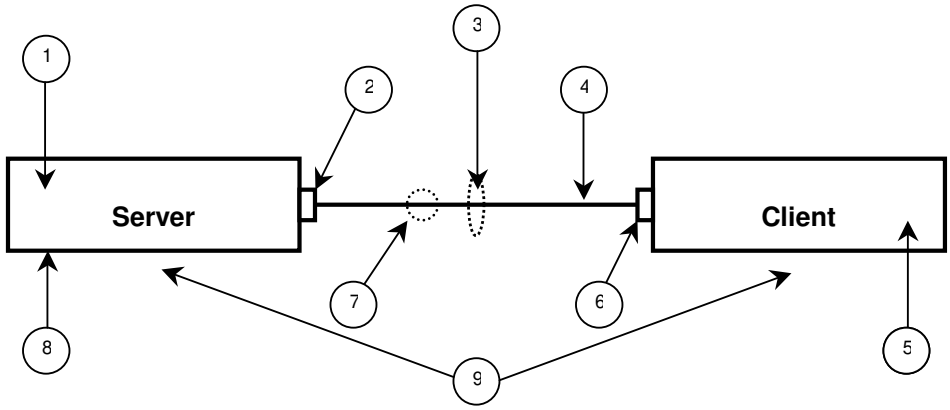

Figure 1: Resolving Architectural Mismatches

Of the list of techniques described by Shaw, there are only a few that are good candidates for a multi-protocol file system. There are several techniques that apply directly to the client. These are probably not desirable because in a multi-protocol system, the large number of clients means that each client would need to be modified. Furthermore, if modifying the client were a legitimate solution, then it would be much simpler to simply modify the client with a completely new client. Therefore, the solutions such as finding common semantics (4), making the client multilingual (5), and providing the client with an import/export converter (6) are probably not viable solutions in this problem space. In addition, the solution of maintaining parallel consistent versions (9) is one that we want to avoid in file systems.

Two of the proposed solutions in the set of architectural solutions describe methods for building an intermediate proxy between the server and the client. A proxy that performed requests on behalf of the client would be transforming the server's semantics to the client's semantics on the fly (3). Furthermore, this proxy machine could

be considered an intermediate semantic (7) that is between the client and server. Overall this technique is a good solution for legacy systems that would not notice the performance overhead incurred by adding the intermediate step. This intermediate could either be a separate machine on the network or another process running on the same machine as the file server. This proxy would either simply translate from one semantic to another or translate multiple client formats into a common format.

The remaining set of solutions describe useful techniques for building a single multi-protocol server that would be able to communicate directly with clients from multiple file systems. The notion of changing a server's semantic to match the expected semantic of a client (1) is one method that could be used to resolve a semantic mismatch. Providing an abstraction of the client's semantics (2) and attaching an adapter or wrapper to the server (8) is how the multi-protocol server can appear to multiple clients as if it is a native server for each particular file system.

4 Analysis of semantic differences

This section examines each category of semantic mismatch in detail to determine exactly some of the issues that must be resolved as well as suggested techniques to resolve some of them. Where applicable, information collected from the file system trace parsing and semantic analysis tool will be used to get a better insight into the magnitude of the mismatch problem.

The table below enumerates all of the file system events that were captured and analyzed by the file system semantic analysis tool. This table shows a number and percentage for each file system event. Since each of the file system traces collected information about events, some traces did not capture all of the file system activity that occurred. Therefore, it was useful to have more than one set of traces to get a better picture of the file system activity.

	Appleton		DFS		Seer		Total
	Number of Ops	% of Total Ops	Number of Ops	% of Total Ops	Number of Ops	% of Total Ops	Number of Ops
ACCESS	0	0.000%	12,302	0.101%	2,010,927	2.817%	2,023,229
CHDIR	0	0.000%	53,968	0.443%	761,011	1.066%	814,979
CHMOD	0	0.000%	2,543	0.021%	0	0.000%	2,543
CHOWN	0	0.000%	2,376	0.019%	0	0.000%	2,376
CLOSE	920,254	27.714%	1,091,801	8.954%	11,690,830	16.376%	13,702,885
CREAT	0	0.000%	6,424	0.053%	30,582	0.043%	37,006
DUP	308,251	9.283%	0	0.000%	2,169,285	3.039%	2,477,536
DUP2	37,127	1.118%	0	0.000%	0	0.000%	37,127
EXECVE	46,540	1.402%	74,280	0.609%	2,929,520	4.104%	3,050,340
EXIT	41,790	1.259%	29,465	0.242%	2,470,922	3.461%	2,542,177
FORK	0	0.000%	29,441	0.241%	2,382,617	3.337%	2,412,058
GETSYMLINK	0	0.000%	461,869	3.788%	0	0.000%	461,869
LINK	0	0.000%	377	0.003%	107,330	0.150%	107,707
MKDIR	0	0.000%	78	0.001%	54,845	0.077%	54,923
MMAP	177,261	5.338%	0	0.000%	0	0.000%	177,261
MOUNT	0	0.000%	11	0.000%	0	0.000%	11
OPEN	412,123	12.411%	907,713	7.444%	21,509,553	30.130%	22,829,389
LOOKUP	0	0.000%	5,993,222	49.152%	0	0.000%	5,993,222
LSTAT	0	0.000%	310,787	2.549%	0	0.000%	310,787
READ	549,620	16.552%	* see below		* see below		549,620
READLINK	0	0.000%	2,094	0.017%	0	0.000%	2,094
RENAME	0	0.000%	6,114	0.050%	175,282	0.246%	181,396
RMDIR	0	0.000%	1,496	0.012%	57,015	0.080%	58,511
ROOT	0	0.000%	610,443	5.006%	0	0.000%	610,443
SEEK	354,846	10.686%	1,945,434	15.955%	0	0.000%	2,300,280
SETREUID	0	0.000%	7,495	0.061%	0	0.000%	7,495
STAT	0	0.000%	599,931	4.920%	23,947,132	33.544%	24,547,063
SETTIMEOFDAY	0	0.000%	82	0.001%	0	0.000%	82
SYMLINK	0	0.000%	90	0.001%	13,782	0.019%	13,872
TRUNCATE	0	0.000%	1,040	0.009%	0	0.000%	1,040
UNLINK	19,252	0.580%	6,551	0.054%	999,522	1.400%	1,025,325
UNMAP	520	0.016%	0	0.000%	0	0.000%	520
UNMOUNT	0	0.000%	3	0.000%	0	0.000%	3
UTIMES	0	0.000%	35,710	0.293%	79,312	0.111%	115,022
WRITE	452,948	13.641%	* see below		* see below		452,948
Total	3,320,532		12,193,140		71,389,467		86,903,139

Table 1: File System Event Count

The table above shows counts of the total event activity that was used in the analysis aspect of this paper. It is important to note that only the Appleton traces tracked the details of each read and write event. The DFS traces kept a total number of reads and writes completed during the lifetime of a file descriptor and reported that information at the time of the close operation to avoid increasing the size of the traces dramatically [Mummert96b]. For purposes of analysis in this paper, this information was not used because it was not able to point out when the read and write activity overlapped. Despite this lack of data in the DFS and Seer traces, there was enough evidence from the Appleton traces to draw conclusions about read and write sharing.

Table 2 below presents an enumeration of the total observed concurrent events. In short, it shows that every file system operation suffers from the fact that it may at one point be enacted while another file system client is already using the targeted file. In fact, as shown in the table below there are many instances where the targeted file is operated upon several times while it is in use. In short, this data should motivate the file system designer to resolve the semantic mismatches that arise when these situations occur.

	Total Events	Total Multiple Calls	Avg. Calls per Event
Chmod While Open	2,864	208	0.073
Chown While Open	525	30	0.057
Delete While Open	89,290	133,880	1.499
Link While Open	3,601	9,227	2.562
Rename While Open	8,138	6,920	0.850
RmDir While Open	7,510	19,227	2.560
Stat While Open	236,491	4,055,859	17.150
UTimes While Open	43,623	21,478	0.492

Table 2: Multiple Semantic Events

In order to understand the information provided by the file system semantic analysis tool, it is necessary to observe and count all of the instances of possible errors in the results of the tool. The table below enumerates several classes of errors that are a cause for erroneous results. Many of these errors arise because of the lack of state given at the beginning of the trace data. For example, the Appleton traces contained instances of successful reads and writes on file descriptors that had no entry in the analysis tool's open file table. In short, when the trace began, there must have been several processes that already had files open but were not noticed by the trace gathering procedure. This could result in a falsely lower probability of concurrent events. This event may also occur because of a corruption in the trace data, causing events to be lost. In short, although there is a margin for error in the trace results, the result of the error would tend to be a reduction in the observed concurrent activity. This means that the actual amount of concurrency could in fact be higher than the data shows.

	Error Type	Error Events	Normal Events	Error Rate
Appleton	Read without Open	35,521	549,620	6.071%
Appleton	Write without Open	18,106	452,948	3.844%
Appleton	Exit without Running	3,129	38,661	7.487%
DFS	Exit without Running	160	29,305	0.543%
Seer	Exit without Running	79,932	2,390,990	3.235%
Appleton	Exec without Running	3,257	40,283	7.480%
DFS	Exec without Running	45	74,235	0.061%
Seer	Exec without Running	17,216	2,912,304	0.588%
Appleton	Close without Open	60,180	960,074	5.899%
DFS	Close without Open	76,454	1,015,347	7.003%
Seer	Close without Open	663,063	11,027,767	5.672%

Table 3: Error Rates in Semantic Analysis

4.1 Definition of a File

Regarding the definition of a file, there are several semantic issues that need to be resolved. These issues are:

- handling special types of files
- reading from unwritten portions of files
- using subfiles or aggregate files for parallel file access
- and using file soft links and shortcuts.

There is little disagreement between file systems on read and write definitions for files. For normal files, when a client writes data to a particular byte address in a file, any later client will retrieve the same data by issuing reads to this same address. This is true regardless of the file system that constructed the file.

Special files require special consideration from multi-protocol file systems because they usually follow an entirely different set of rules from normal files. These special files can be named pipes, Unix domain sockets, character device files, or block device files. These files do not obey the traditional read and write rules for files. For example, in Unix an application that opens the special null device file can seek and write to any location but will receive an end of file signal if they try to subsequently read from this location. Another special file, the random device file in Unix, will give concurrent readers different values when they read multiple times from the same location in the file. Other special files give access to hardware devices connected to the machine. Special files are not limited to Unix, in early DOS and still many Windows clients, names that began with the dollar sign, such as '$clock' or '$lpt1,' referred to special files as well.

Because of the nature of special files, it is nearly impossible for a designer of a multi-protocol file system to predict the behavior of an access to a special file without considering each case individually. Furthermore, it is also not clear that accessing special files over a network file system is desirable and in fact has been the cause of several security problems in some network file systems. For example, in some early Windows CIFS servers, clients requesting a special file named "c$" or "d$" would gain access to the entire C or D disk drive respectively, even if the drive was not set to allow sharing. For this reason, it is probably wise for a file system designer to disallow most special files from being used over a network. In general, special files are useful for clients on the local file system but not for remote clients.

