Network File System - NFS

- NFS is a distributed file system (DFS) originally implemented by Sun Microsystems.
- NFS is intended for file sharing in a local network with a rather small number of computers.
- NFS is both a specification and an implementation of a DFS. Today three versions exist of the specification - version 2, version 3, and version 4.
- The first implementation of NFS version 2 was released by Sun in 1984.
- An implementation of NFS version 3 was released by Sun in 1992.
- The 4.4BSD implementation of NFS was based on the protocol specification and not on the Sun implementation.
- The FreeBSD NFS implementation is a direct descendant of the 4.4 BSD code.

NFS Design Goals

- Protocol designed to be stateless.
- Designed to support UNIX filesystem semantics.
- Access controls follow the UNIX file system model.
- The protocol design is transport independent.

Some of the design decisions limits the areas where use of NFS is appropriate:

- The design is based on a fast local network.
- The caching model assumes most files will not be shared.
- The stateless protocol requires some loss of traditional UNIX semantics.

NFS Structure

- NFS operates as a client-server application. The communication is based on RPC:s.
- The NFS specification distinguishes between the services provided by the mount mechanism and the remote file access services.
- Two different protocols are specified - one protocol for mounting and one protocol for performing file operations - the NFS protocol.
- The RPC routines use a coding called XDR (External Data Representation) to code data sent across the network in a way that is independent of the memory architecture (*big-endian* versus *little-endian*).
- Network data is sent using UDP in NFS v2. NFS v3 also allows the use of TCP.
- The NFS protocol can run over any available stream- or datagram-oriented protocol.
- A big problem for NFS running UDP on an Ethernet is that the Ethernet data size limit of 1500 bytes forces the packets to be fragmented at the IP level. If UDP is used there is no error handling below the UDP level, thus if a single fragment is lost all 8 KB have to be retransmitted. This is the main reason why TCP is in fact a better choice than UDP.

NFS Structure - Mount Protocol

- Each file on a server can be identified by a *file handle*, returned to the client by the mount protocol.
- In the freeBSD implementation, the *file handle* is built from a *filesystem identifier*, an *inode number* and a *generation number*.
- The filesystem identifier and the inode provides an unique identifier for the inode to be accessed.
- The generation number verifies that the inode still references the same file as when it was first referenced.
- The use of the generation number ensures that the file handle is *time stable*.
- A time stable identifier allows the system to remember a file identity across transient errors and to detect attempts to access deleted files.

The NFS Protocol

- The NFS Protocol is stateless.
- The server does not need any additional information beyond that contained in the RPC to fulfill the request.
- In practice the server caches recently accessed file data.
- The benefit of the stateless protocol is that there is no need to do state recovery after a client or server crash.
- There are also some drawbacks to the stateless protocol:
  - → In a local filesystem an unlinked file will be accessible until the last reference to it is closed - on NFS it will disappear immediately.
  - → For version 2 of the NFS protocol, all operations that modify the file system must be written to disk before the RPC can be acknowledged.
  - → For a growing file an update may require three synchronous disk writes: one for the inode, one for the indirect block and one for the data block.
  - → Version 3 of the NFS protocol eliminates some synchronous writes by adding a new asynchronous write RPC.

NFS Protocol - Buffering

- The NFS protocol do not specify the granularity of the buffering that is used when files are written.
- Most NFS implementations buffer files in 8 Kbyte blocks.
- If 10 bytes in the middle of a block is modified most NFS implementations will read the 8 Kbyte block, modify it and write it back.
- The FreeBSD implementation keeps additional information that describes which bytes in a buffer is modified.
- If 10 bytes are written in the middle of a block, FreeBSD will read the entire block from the server but only writes back the modified bytes.
- This has two benefits:
  - → Fewer data are sent over the network.
  - → Non-overlapping modifications to a file are not lost.

FreeBSD NFS Implementation

The NFS implementation in FreeBSD was originally written by Rick Macklem based on version 2 of the NFS specification and later extended to include version 3 of the specification.

The version 3 protocol provides the following:

- 64-bit file offsets and sizes.
- An access RPC
- An append option on the write RPC
- A defined way to create special files and fifos.
- The ability to batch writes into several asynchronous RPC:s followed by a commit RPC.

Rick Macklem also made several nonstandard extensions to NFS that became known as *Not Quite NFS (NQNFS).*

FreeBSD NFS Implementation

- The FreeBSD client and server implementations of NFS are kernel resident.
- NFS interfaces to the network with sockets using the internal kernel interface routines *sosend()* and *soreceive()*. (Chapter 11 in the book).
- Less time critical operations such as mount, unmount and determining which filesystems is allowed to be exported, are handled by user-level daemons.
- The server side requires the **portmap**, **mountd** and **nfsd** daemons to be running.
- Full functionality also requires the **rpc.lockd** and **rpc.statd** daemons.
- The **portmap** daemon acts as a registration service for RPC-based services.

