Glitz: Cross-Vendor Federated File Systems

Daniel Ellard
BBN Technologies*

Craig Everhart
NetApp

Theresa Raj
NetApp

ABSTRACT

We propose Glitz, a system to integrate multiple file server federation regimes. NFS version 4 is a significant advance over prior versions of NFS, in particular specifying how NFS clients can navigate a large, multi-server namespace whose constituent parts may be replicated or moved while in use, as specified by NFS servers. This capability is essentially the same as that of previous distributed file systems such as AFS [7]. Sophisticated as this NFS capability is, it does not address the larger problem of building a usable system atop this basic capability. Multiple single-architecture solutions have been proposed, but each of these is based on an architecture for server federation that does not easily admit other members [3, 16, 18]. Glitz allows those multiple-server federations to interoperate and collaborate in a vendor-independent fashion. We give a history of file system federation efforts as well as a detailed tour of Glitz and its benefits for vendors.

Categories and Subject Descriptors

D.4.3 [**Software**]: Operating Systems—*Distributed file systems*

Keywords

NetApp, NFS, AFS, federation

1. INTRODUCTION

With NFS version 4, NFS allowed servers to provide an arbitrarily-large namespace constructed of separate file systems, potentially on separate file servers, that all NFS clients would render in a uniform way. Moreover, these separate file systems could be replicated onto multiple servers while retaining their place in the distributed namespace. They could also be migrated from server to server while still in use, again while retaining their place in the distributed namespace. In short, the NFS client-server protocol now specified how NFS servers could project this advanced distributed file system function to their clients. Using NFS version 4, if a set of NFS servers had an inter-server namespace to project, and/or could replicate those file systems and/or migrate them from server to server, the NFS protocol was now strong enough so that any NFS client anywhere would be able to take advantage of the functions provided by that set of NFS servers.

Several mutually-incompatible schemes arose for management of multiple servers. The IETF made initial efforts at "server to server protocols" that have since stalled. Zhang and Honeyman proposed a scheme very close to that of AFS for the management of a namespace [18]. IBM, in its Glamour project, developed a scheme for projecting a single namespace and replicating and migrating the content of file systems [16]. NetApp® Data ONTAP® operating in cluster mode provided an analogous scheme within a single cluster and sought to expose details of the scheme across the public NFS protocols [3]. All of these efforts have resulted in a Federated File Systems open community to bring this multi-scheme interoperation discussion into the public for wide participation.

The remainder of this paper proceeds as follows. In Section 2 we provide a short history on federated namespaces leading up to the current state of affairs. Following that, in Section 3 we provide an overview of the Glitz protocol. Finally we conclude in Section 4.

2. FEDERATED NAMESPACE HISTORY

Individual networked file systems have existed for many years, most broadly using SMB on Windows platforms and NFS everywhere else. As soon as a file server for either of these protocols is outgrown, and the server cannot be expanded, file server administrators confront the problem of integrating additional resources. This is done most simply by adding a file server and having the file server clients connect to the additional server, usually at a distinct point in the clients' namespace. While it is easy to purchase new file servers, updating many clients to connect to the additional servers is not a scalable solution. Several schemes have arisen for automating this process, some driven from the client side and some from the server.

In NFS (versions 2 and 3), file systems on the server side are treated independently [1, 15]. With NFS versions 2 and 3, clients are responsible for mounting whichever file systems from whichever servers they want, constructing their own local client-side names-

*Work performed while a NetApp employee

paces out of the collection of exported file systems available from available NFS servers. Part of the reason for treating each exported file system independently is that some aspects of NFS access control are controlled by per-file-system export options, so it is important that access to a parent file system not inadvertently grant access to a child file system.

With NFS versions 2 and 3, the chores of managing the local namespaces of collections of clients, particularly in the face of a growing pool of NFS servers, have been tedious and error-prone. Generally, different automounters have evolved for helping clients manage their NFS server mounts [2, 14]. These tools allow site administrators to control how an NFS client will mount file systems from NFS servers; in some incarnations, they guide how different NFS clients can mount independent collections of file systems. These schemes use a *mount map* for directing a client to mount certain remote file systems. The simplest form uses a file local to the client as the source of mount map information; more complex schemes communicate this information over the network, as a network-accessible NIS map or an LDAP map. Different vendors have implemented different versions of such schemes in their NFS clients, which complicates client management in multi-vendor installations.