Another semantic issue that falls into the category of defining a file is the issue of reading from an unwritten portion of a file. In this case, the server has the option of returning to the client either a buffer full of zeros, a zero-length buffer, or an error code. By definition, any could be considered correct because if no data has been written to an address then it should not be possible to read data from that location. However, there are several motivating factors that suggest that a file system designer should return a buffer full of zeros. Consider a server that returns a zero-length buffer, and we will see why this is undesirable. First, if a client receives a zero-length buffer in response to a read request, it has no way to determine that there is more data to be read from that file after a hole in the file. This client may assume there is no remaining data and cease reading. Also, if the client requests to read a range from the file but the beginning of the range started in a file hole, it would difficult to inform the client that the actual data returned did not begin at the requested offset. Next, if we are following a model that only written data can be subsequently read, then we are introducing three possible values that can exist in each bit position of a file: a written one bit, a written zero bit, and an unwritten bit. In addition, because the underlying storage device of many file systems is based on blocks on storage medium, it may be very complex to represent the portions of the blocks that have not been written, especially if byte ranges written are not aligned on the underlying storage device's blocks. Finally, in some complex file aggregation strategies, such as in a software RAID system or in a system where subfiles are distributed over multiple storage nodes, such as the Vesta distributed file, it is very desirable to use the zero-fill strategy [Corbett96, Hartman95]. In these systems, without a zero-filled portion of a file, the client reading a subfile would not be able to determine whether the aggregated file has ended or if it should continue reading from the next subfile. In Amoeba's Bullet File System, this policy of zero filling is enforced in that the server forces files to be contiguous, not allowing unwritten portions of files to exist [Tanenbaum90]. In addition, file systems on write-once devices such as tapes and CD-ROMs do not allow the writer to seek forward in a file while writing. In these file systems, the file writer must fill in the empty portions of the file with zeros. In short, the zero-fill strategy is a safe design strategy that does not break the semantics of any file systems. Although the zero-fill strategy is recommended here, this certainly does not mean to imply that the physical disk blocks for unwritten portions of a file should be allocated for unwritten portions of files.

A final semantic issue that must be examined when defining files for a multi-protocol file system is to consider to what degree to support links to files. Modern network file systems have several different methods of supporting links to files. Sun NFS, AFS, Sprite, and other Unix-derived network file systems support hard-links, which are references to the lower level storage system's location of the file's data contents. Hard links are supported fully in NFS, but are only supported within a single directory on AFS and Coda. The reason for this is that in AFS and Coda, the file system presented to the user is stored as many volumes on many separate machines. Since a hard link is a reference to the location of the file's data in the lower level storage media, it is difficult to support hard links that point to files on other volumes. For this reason, hard links are limited to within a single volume. Because these systems allow entire directories to be moved from one volume to another to provide load balancing, the AFS/Coda hard link creation semantics are restricted even further to within a single directory. These systems also support soft-links, which are references to the name of the file. CIFS clients do not support any form of link, but the Windows platform does offer the creation of file shortcuts, which are somewhat similar to soft-links but contain extra environment information. From the point of view of the underlying storage system, Windows shortcuts are simply another regular file. In short, linking should be considered a helpful tool for users to navigate directory trees quickly. It is not crucial to file system operation, although it has in many cases become an expected portion of a file system.

	Accessing special files	**Handling soft and hard links.**
CIFS (Windows)	Access to special files was initially permitted and later blocked for security.	Neither soft or hard links exist in CIFS.
CIFS (Samba)	Special files appear as zero length regular files.	Links are presented as the actual target file. Broken soft links are hidden from the client.
Andrew	AFS does not support character and block special files.	Hard links can be created within a single directory. Soft links completely supported.
NFS	Accesses to special files can act as accesses to special files on the local or remote machine, or be denied.	Hard links and soft links are completely supprted.
Sample Commercial Multi-protocol	Creation of special files is denied.	Some disallow link creation. Others support link creation and usage to a limited degree.

Table 4: Using Hard and Soft Links in Various File Systems

From the perspective of a user and its client applications, whether the user has access to a file or access to a link to a file makes little difference. For this reason, it is quite easy for a multi-protocol file system to support soft links even to file system clients that do not understand links. The file server can simply treat all accesses to a link as accesses to the targeted file. In fact, the Samba server displays Unix file systems to CIFS clients using this technique. It treats accesses to soft links as if they are accesses to the targeted file, and hides any broken soft links, which are soft links where the targeted file does not exist. Displaying soft links to some clients as if they are the actual data files is one example of a file system designer altering the definition of a file in order to resolve a semantic mismatch. The disadvantage to this approach is that a user of the file system interface may accidentally delete the data file. If client interface does not recognize soft links, the user may not be able to determine which is the data file and which is the soft link, in fact the user would not be able to determine that one of the two files is a soft link. This is potentially an extremely dangerous means of resolving a semantic mismatch because it has the capability for the data file to be deleted by the user unintentionally.

Another method that may be useful in supporting both Unix-related and CIFS client sets is to convert between versions of soft links. That is, display a Unix soft link to all Unix-related client sets and display a Windows shortcut to all CIFS clients. This allows both systems to recognize that they are working with a link rather than the actual file, however, it does incur overhead on the server to create the shortcut file.

Finally, hard links to files are difficult to support because they are references to an internally kept location of the file rather than just the name of the file. On a large distributed network file system, hard links place an undesirable limitation on the layout of the file system's data. They limit the possible storage layouts that the server has because it can no longer freely migrate files from one storage node to another, rather it must make sure that any migration to the targeted file also migrates all hard links that point to the file. For this reason, the file systems that support hard links usually place limitations on them. AFS enforces a policy that hard links can only be created within the same directory, which results in the link and the target file being located on the same physical volume [Howard88]. This policy renders hard links almost unusable. NFS treats hard links the same as Unix, in that hard links must target files

in the same volume. A designer of a multi-protocol file system must evaluate carefully the layout implications that supporting hard links entails. It may be an acceptable policy to allow soft links in all cases, and only allow hard links in limited situations, in this way the designer is not constrained by the physical properties of the hard link.

Supporting hard links in a multi-protocol file system requires that the underlying file system is the implemented with a reference counting system to make sure that hard linked files are only physically deleted when all of the hard links are removed. More work is required by the multi-protocol file system if they want to implement hard links between multiple storage devices. The table below shows the number of concurrent link creations that occurred while a file was in-use in the file system traces.

	Appleton		DFS		Seer		Total	
Link While Open	0	0.000%	2	0.000%	3,599	0.017%	3,601	0.016%
Total Number of Opens	412,123		907,713		21,509,553		22,829,389	

Table 5: In-Use Link Creation Events

Existing multi-protocol products have taken a number of approaches. Quantum's Snap Servers do not allow their users to create soft links or hard links. Network Appliance's products allow the creation of soft links and hard links, but state in their documentation that use of hard links may not be supported by Windows clients. Neither allow access to special device files.

4.2 Definition of a View (Directory)

In terms of the definition of a view or directory, the designer of a multi-protocol file system must make several policy decisions. The issues that must be resolved are:

- determining the view into the file system to support
- handling of soft links to directories
- handling broken soft links to directories
- managing the potential for cycles in a directory tree.

In the strictest sense, directories are not necessary for achieving the file system's goal of providing durable data storage. However, they serve an extremely important task from a user's point of view. Directories allow users to organize the files in the file system. Hierarchical storage systems provide users with the ability to quickly find files because it is simpler to examine a small list of files in a directory than examine all of the files on an entire volume. Most storage systems support hierarchical directory structures to simplify the view of the system by the particular directory selected. Although a hierarchical view of the file system is the method predominantly used, other methods of viewing sets of files exist. The file system can also provide the user a view of the file system where files are viewed by attributes or properties such as in the Semantic File System [Gifford91]. It is quite difficult for a multi-protocol file system to build a mapping between an attribute-based storage system and a traditional hierarchical view. The multi-protocol file system could produce a view to its hierarchical style clients of either the most recent attribute views used in the system, or in the worst-case even a directory for each attribute, this would result in a brute force approach to resolving the semantic mismatch. Thankfully, all commercially used file systems are hierarchical so multi-protocol file system designers will most likely not run up against this issue.

Next, it is important to determine how to handle soft links in file systems that refer to directories. These can be handled in a fashion similar to the method described for files. That is, for systems without notions of soft links the linked directory can be represented to the client as the targeted directory. In addition, broken soft links to directories can be hidden from the client's view. In this way, clients that do not understand soft links do not receive errors when they access the broken link. The result of hiding broken soft links is that two clients from different file systems may see a different list of files when examining the same directory on a multi-protocol server. This may confuse the users, but at least none of the clients will receive errors that they cannot handle.

A subtle error that can trouble some clients that do not have native means of supporting soft links is that they may experience problems when accessing directory structures in which a soft link in the structure refers to a directory at a higher level in its directory hierarchy. In this situation, a cycle in the directory tree can be created. When a directory cycle exists, clients that expect only acyclic directory trees may run indefinitely when conducting a tree traversal operation, since they will never reach the leaf nodes of the tree structure.

There are several techniques that a multi-protocol file system can use to untangle this directory cycle issue. A first pass approach to prevent many cycles is to deny all clients the ability to create a soft link that would result in a directory cycle. Since it always possible to detect a cycle in the directory, the server can easily determine whether it should deny the client's request. The cycle detection would also need to occur when a rename operation is being done to move a soft link into a new location. This is a simple solution that does result in a loss of functionality for some users and applications that take advantage of directory cycles. This approach may prevent a large number of cycles from being created, but it does not prevent all soft link cycles. For example, a rename operation can move a directory containing a soft link into a position in the hierarchy where the soft link creates a cycle. Another method can be used to hide directory cycles from cycle-unaware client sets. This can be done on the server's side by detecting requests from cycle-unaware clients to traverse the same directory several times. The server can choose to permit an arbitrary number of cycles before eventually hiding the soft link from the client. This creates a false leaf node in the directory structure that cycle-unaware clients can use to avoid traversing a directory indefinitely. The benefit of denying the creation of a directory cycle is that the server can determine at link creation time whether to allow or deny the request. In the solution that hides directory cycles, the server must process each link traversal to determine if the client is in a cycle, this could add significant overhead to a server.

4.3 Definition of Sharing

One of the most complex issues that a multi-protocol file system architect must determine is the definition of sharing that will be used in the their file system. This is a source of many of the major differences between file systems. The amount of sharing permitted in a file system is generally a result of the philosophical point of view of the file system designers as to whether simultaneous clients should be viewed as competing or cooperating. When simultaneous clients are viewed as competing, the designers of the file system insert checks to ensure that the behavior of one client does not affect another client. When viewed as cooperating clients, the system designers may provide extra features to help the clients to synchronize their sharing. Since the server is the ultimate arbitrator in deciding which client's operations is permitted, the server can directly control much of the sharing semantics of the file system. This section will describe several sharing issues that need to be resolved by a multi-protocol file system designer. Data collected from by the file system trace semantic analysis tool will be used to shed more light on a particular sharing semantic.

4.3.1 In-Use File/Directory Deletion

One common issue that arises when sharing files is that one of the concurrent processes chooses to delete a file that is opened by a concurrent client. This semantic is handled differently by many file systems and is further complicated when these file system must handle this situation over a network. The multi-protocol designer has the option of either permitting, denying, or delaying the delete attempt. In either case, both the file-using client and the deleting client would like to succeed. This semantic mismatch also arises when the parent directory of a file in-use is removed. Since deleting a directory also deletes all of the files in that directory, the end result is a situation identical to the in-use file deletion semantic mismatch. An application that relies on the semantic of deleting an in-use file and still being able to access the data would be impacted because it after it created some file for temporary usage and before using the file, it may be denied or delayed when it tries to delete the file. This is commonly done by applications such as compilers that create many temporary files during their lifetime, most use this semantic so that in the event that the application crashes the files are still removed.

