Linux NFS Version 4: Implementation and Administration

William A. Adamson
CITI
University of Michigan
Ann Arbor, Michigan, 48103
andros@citi.umich.edu, http://www.citi.umich.edu/u/andros
Kendrick M. Smith
CITI
University of Michigan
Ann Arbor, Michigan, 48103
kmsmith@citi.umich.edu, http://www.citi.umich.edu/u/kmsmith

Abstract

NFS Version 4 has many new features that differentiate it from previous versions of NFS and address shortcomings in areas such as security and WAN performance. In this paper we detail CITI's reference implementation of NFS V4 [RFC3010] in the Linux kernel.

We describe the changes we made to the Linux kernel to implement the protocol, and issues in matching protocol requirements with POSIX behavior in local Linux filesystems. We also discuss the management of client and server state, security and locking mechanisms, compound RPC, and client data caching.

We conclude with a discussion of the challenges in NFS V4 system administration and issues in migrating from and coexisting with earlier versions of NFS. This paper is targeted to file system developers and network administrators who are interested in the design and features of secure, standard, interoperable distributed file systems.

1 Introduction

The tradeoffs of distributed file systems have been known for many years [Khanna]. The advantages of providing a centrally administered shared file space available to geographically separated users versus the issues surrounding performance, security, cross-platform behaviour, and administrative overhead have resulted in several successful (surviving?) products such as NFS, AFS, CIFS, Novell, AppleTalk, etc that emphasize different tradeoff solutions.

The original NFS version 1 protocol [RFC1094] was designed to operate in a LAN environment with an emphasis on simple servers, timely recovery from server and client failure, and good throughput for reading and writing. Security was not an issue, as most of the product space consisted of corporate intranets.

Subsequent NFS versions [RFC1813] have added new features and performance improvements without changing the original model of stateless servers and clients. As the number of networked computers as well as the mobility and number of computer users has increased, NFS finds itself in an environment it was not designed for - requirements for high performance long haul secure file access.

In addressing this new Internet world, NFS V4 is a dramatic departure from its predecessors. The protocol purports to reduce network traffic by use of a new rpc construct, the compound rpc and by client file cache consistency guarantees provided by new delegation operations. Locking is part of the protocol, with support for both share and byte range locks. Strong security is mandated by use of the RPCSEC_GSS protocol, and new ACL's associated with each file. Finally, NFS V4 is an IETF protocol, with an extension path.

Our current implementation of the NFS V4 client and server is ported to the Linux-2.4.4 kernel, and uses the ext2 file system as the underlying local file system. We have, of course, implemented the client and server with an eye to being independent of the underlying local file system - although we do depend on ext2 particulars with regards to our early ACL work.

NFS V4 is a young, complex protocol, and is changing as implementors fit together all the pieces for the first time. We present short descriptions, and detailed discussion on implementation issues surrounding NFS V4 protocol features. We recommend the use of [RFC3010] as a reference for vocabulary, and a more detailed description of the protocol.

2 Namespace Issues

The NFS V4 keeps many of the same high level namespace ideas as previous NFS versions: it is the client that constructs the hierarchical file name space using mounts to build a hierarchy, with NFS servers expressing the exported portions of their local filesystems in an /etc/exports file. There are however major differences in the design of the NFS V4 namespace, with consequences that have yet to be fully explored.

2.1 Server Pseudo File System

The NFS V4 server joins its exported sub-trees with a read-only virtual file system called the Pseudo File System. The idea is that any client can mount the Pseudo FS. Users then browse the Pseudo FS via LOOKUP. Access into exported sub-trees is based on the user's rpc credentials and file system permissions. This is a major change from previous versions of NFS where an IP based access list was associated with each exported sub-tree. This design has several features for the NFS administrators.

- *Client /etc/fstab maintenance*: Note that if the client mounts the root of the Pseudo FS, the client's /etc/fstab entry does not change with changes to the server's export sub-tree list.
- *Server export namespace*: On the server, the local namespaces and export namespace are kept separate, so that either can be reorganized without changing the other.

The pseudo file system pulls together disjoint parts of the exported local file system into a coherent, browsable package for the client.

2.2 Owner, Group and File Permissions

NFS V4 departs from the use of on the wire numeric uid and gid. Instead, UTF-8 [RFC2279] strings are used, expressing owner and group names in the form

```
user@domain-name, group@domain-name
```

where the @domain-name piece of the name is optional. This was done to provide a common form independent of a particular underlying implementation for local storage or presentation to the end user. Since the Linux kernel as well as the ext2 file system expresses owner and group in terms of numeric uids and gids, a translation of the on the wire UTF-8 strings is needed. The name services that resolve names to uid's exist outside of the kernel. The simplest form of such a service is the local /etc/passwd and /etc/group files. Remote data bases such as NIS or LDAP could also be used, providing extended service such as uid mapping. Following Sun Microsystem's lead, we have implemented a user space daemon, GSSD to perform the name to numeric identity translation. GSSD is also part of the RPCSEC_GSS implementation. It communicates with the NFS V4 kernel client and server via an rpc interface. Currently, it uses the local /etc/passwd and /etc/group files for name and id resolution.

