Network File System (NFS)

NFS is a protocol developed by Sun Microsystems. NFS uses the client-server model, where one powerful server is accessed by potentially many clients. This model has several advantages, but also presents interesting implementation challenges.

og-structured file system writes data and metadata in an append-only fashion, essentially treating the disk as a circular log.

Learning Objectives

- Be able to describe how and why statelessness is used in the NFSv2 design
- Be able to describe idempotent operations and how NFSv2 uses idempotency to simplify recovery/design
- Be able to illustrate the client/server model and a typical NFS configuration
- Be able to describe how an application's FS system calls are translated into NFS network messages
- Be able to describe how NFS uses client-side caching to improve application performance
- Be able to describe how NFS uses server-side caching and its limitations

The client-server model

NFS is an open protocol designed by SUN, essentially to create a market for SUN to sell powerful storage servers. There are many benefits to the client/server model.

- simplified the system administration efforts
  - things like backup, encryption, data management, etc. can all be done on a single server rather than at each client machine
- a user can interact with their data from any number of workstations, and receive the same experience/guarantees

In addition to the model itself, NFSv2 was developed around some interesting design principles. It was an influential model that continues to be used to this day (albeit with a slightly evolved design).

How the model works

Clients that use NFS operate as if they were using a local file system. Under normal operation, the fact that client machines are communicating over a network is transparent: clients are presented with the illusion of a local file system.

- Applications issue system calls on their local machine
- The NFS client converts those system calls into a series of network requests to the server
- The server responds to each request, possibly by reading the user's data from its central store
- The NFS client parses relevant data from those responses, and applies those results to the client's view of their data

NFS Design Goals

Since the server is the single most important machine in the protocol (many clients connect to the server, and if the server goes down, the clients are largely unusable), the NFS design prioritizes simple and efficient server recovery. Two design elements that I find particularly elegant are that

- NFS is a *stateless* protocol: all information necessary to complete a given operation is provided as part of the request (i.e., the server does not need to keep any context to understand what the client is requesting, the client always

provides that context)
- (most) NFS operations are *idempotent*: operations can be executed repeatedly with the same effect. In this way, if a server crashes, a client can simply repeat an operation without wondering whether or not it was successfully applied by the server. In either case, the desired outcome will be the achieved after the idempotent operation completes.

NFS implementation

File Handles. To make NFS a stateless protocol, clients package their context into an NFS *file handle**. File handles contain a *volume ID*, an *inode number*, and a *generation number*.

- the volume ID tells the server which of the (possibly many) data volumes the request is targeting (e.g., an NFS server might have separate student volumes and instructor volumes to simplify the management of student files that are cleaned every 4 years and instructor files which persist until retirement)
- the inode number identifies a file system object within a volume (e.g., /faculty/bill/cs333exam.txt might have inode number 73)
- the generation number is useful because inode numbers are reused. Suppose a file is deleted, and then a new file is created. It is possible that the new file recycles the deleted file's inode number. If a client tries to access the deleted file using only the deleted file's inode number, the client would receive surprising data without any indication of an error.

The file handle is an essential part of almost every NFS protocol message.

Client side caching. NFS clients are separated from the server by a network. The latency of accessing a remote machine over a network is much higher than the latency of accessing a local device. Since many file system operations are translated into multiple messages, each of which must cross the network in a *round trip*, NFS clients cache data when possible. At a high level:

- Reads are cached locally; future requests for the same data can be satisfied from the cache.
- Writes are buffered in memory (as they are in most local file systems), and updates are written to the server when explicitly demanded or convenient

This caching introduces a challenge: **consistency**. Multiple clients can open the same file. If they each cache parts of their data locally, their view of a given file's contents can become out of sync in the presence of updates.

- NFS implements *close-to-open* consistency. This does not provide very strong guarantees about how fast updates are propagated, but it does give a model that clients can reason about: when a client closes a file, a future open of the same file will see the latest version.
- NFS uses `GETATTR` requests to check the validity of its cache. The result indicates whether or not the server has a more recent version of the file than the cached version.
    - Since `GETATTR` requests also require a round trip, clients often issue `GETATTR` requests every *N* seconds; this bounds the staleness, but introduces a window of vulnerability.

Server side caching. NFS servers must persist data, but they may also want to buffer updates for performance reasons.

- NFS servers cannot acknowledge a write until it has been made persistent.
- Companies like NetAPP made their money by adding battery-backed memory so they could acknowledge writes quickly. They also use a variant of a log-structured file system to optimize for write performance.

Future Versions

The NFS protocol has had several iterations. The textbook focuses on NFSv2 and omits discussion of NFSv3 and NFSv4. The later versions implement some optimizations to minimize the number of "round trips" between the client and server for common, and to improve client performance by rolling back some of the design principles of v2. To summarize:

- Some state is introduced
- Compounds let you bundle multiple operations into a message, rather than waiting for responses before sending the next message

Questions:

1. How do idempotent operations simplify the design of how clients respond to server crashes/disconnections and recovery?
2. How does statelessness simplify NFS design (e.g., what are some challenges that would be added to the server if it needed to maintain state)?
3. How might removing statelessness improve NFS performance?
4. What are some alternatives to close-to-open consistency? What would be the costs of implementing those alternatives in NFS?