In the local Unix file system, a deleted file is removed from the namespace of the file system, denying any future opens by clients. Clients that have already opened the file can still continue to read and write to the file. Once the last client with the file closes the file it is then deleted from the storage media. In Sun NFS, which targeted Unix clients, this issue was handled by another means. NFS users would expect to be able to continue accessing a file after deletion as they would in their local Unix file system, but using this semantic complicates the handling of clients over a network since the NFS server stateless property means that it would never be able to determine when all clients had closed their handles on the deleted file. To handle this situation, many NFS clients rename the file and hide it from the user, and remove the file later. The client does not actually send the delete command until the user has closed the handle on the now renamed file. An NFS client on another node of the network would be able to continue accessing the file since it has a token that references the inode number of the target file. When the deleting user closes and deletes the renamed file, the network client's token is suddenly invalid. In CIFS, the semantic reflects that of DOS and Windows clients. In this system, no client is permitted to delete an in-use file, including users with administrative permissions. Deletion is not permitted because early versions of these clients were not

aware of any other simultaneous clients. The table below shows the various interpretations of this semantic for several network file systems.

	Results seen by file user	Results seen by file deleter
CIFS (Windows)	User can access the file uninterrupted.	Delete operation fails because file is in-use.
CIFS (Samba)	User can access the file uninterrupted without noticing that the file is deleted.	CIFS interface deletes fail because file is in-use, local deletes on server succeed.
NetWare	User can access the file uninterrupted.	Delete operation fails because file is in-use.
Andrew	User can access the file uninterrupted.	Delete operation is successful.
NFS	User can access the file uninterrupted when the delete is on the same host. Remote delete stops future access to a file. A remote delete with the file open will allow the user to access the file, until the deleting user closes the file.	Delete operation is successful
HTTP/FTP	User can access the file uninterrupted.	Delete operation is successful

Table 6: Handling In-Use Deletes in Various File Systems

Because of the various complications that result from deleting a file while it is in use, it is important for a multi-protocol file system designer to weigh the benefits of supporting these clients against the complexity costs that supporting this semantic entails. There are many policies that a designer can choose that range from denying all in-use file deletions to allowing for some clients and not others, to permitting all in-use file deletions. Regardless of the policy chosen, the semantics for one set of clients will not be the same as those of its native file system. Table 7 below shows the number of *delete while open* and *remove parent directory while open* events that occurred in the file system traces that were analyzed in this project. As the results show, the in-use file deletion events appear in each set of traces.

	Appleton		DFS		Seer		Total	
Delete While Open	3,860	0.937%	767	0.084%	84,663	0.394%	89,290	0.391%
RmDir While Open	0	0.000%	0	0.000%	7,510	0.035%	7,510	0.033%
Total Number of Opens		412,123		907,713		21,509,553		22,829,389

Table 7: In-Use Delete and RmDir Events

The most common reason that file system users use the delete operation is to reclaim storage space on the file server. Users may do this because they are nearing capacity on the file server or they may have an internal quota, limiting the amount of space that they can consume. In any case, the user may be confused by the in-use file deletion semantic because they may be expecting the amount of available space to change after a delete operation. If the file system does not delete the actual data of the file until all of the users have closed, then it may appear that no space was reclaimed.

Whether clients should be permitted to read deleted files is related to how a designer interprets the meaning of the delete. One interpretation may be that the delete operation is intended to destroy the file completely and therefore it should stop all accesses to a file, including accesses by currently opened clients. A more commonly held interpretation of the delete operation is that the operation is simply removing the file's name from the namespace of the file system. In this view, it is not an operation that affects the ability to read and write to a file since it does not affect the permissions of the file. Therefore, if a user's intention is to immediately cease all accesses to a file, then a permission change or other security operation should be used rather than a deletion. In this interpretation of the delete command, since the operation does not affect access permissions, it should not affect clients that already have opened the file for usage. In general, this is the viewpoint subscribed to by most network file systems. The exception is that in NFS, a file user may be denied access to a file that is deleted by another user. In this situation the NFS server attempted to provide the client with Unix delete semantics but did not realize the file was in use by another client.

If a multi-protocol file system designer accepts the premise that the delete operation is not an access control operation, then the designer must make sure that the file deletion does not affect the ability of the current client to access the file. The only decision left for the designer is whether to permit the delete operation or deny the operation. In many file systems, clients use the ability to delete a file after opening to guarantee the removal of their

temporary files. Since many applications and client sets rely on the ability to successfully delete opened file, the designer must decide whether to allow in-use file deletion in all cases or only allow it when a file is not in use by a client whose semantic prevents deletion. In either case, the designer will be breaking the semantic of one client set.

4.3.1.1 Possible solution

In order to allow access to a deleted in-use file, the multi-protocol server must maintain state about which files are in use, so that the actual file data is not deleted. Clients that have semantics that care about the server maintaining state about their open files must send their open and close requests to the server. In this case, the multi-protocol server is guaranteed that it will be informed when the file-using client has completed its usage. Therefore, the multi-protocol server can honor all of the deletion requests of all clients. It can then either allow the deleted file to appear in the namespace of the file-using client giving the appearance that it has not yet been deleted, or it can simply guarantee the file-using client will be able to continue reading and writing the file. Since the file-using client, who is preventing complete deletion, must inform the server when it is done using a file, the multi-protocol server is guaranteed that it will know when to garbage-collect this deleted file. This solution for in-use file deletion technically breaks the semantics of CIFS clients, because their semantic requires that the file not be altered in any way while in-use; however, the behavior of a CIFS client using the deleted file would not be affected. Moreover, the multi-protocol server could even hide the fact that the file was deleted from the CIFS client, in this way the client would not be able to detect that the semantic was altered.

4.3.2 In-Use File Permission Change

Another issue that can cause semantic mismatches occurs when a file's access information is changed while the file is being used. A permission change may cause a semantic mismatch when the permissions on a file are changed to revoke the permissions for a user that currently has the file open. From the perspective of a file system designer, it is not possible to determine whether the user's permission change was an intentional attempt to deny future reads and writes to the file by the in-use client, or whether it is a request to deny any more opens on the file by this client. The access information changing can be in the form of a change of ownership on the file or a change of permission. In Unix-derived systems such as NFS, AFS, Coda, and Sprite these are the *chown* and *chmod* operations respectively. In Netware and recent implementations of CIFS, this is a change to a file or directory Access Control List. The table below shows the result of this semantic in multiple network file systems.

	Operation result seen by user	**Operation result seen by perm changer**
CIFS (Windows)	User can access the file uninterrupted.	Permission change is successful.
CIFS (Samba)	User can access the file uninterrupted.	Permission change is successful.
NFS	File owner access the file uninterrupted. Other users can not read from or write to the server, but can continue accessing their locally cached data.	Permission change is successful.
Andrew	User can access the file uninterrupted.	Permission change is successful.
HTTP/FTP	User can access the file uninterrupted.	Permission change is successful.

Table 8: Handling In-Use Permission Changes in Various File Systems

The table below shows the occurrence of permissions on a file being changed while it is being used. In general, this event is much less frequent than the in-use file deletion. Furthermore, the file system trace semantic analyzer tool searched for any changes in the file's permission or ownership; therefore, many of these events may not have affected the access rights of the current user.

	Appleton		DFS		Seer		Total	
Chmod While Open	0	0.000%	2,864	0.316%	0	0.000%	2,864	0.013%
Chown While Open	0	0.000%	525	0.058%	0	0.000%	525	0.002%
Total Number of Opens		412,123		907,713		21,509,553		22,829,389

Table 9: In-Use Permission Changing Events

There are primarily two interpretations to this semantic that are used by network file systems. Most of the Windows and Unix network file systems follow the semantic that the act of opening a file means that the client is permitted future operations on the file. The file server issues a contract to the opener of the file that can be utilized as a handle to allow future accesses to the file. A different semantic interpretation is used in Sun NFS. Since the file server does

not maintain a list of the contracts it has issued to clients, an NFS server must recheck the permissions of the accessing client each time the client needs to read or write file data to the server [SunMicro89, SunMicro95]. Because the server rechecks the permissions of the client, the server will stop the accesses of a client when the permissions on the file change. There is an important exception to this policy. The NFS server always permits accesses by the file's owner despite the permissions. By using this exception the owner of the file would be granted Unix semantics, it is able to continue accessing a file despite the permission bits once the file has been successfully opened, and the server is not burdened with maintaining a list of open files, since it could result in a loss in performance and complex failure handling [Gobioff97].

For the designer of a multi-protocol file system, either semantic choice can be supported, but there will be immediate consequences on the applications running on the system. If the designer chooses to allow accessing in-use file to continue, it would need to be able to maintain a list of clients and the files that they have opened. This would be difficult to do if the multi-protocol system had an NFS interface, since the server would not receive open and close requests from the NFS clients. Therefore, the multi-protocol server would not be able to determine if an incoming NFS request should succeed, because the NFS client may be using an old file handle, or fail, because the NFS client has closed and is reopening the file.

The designer of the multi-protocol file system may wish to keep very little state about its clients or wish to use a strict security model. In this case, the server can be designed to check the permissions of each client on each operation. Choosing this semantic will break the semantic of many file systems, since it could stop many clients' ability to access files even after they have obtained a contract from a server. However, although breaking the semantics of client sets is not desirable and may result in an application crash or halt, the file system designer may decide that this broken semantic is an acceptable price for strictly following the security model. This guarantees to the permission-changing client that the new permissions will be adhered to from that moment forward. In short, in environments that depend on the adherence to a strict security model it may be acceptable to break the semantic of a client so that the security of the system remains well defined. The designer may be able to avoid the crashing of a client application by reporting that the device has been removed or some other error condition that the client would recognize. This breaking of the client's definition of an error is discussed further in Section 4.6: Definition of an Error.

4.3.2.1 Possible solution

For this semantic, one possible solution is that the multi-protocol file system presents each file system interface with the semantic that it expects. In each of the file systems listed in the table above, the operation of the permission-changing client succeeds. The difference occurs for the file using client, most file system permit accesses to continue, but NFS clients would fail since they would not maintain a handle on the server file. The multi-protocol designer can now determine whether it wants the server to deny future NFS operations since this is an outcome expected by NFS clients, or if it is acceptable to permit the access operations since this is the semantic provided on Unix, which NFS designers were attempting to mimic.

4.3.3 In-Use File Property Read and Change

Another event that could occur frequently in a multi-protocol file system is the modification or reading of a file's property set while the file is being used by another client. For example, in many instances it is useful for an application to update the time attribute while using the file. This is typically done as a method of notifying a cooperating application. It is also fairly common to read the properties of the file with the *stat* command after the file has been opened. The table below shows the number of times that these kinds of events occurred in the file system traces.