The connection between the principal piece and the realm piece of the NFS V4 UTF-8 owner or group name is open to interpretation. Thus, similar names could possibly be the same user, or could indicate different users.

```
1. alice@
2. alice@umich.edu
3. alice@cable.net
4. "/C=US/ST=Michigan/L=Ann Arbor
   /O=University of Michigan
   /OU=TEST -- CITI Client CA v1
   /CN=Alice L. User 1
```

```
     (07749036)/Email=alice@UMICH.EDU"
```

The first name indicates a local UNIX user, the second name could be a user in the UMICH.EDU Kerberos V5 realm, the third name, a DNS name from the local cable company, and the forth name, the owner of an X.509 PKI certificate. These four names could all represent the University of Michigan user alice and she could attempt to access NFS V4 files under any one of theses names, using one of many rpc security flavors.

To further complicate matters, the same name can have different meanings. For example, the second name could represent a Kerberos V5 principal, or a DNS realm name depending on the RPC credentials that accompany the request. Since there is no credential associated with a name in an ACL, there is no difference in file system access for different security contexts. What is desirable is to export a file system with multiple security flavors, and have ACL's control access. For example, a file system could be exported with both RPC_AUTH_GSS/Kerberos V5 and AUTH_UNIX security flavors, the intention being to allow AUTH_UNIX users read-only access, and Kerberos V5 users write access.

We have just begun our ACL and PKI security implementations, and anticipate lively discussions on these issues, perhaps resulting in protocol changes.

3 Compound RPC

NFS V4 defines a new RPC procedure, the compound RPC, which subsumes all previous NFS RPC procedures with the exception of the null procedure. The previous NFS RPC procedures make up the bulk of the compound rpc components, the idea being to reduce network traffic by combining what used to be independent RPC's into one compound RPC. NFS operations consume and excrete file handles. The NFS V4 server maintains a current file handle in global state that holds the input and output file handle parameter used by most NFS operations. Several new NFS operations manipulate the current file handle to set a starting point for processing compound operations, and return the result to the client. The operations in a compound rpc are processed from the top down, each operation using the filehandle output by the previous operation. Note that the compound rpc processing is not atomic, and that error handling stops at the first operation that returns an error.

One result of this change in the rpc procedure structure is that the rpc layer can no longer tell what operation is being requested, because all rpc's now have the procedure number of 1. The payload needs to be xdr'ed and parsed before the list of compound operations is exposed. One area affected by this layering change is the implementation of the replay cache. Coupled with the NFS V4's mandate of a strict serialized replay cache for lock mutating operations, the replay cache can no longer be implemented in the rpc dispatcher, but instead requires more processing (xdr decoding) before replay determination can occur.

Here is the mount compound rpc that obtains the pseudo file 2 system root file handle, and which obviates the need for mountd.

```
PUTROOTFH  -  Places the root of the
              Pseudo FS as the
              Current FH.
LOOKUP     -  Only used if the client is
              not mounting the Pseudo FS
              root.
GETATTR    -  Return the Current FH
      attributes
GETFH      -  Return the Current FH
```

4 Managing NFS V4 State

NFS V4 is explicitly stateful, in contrast with earlier versions of NFS, which are stateless in principle, but rely on an auxiliary stateful protocol (NLM) for file locking. Among other things, this means that file locking operations are now part of the NFS protocol proper, eliminating the need for separate rpc.statd and rpc.lockd daemons. In this section, we describe the NFS V4 state model in detail.

NFS V4 state is first instantiated via the SETCLIENTID operation, which the client uses to establish a 64-bit "clientid" for itself on the server. The clientid, and all other NFS V4 state is *lease-based*, meaning that it expires if the client performs no state–manipulating operations within the server's lease period, a timeout period chosen by the server. There is a special state–manipulating operation, RENEW, which has no effect beyond renewing the client's state on the server. Otherwise idle NFS V4 clients will emit periodic RENEW operations.

The clientid is the highest level of NFS V4 state; one clientid exists per client. The next highest is the NFS V4 *lockowner*; it is intended that one lockowner exist per distinct process on the client, although the client is free to define them according to its convenience. When a file is opened using the OPEN operation, the client specifies a lockowner; if this lockowner does not exist yet, the server will instantiate one. NFS V4 lockowners are always created in this way. The lockowner has three separate (we believe) meanings in NFS V4:

- It has the "obvious" meaning: the unit of contention for locks.
- It is the unit of serialization: OPEN, CLOSE, and LOCK operations (plus a few closely related operations) are serialized on a per–lockowner basis, meaning that before the client can send the (N+1)–th such operation for a given lockowner, it must wait for completion of the N–th.
- It is the unit of open file ownership: a file can be opened at most once by each lockowner.

