ASSIGNMENT-3

Q.1) How File Service differ from File Server.

File Server-

A file server is a central server in a computer network that provides file systems or at least parts of a file system to connected clients. File servers therefore offer users a central storage place for files on internal data media, which is accessible to all authorized clients. Here, the server administrator defines strict rules regarding which users have which access rights: For instance, the configuration or file authorizations of the respective file system enable the admin to set which files can be seen and opened by a certain user or user group, and whether data can only be viewed or also added, edited, or deleted. It is a service software running on a single machine.

File Service-

File service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. A software entity running on one or more machines. A service provides a particular type of function to a set of possibly unknown clients.

Q.2) Which is more beneficial between distributed file system (DFS) and file system in Centralized systems.

Distributed File System is far more beneficial than Centralized File System.

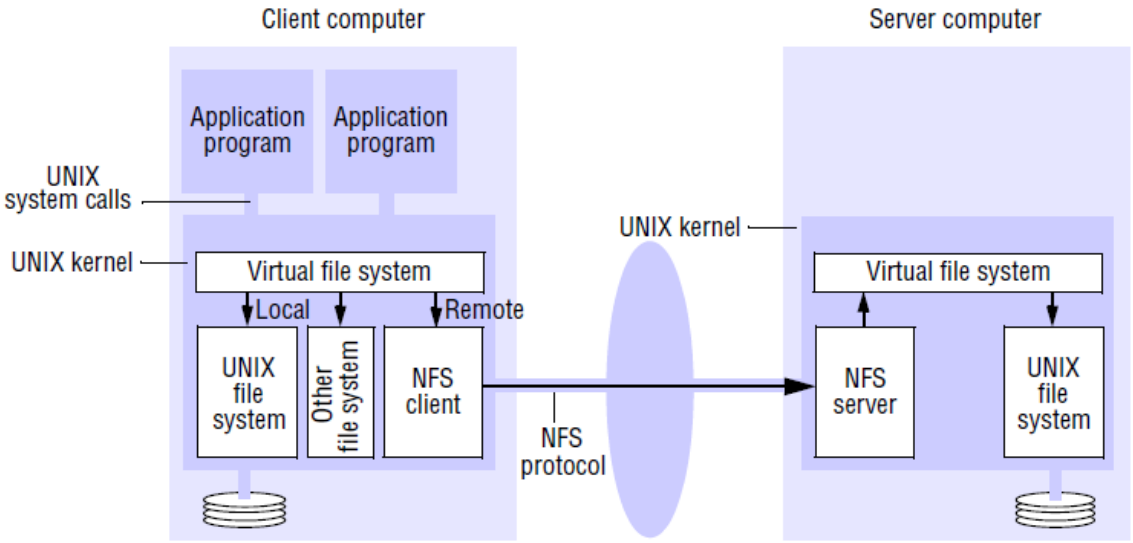
1.) In a centralized file system there is single central server, so in case of server failure all the operations would be stopped. Whereas in DFS there are multiple servers so partial failures does not affect overall functioning of system.
2.) In a centralized file system there is no concept of replication, so in case a file gets corrupted we don't have any backup. Whereas in DFS there can be multiple copies of file on different systems.
3.) Sharing of files between systems is far more easy compared to centralized File system.
4.) DFS is more scalable than centralized file system.
5.) DFS provide more flexibility in terms of implementing new file services.
6.) A DFS can have the computing power of multiple computers, making it faster than centralized system.
7.) As there are multiple servers it can handle more number of clients file requests and reduces network traffic.
8.) It provides persistent storage of files.

Although DFS is beneficial centralized file system also has some advantages over DFS:

1.) Implementation is far more easy than DFS.
2.) Security maintenance is easy compared to DFS as DFS contains multiple system at different location through interconnection network.
3.) Troubleshooting a centralized file system is easy.

Q.3) Explain Layer Structure of NFS.

All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system. The NFS client and server modules communicate using remote procedure calls.



1.) **Virtual file system -** NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way. The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate local system module . The file identifiers used in NFS are called *file handles*. A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file. The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file. A VFS structure relates a remote file system to the local directory on which it is mounted. The v-node contains an indicator to show whether a file is local

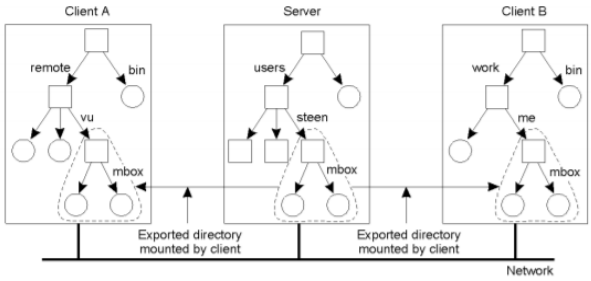
or remote. If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation). If the file is remote, it contains the file handle of the remote file.