Windows clients and servers have traditionally used the Server Message Block (SMB) protocol for remote file access, rather than the NFS protocol. Like NFS versions 2 and 3 clients, early SMB client-server installations allowed clients to independently choose the server file systems that were to be mounted and accessed remotely, requiring each SMB client to be configured and managed independently when the set of servers changed. In 1996, Microsoft released SMB changes designed to simplify this management, allowing one SMB server to redirect SMB clients to another SMB server's export point with the MS-Dfs facility [9]. Windows NT 4.0 servers could export a Dfs root volume that clients would need to mount individually, but which could then link to file services on other servers. The administrator for the Dfs root machine could create a junction point in the exported file namespace that redirected SMB clients to a point within another file server. This technology has grown with further Windows releases, but they share the property that an object in the file namespace of one server directs clients to continue interpreting file path names at another server. This technology dramatically simplifies how clients are managed: instead of each client having to be configured to be aware of server namespace changes, those namespace changes happen by the administrator's operation on a parent file server, and clients learn the new configuration directly from the server.

The AFS distributed file system initially projected a single namespace for a site, then expanded its namespace to address sites around the world [7, 17]. As it progressed beyond experimental status, it used a scheme that had been popularized by NFS: the division of the namespace into disjoint connected sub-trees (like separate server-side file systems), connecting the sub-trees with mounts. Unlike NFS's client-side mounts, in which any client could mount any file system anywhere, AFS used server-directed mounts. The client mounted the site's root file system (called a volume in AFS), and then other AFS volumes were logically mounted on specific leaf objects in that root volume. Volumes could be located on any AFS server, and read-only instances of them could be replicated for better availability and load balancing. The logical AFS volume mounts were specified by the AFS servers; an AFS client would not make decisions about which volume went where.

This server-specified AFS volume mounting worked as follows. AFS volumes were independent file systems with a root directory. They were created and given a unique alphabetic name by an administrative command. When creating an AFS volume, an entry was made in the AFS site's Volume Location DataBase (VLDB), a replicated database accessible over a network access protocol. The entry for the volume contained the volume name and a list of its locations; the database was indexed by volume name. If that volume was to be mounted as a subdirectory of a parent volume, the administrator would issue another administrative command to create a mount point, which was an object in a directory in the parent volume that contained the name of the volume to be mounted. The AFS client, encountering this mount point object, would consult the site's VLDB to determine where the servers were that held find an instance of the (possibly replicated) volume; it would then contact one of those servers to learn about the contents of that AFS volume and continue navigating a path name.

An example will help illustrate this mechanism. Suppose that the AFS root volume `root.afs` contains the path `/user`. The AFS client at this site would conventionally mount the `root.afs` volume on the local directory `/cmu`, so the paths `/cmu` and `/cmu/user` were accessible from that AFS client machine. Suppose the AFS administrator creates the AFS volume `user.cfe` on server machine server2. This creates the given AFS volume as an empty directory and creates a VLDB entry with a volume name of `user.cfe` and a location of server2. The administrator then can create a mount point for this volume from an AFS client, directing the AFS volume `user.cfe` to be mounted at a client-visible path name; suppose that this path is given as `/cmu/user/cfe`. The mount point is a special object in the `root.afs` volume, created as a new object named `cfe` in the `/user` directory within the `root.afs` volume, with data naming the `user.cfe` volume. When any AFS client at the site then explores the name `/cmu/user/cfe`, it will navigate the `root.afs` volume to reach the `/cmu/user/cfe` object, observing that it is to be interpreted as an AFS mount point object. It will then learn the volume name `user.cfe` from the mount point object, consult the site's VLDB to learn where the volume is located, and then it will consult server2 to determine the actual content of the `user.cfe` volume, which at this point is still an empty directory.

One might imagine that the VLDB concept is extraneous to the AFS design, and that an AFS mount point might simply name the AFS volume and its server. But this would be to remove an important piece of functionality. As already mentioned, read-only AFS volumes can be replicated; they can also be moved from one server to another, even while they are in use. Generally, volumes with many mount points are replicated to enhance the availability of the sub-volumes. If one of these sub-volumes is moved from one server to another, the published location of that sub-volume must be updated. If the location were to be given in the AFS mount point itself, then all the AFS mount points in each of the replicas of the parent volume would need to be updated. But with the VLDB architecture, only the single location of the moved AFS volume within the VLDB needs to be updated. Similar considerations apply if the sub-volume itself is replicated: rather than having to add the new replica location to its mount point in all replicas of its parent volume, we can add the new replica location to the volume's entry in the VLDB.