The serialization of certain operations referred to above is needed to guarantee replay protection for state–establishing (non–idempotent) operations. A sequence number is associated with each lockowner, and an NFS V4 server caches the most recent non–idempotent operation on a per–lockowner basis, so that it can be replayed if a duplicate sequence number is received.

Finally, the lowest level of state is the NFS V4 *stateid.* Stateids are instantiated by OPEN and consumed by CLOSE; each stateid identifies an open file (much like a Unix file descriptor). Each stateid is associated to a particular lockowner, and to a particular file. Furthermore, at most one stateid can be associated to a particular (lockowner, file)–pair; in other words, an NFS V4 lockowner cannot open a file more than once. Stateids are required for file–manipulating operations, such as file locking, read and write, and file truncation.

In order to support the NFS V4 features of Win32 share semantics, lease based non-blocking byte range locks, file delegation, and the compound rpc, the NFS V4 client and server need to manage state at various scopes ranging from global to per file. When a file is delegated to a client, the client manages all state associated with the delegated file, including delegation to other processes, as well as share and byte range locking. Thus the client and the server maintain much of the same state.

4.1 Linux and NFS V4 Lockowners

A major client–side design decision is: how should the client allocate its NFS V4 lockowners? We are currently deliberating between two alternatives:

- 1. *One lockowner per pid.* The NFS V4 RFC specifically advocates mapping "a thread id, process id, or other unique value" to an NFS V4 lockowner. However, we have encountered unexpected implementation issues associated with using this type of mapping for Unix clients. The issues arise when trying to implement the correspondence between Unix file descriptors and NFS V4 stateids. First, multiple file descriptors might correspond to the same stateid, because a Unix process can open the same file under multiple descriptors, and at most one NFS V4 stateid can be associated to a given (lockowner, file)–pair. Second, multiple stateids might be associated to a given descriptor, because a Unix process might share the descriptor with a child process, and a separate stateid is required for each NFS V4 lockowner. Thus it is required to implement a many–to–many mapping between file descriptors and stateids. This mapping is difficult to implement correctly in all cases, particularly since it interacts with many other subsystems, for example, the mmap() subsystem. This unforeseen implementation complexity has led us to consider a second option:
- 2. *One lockowner per inode.* An alternative is to define a single lockowner for each inode. An OPEN is sent to the server when the first process opens the inode, and the matching CLOSE is sent when the last process closes it. This avoids the issues of choice 1, but there is a tradeoff: the client now has to assume part of the responsibility for detecting lock conflicts.

> The server does know which parts of the file have been locked by the client as a whole but only the client knows which of its locks correspond to distinct processes. So, if the client wants to lock a region, for example, it first must check whether any other processes have set conflicting locks, then send the LOCK request to the server, which will check whether and other *clients* have set conflicting locks.

Philosophically, we believe that this tradeoff in implementation complexity is the result of the NFS V4 lockowner being "overloaded" with two meanings: it are the unit of (a) lock contention, and (b) open file ownership. It is possible for our client to define its lockowners in a way which fits cleanly with either of these meanings, but not both. Defining per–pid lockowners (choice 1) will result in a good fit with (a) but not (b). Defining per–inode lockowners (choice 2) will result in a good fit with (b) but not (a).

In addition to the implementation complexity, there are also performance tradeoffs to be considered. Our work is not yet to the point where we can conduct realistic performance benchmarks, but our prediction is that file locking will be slower under choice 1 than choice 2, because of coarser serialization, whereas ordinary file I/O may be slightly faster because each inode never needs to be opened more than once.

As of this writing, our implementation uses per–pid lockowners (choice 1), but with some significant outstanding issues. We are considering switching to per–inode lockowners (choice 2), since we feel it would reduce the overall implementation complexity. Furthermore, it would have the advantage of confining the extra complexity to the locking subsystem only, where it can be more easily audited for bugs.

As a final note, we remark that the issues raised in this section have also been raised to the NFS V4 protocol designers, and it is quite possible that the protocol will be modified somehow in order to alleviate these issues for Unix clients. The issues do exist at the time of this writing, but they may be eliminated in the final version of the NFS V4 RFC.

4.2 Linux and NFS V4 OPEN

Another client-side difficulty that we have encountered is the implementation of open(). The NFS V4 OPEN operation has semantics similar to the POSIX open() system call, in the sense that:

- OPEN can be used to simultaneously create and open a file, with an atomicity guarantee.
- Files must be opened by name, i.e. the parameters to the OPEN are the name of the file being opened, and the filehandle of the containing directory. It is not possible to open a file by filehandle, at least at the time of this writing.

