Very Simple Network File System (VSNFS)

KarthikBalaji Gandhi (findkb@gmail.com)
Prabakar Radhakrishnan (prabakarcse@gmail.com)
Praveen Krishnamoorthy (kpraveen85@gmail.com)

Problem Statement :

Design a very simple network filesystem(VSNFS). Such a filesystem can be used as a template for quick creation of network filesystems tuned for specific requirements.

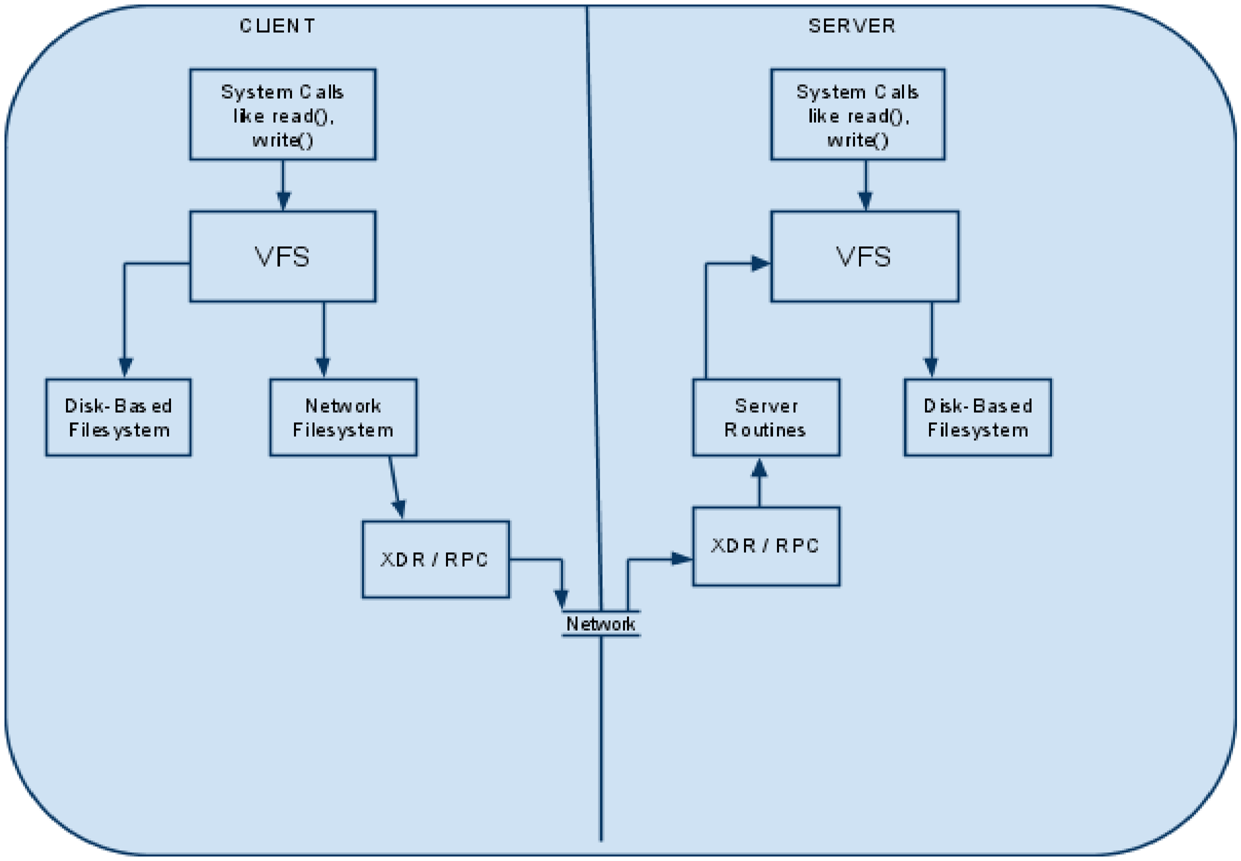
Introduction :

Network filesystem enables file access over a network through the usage of Remote Procedure Calls. Notable network filesystems include Sun's Network File System(NFS), Andrew FileSystem, Common Internet FileSystem(CIFS). The modern day Network Filesystems are very complex and hard to comprehend.

The motivation in taking up this project is to understand the intricacies of a filesystem by designing and implementing a bare minimal network filesystem that caters to the user request for the remote access of files. It is always useful to have a design template which can be extended and tuned for specific requirements.

Our design is loosely based on the design of NFSv2 specification(RFC#1094). VSNFS will be implemented as a loadable kernel module in linux kernel v2.6.31.6.

Design :



As seen from the above diagram there are three important parts in the network filesystem.

- VSNFS Protocol
- VSNFS Server
- VSNFS Client

VSNFS Protocol:

Like NFS protocol we will also be using Sun Remote Procedure Call (RPC) mechanism. The remote procedure call framework is designed to be transport protocol independent. We will be using TCP/IP as our transport mechanism. The protocol is defined in terms of set of procedures, their arguments and results. The two categories of NFS procedures that we are planning to implement are filesystem level operations and file level operations.

filehandle is one of the most commonly used NFS procedure parameters. This is provided by the server and is used by the client to reference a file. Client does not care about the contents of the file handle but it is used in almost all file operations. We will be defining filehandle similar to the NFS implementation in linux kernel.

```
 #define VSNFS_MAXFHSIZE           128
 struct vsnfs_fh {
         unsigned short          size;
         unsigned char           data[VSNFS_MAXFHSIZE];
 }
```

Server will fill the data and it will be blindly used by the client as reference. Also there are certain attributes associated with each file. We define it based on the NFSv2 specification and it looks like,

```
struct fattr {
              ftype        type;
              unsigned int mode;
              unsigned int nlink;
              unsigned int uid;
              unsigned int gid;
              unsigned int size;
              unsigned int blocksize;
              unsigned int rdev;
              unsigned int blocks;
              unsigned int fsid;
              unsigned int fileid;
              timeval      atime;
              timeval      mtime;
              timeval      ctime;
          };
```

Since we aren't implemeting any permission checks or set/get attribute procedures, the notion of file attribute is not very useful for us. We provide this as a skeleton for future enhancements.

