UNIT-IV

Distributed File System

Distributed File System:

Definition: A **Distributed File System (DFS)** as the name suggests, is a file system that is distributed on multiple file servers or multiple locations. It allows programs to access or store isolated files as they do with the local ones, allowing programmers to access files from any network or computer.

The main purpose of the Distributed File System (DFS) is to allows users of physically distributed systems to share their data and resources by using a Common File System. A collection of workstations and mainframes connected by a Local Area Network (LAN) is a configuration on Distributed File System. A DFS is executed as a part of the operating system. In DFS, a namespace is created and this process is transparent for the clients.

DFS has two components:

- **Location Transparency –**
  Location Transparency achieves through the namespace component.
- **Redundancy –**
  Redundancy is done through a file replication component.

In the case of failure and heavy load, these components together improve data availability by allowing the sharing of data in different locations to be logically grouped under one folder, which is known as the "DFS root".

It is not necessary to use both the two components of DFS together, it is possible to use the namespace component without using the file replication component and it is perfectly possible to use the file replication component without using the namespace component between servers.

Features of DFS :

- **Transparency :**
  - **Structure transparency –**
    There is no need for the client to know about the number or locations of file servers and the storage devices. Multiple file servers should be provided for performance, adaptability, and dependability.
  - **Access transparency –**
    Both local and remote files should be accessible in the same manner. The file system should be automatically located on the accessed file and send it to the client's side.
  - **Naming transparency –**
    There should not be any hint in the name of the file to the location of the file. Once a name is given to the file, it should not be changed during transferring from one node to another.
  - **Replication transparency –**
    If a file is copied on multiple nodes, both the copies of the file and their locations should be hidden from one node to another.
- **User mobility :**
  It will automatically bring the user's home directory to the node where the user logs in.
- **Performance :**
  Performance is based on the average amount of time needed to convince the client requests. This time covers the CPU time + time taken to access secondary storage + network access time. It is advisable that the performance of the Distributed File System be similar to that of a centralized file system.
- **Simplicity and ease of use :**
  The user interface of a file system should be simple and the number of commands in the file should be small.
- **High availability :**
  A Distributed File System should be able to continue in case of any partial failures like a link failure, a node failure, or a storage drive crash.
  A high authentic and adaptable distributed file system should have different and independent file servers for controlling different and independent storage devices.
- **Scalability :**
  Since growing the network by adding new machines or joining two networks together is routine, the distributed system will inevitably grow over time. As a result, a good distributed file system should be built to scale quickly as the number of nodes and users in the system grows. Service should not be substantially disrupted as the number of nodes and users grows.

- **High reliability :**
  The likelihood of data loss should be minimized as much as feasible in a suitable distributed file system. That is, because of the system's unreliability, users should not feel forced to make backup copies of their files. Rather, a file system should create backup copies of key files that can be used if the originals are lost. Many file systems employ stable storage as a high-reliability strategy.
- **Data integrity :**
  Multiple users frequently share a file system. The integrity of data saved in a shared file must be guaranteed by the file system. That is, concurrent access requests from many users who are competing for access to the same file must be correctly synchronized using a concurrency control method. Atomic transactions are a high-level concurrency management mechanism for data integrity that is frequently offered to users by a file system.
- **Security :**
  A distributed file system should be secure so that its users may trust that their data will be kept private. To safeguard the information contained in the file system from unwanted & unauthorized access, security mechanisms must be implemented.
- **Heterogeneity :**
  Heterogeneity in distributed systems is unavoidable as a result of huge scale. Users of heterogeneous distributed systems have the option of using multiple computer platforms for different purposes.

Working of DFS :

There are two ways in which DFS can be implemented:

- Standalone DFS namespace –
  It allows only for those DFS roots that exist on the local computer and are not using Active Directory. A Standalone DFS can only be acquired on those computers on which it is created. It does not provide any fault liberation and cannot be linked to any other DFS. Standalone DFS roots are rarely come across because of their limited advantage.
- Domain-based DFS namespace –
  It stores the configuration of DFS in Active Directory, creating the DFS namespace root accessible at \\<domainname>\<dfsroot> or \\<FQDN>\<dfsroot>

SUN NFS-Network File System

The earliest successful distributed system could be attributed to Sun Microsystems, which developed the Network File System (NFS). NFSv2 was the standard protocol followed for many years, designed with the goal of simple and fast server crash recovery. This goal is of utmost importance in multi-client and single-server based network architectures because a single instant of server crash means that all clients are un serviced. The entire system goes down.

Stateful protocols make things complicated when it comes to crashes. Consider a client A trying to access some data from the server. However, just after the first read, the server crashed. Now, when the server is up and running, client A issues the second read request. However, the server does not know which file the client is referring to, since all that information was temporary and lost during the crash.

Stateless protocols come to our rescue. Such protocols are designed so as to not store any state information in the server. The server is unaware of what the clients are doing — what blocks they are caching, which files are opened by them and where their current file pointers are. The server simply delivers all the information that is required to service a client request. If a server crash happens, the client would simply have to retry the request. Because of their simplicity, NFS implements a stateless protocol.