During NFS V4's design, it was intended that NFS V4 OPEN operations would correspond to POSIX open() calls in a roughly one–to–one fashion. In the Linux VFS, open() is implemented in three steps:

- 1. The VFS calls the filesystem's lookup() method, to look up the file by name, and instantiate an inode for it.
- 2. If the file does not already exist, and O_CREAT was specified in the open(), the filesystem's create() method is called to create it.
- 3. The filesystem's open() method is called to open the file.

Unfortunately, in a network filesystem, this implementation is prone to several race conditions against other clients. For example, after the lookup() is done in step 1, another client could delete the file before the open() in step 3 (or worse yet, replace it with a different file). Protecting against all race conditions of this type is quite difficult. This issue does not arise for local filesystems because the VFS acquires semaphores which "lock out" other processes and effectively make the combination of (1)–(3) atomic. However, in NFS V4, it is not possible to "lock out" other *clients* in this way, and we are presented with an inherent race condition.

Our point of view is that the root of this problem is that there is no way for a Linux file system to process the entire open() system call at once; instead the VFS splits the open() into three pieces in separate filesystem methods. We have created an experimental solution by defining a new filesystem method, called lookup_open(), which receives all the parameters to the open() system call. The NFS V4 client's lookup_open() method simply bundles these parameters into a single NFS V4 OPEN and sends it to the server. If a filesystem chooses not to define the lookup_open() method, the VFS will fall back on the default behavior of splitting the open() into the three steps above. In this way, backward compatibility is retained, while solving the race condition in NFS V4. We emphasize that this is only an tentative, experimental solution, and will probably change substantially. We do feel that some modification of the Linux VFS is needed to implement open() correctly in NFS V4, but we do not yet have a definitive idea of what the "right" change is.

4.3 Linux and NFS V4 Locks

NFS V4 defines a LOCK operation, which is designed to implement byte–range file locking. This operation is part of the NFS V4 protocol proper, in contrast with NFSv3, which uses the auxiliary protocol NLM (and its associated daemon, lockd) for file locking. One important difference between locking in NFS V4 and locking in NLM is the lack of dependence on a callback mechanism. When an NLM client waits for a blocking lock, and the lock becomes available, the server sends an RPC to a "callback" service on the client, to inform the client that the lock was granted. In contrast, an NFS V4 client must poll the server, repeatedly requesting the lock until it is available. This design decision, although disadvantageous from a performance standpoint, was made because it only requires one–way reachability between client and server, and will always work correctly in the presence of intervening firewalls, NAT boxes, etc.

Byte–range locking in NFS V4 was intended to operate in a heterogeneous Unix/Win32 environment. However, one significant difference between byte–range locking on these two platforms is the treatment of subranges. In a POSIX lock request, arbitrary subranges are allowed. For example, bytes [0,10] of a file could be locked with one request, and bytes [3,6] unlocked in a subsequent request, leaving the ranges [0,2] and [7,10] locked. In contrast, Win32 platforms enforce a strict one–to–one correspondence between byte ranges, so such a locking request would be disallowed. NFS V4 tries to accommodate both types of server environments by allowing servers to enforce a one–to–one correspondence between locking ranges if necessary. However, servers are encouraged to support more general locking requests if feasible.

This presents a problem for Unix clients, since they cannot assume that the server can process subrange locking requests. Consider the example above, where a request is made to unlock bytes [3,6]. A Unix NFS V4 client would have to simulate this request by factoring it into three separate requests to the server:

- 1. A request to unlock the entire byte range [0,10].
- 2. A request to lock bytes [0,2].
- 3. A request to lock bytes [7,10].

Naturally, this presents a race condition if, for example, another client sets a lock on bytes [0,2] between steps 1 and 2! In this event, unlocking the range [3,6] would result in the "loss" of the lock on the range [0,2], which probably spells disaster for the client.

We have tried to tame this race condition as much as possible by implementing two types of locking in our client: POSIX locking and "strict locking". If POSIX locking is enabled, our client will assume that the server is capable of processing arbitrary subrange requests (which would be the case for our Linux server or a typical Unix server). If "strict locking" is enabled, our client will assume that the server supports only the minimum subrange requests mandated by NFS V4 (which would be the case for a typical Win32 server), and factor its locking requests appropriately. In the event that a lock is lost due to the race condition, the client will log a panic message in the syslog. The two types of locking are currently selectable as a mount option, so that the system administrator can determine whether the server is a Unix or Win32 server, and make the selection accordingly. However, it is our understanding that the protocol will soon be changed so that the locking type can be autonegotiated between client and server at mount time, making it transparent to the user.