The previous example illustrated an early form of the AFS site-wide namespace. Later forms could encompass multiple sites, potentially scattered across the world. This form was based on the concept of the *cellular mount point*, in which the AFS mount point contained not only the name of the AFS volume to be mounted but also the name of the AFS cell, or installation, in which the target AFS volume existed. This construction relied on two configuration files stored locally on each client: the `CellServDB` file, containing mappings from cell names to the addresses of their VLDB servers, and the `ThisCell` file, naming the local cell, whose `root.afs` volume was initially to be mounted, conventionally on the workstation directory `/afs`. In general, if `foreign.org` was the name of an AFS cell other than the local one, `/afs/foreign.org` was housed in the local site's `root.afs` volume as a cellular mount point for the AFS volume `root.cell` in the cell `foreign.org`. The client encountering this name would then consult its `CellServDB` data to learn the VLDB servers for `foreign.org`, then consult those servers for the location of its `root.cell` volume, finally mounting that volume's root directory in the client path `/afs/foreign.org`. Thus, AFS achieved a multi-organizational namespace, but it relied on each organization having a special mount point in every other organization's `root.afs` AFS volume, as well as every AFS client machine having an entry describing the location of each organization's VLDB servers in its `CellServDB` file.

DCE/DFS reused the AFS inter-volume single-site scheme, in which the AFS volume was renamed to be the DCE/DFS fileset, and the AFS VLDB renamed as the DCE/DFS Fileset Location DataBase, or FLDB [8]. The global namespace was provided by a different scheme, cellular mount points were eliminated. Instead, the root directory was conventionally named `...`, and names under that directory were interpreted as DNS domain names or X.500 names. In either case, the given organization registry was consulted to determine the location of a DCE name server for the named organization, which would then contain a referral object directing a DCE/DFS client to the FLDB server addresses for that organization. The DCE/DFS client looked up the reserved fileset name `root.dfs` and then proceeded as with AFS.

NetApp Data ONTAP operating in Cluster-Mode uses a similar inter-volume structure within a single cluster [3]. It retains the AFS terminology of volume and VLDB but not the implementation. The volumes reside on D-blades or on combinations of them. The inter-volume link is called a junction, but it is interpreted by the N-blades, which act as proxies for remote clients, redirecting requests to the appropriate D-blades. The junction does not contain a key for the VLDB; rather its file identity itself serves as a key. Because NetApp Data ONTAP operating in Cluster-Mode serves protocols such as NFS versions 2 and 3, for which clients cannot accept referrals, the N-blade serves as a proxy for all requests, even for remote D-blades. NFS version 4 allows some of this function to be performed by advanced NFS clients.

The IETF's NFS 4.0, first specified in 1999, introduced into NFS the possibility that an NFS file server could support a multi-server namespace, intending at first to support only Replication and Migration [12]. An NFS 4.0 server can return an error code for a name lookup operation and can direct, with the `fs_location` attribute of the looked-up name, the client to continue its pathname evaluation with a different file system on another server. This aspect of the protocol, initially crafted to support file system migration, can also describe any inter-file system reference. Any protocol that would support such a mechanism could be leveraged to make it appear that an inter-server file system reference has resulted from a prior file system migration. This capability has been enhanced in the new NFS 4.1 draft (currently an IETF draft).

NFS 4, being a network protocol specification, does not dictate how a parent file server is to know what `fs_location` attributes should be returned to a client. A naive implementation, for example, might store the actual file system location (e.g., server name and path) as data in a parent file system. As noted above, though, such a scheme would work for creating a collection of file systems that refer to each other, but it would complicate file system migration, particularly if file system replication were supported as well. IBM Research's Glamour project defined a scheme atop NFS 4 that used database functionality comparable to AFS's VLDB as an indirection mechanism: the NFS file server itself observed server-side objects that contained a database key, which the file server looks up in the installation's database via the LDAP protocol. The database entry contains server addresses and parameters, which are then returned to the NFS 4 client as `fs_location` attributes.