	Appleton		DFS		Seer		Total	
Stat While Open	0	0.000%	1,829	0.201%	234,662	1.091%	236,491	1.036%
UTimes While Open	0	0.000%	30,054	3.311%	13,569	0.063%	43,623	0.191%
Total Number of Opens		412,123		907,713		21,509,553		22,829,389

Table 10: In-Use Property Changing Events

In each of the file systems analyzed, it was always possible to read the properties of the file and even change the properties of the file while the file was in use. In general, this is safe because the file property set is a table of flags

that is not directly related to the file. In most file system implementations, the properties are informational and have no effect on the behavior of a client that is using the file.

However, there are applications that rely heavily on the file properties maintained by the file system. These are applications such as *make* that perform actions based on the timestamps of the file. Another example is an application that decided if it should perform a task based on the size of the file. These are both properties of the file, which are modified as side effects of writing. Therefore, if the client is caching its write operations, a concurrent file property reader will not see the correct values for the file size or the modified timestamp since they will not be updated until the cached write is sent to the disk. Unless the writing application chooses to close and sync the data to disk, this information will be unavailable to any other client.

4.3.4 In-Use File/Directory Renaming

Another semantic issue that can arise in a multi-protocol file system is that a client renames an open file or a directory containing an open file. In this situation, the designer needs to determine whether or not the rename operation can be allowed to succeed. In addition, if the rename operation is permitted the designer needs to determine whether the client that has opened the file should be allowed to proceed or if it should be forced to access the file using the new path. The table below shows the frequency of this event as detected by the file system trace analysis tool.

	Appleton		DFS		Seer		Total	
Rename While Open	0	0.000%	2,762	0.304%	5,376	0.025%	8,138	0.036%
Total Number of Opens	412,123		907,713		21,509,553		22,829,389	

Table 11: In-Use Rename Events

The rename semantic issue is very similar to the file deletion semantic; in fact, the rename operation could be viewed as a composite of several operations. Hard linking the target file followed by a deletion of the original file is sufficient to do the rename. If this interpretation of the rename operation is used, then the solution to this semantic mismatch should resemble the solution to the deletion semantic.

	Results seen by file user	**Results seen by file renamer**
CIFS (Windows)	User can access the file uninterrupted.	Rename operation fails because file is in-use.
CIFS (Samba)	User can access the file uninterrupted without noticing that the file is renamed.	CIFS interface renames fail because file is in-use, local renames on server succeed.
NetWare	User can access the file uninterrupted.	Rename operation fails because file is in-use.
Andrew	User can access the file uninterrupted.	Rename operation is successful.
NFS	User can access the file uninterrupted.	Rename operation is successful.
HTTP/FTP	User can access the file uninterrupted.	Rename operation is successful.

Table 12: Handling In-Use Renames in Various File Systems

Another way to look at this semantic is to examine it keeping in mind the various logical pieces of storage that are provided by the file system. The file system provides data storage to file system clients and the ability to store attributes and properties about each file and directory. The name of the file or directory is one of the many file attributes that are kept by the server. Some file systems such as CIFS prevent the rename operation from continuing unless the object is not in-use. Although CIFS prevents the file name from being changed, it will allow all of the other properties of the file to be changed while the file is in-use. CIFS clients are not actually affected by the file renaming, but it technically violates guarantees that are made to them at file open time. Other network file systems such as AFS, NFS, and Sprite will permit the rename and it will not affect the simultaneous user of the file, since the user is accessing the file indirectly through a file handle.

4.3.4.1 Possible solution

A multi-protocol file system designer can support the rename prevention semantic desired by these CIFS clients, since the server is informed by the clients when it has completed using the file, with a close operation. Therefore, the server has the ability to support the rename prevention semantic if desired. However, most file systems grant access to the contents of the object through the client's use of a handle, not through the use of the filename, including the CIFS system. For this reason, the server can permit the rename to proceed without affecting the operation of the file user. Although this would break the semantic as derived from the behavior of existing systems, it would follow the

behavior that these systems use for other all other object attributes and properties. The CIFS client would still be able to access the file data as it had done before the rename operation. In short, since this operation affects neither the security status of an object nor the data contents of the object, it can be allowed to proceed without fear of raising security or data integrity issues. Furthermore, much like the proposed possible solution to the in-use file deletion semantic, to protect CIFS clients further, the multi-protocol server could hide that the file was renamed from the client, in this way the client would not be able to directly detect that the semantic was altered. In this scenario, multiple clients would temporarily have different views of the namespace, however the goal of protecting a client from the assumption a file system error would be achieved.

4.3.5 Concurrent Reading and Writing

A direct consequence of allowing files and directories to be presented in a network file system is that eventually multiple users will access these files simultaneously. For executable programs or other files that are read but not written, sharing access to these files does not produce any semantic issues. However, when file system clients intend to write concurrently to the same file or when some clients read and other clients write to a file, then important issues concerning the sharing of data must be considered carefully by file system designers. This issue is especially complex when the clients are accessing files over a network, and is even further complicated when the files are being accessed concurrently on a multi-protocol file system. In this type of scenario, the designer must decide how to resolve potentially conflicting requests from multiple clients [Allison99b].

For each of the client sets that the multi-protocol file system supports, the designer must determine whether it is acceptable to break the expected semantics of that client set in order to support the intended semantics of another. Table 13 shows dramatically how often this semantic issue arises. With the data available in the Appleton file system traces it was possible to examine each read and write system call that occurred. The results gathered by the semantic analysis tool show that multiple applications frequently had the same files open and issued concurrent read and write calls to those files before closing them. The data shows that write after write sharing, where write activity by one application occurs after write activity by another application, happens very frequently, especially for temporary files, while read after read sharing primarily occurred on data files. The data in the file system traces also showed that when multiple applications were concurrently accessing a file, the file was usually subject to a large number of concurrent accesses, not just a single concurrent access. Table 14 below shows this information. It shows, for example, that when concurrent activity occurs in a Write after Read situation, the clients will perform on average over 160 concurrent IO operations before they stop sharing the file. In short, the data shows that this activity is inevitable in a multi-user network file system.

	Total Reads	Total Writes	Read After Reads	Read After Writes	Write After Reads	Write After Writes
Appleton Trace Data Source	549,620	452,948	104,812	68,688	107,854	205,699
As Percentage of Reads			19.0699%	12.4974%	19.6234%	-
As Percentage of Writes			-	15.1647%	23.8116%	45.4134%

Table 13: Concurrent Read and Write Activity

	Read After Reads	Read After Writes	Write After Reads	Write After Writes
Total number of calls during concurrency	13,275,231	1,632,833	17,348,821	933,641
Average number of calls per concurrent session	126.66	23.77	160.85	4.54

Table 14: Multiple Concurrent Read and Write Events

In some network file systems such as NFS and Sprite, the semantic of concurrent accesses were intended to mimic to various degrees the behavior of a local Unix file systems. In this sharing semantic, commonly referred to as *Unix sharing semantics*, writes that are done by a client will be observed by all later concurrent readers. In addition, in this level of semantics the write system call is atomic, meaning that all read system calls will either see none of the effects of the write or the entire write. This is a difficult property to provide in a network file system because in the case of a large write that is spread over multiple blocks, the client may be slowly writing back the data to disk over the network as another client is reading the data. The reader may actually see pieces of the modified data instead of the results of the entire write operation. It is important to notice that although both NFS and Sprite are said to

provide clients with Unix sharing semantics, they differ dramatically on the consistency of the guarantees that they each make. Sprite updates all concurrent clients with the contents of each write as it occurs. In NFS, clients are allowed to cache their read data, and periodically check with the server to validate their data; therefore, concurrent NFS readers will not see a write as it occurs rather only after a period of time has passed. To provide Unix sharing semantic to clients, the server allows files to be opened and modified simultaneously by multiple clients. The server must then present the modified portions of the file to any concurrent clients. In order to allow a client reader to observe writes that occur, the caching policy of the client must recognize that the data maintained in its cache may be invalid. From the perspective of the file system designer, the client can either query the validity of its cached data from the server or the server can push changes or notification of changes to any clients when data is modified, such as through a callback [Douglis91, Nelson93, Nelson87, Srinivasan89].

A second common policy used in concurrent sharing is *session semantics*, also commonly referred to as *private copy until close semantics*. In these semantics, of which AFS version 2 is the notable example, multiple clients access the same file by each working on privately held local copies of the file's data. When a client writes to the file, it changes only its own privately held copy. When the client has finished modifying the file, it can send the entire modified file to the file server. In this way, the server need only maintain the most recently modified version of the file. AFS version 3 changed this policy by allowing the client to optionally write back data to the server, before the file is closed, in 64 kilobyte blocks. This allowed the client to handle the performance penalty associated with the whole file transfer for very large files. Furthermore, the server does not need to be involved in sending updates to all of the clients for a particular file. In Amoeba's Bullet file system each file is immutable, meaning that all readers see one file while a writer changing a file is actually creating an entirely new file [Douglis91, Kistler92].

If a multi-protocol file system designer intends to support client sets from multiple sharing semantics, it is important to recognize that this sharing issue has several important factors. This issue determines the user's perceived behavior of the system and is very dependent on the capabilities of the file system client, especially the caching policy of the client.

In terms of the behavior perceived by the user, some applications may attempt to take advantage of their system's sharing semantic by using it as a primitive foundation upon which to build a means of communication. For example, multiple clients could build a logging facility upon the sharing semantic of their file system. In this system, all of the log entries written to file by multiple clients appear to the other clients since the file system would push the changes to the concurrent clients. On a Unix system, this can be done by have the clients interested in reading the log run the *tail –f* command, which keeps the file open and shows changes to the file as they are appended to the end of the file. Messages can be appended to the log using the *echo* program with the append redirector (>>). If a designer built a multi-protocol file system with underlying AFS-style session semantics, then applications running on this file system that expect to be able to use the file system to communicate with concurrent users would no longer work.

The other issue related to the semantic of concurrent file accesses is the internal capabilities of the client, especially the caching policy of the clients. In clients with primarily Unix semantics, it is important to update the server with the most recent changes to the file, for this reason a write-through caching is sometimes used. In write-through caching, file writes occur to data stored in the local file system cache and are also simultaneously sent to the file server. In NFSv2 and Sprite clients send their writes directly to the server in a write-through manner. In Linux NFSv2, and to a lesser degree in NFSv3, the clients use write-back caching. In write-back caching, file writes occur to data stored in the local file system cache, these writes are sent to the file server when the client needs to evict the written data from the cache or after a certain time period. In AFS, clients implement session semantics, which is a special case of write-back caching more properly termed copy-back caching. In this style of caching, file writes again occur to the local cache but are only sent to the file server when the file is no longer in use by the client.

In the newer implementations of the CIFS file system, clients can use 'opportunistic locks' to determine the caching policy of the client. An opportunistic lock, or oplock in CIFS, is very similar to a resource callback. It is not actually a lock but is a certification from the server that the client can cache the entire contents of a file locally. By holding an oplock on a file, a client can essentially use session semantics for accessing the file. If the server breaks the oplock, it notifies any holders of the oplock. These clients write any pending changes back to permanent storage, and then change their caching policy. The clients change their behavior to implement strict Unix semantics for the file accesses by forwarding all read and write requests directly to the server rather than accessing data in the local cache.