Note that this "lost lock" race condition can only occur if the client actually makes locking requests for proper subranges, which seems to happen infrequently in practice. Furthermore, it can only occur if the client is a Unix client, and the server is a Win32 server, which is not a typical configuration. Nevertheless, users of NFS V4 should be aware that *there is an inherent race condition in file locking between Unix clients and Win32 servers.*

5 Delegation

File delegation is a feature of NFS V4 which is intended to enhance performance by providing clients with cache consistency guarantees, obviating the need to send frequent revalidation RPC's to the sever.

Whenever a client opens a file, the server can optionally return either a *read delegation* or a *write delegation* for the file to the client, as part of the OPEN response. A read delegation is a guarantee to the client that no other clients have opened the file for writing; thus the client is free to cache the file aggressively (in a read-only fashion) without need for periodic cache consistency checking. A write delegation is a guarantee that no other clients have opened the file at all; thus the client is free to defer and gather writes, before sending them to the server. The policy for granting delegations is implemented by the server; clients have no ability to request or refuse delegations. However, a client can return a delegation at its choosing, so even though a delegation cannot be refused, it can be returned immediately in a separate RPC.

The cache consistency guarantees provided by delegation are similar to those provided by file locking, but delegation differs from locking in several important respects. First, a file lock is granted to a particular process on a particular client, whereas a file delegation is granted to the client as a whole. Second, file locks are requested by clients, if exclusive access is needed for correctness reasons, whereas delegations cannot be requested by clients; the server assigns them to clients in an attempt to increase performance. Third, a file lock can be maintained indefinitely; delegations can be recalled by the server at any time, as we now explain.

After the server grants a delegation, it may receive an OPEN request from a different client which would conflict with the delegation. In this event, the server employs a callback mechanism to recall the delegation, before responding to the OPEN. Therefore, before the server will assign delegations to a client, it must verify that the callback service is available which is done when the client does its SETCLIENTID. If the callback service is unreachable (e.g., because of intervening NAT boxes or firewalls), the client will be unable to participate in file delegation (which is strictly a performance-enhancing feature), although all other features of NFS V4 will still be available. In particular, file locking will still work, since it is implemented using a polling scheme rather than a callback mechanism. A major design goal in NFS V4 was that it should not rely on the presence of a callback service. This is in contrast with NFSv3's lockd, which depends on callbacks.

In addition to allowing aggressive data caching, delegations can improve performance by allowing a client to perform certain file operations locally, without sending an RPC to the server. This is particularly true in the presence of a write delegation, which ensures that the client has exclusive access to the file. In this case, the client can "localize" almost all file operations, including utime() requests, file opens/closes, reads/writes, and file locking.

At the time of this writing, our implementation of file delegation is in a state of active development. The basic mechanism of exchanging delegations between the client and server is implemented, including the callback service, but the optimizations available to the client in the presence of a delegation are not. We have yet to modify the client's data caching policy, and localize certain file operations, when a delegation is held. So far, we have not encounted any serious issues implementing file delegations in Linux.

Finally, we note that NFS V4 delegations exist only for regular files; it is not possible to associate a delegation with a directory. As a future research project, we speculate that directory delegation may offer performance improvements similar to those offered by file delegation. By analogy with file delegation, a read delegation for a directory would guarantee to a client that no other clients may change the directory contents while the delegation is held. A write delegation would guarantee that no other clients may read or change the directory contents while the delegation is held. If successful, directory delegation could be incorporated into a future version of NFS V4 under the NFS V4 "minor versioning" mechanism.

6 RPCSEC_GSS Implementation

NFS V4 requires the RPCSEC_GSS protocol for strong security. The use of the RPCSEC_GSS security flavor in NFS V4 differs from it's use in previous versions of NFS in several ways.

- *Mandated Use*: There are two mandatory GSS-API mechanisms, Kerberos V5 [RFC1964] and LIPKEY [RFC2847] which is layered on SPKM-3 [RFC2847]. There is a per mechanism ordered list of integrity, confidentiality, and key distribution algorithms that must be implemented.
- *Machine certificates*: The SETCLIENTID negotiation should be done over an rpc channel secured by either Kerberos V5 or LIPKEY mutual authentication using machine credentials.
- *Multiple mechanisms per export* NFS V4 allows the security flavor and triple to be different on a per file basis.
- *Security Negotiation* The SECINFO operation is an automated means of negotiating security as the file system is explored by the user.

While Kerberos is a well known and widely implemented security product, SPKM versions and LIPKEY exist only as RFC's. The SPKM3 security mechanism is based heavily on the SPKM1 and SPKM2 [RFC2025] mechanisms. SPKM3's differs from the previous SPKM versions mainly in that there is no requirement for initiator X509 credentials. In this case, the initiator sends an unauthenticated create context messages to the target, and the target returns its credentials to the initiator. Thus, at its lowest QOP level, SPKM3 is open to a man-in-the-middle attack where the attacker poses as the target to the initator and as the initiator to the target. Upon receiving the target X.509 credentials, the initiator checks the target credential signature, looking for a common signer. Thus in order for the man-in-the-middle attack to work, the attacker needs to have its fake target credentials signed by a member of the initiators certificate chain. SPKM3 does allow for the use of initiator credentials, which prevents the man-in-the-middle attack.