Glamour initially anticipated that other file servers, acting as NFS 4 servers but not with Glamour extensions, could be referenced in the Glamour-projected namespace [16]. Essentially, an entry could be made in the Glamour fileset location database that referred to a file system on a non-Glamour file server: all the entry would need to contain is the file server's name or IP address and the name of the server's file system. The `fs_location` attributes would cause an NFS 4 client to mount the non-Glamour file system in the Glamour-exported distributed file space. Using this technique, a Glamour federation could be said to span both Glamour and non-Glamour file servers, albeit in an asymmetric way. All file systems in the distributed namespace need to be present in the Glamour fileset location database. Such a system is more brittle than an all-Glamour scheme. If the location of a Glamour fileset (a fileset, or NFS file system, hosted on a Glamour file server) were to change, the Glamour fileset location database would be updated; if, for example, a fileset replica were created, its location could be recorded in the database as an alternative for the NFS client. But a non-Glamour fileset might be moved to a server with a new network address or to a different mount point within the non-Glamour server, and it would require a special effort to update the fileset's location in the Glamour database. Clearly, managing Glamour resources was simpler than managing non-Glamour resources under this scheme.

These multiple schemes for managing multiple file servers were quickly balkanizing a territory that customers would prefer to stay open. Administrators in the business of building a multi-server federation do not want to buy it all from one vendor and then be stuck with that vendor forever. They would much prefer to be able to buy parts from different vendors, perhaps at different times, and to use those parts in a single coordinated multi-server file system for their organization. The goal of the open Federated File Systems project and protocols is to allow them to do so. Multiple vendor multi-server architectures can coordinate with each other through the Fed-FS protocols and provide a common service.

3. THE GLITZ PROTOCOL

Glitz is the name of the NetApp project originally devised as a means of reconciling the IBM Glamour file system federation scheme and NetApp Data ONTAP operating in Cluster-Mode. After initial joint work between NetApp and IBM toward that end, we have gone public with an open protocol-development effort called Federated File Systems, but in this paper the result is referred to as Glitz. The open discussion group is intended to allow servers and server collaboration schemes from all vendors to interoperate. It was announced on the IETF's NFSv4 working group mailing list and at a Birds-of-a-Feather session at the 2007 USENIX FAST conference.

3.1 Glitz Protocol Requirements

The first phase of the Glitz project was to organize a cross-vendor group of people interested in federated file systems based on NFSv4 and work with that group to define a set of requirements for a federated file system protocol.

The first task was to agree on some basic assumptions and terminology. A fundamental step was settling on a definition of federation and agreeing that it is essential to the success of the protocol. What we mean by federation is that different parts of the system may be administered independently, with no centralized authority (beyond what is already necessary to provide shared infrastructure services, such as DNS and IP routing). Joining a Glitz federation does not require giving up control of one's servers or other resources—it requires only adherence to the protocol.

A corollary of the principle of federation is that the owner or administrator of a resource (such as an NFS server) retains control over that resource. For example, the information about where the volumes owned by a specific administrator are located is controlled by that administrator. The administrator of a cluster of servers can migrate a volume from one server in the cluster to another without the cooperation (or even acquiescence) of any other administrator, and clients accessing that volume will discover that the volume has been migrated without any assistance from their administrators.

A federation is composed of federation members. Functionally, each federation member consists of a set of services and the principles of the administrators of those services. In more concrete terms, a federation member usually consists of a set of servers and the system administrators of those servers.

3.2 Requirements Overview

The most important requirement—both in terms of the viability of the protocol and the impact it has on the design—is that it must be possible to use the Glitz protocols without any modification to clients [10]. Glitz must require no changes to the NFSv4 protocol. A goal related to this requirement is that the protocols must entail only minor modifications to the servers.

The second key requirement is that a client should require as little *a priori* knowledge about the implementation, general structure, or composition of the federation as possible. Ideally, the client should be able to discover a root of the global namespace without explicit configuration (*i. e.* via a simple discovery process) or by a minimal configuration (*i. e.* specifying an NSDB node location or an explicit FSL), if the local network does not support discovery.