NFS - Mountd

The interactions between a client and the server when mounting an NFS file system is shown in figure 9.2

The mountd daemon handles two important functions:

1. On startup or after a HUP signal, **mountd** reads the /etc/exports file and creates a list of hosts to which each local filesystem may be exported. The list is passed into the kernel and stored in the mount struct.
2. Client mount requests are directed to the **mountd** daemon. After verifying that the client has permission to mount the requested file system a *file handle* is returned.

NFS - nfsd

- The **nfsd master daemon** forks off children that enters the kernel using the nfs-specific *nfssvc* system call.
- The children remains kernel resident providing a process context for the RPC daemons.
- Typical systems run four to six **nfsd** daemons.
- If **nfsd** is providing datagram service it creates a datagram socket.
- If **nfsd** is providing stream service, connected stream sockets will be passed in by the master nfsd daemon in response to connection-oriented requests from clients.
- When a request arrives on a socket where is a callback from the socket layer that invokes the *nfsrv_rcv()* routine.
- The *nfsrv_rcv*() call takes the message from the socket receive queue and passes it to an available nfsd daemon.
- The **nfsd** daemon verifies the sender and passes the request to the appropriate local filesystem implementation for processing.
- When the result returns from the filesystem it is returned to the requesting client.

NFS - rpc.lockd and rpc.statd

- The **rpc.lockd** daemon manages locking requests for remote files.
- The **rpc.statd** daemon cooperates with **rpc.statd** daemons on other hosts to provide a status monitoring service.
  - → The daemon accepts requests from programs running on the local host (typically rpc.lockd) to monitor the status of a specified host.
  - → If a monitored host crashes and is restarted the rpc.statd daemon will inform the other daemons about the crash when it is restarted.
  - → By using the rpc.statd service, crashes will be discovered and the locks held by a crashed host will be released - otherwise such locks may be held indefinitely.

NFS - nfsiod

- The purpose of the **nfsiod** daemons is to do asynchronous read-aheads and write-behinds.
- If *nfsiod* is not used, each read or write of an NFS file that cannot be serviced from the local cache must be done in the context of the requesting process which have to wait for the operation to complete.
- The client server interaction when *nfsiod* is used is illustrated in fig. 9.3

Client-Server Interactions

- A local filesystem is not affected by network disruptions because it will work unless where is a catastrophic crash at the local machine in which case there is nothing to do.
- By contrast, the client of a network file system must be prepared to deal with the case that the client is still running but the server becomes unreachable or crashes.
- Each NFS mount point provides three alternatives for dealing with server unavailability:

1. *Hard mount.* The client will continue to try to connect the server indefinitely. this is the default behavior.
2. *Soft mount.* The client retries an RPC a specified number of times an then returns a transient error. The problem with this type of mount is that many programs do not expect transient errors from I/O system calls.
3. *Interruptible mount.* The process will wait forever but it sleeps at a level that can be interrupted by a signal.

RPC Transport Issues

- Since UDP does not guarantee datagram delivery, a timer is started and if a timeout occurs the RPC request is retransmitted.
- This may create problems for non-idempotent operations if an RPC is retransmitted after it has been received by the server.
- A recent-request cache is normally used at the server to minimize the negative effect of duplicated RPC:s.
- The amount of time a client waits before resending an RPC is called the *round-trip-timeout (RTT).*
- Because the round trip time is highly variable it is difficult to find a good value for RTT.
- If an Ethernet is used the standard-sized 8-Kbyte datagrams will be fragmented at the IP level into at least six fragments that fits the Ethernet maximum packet size of 1500 bytes.
- If a single of these fragments is lost the entire RPC must be retransmitted.
- Almost all these problems will disappear if TCP is used as a transport protocol.
- Since TCP provides a reliable data stream all retransmissions will be done at TCP level.
- In NFS version 3 TCP is the first alternative. NFS version 4 is also specified to use TCP.

Security Issues

- NFS versions 2 and 3 are not secure.
- There has been some attempts to improve the authentication but since all data are sent in clear text, the security is still limited.
- NFS export control is at the granularity of local filesystems.
- Associated with each local file system mount point is a list of the hosts to which the file system may be exported.
- When NFS is running UDP this list is checked for every RPC. If NFS is running TCP the check is made when a connection is established.
- The user and group permissions are checked based on the client UID and GID.
- This requires that the client and server use the same UID and GID assignments.
- NFS have been extended to use Kerberos authentication but it is not well defined and interoperability cannot be guaranteed.

Performance aspects

- To improve performance client caches is used.
- A problem with client caches is that caches at different clients may contain inconsistent data.
- There are three possible write strategies for the caches:

1. Synchronous writing. The write system call do not return until the data is stored at the server.
2. Delayed write. The write system call returns immediately. Writes to the server are delayed until the cache is full.
3. Asynchronous write. The write to the server is started immediately but the write system call returns before the write completes.

- The FreeBSD NFS implementation uses asynchronous writes while a file is open but synchronously waits for all data to be written when the file is closed.
- The implementation will query the server about attributes of a file at most once every 3 seconds.