On currently developed network file systems, there is wide disparity on the sharing semantic that is provided to the clients. In Table 15 below, the each table entry shows the view of the file as perceived by a concurrent reader or writer from a particular file system. It shows the range of sharing behavior from the strict session semantics that AFS uses to the Unix semantics of the Sprite system. Of all of these network file systems, CIFS uses the most flexible policy that gives clients the ability to specify a deny-mode flag that determines whether to allow or deny concurrent read or write access.

	Reader's view of file	**Writer's view of file**
CIFS	Depends on deny-mode flags of the writer. Reader may be denied access to the file. Reader may see changes as they are written.	Writer will succeed if allowed by deny-mode flags.
NFS	Reader may see old data, if it has not contacted the server recently. Reader may see partial writes depending on how quickly the writes are flushed to disk.	Writer will succeed.
Sprite	Reader sees changes as they are written.	Writer will succeed.
Andrew (version 2), Amoeba (Bullet)	Reader will not see any of the file writes.	Writer will succeed.
FTP, HTTP	Reader will see partially changed file.	Writer will succeed.

Table 15: Handling Concurrent Reads and Writes in Various File Systems

A special case concerning concurrent writes can occur for some file systems that support append mode writing. For these file systems special care is taken when the file write that occurs is an append. In this case, the file system designer must decide whether or not they wish to support the semantic that no two concurrent appends should ever overlap. In order to support this, the server maintaining the file data must honor the atomicity of each append write and serialize each append so that they do not overlap. In effect, the entire data for the first append must be completely written to the server before the next append is permitted. In this instance of concurrent write access, the file system designer may decide that it is acceptable for the client to cache appends to the end of the file as if they were actually writes starting at a known offset (the cached end-of-file location). A system using this semantic may appear to be dropping some portion or all of an append write, since two concurrent append writers using this system may convert separate appends into writes beginning at the same cached end of the file location. Once written back to the server, one of the appends will appear intact while the other would be totally or partially overwritten. Designers may determine this to be an acceptable trade-off to avoid the synchronization overhead. Since it is primarily mail server applications and logging tools that take advantage of serialized concurrent appends, the designers must consider the importance of these applications in the deployment environment.

4.3.5.1 Possible solution

There are many reasons why supporting concurrent reading and writing is desirable. Clients that have access to a file system can quickly communicate by simply reading and writing to a shared file. In this scenario, there is no need to setup a separate network socket connection and negotiate over a communication protocol. There is also no need to worry about the failure semantics for the connection. In short, the file system is undeniably an easy and powerful method of exchanging data. This is definitely the most desirable method when all of the clients share a common native file system client. Unfortunately, when a designer is considering building a system to handle multi-protocol file system client the problem is suddenly exacerbated by clients that are attempting to perform read and write sharing according to their own data caching policies. The designer is forced to instead find a feasible solution that will work for all clients.

The primary reason that concurrent file usage is such a fundamental semantic problem is because of the philosophy that a file system can be used as a communication medium for applications. Because of this overloaded usage, file system designers go out of their way to support applications that use this feature. However, since this study is primarily concerned with clients of multi-protocol network file systems, it seems that since a network exists perhaps the file system is no longer the best tool to provide reliable message-passing medium. In addition, research on the file access patterns of clients for the Andrew project determined that true concurrent file accesses are rare [Howard88]. Most concurrent accesses occur by multiple threads of the same application and not by multiple applications on the network. Therefore, the set of applications that are using the file system to communicate is small. In fact, it seems that role is better served by message passing through sockets, datagrams, or even distributed shared

memory. If the network file system is no longer viewed as the best tool to provide communication between processes, the disagreement between the implementations of sharing semantics is greatly reduced. A file system designer that does not have to confront the meaning of the users' operations but can focus instead on improving the performance of the file system. Since many network file systems have already diverged from the strict Unix semantic model (Amoeba, Andrew, Coda, CIFS, NFS), it seems that the multi-protocol file system designer should examine the strengths of this choice to see if it fits with the needs of their client base [Kistler92, Leach97, Sandberg85, Tanenbaum90].

One important factor to remember when designing a multi-protocol file system is that the sharing semantics of the system depend greatly on the capabilities of the client. Unless the designer can manipulate or replace the file system client, the client's limitations must be accepted as is. For example, a multi-protocol designer that intends to support Andrew clients must remember that no matter how the multi-protocol file server is built it will not be able to convert the client to use Sprite-like sharing semantics. The server has no interface to update the client with the new data. However, a designer that was supporting a Sprite interface could force the Sprite clients to use session semantics by simply not notifying the client of changes. In short, it is not possible to provide session semantic clients with Unix semantics, but it is possible to provide Unix semantic clients with session semantics.

If a file system designer is building a file storage appliance, with the intent that clients from multiple protocols access it, each protocol having been designed with different design issues in mind, it is necessary to provide the core functionalities of data storage and sharing data and eliminate the file system's role as a medium for inter-process communication. If the file system is not being used a means of communication, then it is not necessary to keep the clients updated with the most recent version of the file. Therefore, it is possible to provide all clients from all semantic sets with session semantics and allow multiple writers and multiple readers to operate concurrently without interfering with each other. Multiple threads and processes on a local machine would still be able to communicate through the file system locally because they would be utilizing a shared cache. This policy does break the semantic of some file systems, but it does not put any client in a situation that would cause it fail unexpectedly.

4.3.6 File Locking

An issue related to the ability of network file system users to share files is the ability to lock files or portions of files in a network file system. This is an extremely important issue because it is a layer of abstraction that can be used to hide any sharing semantic mismatches. For example, a designer of a file storage appliance can take advantage of the file locking of a client set to hide a semantic mismatch at the concurrent access level. The designer may not want to allow concurrent writes, so it may leverage the locking layer to deny write accesses by another client. In this manner the designer can effectively hide the semantic mismatch beneath a layer of abstraction. The concurrent application would not fail since it would respect the locking layer of the file system [Borr98].

File locks play two important roles that need to be considered by the file system designer. The role of the lock can be either an access lock or an observer lock. This property determines the task that the user is attempting to accomplish with the lock. Access locks are designed to help applications synchronize their usage of portions of the target file; observer locks are locks that allow an application to be notified when changes occur to a file.

The observer lock is a tool a client can use if it wishes to be notified when a certain event occurs on a file, this event is the modification of the file, such as by writing, renaming, deleting, etc. In this way, the owner of the observer lock can be notified (also called triggered) when changes occur to a file or directory in the file system. This lock is predominantly used in systems that have file system browsing applications; the browsing application will typically take an observer lock on all of the files and directories in which the user is browsing. The application can then update the view of the file system for the user as it changes. The range of observer lock is the entire file.

The other role of a file system lock can be as an access lock. The lock owner uses an access lock to prevent certain kinds of access to the file. There are two types of access locks, shared or exclusive locks. The owner of a shared access lock is guaranteed that only other owners of that specific shared lock is allowed to access the file. The owner of an exclusive access lock is guaranteed that no other user is allowed to access the file. The second property of an access lock is the range of the lock. This can be either whole-file or byte-range lock and determines the portion of the file that the user intends to use the role of the lock upon. The final property of an access lock is the enforcement policy. Because many network file systems were implemented before a locking system was developed, there is a divergence between the enforcement policies of many access locks. In particular, many file systems support either a

mandatory or an advisory enforcement policy for access locks, sometimes even both simultaneously. In a mandatory lock system, a user is guaranteed that all other clients that wish to use the file must first acquire a lock. The lock is then respected by all reading and writing operations. In the advisory locking system, the locks are respected only if the client chooses to acquire them. A misbehaving client may choose not to acquire an advisory lock and could then go ahead accessing the file unprotected. In short, the advisory locking system requires that the clients choose to obey the locking semantics.

In a multi-protocol file system it is important to recognize that the various locking systems of the clients can be conflicting. A file system that provides a mandatory access locking system to a client would not be able to recognize the intentional breaking of a lock's semantics that is possible in the advisory locking system.

4.3.6.1 Possible solution

Although the differing semantics of the locking systems presents the file system designer with a semantic mismatch, the designer may be able to solve this situation by relying on the file server as the clearinghouse for all of the locking protocol requests. In most network file systems, the client gains a lock on a file by requesting it from a file server. If the server treats all of the requests as mandatory locks, it can deny any requests for advisory locks and even deny requests for access that attempt to circumvent the locking process. In this way the server can enforce the locking semantic that protects the client to the maximum degree. To do this, the designer must be willing to reply with an error code to requests that would otherwise proceed successfully. The server could send any error to the client that would effectively inform the client to retry the request at a later time. In general, the system users would notice that any access errors that occurred were the result of some violation of the sharing semantics. This is especially true because of the rarity of the true concurrent file sharing.

4.4 Definition of Security

Another definition that must be decided by the designer is which security policies that should be used in the multi-protocol file system. In any system that is connected to a public network and hosts files for many users, it is important to provide security to protect the files belonging to users from malicious or inadvertent access. Concerning the security of the file system, there are several issues that need finalization. Issues such as authenticating commands, controlling which users have access to files, controlling the permitted operations, and securing communication.

First, authentication allows the server to correctly identify the users of the system. The server can then combine the access control and access permission security components to describe who is allowed perform what action on a file system artifact. Finally, by communicating securely the server can protect the contents of the data being viewed by users.

4.4.1 Authentication

A major obstacle that designers tackle in a multi-protocol file system is solving the semantic mismatches of the various authentication systems that need to be supported. The authentication system is the means used to identify the clients of file system to the server. All of the other security decisions that the server makes are based on knowing the identity of the client.

Network file systems have many means of granting authentication. The FTP protocol grants access to clients that send a valid username and password unencrypted over the network. In NFS, authentication is granted based on the values of the user and group identifiers (uid, gid) supplied by client machine to the server. In this model, a client need only know the identifier of a privileged user in order to gain access to files accessible to that user. The NFS server has no way to certify the validity of the client's response. In CIFS, the client gains access to the file system resources through a token granted to the client by a password server. The password server can be the file server itself or it can be a separate machine. This system is more secure than the NFS method, except that the early versions of this protocol sent passwords over the network unencrypted. Later versions used a simple cryptographic scheme, but still maintain unencrypted copies of the password on the client's local file system [Allison99a]. In AFS, clients are authenticated through the use of cryptographic tokens granted by a secure Kerberos server. From a security perspective the most important issue is who is the agent authenticating the identity of the user. In NFS, the client machine performs this operation. The AFS and CIFS models are much more secure because the multi-protocol server does not have to trust the client to authenticate the user correctly. Instead the client provides a token from the user that the server can examine and validate on its own.

A multi-protocol file system designer must recognize that by their nature the authentication system of the file system is a portion of the file system semantics that cannot be resolved without altering the code that the client runs. Furthermore, the designer should realize that by supporting file systems with weak authentication semantics, the security is reduced to the level of the weakest system. The designer of the multi-protocol system can support the authentication systems of each system, but it needs to maintain a mapping from the native file system's identification scheme into a unique identifier. In this way, a user would have identical permissions no matter which file system interface it used to access the file system.

4.4.2 Access control

One major security concern for file systems is controlling the list of users that have permission to access a file or directory. There are two primary access control related semantic issues that the multi-protocol file system needs to be able to handle. The first is that the designer needs to recognize which directories need to be examined when granting access to a resource. Second, the designer needs to determine how to resolve the various kinds of access control lists used by different file systems.