We require that there exist some mechanism that allows all of the entities that participate in the federation protocols to mutually authenticate, either as principals or by roles. Note that authorization is accorded to a principal in the context of federation member—an administrator with root privileges within one federation member might not (and usually will not) have any administrative privileges in the context of any other federation member.

We assume that all entities execute the protocols correctly and in good faith, or have well-defined and detectable failure characteristics. The behavior of the federation is undefined in the case of Byzantine behavior among the entities in the federation.

We also explicitly assume that all of the federation members share some common infrastructure, such as DNS and IP routing. We assume that federation members can express the network location of any service they wish to make accessible to other federation members via globally meaningful and unambiguous mechanisms (such as DNS names). These mechanisms need not be persistent. For example, if one federation member informs another that a particular volume is hosted on server moog.netapp.com, it is reasonable to expect that this information is correct and usable for some period time extending into the future, but there is no guarantee that this response will always be correct, or even that server `moog.netapp.com` will continue to exist or have a DNS entry.

3.3 Terms and Definitions

Current server collections that provide a shared namespace usually do so by means of a service that maps file system names to file system locations. We refer to this server as a namespace database service (NSDB). In some distributed file systems, this service is embodied as a volume location database (VLDB), and may be implemented by LDAP, NIS, or any number of other mechanisms.

The basic building blocks of the namespace are *filesets* and *junctions*. A fileset is the abstraction of a file system. The fileset abstraction implies very little about how the fileset is implemented, although in the simplest case a fileset can be implemented by an exported file system. A fileset is a directory tree that may contain ordinary files and directories. A fileset may also contain a new type of object, called a junction, which is a reference to another fileset. A junction is a link from a path in a fileset to an object in another fileset.

Each fileset has a globally unique fileset name (FSN) that is used as the identifier of the fileset. If a fileset is replicated, then all replicas have the same FSN. If a fileset is migrated from one server to another, the FSN remains the same. Each implementation (*i. e.* each replica) of a given fileset is specified by a fileset location (FSL).

The NSDB is a federated service. Rather than a monolithic NSDB that manages all of the namespace information for the entire federation, the NSDB service is partitioned so that each administrative domain (or set of allied servers) has its own NSDB service instance, called an NSDB node, that is responsible for providing authoritative information about filesets hosted by those servers. Figure 1 shows two members of a federation.

The primary purpose of the NSDB service is to provide a mapping between FSNs and FSLs. Junctions only record the FSN of the target fileset, not the FSLs of instances of that fileset. This level of indirection is necessary to satisfy the requirements of federation; if a fileset is migrated from one location to another, the filesets containing junctions that refer to the migrated fileset do not need to be changed. The only changes that are required are confined to the local administrative domain—migration of the fileset itself, and the update to mapping from the FSN of that fileset to its new FSL.

Each FSN has two components: a junction key and an NSDB lo-

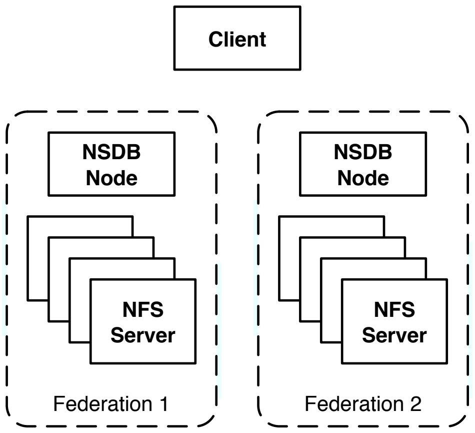

Figure 1: Federation overview showing two independent administrative domains.

cation. The junction key is a UUID that is used to globally identify the fileset. The NSDB location is an identifier for the NSDB node responsible for the authoritative information about the fileset. Information about an FSN may be cached or replicated across more than one NSDB node, but only the NSDB node named by an FSN is treated as the definitive source of information about that FSN.

Note that the NSDB location is not a literal location (*i. e.* an IP address and port number), but is an identifier used to find the correct NSDB node instance. This identifier is typically a DNS name which is resolved to an SRV record for the NSDB service, which in turn resolves to the IP location of the instance(s) of the NSDB [6].