The required LIPKEY security mechanism is layered on top of SPKM3. LIPKEY encrypts a user's login name and password in the session key negotiated by SPKM3, providing a means of initiator identity that requires no credentials. Since NFS V4 clients are required to have machine credentials [RFC3010], we can imagine using the machine credentials as initiator credentials in the SPKM3 layer, and then using the LIPKEY user name and password to identify the user.

6.1 User Level Implementation

Our user level implementation is based on the MIT Kerberos V5 gssrpc [RFC2203] which was incomplete, and needed many changes in order to meet the RPSEC_GSS protocol specfication. The MIT Kerberos V5 GSS-API implementation is quite complete, and we could just use it. We have enabled the GSS-API mechanism glue layer provided in the MIT Kerberos V5 source which presents the GSS-API layer to the ONC RPC, and switches on the mechanism type. The combination of GSS-API and ONC-RPC is complicated by the need to merge similar functionality such as transport independence and the sequencing of portions of the transport steam. We have yet to completely merge the GSS-API concept of multiple channels into the ONC-RPC, but other than that, our user-level RPCSEC_GSS Kerberos 5 mechanism implementation is mostly complete.

There is no previous SPKM implementation to base our work on, so we started from scratch, and as of this writing, the SPKM3 implementation is incomplete. SPKM is designed to to be compatible with the X.509 [X.509] public key infrastructure, which naturally leads to the use of OpenSSL libraries for our open source user-level implementation. The required ASN1 [ISO8825] encoding and decoding turned out to be the main bottleneck to progress. We are using the Valicert *asn1parse* compiler [asn1parse] to produce the ASN1 DER encode and decode functionality which links against SSLeay-0.9b libraries. [1] The Valicert asn1parser was designed to handle all the ASN1 syntax needed for the SSLeay crytpo code, and SPKM3 requires ASN1 syntax not currently handled by the parser. For this reason, the SPKM3 ANS1 encoding/decoding of the InitialContextTo-

[1]Valicert is soon to come out with an OpenSSL compiler which we will switch to.

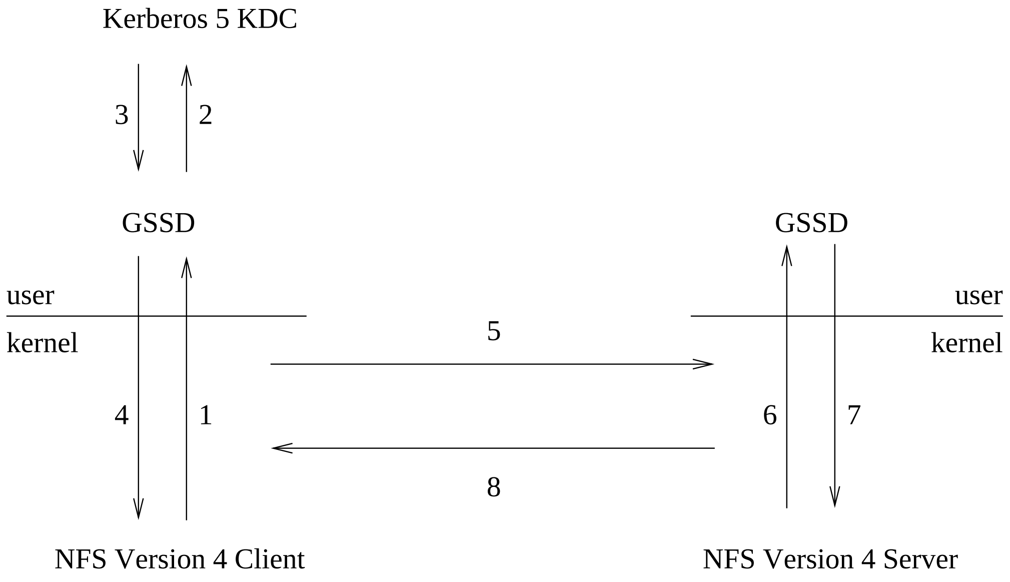

Figure 1: Kernel RPCSEC_GSS Kerberos V5 context creation. 1,4,6,7 GSSD RPC interface calls. 3,2 Kerberos V5 mutual authenticated KDC calls. 5,8 RPCSEC_GSS Null procedure calls.

ken is done with routines from the Kerberos V5 gssapi libraries with the payload encoding/decoding done by asn1parser generated routines.