First, it is important to resolve which objects need to be examined when a user requests access to a file system object. In the general case, when a user wants access to a file system resource it provides a pathname to the file system that refers to the resource. The file system client will examine the pathname and traverse down through the directory hierarchy until it reaches the named object. If any of the directories along the traversal have access control information that denies the user, the traversal will fail and the user will not be able to access the object. In some network file systems, such as NFS and AFS, these are not the semantics that are provided to the client. In these file systems, the client can specify the parent directory and the target object, or simply the target object, and avoid traversing through the file system hierarchy. The advantage of this approach is that it avoids the expense of examining the entire directory path from the root directory to the target file. This is important for a network file system such as AFS since each directory may be located on a separate volume server, and it may be difficult to assure that the permissions of a directory higher in the hierarchy do not change after the traversal. However, although checking only the parent directory and target object's permission information is faster then getting the entire traversal information, it may be more complex for some users. Some users may set the permissions to deny access to a private directory but may then be surprised to find out that access can still occur to subdirectories of the private directory. In CIFS, unless the user has specific bypassing permissions, the user must be permitted by each directory component along a path [Gobioff97]. This issue is important for multi-protocol designers because it determines which access control information is used to prevent unwanted access. The designer must examine the trade-offs so a balance is reached between the performance penalties of examining each directory along a pathname traversal, and providing clients with an access control checking policy that is potentially counter-intuitive.

The next important access control issue is enumerating which users can be specified to have access to a file. Most file systems control access through the use of an access control list. Network file systems such as NFS and Sprite use a special case of an access control list that is derived from Unix and has only three entries. These entries contain one entry for the file's owner, the owner's group, and another entry for all other users; these three entries make up the user-group-other (owner-group-universe) list. Other file systems such as AFS and CIFS use access control lists that are not limited to three entries. These access control lists can contain the names of users, the names of groups, or a name for everyone else. In this manner the names of all users permitted to access the file can be enumerated above and beyond the standard three Unix-style entries.

For a designer of a new multi-protocol file system the unbounded access control list style is certainly more desirable than the user-group-other style. This style of access control allows management that is more precise control over access to files and directories. If a multi-protocol file system designer builds an access control list infrastructure, then it can support access from client sets that use the access control list style as well as client sets that use the user-group-other style. To support client sets that use the user-group-other style, the multi-protocol system can provide an access control list that has three entries: one entry for the owner, one entry for an associated primary group, and one entry for all others. To support client sets that use access control lists, the multi-protocol system can provide an access control list that maps users and groups from one file system to another. Although a multi-protocol file system can be created which uses simple access control lists that are equivalent to the user-group-other style to support one client set, and uses mappings of access control lists to support the second client set, there are limits to this solution. If the designer chooses to allow a full access control list system, it cannot fully support user-group-other style clients

because it will not be able to display these access control lists to these clients. The user-group-other style clients can only understand a three-entry access control list. Since there must be one entry for the file owner and another for all other users, that means there is only one group access control entry remaining to be used for mapping the rest of the access control list.

In short, it is impossible to display the full set of users and groups of an access control list in the user-group-other style. However, since the file system server grants accesses, the server has the ability to grant access to files even when the view of the access control information displayed to the client states otherwise. One solution that could be implemented is that when a client that only understands the user-group-other style requests access control information then the multi-protocol file server can send a user-group-other list that contains a group that permits or denies the user based on the value of the actual permission check. In this way, the client may believe it is accessing a server that is using the user-group-other style. A major limitation to this method is that in many file systems the client system may cache the access control information so that if another client accesses the file it can simply check the permissions of the client without consulting the server. To avoid this problem, the multi-protocol server may be able to send the client a user-group-other list that contains an unknown group, in an attempt to force the client to check with the server directly rather than making access decisions on its own.

In short, when managing clients that cannot understand a rich access control representation, it is important to choose a policy that guarantees that the server fulfils its goal of protecting files from disallowed access. It is safer, from an application and security perspective, to display that these clients can access a file and for them to later determine that they cannot, than to display that these clients cannot access a file and for them to later determine that they can. This is an important policy for all security issues for the multi-protocol server. The server may display incorrect information showing that a user is allowed, but it should never incorrectly display that a user is not allowed. This policy means that the system will never give its users a false sense of security. In effect, it may be more acceptable to display that an unwanted user has permission to access a file but are denied access when they try, than to claim that an unwanted user does not have access when in fact they actually do.

4.4.2.1 Possible solution

To control fully the access control and access permissions of a file, one solution that a multi-protocol designer could choose to implement would be an access control list system in which access control entries could contain a list of both positive rights and negative rights. This is useful because it may be desirable to create an access control entry that represented a subset of a group of users. Rather than enumerate each user that is granted access, the server can issue a positive access control entry to a group and then issue a negative access control entry to the users in that group that are not allowed to access the file. Although negative access control entries are not necessary to support all the semantics of access control lists, using negative access control entries can reduce the amount of space needed to represent a complex access control list.

To fully represent the semantics of most file systems it may also be necessary to provide an access control list for both files and directories, as well as a default access control list to use for newly created files and directories. In this manner, the permissions of new files and directories can be inherited from the directory in which they are created.

In effect, by building support for a complex access control list scheme, the multi-protocol file system designer can guarantee that any access control list constructed by a file system client can be converted efficiently into its internally stored system.

4.4.3 Access permission

The next important semantic issue that needs to be resolved in a multi-protocol file system's security is its access permission definition. That is, it must define what operations users can perform. When the definition for file system access permissions is linked with its access control system, the result is a means to map each permitted users to a set of operations that it is allowed to perform.

As with the access control issue, there are several sets of access permissions that have been used in file systems, both for convenience and historical reasons. In Sun NFS and Sprite, the set of possible access permissions derives from the set used in Unix, that is permission set consists of Read, Write, and Execute (rwx). From these three permission bits, the server determined whether a client was allowed to perform all of its operations. In AFS, it was recognized that it was desirable to represent access permissions at a finer granularity. To accomplish this, the AFS

set of permissions consisting of Read, Lookup, Insert, Delete, Write, Lock, and Administer (rlidwka) was used. With this larger set of permission bits available, users could more precisely control which operations were allowed. CIFS and NetWare also have designed their own set of permissions and now like many semantic issues, finding a common ground between each of these file systems' methods has become quite difficult [Hitz98, Novell94]. The permission bits for several network file systems are enumerated in the table below. Although many file systems use common names for some permission bits, such as the "read" permission bit, it is important to keep in mind that the operations that this permission bit allows varies dramatically from one file system to the next.

NFS/FTP	CIFS	Andrew	NetWare
Read (R)	Read (R)	Read (R)	Supervisor (S)
Write (W)	Write (W)	Lookup (L)	Read (R)
Execute (X)	Execute (X)	Insert (I)	Write (W)
	Delete (D)	Delete(D)	Create (C)
	Change Permissions (P)	Write(W)	Erase (E)
	Take Ownership (O)	Lock(K)	Modify (M)
		Administer(A)	File Scan (F)
			Access Control (A)

Table 16: Permission Bits for Various File Systems

The basic semantic question that all of these permission strategies address is the whether an operation should be permitted or denied. Fundamentally, this is a Boolean operation. Since no file system operations are allowed to be partially completed, the permission check process results in an equation where the result is either true or false, permitting or denying the operation.

To support the security model requested by clients, the multi-protocol file system must be able to support operations to add or remove permissions as requested by the users. As users change permissions on an access control entry, they add and remove permissions from the permission set of the access control entry. This permission set is an abstract collection that contains all of the operations that the user named on the access control entry is allowed to complete. To decrease the amount of space required to store the permission set, most file systems overload related permissions to reduce the total number of Boolean variables in the permission set. For example in Sun NFS, the directory Write permission is overloaded to allow several operations such as file and directory renaming, file and directory creation, and file and directory deletion. However in CIFS, the deletion of files and directories is granted by a distinct Delete permission, and in AFS the creation of files and directories is granted by the Insert permission.

The access permission issue suffers from similar problems as those related to the access control issues that were described earlier, that is, clients that use smaller permission sets, such as Sun NFS clients, cannot view complex access permission sets that are used by other clients. Therefore, if these clients observed the permission set on an access control entry, they may not be able to determine if an operation is allowed because there may not be a direct mapping into their file system of each of the set permission flags. Although it is impossible to create an exact mapping between the permission sets of each file system for display purposes, it is possible to support each file system's permission adding and removal operations.

Supporting all of the permission adding and removal operations of various file systems can be done by creating a large internally used permission set, in which each interface operation on the multi-protocol server has an associated Boolean flag. In this way, there is a single flag that allows or denies each operation. By using this large permission set, the multi-protocol system designer can support the permissions of every file system that it supports. Since there is a one-to-one mapping between Boolean permission bits and interface functions, there is no permission that cannot be represented by the server. It can represent each permission flag of a particular file system as the logical 'or' operation of one or more of its internally maintained permission flags. Since the multi-protocol server uses a Boolean flag for each operation, there is no way that two clients from different client sets can issue permission add and removal operations that will result in an unknown state on the server.

To display the access permissions to a client, the multi-protocol designer should follow the policy that it is safer to display that an operation can be completed and later deny the operation, than to display that an operation can not be done and later accept it. For example, consider the fictitious situation where a multi-protocol file system is supporting the semantics of the Alpha and the Beta file system. In the Alpha file system there is an access

permission named “Read” that allows a client to read a file’s contents, lookup the name in a directory, and get the attributes of the file. In the Beta file system, an access permission also named “Read” allows clients to read a file’s contents and lookup the name in a directory. If an Alpha client adds the Alpha-Read permission to an access control entry and a subsequent Beta client removes the Beta-Read permission, the multi-protocol file system must be able to display a legitimate permission set to each of the Alpha and Beta clients. From the perspective of the Beta client, the multi-protocol file system can display that the Beta-Read permission is not granted, since clients cannot read a file’s contents or lookup the name in a directory. From the perspective of the Alpha client, the Alpha-Read permission has been partially revoked, clients cannot read a file’s contents or lookup the name in a directory, but they can still get the attributes of the file. Because the access control entry has part of the Alpha client’s “Read” permission, the multi-protocol file system should display to this client that it has the Alpha-Read permission. In this way, the system is not misleading an Alpha client into believing that the “Read” permission is not allowed. In this scenario, the server will always understand the full interpretation of the permission set so it can determine whether to allow any operation.

Existing multi-protocol solutions follow a range of policies, many simply use Unix style permissions and access control lists, converting all accesses to this style. Other multi-protocol solutions such as Network Appliance’s filers have chosen a separate way to handle access permissions and access control for their file system [Hitz98]. In their file system, users can set that an entire directory or directory tree follows either NFS or CIFS style semantics. Non-native accesses to these directories are converted into the style for the directory. In addition, clients are allowed to set directories to be mixed-mode. In the mixed-mode directory, access style is set on a per-file basis. In this way, files created on the NFS interface have NFS access permissions and a Unix style access control list, and files created on the CIFS interface use the corresponding CIFS style. Although, this approach does not provide the user with a multi-protocol solution, it is easy for a user to understand since it they can consider their files as either “NFS files” or “CIFS files.”