In contrast to most contemporary peer-to-peer services, which strive to make services location-oblivious (*i. e.* any node can host any object), the administrative requirements of federation are much easier to satisfy if the location of the authoritative copy of each object—such as the mapping between an FSN and its FSLs—is known a priori. Contemporary peer-to-peer services make very weak assumptions about the reliability, availability, and integrity of each node in the system, and must cope with problems such as churn [5]. Our basic assumptions are very different: we assume that NSDB nodes are long-lived, maintained diligently, and considered part of the core infrastructure of each federation member (on par with DNS servers, border gateway routers, mail servers, and NFSv4 servers). Although it is not mandated by the protocol, the best practice is to implement each NSDB node as a fault-tolerant, replicated service (rather than a single server). This makes it practical to encode an NSDB location as part of the FSN.

It is important to note that the Glitz protocols are used to create, manage, organize, and report information about the implementation of the namespace, but are not used to manipulate the underlying fileset implementations. For example, the NSDB is used to track the mapping between FSNs and FSLs, not control the actual FSLs. Adding a new FSN to FSL mapping does not create a new instance of the fileset, nor does changing a mapping cause a fileset to migrate, nor does removing a FSL from a mapping delete a replica. It is the responsibility of the administrator to ensure that any changes to the implementation of the namespace are recorded correctly in the NSDB, but the administrator does not make these changes via the NSDB.

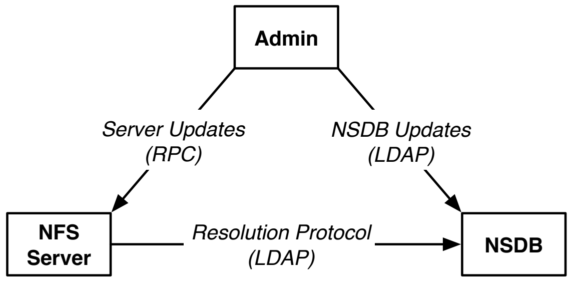

Figure 2: The three Glitz client/server protocols: NFS server/NSDB, Admin/NFS server, and Admin/NSDB.

It is not the intent of the federation to guarantee namespace consistency across all client views. Since different parts of the namespace may be administered by different entities, it is possible that a client could be accessing a stale area of the namespace managed by one entity because a part of the namespace above it, managed by another entity, has changed.

It should be possible, using a system that implements the interfaces, to share a common namespace across all the fileservers in the federation. It should also be possible for different fileservers in the federation to project different namespaces and enable clients to traverse them.

There are three types of entities that participate in the Glitz federation protocols: NFSv4 servers, NSDB servers, and administrators. The administrators send requests to the NFSv4 servers and the NSDB servers in order to manage the namespace, and the NFSv4 servers use the NSDB servers in order to resolve the current location of filesets. These three types of interaction define three subprotocols, which we describe in more detail in the next sections. How the protocols fit together is illustrated in Figure 2.

Since the NSDB is a directory service, it makes sense to implement it on top of an existing directory service, such as LDAP [11]. LDAP already supports security and replication for fault tolerance and availability, and therefore the NSDB can be implemented easily as an ordinary LDAP service.

The current draft specifies that the protocol that the administrator uses to create or destroy junctions on the server is not expressed in LDAP, but rather in ONC/RPC [13]. Like LDAP, ONC/RPC has already been extended with standardized security mechanisms, so that these administrative tasks can be done securely. Using RPC on the server is a natural fit, because the NFS protocols are defined using the same mechanisms and therefore we are not introducing onerous new administrative chores, such as setting up and maintaining both secure RPC and secure LDAP for the same service. (Since the NFS server must communicate with the NSDB in order to perform junction resolution, the NFS server must be an LDAP client, but this is a relatively small burden compared with requiring that the NFS server must also be an LDAP server.)

3.4 Querying and Updating the NSDB

The contents of the NSDB define most of the aspects of the global namespace. Changes to the set of filesets, their properties, or the location(s) of their implementations must be reflected by changes to the contents of the NSDB. There are several tables and mappings maintained by the NSDB, and the contents for each NSDB node can be edited by the administrator of that node. These include:

- The mapping from FSNs to FSLs. This mapping is used by the NFS servers to perform FSN resolution.
- The list of FSNs managed by this node. Knowing the set of FSNs managed by a node is useful to the administrator (or more likely, the tools used by the administrator) in order to build a catalog of all filesets under administration.
- The mapping from FSNs to annotations.
- The mapping from FSLs to annotations.