2.) Client integration- It is integrated with the kernel and not supplied as a library for loading into client processes so that:

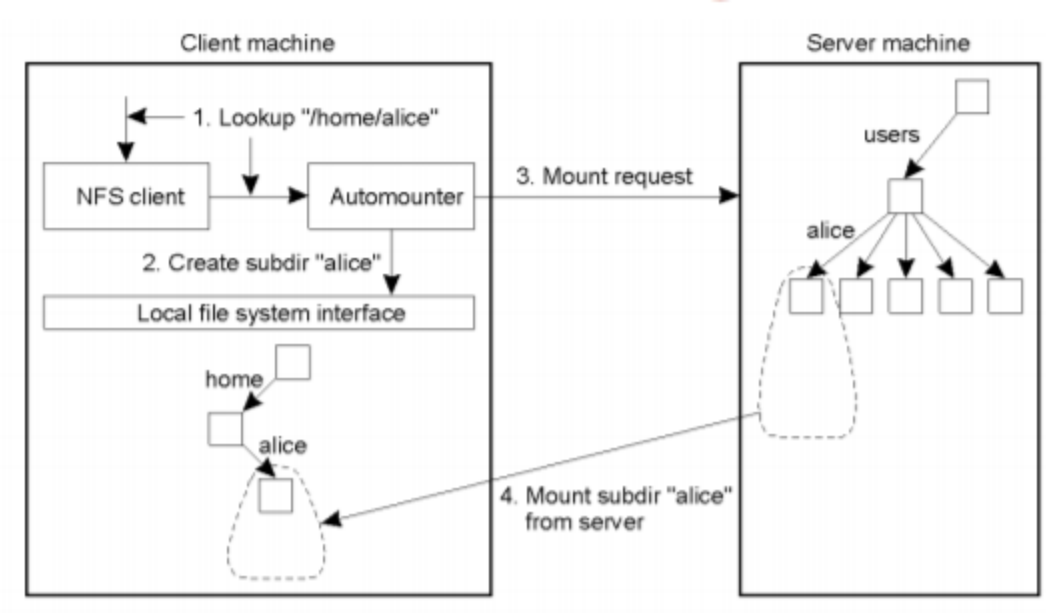
1.) User programs can access files via UNIX system calls without recompilation or reloading;
2.) A single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);
3.) the encryption key used to authenticate user IDs passed to the server (see below) can be retained in the kernel, preventing impersonation by user-level clients.

The NFS client module cooperates with the virtual file system in each client machine. It operates in a similar manner to the conventional UNIX file system, transferring blocks of files to and from the server and caching the blocks in the local memory whenever possible. It shares the same buffer cache that is used by the local input-output system.

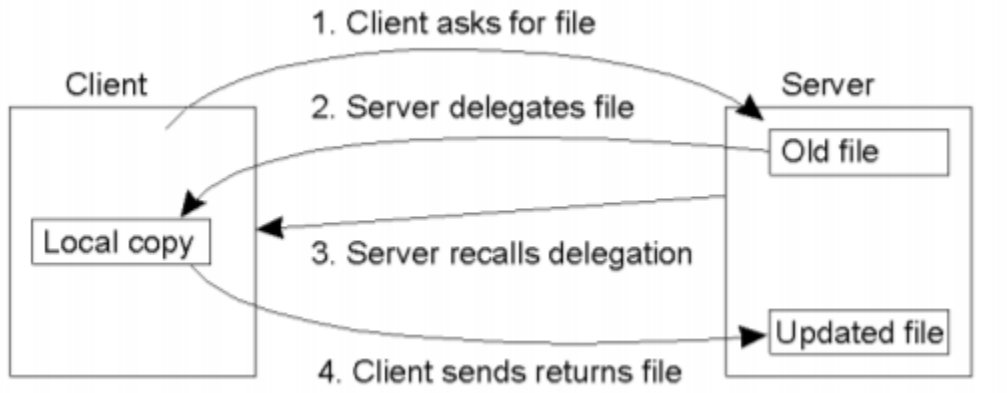
3.) Mount service -The mounting of subtrees of remote file systems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local file systems that are available for remote mounting. An access list is associated with each file system name indicating which hosts are permitted to mount the file system.



4.) Automounter - The automounter was added to the UNIX implementation of NFS inorder to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client.

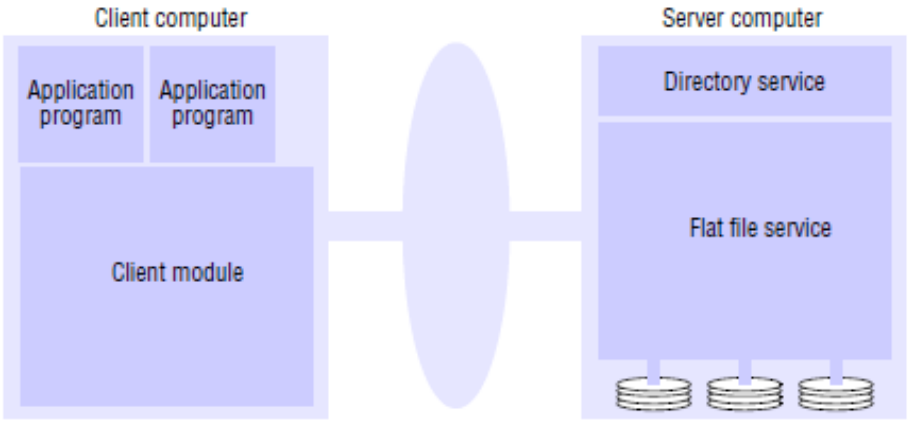


5.) Client Caching - The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.



Q.4) Discuss all modules of File Service Architecture.

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a flat file service, a directory service and a client module.



The division of responsibilities between the modules can be defined as follows:

1.) **Flat File Service** - The flat file service is concerned with implementing operations on the contents of files. Unique file identifiers (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

2.) **Directory service** - The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

3.) **Client module** - A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

Figure 12.7 Directory service operations

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.