4.4.4 Communicating Securely

Another important semantic issue that defines the security of a multi-protocol file system is the means of protecting and securing the communication between the clients and the server. There are several aspects to communication that must be secured. The server should be able to authenticate where communication comes from, guarantee the integrity of communication, ensure that communication between the client and server is private, and avoid replayed communication. If each of these aspects can be guaranteed then the communication between the server and its clients is fully secured [Gobioff97].

The security of a file-system is one semantic issue that cannot be changed without changing the communication protocol used by the clients. That is, unless a particular client set has the ability to use encryption to guarantee that communication is private then there is no behavior that a multi-protocol server can do that will elicit encrypted communication. Therefore, there is no means of resolving the security of communication between file systems, the server must be able to understand and use the communication convention of each supported client set. As in any security issue for a multi-protocol system, the security level of the entire system is the security of the weakest interface.

For the a designer of a new network file system, it is important to recognize the need for communication security and the steps required for building this security. To authenticate communication, the server and the client need to be able to agree on a shared secret that can be used by the server to determine the identity of the party it is communicating with. In AFS, authentication can be done through the use of Kerberos tokens. To guarantee the integrity of communication, the client can construct a digest of the communication, in which the digest is constructed through the use of a shared secret key. To keep communication private, the server and client need to agree on a session key that can be used to keep communication encrypted. Finally, to avoid an unintentional or malicious replay of communication, the server needs a simple means to verify that the communication it sees is fresh. The client and server can agree on a system of nonces to avoid replayed communication.

4.5 Definition of Properties

The fifth category of file system semantics that needs to be resolved in a multi-protocol file system is the properties of files and directories that will be used by the file system to classify directories and files and determine high-level information about directories and files. Although file and directory properties are information about the files and directories rather than the actual file and directories themselves, it is still important that this information be retrieved

quickly. In fact, in research done for the NFS and AFS file system the retrieval of the properties and attributes of files and directories was an extremely common operation [Spasojevic96]. The naming of files and directories is one major aspect of the semantics of file system properties. The attribute set of files and directories is the other major aspect to the semantics of the file system properties.

	Naming	File Attributes	Dir Attributes	File Dates	Dir Dates
NFS, Andrew, Unix	• case sensitive • any character except "/" • names beginning with "." are hidden	size link count owner group inode number	size link count owner group inode number	changed modified accessed	changed modified accessed
CIFS	**Naming**	**File Attributes**	**Dir Attributes**	**File Dates**	**Dir Dates**
	• case insensitive • any character except "\/:*?"<>|" • older clients must be able to request an 8.3 name (*) • newer clients can use Unicode	size size used owner read-only archive hidden system compressed	size size used owner read-only archive hidden system compressed	created modified accessed	created
NetWare	**Naming**	**File Attributes**	**Dir Attributes**	**File Dates**	**Dir Dates**
	• case insensitive • any character except "\/:*?"<>|" • older clients must be able to request an 8.3 name (*)	archive needed copy inhibit delete inhibit execute only hidden index normal purge rename inhibit read only read write shareable size system file transactional don't compress don't migrate immediate compress compress can't compress migrated	delete inhibit hidden normal purge rename inhibit size system don't compress don't migrate immediate compress	created modified accessed	created
* an 8.3 name is a name consists of an up to eight letter word, followed by a period, followed by up a three letter "extension"					

Table 17: Properties from Various File Systems

Although the multi-protocol server cannot affect how the file system clients construct names, it can control the mapping between how names of directories and files are mapped between multiple file system protocols. In early file systems, the clients were limited to short filenames, a small character set, and case insensitivity. In more recent file systems, the filename length limit has been extended and the set of valid characters has been expanded. Both of these changes to the filename restrictions have made the task of the multi-protocol server more complex. The multi-protocol server can choose to accept the filename restrictions of the client sets it is supporting or it can construct a mapping between the incompatible naming systems. Although the multi-protocol server can support complex naming systems such as Unicode, it must be able to map this set of characters into the smaller character set of other supported client sets. It also must be able to map the long file names from one file system client into the shorter file length limits of supported client sets.

To construct a mapping from longer filenames with complex character sets to the limits of another client set is essentially a hashing problem. Although it is possible to map names from one file system to another by hashing the complex filename, the users of the simpler file system desire that the file and directory name be something related to the original filename. For most users that construct filenames using the ASCII character set, mapping from a longer to a short filename could be a simple truncation of the filename. However, for client sets that use Unicode characters constructing an acceptable mapping that can be used by CIFS clients that use the DOS (8.3) 11-character limit, for example, may be impossible without input from the file users. To solve this issue, the multi-protocol system

designers can either construct a mapping that they hope the users of the system will accept or provide a tool to the users to control the naming of the files and directories.

Another semantic issue that must be resolved is supporting the set of attributes used by all of the supported client sets. Attributes are the set of variables that can be used to determine certain high-level information about the files or directories. Each network file system provides their users with the ability to store information about the files such as whether the file or directory is hidden, if it is read-only, time-stamps about the creation and modification of the file, and the size of the files. There are several major classes of attributes that file systems store: informational attributes, overloaded access permission attributes, and timestamp attributes.

Most attributes that file systems manage are considered informational; these are attributes that do not affect how clients use the files and directories. There are some attributes however, such as the archive attribute whose existence is relied upon by some applications. For example, the archive attribute is used by many backup applications to determine whether or not a file has been backed up, the application can clear all of the archive bits on a set of files and then set the flags as it progresses. A designer of a multi-protocol file system should examine the set of attributes that the applications running in the target environment will require to contain valid information, and then choose these attributes in addition to any other informational attributes they feel they have storage space to support. Some of these properties, such as the file size and file id number (inode number), are useful to both the file system internals and to the users of the files. Other attributes, such as CIFS' system flag, are flags that exist for historical reasons. Since these flags are relatively unused, a file system designer could choose not to support changes to them but rather to always display default values of these attributes if queried for them by clients. On the other hand, since the space requirements for these flags are relatively small, only one bit per file, if space is available it should be provided for these informational flags.

Another set of attributes that are commonly provided by file systems are access permission attributes; these are attributes such as the executable flag, the read-only flag, and the hidden flag. These attributes are sometimes used by file systems as another means of access control. For example, many file systems grant a separate access permission to allow users to execute a file, and many file systems also allow users to set a read-only flag. With regard to the execute access permission, this may give users the false perception that someone who is not granted an execute permission on a network file system cannot execute a file. In fact, a user that has the ability to read the contents of a file can simply copy the contents of the file, and later execute it on a local file system. This problem with the execute access permission is a security issue that should be solved by using an executable attribute to designate to operating systems that a file can be executed. The read access permission should be used to control who is able to read the executable data in the file. Similarly, some file systems allow read-only and hidden attributes to be set, again these attributes may give the false impression to users that the file cannot be written to, or that the file cannot be observed by other users. In fact, these attributes are only suggestions to the client applications and can be overridden by them if desired. If a user actually wants to enforce these policies, it must use the access permissions instead of attributes. Therefore, although these attributes can be supported, designers of new file systems that use these permissions should make sure that is obvious to users that these attributes only suggest behavior and are not enforced by any secure means.

Samba has some unique methods for maintaining non-native attributes, it stores the CIFS archive bit in the underlying system's Unix user execute permission bit. In addition, Samba also returns successfully when requests arrive to change the hidden attribute; however, the actual value of the hidden attribute on a later attribute retrieval is unchanged.

The final set of attributes that are commonly provided by file systems are timestamps. Timestamps can exist for the creation time, time of last access, and times of last modification to data or attributes. Timestamps are commonly used by programs that rely on the date information for the coordination of tasks such as archival backup and source code compilation. For many of these tasks the precision and accuracy of the timestamps is not important. For these purposes, the timestamps on a file only need to inform the application whether the file has changed since the timestamp was last examined. Research at Cornell on Windows file system traces revealed that the file system's timestamps were often incorrect and only used for informational purposes [Vogels99].

Semantically, timestamps present a unique problem because some filesystems such as NFS prefer the server to update the access timestamps for a file; others such as AFS prefer the client to set this information. In a system that

is compiling files over the network, clock skew between the server and the client can cause some files not to be built. Another influence on timestamps is the caching policy of the clients. If the server constructs the timestamps it may create its timestamp based on the time the cached data was written back rather than when the file was logically modified by a write.

In the multi-protocol system, the server receives all requests for attribute modification; therefore, it may be possible for the server to make sure that the timestamps set by the client are at least as recent as the current time on the server. In this way, applications using the values of client created timestamps will not see the timestamps for some files drift behind the timestamps of files kept on the server.

4.6 Definition of an Error

The final semantic category that should be examined by a multi-protocol file system designer is the definition of errors used by the supported file systems. This category of semantics is important because it is important for a file system designer to report errors to clients that will prompt the client to a desired behavior. For some semantics that the file system designer cannot resolve, the designer may choose instead to report errors even when an actual error does not exist. In this way, the designer may be able to enforce desired behaviors in some file system clients. The final aspect to the definition of an error that will be discussed is what is called *bug for bug compatibility*. This is important when considering applications that rely on particular behaviors of the system, regardless of whether the behavior is documented or even intentional.

To control the sharing behaviors of clients, it may be desirable to report to one client that a certain file is temporarily unavailable rather than resolve incompatible sharing semantics. The error may trigger the client to retry later, or it may trigger the client to change its set of parameters and in this way comply better with the intended semantics of the multi-protocol file system. To do this, the designers of file systems should examine the set of errors that can be reported to clients. It should even examine error conditions for situations that may not necessarily apply to the server, such as the behavior of clients when responding to removable devices or write-only devices. It may also be useful to examine the behavior of clients when faced with access permission errors. By taking advantage of these classes of errors, the multi-protocol file system may be able to elicit desirable behavior from clients.

Another reason that it may become desirable to break a client set's definition of an error is to protect the client from harm that can occur from other semantic mismatches. If the designer chooses to radically alter the behavior of a server, the unexpected behavior of the server may cause any range of effects to the client applications. However, even if the error was outside the expected set of errors, it may be possible to hide the strange behavior of the server from the clients by displaying to clients an error condition that they would be able to recognize and even recover. In short, it is not important from the designer's perspective whether the meaning of the error code is correct as long as the client does the intended behavior.

Another important issue in the definition of an error is the degree to which the multi-protocol file system needs to be *bug-compatible* with the server it is being designed to imitate. Being bug compatible means that the multi-protocol server may be required to match errors in earlier versions of the file system. This is an extremely important issue since some applications may be built to rely on a particular nuance of the file system and if the multi-protocol file system does not exhibit the same behavior then the application may fail, even if the original behavior was a bug. An example of this is applications that are programmed to be fault tolerant. These programs may have a high degree of dependence on very particular aspects of the file system, such as the error codes that are returned. The aspects of the file system that the application depends upon may not be documented but may be the result of the programmer trying a particular access and then modifying the application code based on the result. Unfortunately for the multi-protocol designer this means that to become completely bug-compatible with an existing file system server, the designer may have to enumerate and search all interactions with the server will all possible input values. The result is that the designer is faced with an unbounded error set. Moreover, this examination can only be done by either scrutinizing the source code of the system or by reverse engineering a finished product. Since this puts the designer in an uncomfortable position, perhaps the only reasonable approach the designer can take is to examine the deployment environment and seek to be bug-compatible enough to allow the key applications running in the environment to be successful.