6.2 Kernel Implementation

A main feature of the RPCSEC_GSS protocol is the ability to add security mechanism that adhere to the GSS-API specification without changes to the ONC RPC layer, yet the available open source security code such as Kerberos V5 and OpenSSL is not kernel safe. For this reason, we have left a good deal of the security functionality in a user space daemon, GSSD, which communicates with the kernel NFS V4 via an rpc interface. GSSD communication requires the loopback source address and a reserved port, with AUTH_UNIX root credentials. GSSD's main job is gss_context creation, and services such GSS-API calls as *gss_init_sec_context()*, and *gss_accept_sec_context()*. GSS-API calls that involve encryptions such as *gss_wrap(), gss_unwrap(), gss_verifymic()* and *gss_getmic()* are serviced in the kernel.

The addition of a user level daemon adds complexity into the already complex gss_context creation dance. Figure 1 illustrates the RPC communication needed to establish a Kerberos V5 GSS context. Note that Kerberos V5 user credentials need to be in place prior to these steps. The first two steps show the call through the GSSD upcall interface to the Kerberos V5 data base requesting a Kerberos v5 service ticket for the NFS V4 server. The requested service ticket is delivered to GSSD in step 3 where the client side of the service ticket is decrypted and passed to the NFS V4 client along with the server portion of the service ticket in step 4. In step 5, an RPCSEC_GSS overloaded Null procedure carries the server portion of the service ticket to the NFS V4 server, which forwards it to GSSD in step 6 to be decrypted. The results are passed back to the NFS V4 server in step 7, and a status is returned to the NFS V4 client in step 8.

0ur current design of using the GSSD upcall interface for context creation allows us to use our user level mechanism switching and spkm3 implementation. We are in the process of developing a kernel credential cache.

7 Access Control Lists

Our ACL implementation is based on the EA/ACL patch for Linux [ACL] , which is still under development, but in our experience has been quite stable and complete. We have only started implementing ACL's recently, and our implementation is still in an early stage, so we simply outline the roadmap for our work here.

The Linux ACL path implements Posix ACL's, as described in [posix1003] However, NFS V4 has its own ACL model, which is much more rich and finely granulated than the Posix model. To bridge the gap, we have identified a subset of the NFS V4 ACL model which is functionally equivalent to the Posix model. For our initial ACL implementation, our client and server will always reject ACL's which are not in this subset. In this way, we hope to make Posix ACL's work transparently in a homogeneous, purely Linux environment, leaving open the possibility that they will not work correctly in the presence of other platforms.

Eventually, we would like to see support for the full range of NFS V4 ACL's. The difficulty, however, is ensuring that the additional, non-Posix ACL's are enforced by the server's filesystem itself. Implementing NFS V4 ACL's would be fairly straightforward if we were free to enforce the ACL's in the NFS V4 server code, but in our opinion, this is dangerous, since by accessing the files outside of NFS V4 (for example, through ftp or local access on the file server itself), the ACL's could be bypassed. We believe that to avoid potentially dangerous security–related surprises, an implementation of NFS V4 ACL's in the fs layer itself is called for, but it is unclear whether this will be feasible within the current scope of our project.

8 Administration

Although NFS V4 shares many features of previous NFS versions, there are distinct differences in features such as namespace and security that effect administrative tasks. In this section, we describe these differences.

8.1 Server Administration

As in previous versions of NFS, server options are expressed in the /etc/exports file. One of the first tasks in server administration is to decide what security infrastructure will be used to protect exported subtrees. Currently, the the Linux server Pseudo filesystem is exported as a read only file system with the AUTH_UNIX security flavor. The Pseudo file system name space is constructed by reading entries in the /etc/exports file. We follow Sun Microsystem's example and allow selection of security options on a per export basis expressing security option choices via an /etc/exports file syntax developed by Sun. Exported file systems no longer need to express access in terms of client IP addresses,

Every entry in the /etc/exports file looks like this.

```
(pseudo path) (export path) [ro] [sec=[:]]
```

- *Pseudo path* is the path of the pseudo system. This is the file system that the client will see. You can make any hierarchal virtual file system you want with these pseudo directories. The last directory is the mount point where the exported directory will be mounted.
- *Export path* is the path to the actual directory that you will be exporting.
- *Mount Options: ro* is to make the exported directory read-only. This is optional.
- *sec=option[:option]* lists the security option(s) required by the exported file system.

Here's a list of valid security options Note: currently, only krb5 is implemented.