Filesystem level operations that are build into the protocol are,

- *mount("host", "/export/dir")* - This will mount the remote filesystem locally. This internally creates a RPC client. Returns the file handle to the root of the filesystem.
- *unmount("local_mount_point")* - Unmount the mounted VSNFS at the given mount point.

File level operations which usually take a file handle as one of their essential parameter are,

- *lookup(dir_fh, filename)* - If the file is found, creates and returns a new file handle and corresponding attributes to it.
- *create(dir_fh, filename, attr)* - Creates a new file with the specified attributes and returns its filehandle and its attributes.
- *remove(dir_fh, filename)* - Remove the file from the directory and returns the status.
- *read(fh, offset, count)* - Returns up to count bytes of data to a file starting from offset. Also returns file attributes.
- *write(fh, offset, count, data)* - Writes count bytes of data starting from offset. Returns the new attributes after the writing.
- *mkdir(dir_fh, name, attr)* - Creates a new directory in dir_fh with name and associated attr and returns new fh with its attributes.
- *rmdir(dir_fh, name)* - Removes the directory name from the parent directory dir_fh. Return the status.
- *readdir(dir_fh, cookie, count)* - Returns upto count bytes of directory entries from directory dir_fh. We use a opaque pointer to the next directory entry called a cookie. This cookie is used in in subsequent readdir calls to start reading further in the directory. A call to readdir with cookie of zero returns entries starting from first entry.

Every procedure will have a procedure number associated with it. This procedure number is used with the RPC when requesting service from the VSNFS server.

Every option requires either a directory handle or a file handle. So it is necessary during the mount time to obtain a file handle for root of the filesystem. This is done in the mount operation itself. For simplicity in implementation, we support very few features in the initial version of the VSNFS protocol.

- We limit the number of files in root directory so that server can maintain a table matching *filehandles* with *inodes*.
- There is no permission checks for clients.
- Only one mount point.
- Can serve only one client.
- To start with the files would also be transferred as a whole across the network.

VSNFS Server:

On the server side the user starts a new vsnfs kernel thread with the following command,

# vsnfs.start /export/dir

This registers a new RPC framework and waits for RPC's from the client. Whenever there is a request from the client that modify the files in the server, the server need not flush all modified data to the disk before returning from the call (i.e. the actual server disk write is asynchronous). The server just updates the disk cache and it is flushed to the disk as and when appropriate. This simplifies our server design.

The interesting part we read from the RFC is about what represents a filehandle, which is being used in all the procedures described in the above protocol design. When a filehandle created is handed out with an inode number that is later removed, and that inode was reused to point to some other file, we must have a way to identify that they are different files. So as described in the RFC we add a integer called generation number to the inode. This will keep track of the generation of inode being used. This is incremented everytime an inode is freed. Ideally a filehandle comprises of an inode number, generation number and VSNFS ID.

VSNFS Client:

The ultimate goal of the client is to provide transparent access to the files in the remote server. By transparency ,we emphasize that the users should be able to carry out any basic task in the network filesystem like he/she could in a real disk-based local filesystem. This is done with the help of the VFS layer. Once the client decides to access files in remote VSNFS server, it issues a mount command,

#mount -t vsnfs hostname:/export /mountpoint

When mount type is specified as vsnfs the corresponding module is automatically loaded and it registers itself with the VFS layer. Once the filesystem is mounted client can access the remote files like local files by issuing RPC to the VSNFS server.

Development Plan:

WEEK	MEET WITH SPONSOR	ACTIONS	RESULTS
11/16/09	11/20/09	Implementation of basic filesystem interface. Both client and server.	Only mount would be successful. If possible we will use vfs methods for certain operations or some hard-coded procedures.
11/23/09	11/27/09	Implementation of as many procedures as possible	Demonstrate the implemented procedures.
11/30/09	12/04/09	Wrap up the main development.	Most of the functionality should work though with some bugs.
12/07/09	12/09/09	Fixing bugs and running benchmarks.	Filesystem should be funtional . May have some less obvious bugs :). Call it beta !!

Future Enhancements:

- External Data Representation (XDR) to describe protocols in a architecture-dependent way.
- Authentication & permission checks for client request.
- Support for many clients and extending for several export points.

Demo Plan:

The VSNFS server and the VSNFS client would be made to run in separate VMs and one of the directory in the server would be exported to the client. The client should be able to mount this directory into its mount point and should be able to access it as though it resides in a local filesystem. Every implemented procedure(eg read,write) would be demonstrated.

Conclusion:

We aim to implement this as a separate module which doesn't affect other subsystem. The option of implementing this as a stackable network filesystem is also kept open.

References:

- RFC 1094 : “Network File System Protocol Specification”
- RFC 1057 : “Remote Procedure Call Protocol Specification”
- http://www.cs.yale.edu/homes/arvind/cs422/doc/nfs.pdf
- http://en.wikipedia.org/wiki/Network_File_System_(protocol)