Each object in the NSDB may be annotated to provide extra information that may be helpful the servers or the administrators. For example, an FSN may be given an annotation describing its purpose (*e. g.* "Dan's home directory" or "Snapshot of the Pathfinder work area on April 12th, 2008"). Descriptive annotations are not necessary for the correct execution of the protocol, but can be very useful to the administrator.

Each annotation is a name/value tuple, and there are operations for inserting, deleting, modifying, or searching annotations. In the initial draft, the names and interpretations of annotations were left completely up to the administrator, but there is ongoing discussion about codifying some of the annotation names and the interpretation of their values so that they can be used to represent standard information (such as mount options, owner, creator, and similar attributes).

3.5 Creating, Deleting, and Updating Junctions

While the NSDB is responsible for keeping track of filesets and their properties, the file servers are responsible for storing and maintaining the junctions that stitch together these filesets into the namespace.

There is a considerable diversity of opinion about what form the server should export to the administrator. We begin with a discussion of the interface as it exists in the proposal as of 2008, and then describe some of the proposed alternatives and the tradeoffs between them.

The basic methods the interface must provide are:

- Create a junction from a given location in the local namespace of a fileset hosted on the server to an arbitrary FSN.
- Delete a junction from a given location in the local namespace, removing the corresponding name from the namespace.

Despite the apparent simplicity of these interfaces, there is much more detail that must be provided. One of the problems is how to specify the local pathname. Defining an object in terms of its pathname can lead to confusion on an NFS server, because the same filesets may appear in more than one namespace, and the same pathname in two different namespaces may name objects in different filesets. For example, the NFSv4 pseudo file system presents, to clients, a namespace constructed from the file systems exported by a server. This exported namespace may overlap or conflict with the local namespace (and there has even been a proposal that a server may present different clients different pseudo file systems, making paths in the pseudo file system even more difficult to reason about).

The current proposal is that it must be possible to specify the location in terms of a server-local pathname. Since these paths are generally not visible outside the server, this implies that the administrator must have direct access to the server file system and its namespace, but this assumption does not seem unreasonable because this operation also requires the ability to modify elements of the namespace.

The current proposal also permits a path location to be specified in terms of the pseudo file system (or relative to an export), but this functionality is not required. If the location is specified in this manner, a flag is set in the request to ensure that the server interprets the pathname in the appropriate context.

The second issue is how the administrator learns the value of the FSN to add. We assume that the administrator discovers the FSN either directly from another administrator or by browsing through the contents of an NSDB node (which presumes knowledge about which NSDB node to browse).

An important aspect of these interfaces is that there is little error checking; it is possible to create a junction that refers to an FSN that does not yet exist or that is not implemented by any accessible FSL. This is useful, however, because it permits the namespace to be constructed (or reconstructed, in the case of disaster recovery) in an arbitrary order. The administrator can verify whether an FSN is valid (*i. e.* actually is known to the given NSDB node, and has valid, available FSLs) when the junction is created (or at any particular instant in time), but there is no guarantee that validity will hold at any later point in time.

In an earlier draft of the protocol, the administrator could query a file server to discover the FSN of any client-visible fileset (either visible in the global namespace, or as an export). The assumption was that the administrator would learn, through some mechanism, the export host and path for some FSN, learn the FSN for that host and path from the host, and then construct the junction. In essence, given an FSL, the administrator could ask the server named in the FSL for the FSN implemented by that FSL. This has a certain elegance (all the administrator needs to see are human-readable representations of FSLs), but this convenience comes at a high cost. First, this increases the amount of information that the server must maintain, and the degree to which the server must be modified in order to support Glitz. More importantly, it increases the number of methods that each server must support. In the current draft, only servers that host filesets that contains junctions need to support this part of the protocol at all—it is possible for a standard, non-Glitz server to implement a fileset that serves as a leaf of the namespace. Since such servers never need to resolve a junction, or answer queries about what FSNs they host, they can be implemented using unmodified, legacy NFSv4 servers. This is particularly practical because it means that an administrator can create a federated file system from a mix of Glitz-aware and Glitz-oblivious servers; it is not necessary upgrade all of the servers in order to use Glitz. The analogous capability of Glamour was a weakness because it required a vendor-specific federation mechanism, but Glitz, being an open protocol, could be implemented on anyone's system.