4.7 Summary

As this paper has shown, there are a number of semantic mismatches that can arise when building a multi-protocol network file system. In order to choose an approach to use when faced with a semantic mismatch, a designer must have a good understanding of the problems that may arise if one solution is chosen over another. Semantic mismatches in a multi-protocol network file systems can create many problems for users of the file system and the applications they run. The effects of a semantic mismatch can range widely in their severity. Some mismatches may cause an application to crash or data to be lost, others may cause an application to fail unexpectedly, and still other mismatches may cause the user to become confused by the system's behavior.

In general, without considering the details of a specific multi-protocol file system deployment, we can choose several semantic issues that take highest priority because of their expected severity. Concurrent read and write sharing semantics are of vital importance because this semantic influences the perceived reliability of the system for storing a user's data. As we have seen, choices of resolving these sharing semantic mismatches may cause some applications to crash if they expect sharing activity to be handled differently. Another of the most important semantic issues in a multi-protocol file system is bug compatibility. The reason this is important is because an application that expects to run on one file system may fail unexpectedly, silently, or with an unusual error message when run on the multi-protocol system. Finally, another crucial issue in multi-protocol systems is the security of communication because it is an issue that cannot be resolved and can result in a situation where the multi-protocol system has a vulnerable backdoor. In conclusion, when the designer of a multi-protocol file system considers the implications of the techniques used to resolve semantic mismatches, they can construct a ranking of the severity of the mismatches. Depending on the particular deployment, the users and the intended usage of the system, the designer may rank the importance of set of mismatches higher or lower. The designer needs to have an understanding of the intended deployment of the system and an understanding of the applications that may run on the finished multi-protocol system. In addition, the designer needs to have good knowledge of the users to be able to determine what solution to a semantic mismatch is most desirable. It may be more desirable to the end user to be provided with a multi-protocol file system that does not match exact semantics, but is instead easier to understand and predict than a system that approximates the semantics.

5 Conclusion

In the future the demand for a multi-protocol network file system, which satisfies the demands of clients from many different file system interfaces, will increase. In particular, it will be necessary to provide a large set of heterogeneous clients with access to a common set of files. This information sharing, although very desirable, places a difficult demand on a file system designer. The designer no longer has the luxury of considering only one set of similar clients. Instead, the set of clients demanding access to the shared data will be from many network file system interfaces and from many different client platforms.

In order to serve these clients effectively, the file system designer must have an appreciation of the design goals of each of the file systems. By examining the goals of the network file system, the designer can understand the reasons that some semantic choices were made. The designer can then enumerate all of semantic mismatches that may arise by examining each of the categories of semantic mismatches.

Once the mismatches have been enumerated, the file system designer can use various techniques, such as file system trace analysis, to determine the actual behavior of the target clients. When the designer has a clear understanding of the initial goals of the file system, how the goals motivated specific semantic choices, and the actual behavior of clients, the designer can then decide how best to resolve the semantics of this file system. However, before choosing any solution the designer must understand the costs and benefits of supporting a specific semantic of a client set. It may be the case that the costs of supporting the semantic are too high or that the benefits are too low. In these cases, the designer must be willing to accept to the consequences of sacrificing a particular client set's semantic so that the goal of providing a filing service to many client sets can be achieved.

In short, the problem of resolving file semantics is difficult. This paper primarily examined the file system semantics for NFS, CIFS, and AFS. By analyzing a file system using the mismatch category model that was presented, a designer can successfully determine many of the mismatches that a file system will face. The designer can then use traces of file system to discover the probability of a semantic mismatch. With this knowledge in mind the designer will be able to successfully resolve the semantic differences of their targeted file systems so that their multi-protocol file system can support a wider and more heterogeneous set of clients.

6 References

[Allison99a] Allison, B., CIFS Authentication and Security, Network Appliance, 1999.

[Allison99b] Allison, B., R. Hawley, A. Borr, M. Muhlestein, and D. Hitz, File System Security: Secure Network Data Sharing for NT and Unix, Network Appliance, 1999.

[Appleton] Appleton R., Prefetching File Systems- Traces of Disk Activity, http://euclid.acs.nmu.edu/~randy/Research/traces.html.

[Berners-Lee96] Berners-Lee, T., R. Fielding, and H. Frystyk, Hypertext Transfer Protocol (HTTP/1.0), RFC-1945, IETF Network Working Group, May 1996.

[Borr98] Borr, A., SecureShare: Safe Unix/Windows File Sharing through Multiprotocol Locking, Proceedings of the 2nd USENIX Windows NT Symposium, August 1998.

[Brown00] Brown, A. and D.A. Patterson. "Towards Availability Benchmarks: A Case Study of Software RAID Systems." Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, June 2000.

[Corbett96] Corbett, P. and D. Feitelson, The Vesta Parallel File System, ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996.

[Devarakonda96] Devarakonda, M., B. Kish, and A. Mohindra, Recovery in the Calypso File System, ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996.

[Douglis91] Douglis, F., M. Kaashoek, A. Tanenbaum, and J. Ousterhout, A Comparison of Two Distributed Systems: Amoeba and Sprite, Computing Systems, Vol. 4, No. 3, December 1991.

[Fielding97] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, Hypertext Transfer Protocol (HTTP/1.1), RFC-2068, IETF Network Working Group, January 1997.

[Gifford91] Gifford, D., P. Jouvelot, M. Sheldon, and J. O'Toole, Semantic File Systems, Proceedings of the Thirteenth ACM Symposium on Operating Systems, Vol. 25, No. 5, October 1991.

[Gobioff97] Gobioff, H., G. Gibson, and D. Tygar, Security for Network Attached Storage Devices, School of Computer Science, Carnegie-Mellon University, October 1997.

[Gray96] Gray. J., Hellan, P., O'Neil, P., and D. Shasha, The Dangers of Replication and a Solution, SIGMOD, Montreal, Canada, June 1996.

[Gronvall99] Gronvall, B., A. Westerlund, and S. Pink, The Design of a Multicast-based Distributed File System, Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, February 1999.

[Hartman95] Hartman, J. and J. Ousterhout, The Zebra Striped Network File System, ACM Transactions on Computer Systems, Vol. 13, No. 3, August 1995.

[Hitz98] Hitz, D., B. Allison, A. Borr, R. Hawley, and M. Muhlestein, Merging NT and Unix Filesystem Permissions, Proceedings of the 2nd USENIX Windows NT Symposium, August 1998.

[Hitz99] Hitz, D., Building a File Service Infrastructure for Unix and Windows NT clients, Network Appliance, 1999.

[Howard88] Howard, J., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, Scale and Performance in a Distributed File System, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.

[Kiselyov99] Kiselyov, O., A Network File System over HTTP: Remote Access and Modification of Files and Files, 1999 USENIX Annual Technical Conference, FREENIX Track, June 1999.

[Kistler92] Kistler, J. and M. Satyanarayanan, Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992.

[Kuenning] Kuenning, G., The SEER Predictive Hoarding System, http://fmg-www.cs.ucla.edu/geoff/seer_traces.html.

[Kuenning94] Kuenning, G., G. Popek, and P. Reiher, An Analysis of Trace Data for Predictive File Caching in Mobile Computing, Proceedings of the 1994 Summer Usenix Conference, 1994.

[Leach97] Leach, P. and D. Naik, A Common Internet File System (CIFS/1.0) Protocol, IETF Network Working Group, December 1997.

[Lee99] Lee, Y., K. Leung, and M. Satyanarayanan, Operation-based Update Propagation in a Mobile File System, Proceedings of the USENIX Annual Technical Conference, June 1999.

[Mann94] Mann, T., A. Birrell, A. Hisgen, C. Jerian, and G. Swart, A Coherent Distributed File Cache with Directory Write-Behind, ACM Transactions on Computer Systems, Vol. 12, No. 2, May 1994.

[Mummert96a] Mummert, L., J. Kistler, T. Kroeger, and M. Satyanarayanan, Coda Project Traces and DFSTrace, http://csl.cse.ucsc.edu/projects/DFSTrace/.

[Mummert96b] Mummert, L. B., and Satyanarayanan M., *Long Term Distributed File Reference Tracing: Implementation and Experience*, Software Practices and Experiences, Vol. 26, No.6, 1996.

[Nelson93] Nelson, M., Y. Khalidi, and P. Madany, The Spring File System, Sun Microsystems Inc., February 1993.

[Nelson87] Nelson, M., B. Welch, and J. Ousterhout, Caching in the Sprite Network File System, Proceedings of the 11th ACM Symposium on Operating Systems Principles, Vol. 21, No. 5, 1987.

[Novell94] Novell Inc., File System Security, http://developer.novell.com/research/appnotes/1994/april/03/06.htm, April 1994.

[Novell98a] Novell Inc., Novell Storage Services for NetWare 5, http://www.novell.com/documentation/lg/nw5/pdfdoc/nss__enu.pdf, July 1998.

[Novell98b] Novell Inc., Traditional File Services for NetWare 5, http://www.novell.com/documentation/lg/nw5/pdfdoc/trad_enu.pdf, July 1998.

[OSF] Open Software Foundation, File Systems in a Distributed Computing Environment.

[Pawlowski94] Pawlowski, B., C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, NFS Version 3 Design and Implementation, The USENIX Association, June 1994.

[Postel85] Postel, J. and J. Reynolds, File Transfer Protocol (FTP), RFC-959, IETF Network Working Group, October 1985

[Sandberg85] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, Design and Implementation of the Sun Network Filesystem, USENIX Conference Proceedings, USENIX Association, 1985.

[Satya89] Satyanarayanan, M. Integrating Security in a Large Distributed System, ACM Transactions on Computer Systems, Vol. 7, No. 3, Aug. 1989.

[Satya92] Satyanarayanan, M., The Influence of Scale on Distributed File System Design, IEEE Transactions on Software Engineering, Vol. 18, No. 1, January 1992.

[Satya96] Satyanarayanan, M. and Spasojevic, M., AFS and the Web: Competitors or Collaborators?, Proceedings of the Seventh ACM SIGOPS European Workshop, September 1996.

[Shaw95] Shaw, M., Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging, Symposium on Software Reuse, April 1995.

[Spasojevic96] Spasojevic, M. and M. Satyanarayanan, An Empirical Study of a Wide-Area Distributed File System, ACM Transactions on Computer Systems, Vol. 14, No. 2, May 1996.

[Srinivasan89] Srinivasan, V. and J. Mogul, Spritely NFS: Experiments with Cache-Consistency Protocols, Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, May 1989.

[SunMicro89] Sun Microsystems Inc., NFS: Network File System Protocol Specification, RFC-1094, Sun Microsystems Inc., March 1989.

[SunMicro95] Sun Microsystems Inc., NFS Version 3 Protocol Specification, RFC-1813, Sun Microsystems Inc., June 1995.

[Tanenbaum90] Tanenbaum, A, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen, and G. van Rossum, Experiences with the Amoeba Distributed Operating System, Communications of the ACM, Vol. 33, December 1990.

[Vogels99] Vogels, W., File System Usage in Windows NT 4.0, Department of Computer Science, Cornell University, 1999.

[Watson99] Watson, A. and P. Benn, Multiprotocol Data Access: NFS, CIFS, and HTTP, Network Appliance, 1999.

[Westerlund98] Westerlund, A., and J. Danielsson, Arla: A Free AFS Client, USENIX 1998 Annual Technical Conference, FREENIX Track, June 1998.