1. none	(AUTH_NONE)
2. sys	(AUTH_SYS)
3. dh	(AUTH_DES, old diffe-hellman)
4. krb5	(RPCSEC_GSS Krb5 authentication)
5. krb5i	(RPCSEC_GSS Krb5 integrity)
6. krb5p	(RPCSEC_GSS Krb5 protection)
7. spkm3	(SPKM3 authentication)
8. spkm3i	(SPKM3 integrity)
9. spkm3p	(SPKM3 protection)
10. lkey	(LIPKEY authentication)
11. lkeyi	(LIPKEY integrity)
12. lkeyp	(LIPKEY protection)

An NFS V4 /etc/exports file looks like this:

```
/foo          /export
/goo/dir1     /usr/local   ro
/goo/dir2     /usr/share   ro   sec=dh:lkeyi
/goo/dir3     /usr/man          sec=sys
/goo/dir4     /usr/doc          sec=krb5:spkm3
# a comment
```

The server needs an identity and accompanying credentials in both the Kerberos V5 and LIPKEY (PKI) security realms in order to support the required RPCSEC_GSS security mechanisms.

We are developing a NFS V4 server Tcl/Tk administrative tool that communicates with the NFS V4 server via the NFS V4 *ioctl* interface. Using the tool, administrators will be able to start and stop server threads, edit the /etc/exports interface, list information about attached clients including users, lockowners and open state, and reap state such as deadlocked locks. The tool will also supply a message interface to NFSV4 debug messages supplying a window front end to the existing sysctl debug granularity settings.

We recommend running the NFS V4 server as a file server only, not permitting local access. NFS V4 has features such as Kerberos V5 and LIPKEY security mechanisms not supported by the local file system, which makes local access problematic.

8.2 Client Administration

As was mentioned in Section 6, it's recommended that the establishment of a clientid via the SETCLIENTID negotiation be done under a secure mutual authentication. We currently require the use of client machine credentials for SETCLLIENTID and callback communication.

We recommend that clients always mount the root of a servers Pseudo filesystem, allowing servers to change their exports under the Pseudo filesystem. The automounter is no longer needed. Currently, there is no uid and gid mapping service, nor security realm principal name mapping service, so attention needs to be given to matching local and remote uid and gid's.

8.3 Issues in Migrating from NFS version 3

A goal of previous versioning of NFS was backward compatibility. Not so with NFS V4. This protocol is such a departure from previous versions there is no common client nor server. Exactly how to migrate data from NFS V3 to NFS V4 is unknown. One hopeful migration strategy is to configure an NFS V4 server such that the same data could also be exported from an NFS V3 server. This might be possible under certain conditions. NFS V4 locking is very similar to NLM mechanisms. The NFS V4 server could export using security mechanisms common with NFS V4 such as AUTH_UNIX. The NFS V4 sever could use UNIX file permissions instead of ACL's, etc. If this is possible, then this would allow clients to slowly switch from an NFS V3 base to an NFS V4 base.

9 Conclusion

The NFS V4 protocol is still in a developmental stage. There are very active commercial ports in progress, including Solaris, Network Appliance Filer, Windows98/NT/2000, and other ports planned. We are hopeful that as the protocol gains momentum, our work can be included into future Linux kernel releases.

10 Acknowledgments

We would like to thank Sun Microsystems for sponsoring CITI's NFS Version 4 Open Source Reference Implementation Project. We also acknowledge Dug Song, Jake Moilanen, and Weston Adamson for their work on the project.

11 Availability

Our source, documentation, and performance results are available at

`http://www.citi.umich.edu/projects/nfsv4/`

References

[RFC1813] B. Callaghan et al., *NFS Version 3 Protocol Specification*, RFC 1813(June 1995)

[RFC3010] S. Shepler et al., *NFS version 4 Protocol*, RFC 3010(December 2000)

[RFC1094] Sun Microsystems, Inc.,*NFS: Network File System Protocol Specification*, RFC 1094(March 1989)

[RFC2279] F. Yergeau, UTF-8, a transformation format of ISO 10646, RFC 2279, (January 1998)

[RFC1964] J. Linn et al., The Kerberos Version 5 GSS-API Mechanism, RFC 1964 (June 1996)

[RFC2847] M. Eisler, LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM, RFC 2025 (June 2000)

[RFC2025] C. Adams, The Simple Public-Key GSS-API Mechanism (SPKM), RFC 2025 (October 1967)

[RFC2203] M. Eisler et al., RPCSEC_GSS Protocol Specification, RFC 2203 (September 1997)

[RFC2459] R, Yousley et al., Internet X.509 Public Key Infrastructure Certificate and CRL Profile, RFC 2459 (January 1999)

[ISO8825] M. Rose et al., Information Technology - Open Systems Interconnection Specificaton of Basic Encoding Rules fro Abstract Syntax Notation One, ISO/IEC (1990)

[asn1parse] Valicert ASN1 Parser `http://www.valicert.com/developers/download.html`, last referenced June 2001.

[ACL] Extended Attribute and Access Control Lists for Linux `http://acl.bestbits.at`, last referenced June 2001.

[posix1003] IEEE, IEEE Draft P1003.1e Draft Standard 17, (March 10,1999)

[Khanna] R. Khanna, *Distributed Computing Implementation and Strategies*, PTR Prentice Hall (1994).