4. CONCLUSION

We have described Glitz, a collection of protocols that jointly allow NFS file server collaborations to interoperate. This work has led to a deep collaboration in the Federated-FS effort within the NFSv4 working group of the IETF, within which three RFCs are expected to be published, allowing not only namespace integration, but also the management of file system replication and migration in a vendor independent fashion. We also expect that the integration scheme will support Microsoft's CIFS [9] file system protocols, allowing organizations the ability to share complex, inter-vendor namespaces and server administration schemes for the benefit of both their NFS and their CIFS clients.

Acknowledgements

The authors wish to thank James Lentini, Mark Storer, and John Strunk for their invaluable contributions to this paper.
NetApp, the NetApp logo, Go further, faster, and Data ONTAP are trademarks or registered trademarks of NetApp, Inc. in the United States and/or other countries.

5. REFERENCES

[1] CALLAGHAN, B., PAWLOWSKI, B., AND STAUBACH, P. NFS version 3 protocol specification. RFC 1813, June 1995.

[2] CALLAGHAN, B., AND SINGH, S. The autofs automounter. In *Proceedings of the USENIX Summer 1993 Technical Conference* (Cincinnati, OH, June 1993).

[3] EISLER, M., CORBETT, P., KAZAR, M., NYDICK, D. S., AND WAGNER, J. C. Data ONTAP GX: A scalable storage cluster. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)* (San Jose, CA, Feb. 2007), pp. 139–152.

[4] EVERHART, C., MAMAKOS, L., ULLMANN, R., AND MOCKAPETRIS, P. New DNS RR definitions. RFC 1034 and 1035, Oct. 1990.

[5] GODFREY, P. B., SHENKER, S., AND STOICA, I. Minimizing churn in distributed systems. *ACM SIGCOMM Computer Communication Review 36*, 4 (Oct. 2006), 147–158.

[6] GULBRANDSEN, A., VIXIE, P., AND ESIBOV, L. A DNS RR for specifying the location of services (DNS SRV). RFC 2052, Feb. 2000.

[7] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS) 6*, 1 (Feb. 1988), 51–81.

[8] KAZAR, M. L., LEVERETT, B. W., ANDERSON, O. T., APOSTOLIDES, V., BOTTOS, B. A., CHUTANI, S., EVERHART, C. F., MASON, W. A., TU, S.-T., AND ZAYAS, E. R. DEcorum file system architectural overview. In *In Proceedings of the Summer 1990 USENIX Conference* (Anaheim, CA, June 1990).

[9] LEACH, P. J., AND NAIK, D. C. A common internet file system (CIFS/1.0) protocol, Dec. 1997.

[10] LENTINI, J., EVERHART, C. F., ELLARD, D., TEWARI, R., AND NAIK, M. Requirements for federated file systems. RFC 5716, Jan. 2010.

[11] OPENLDAP FOUNDATION. Lightweight directory access protocol (LDAP): Technical specification road map. RFC 4510, June 2006.

[12] SHEPLER, S., EISLER, M., AND NOVECK, D. Network File System (NFS) version 4 minor version 1 protocol. RFC 5661, Jan. 2010.

[13] SRINIVASAN, R. RPC: Remote procedure call protocol specification version 2. RFC 1831, Aug. 1995.

[14] STEWART, J. N. AMD - the Berkeley automounter, part 1. *;login* (May 1993).

[15] SUN MICROSYSTEMS, INC. NFS: Network file system protocol specification. RFC 1094, Mar. 1989.

[16] TEWARI, R., HASWELL, J. M., NAIK, M. P., AND PARKES, S. M. Glamour: A wide-area filesystem middleware using NFSv4. Tech. Rep. RJ10368 (A0507-001), IBM Research Division, July 2005.

[17] ZAYAS, E. R., AND EVERHART, C. F. Design and specification of the cellular andrew environment. Tech. Rep. CMU-ITC-88-070, Information Technology Center, Carnegie Mellon University, Aug. 1988.

[18] ZHANG, J., AND HONEYMAN, P. Naming, migration, and repliction for NFSv4. Tech. Rep. 06-01, Center for Information Technology Integration, University of Michigan, Ann Arbor, MI, Jan. 2006.