File Handles:

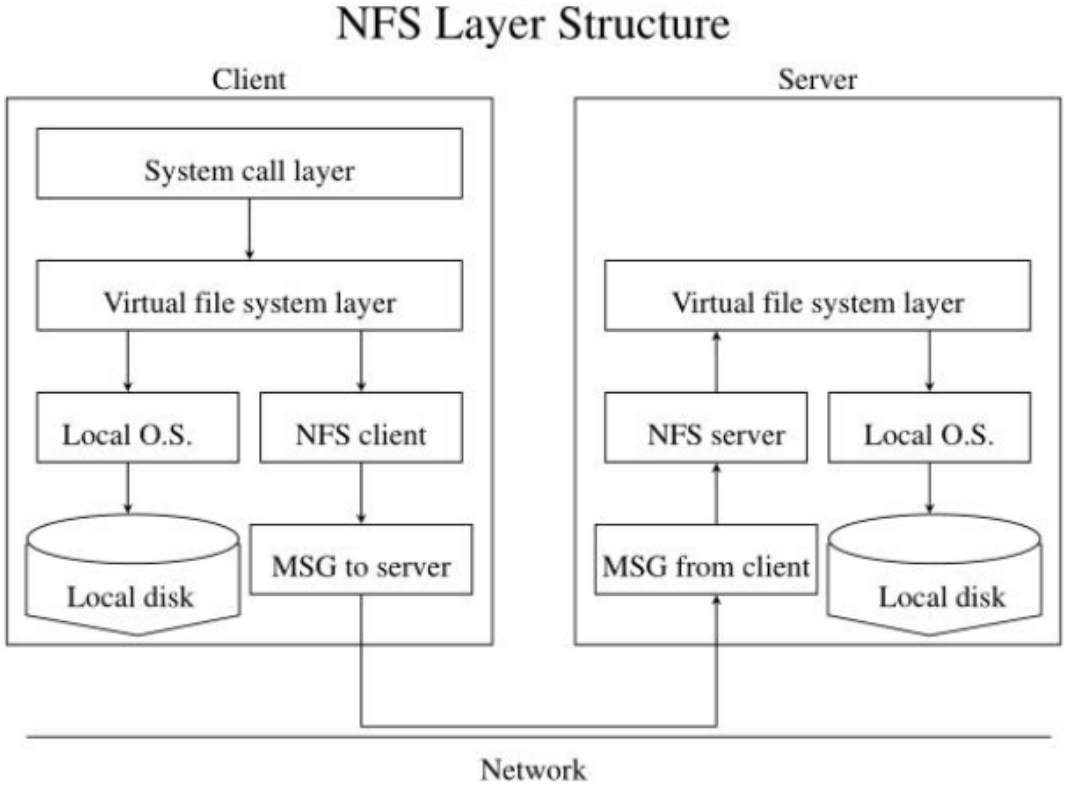
NFS uses file handles to uniquely identify a file or a directory that the current operation is being performed upon. This consists of the following components:

- **Volume Identifier –** An NFS server may have multiple file systems or partitions. The volume identifier tells the server which file system is being referred to.
- **Inode Number –** This number identifies the file within the partition.
- Generation Number – This number is used while reusing an inode number.

File Attributes:

"File attributes" is a term commonly used in NFS terminology. This is a collective term for the tracked metadata of a file, including file creation time, last modified, size, ownership permissions etc. This can be accessed by calling stat() on the file.

Network File System Architecture:



NFS Architecture

- The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system.
- In most cases, all the clients and servers are on the same LAN.
- NFS allows every machine to be both a client and a server at the same time.
- Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its sub-directories, so the entire directory tree is exported as a unit.
- The list of directories a server exports is maintained in the /etc/exports file, so these directories can be exported automatically whenever the server is booted.
- Clients access exported directories by mounting them. When a client mounts a directory, it becomes part of its directory hierarchy.
- A diskless workstation can mount a remote file system on its root directory, resulting in a file system that is supported entirely on a remote server.
- Those workstations that have a local disk can mount remote directories anywhere they wish. There is no difference between a remote file and a local file.
- If two or more clients mount the same directory at the same time, they can communicate by sharing files in their common directories.

NFS Protocol:

- A protocol is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients.
- As long as a server recognizes and can handle all the requests in the protocols, it need not know anything at all about its clients.
- Clients can treat servers as "black boxes" that accepts and process a specific set of requests. How they do it is their own business.
- *Mounting*:
  - A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy.
  - The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted.
  - If the path name is legal and the directory specified has been exported, the server returns a file handle to the client.
  - The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information.
  - Subsequent calls to read and write files in the mounted directory use the file handle.

NFS Implementation:

- It consists of three layers:
  - *system call layer*: This handles calls like OPEN, READ, and CLOSE.
  - *virtual file system (VFS)*: The task of the VFS layer is to maintain a table with one entry for each open file, analogous to the table of I-nodes for open files in UNIX. VFS layers has an entry, called a v-node (virtual i-node) for every open file telling whether the file is local or remote.
  - *NFS client code*: to create an r-node (remote i-node) in its internal tables to hold the file handles. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in the local operating system. Thus from the v-node it is possible to see if a file or directory is local or remote, and if it is remote, to find its file handle.

CODA File System (Constant Data Availability):

- CODA(constant data availability) is a distributed file system that was developed as a research project at Carnegie Mellon University in 1987 under the direction od Mahadev Satyanarayan.
- This goal has led to advanced caching schemes that allow a client to continue operation despite being disconnected from a server

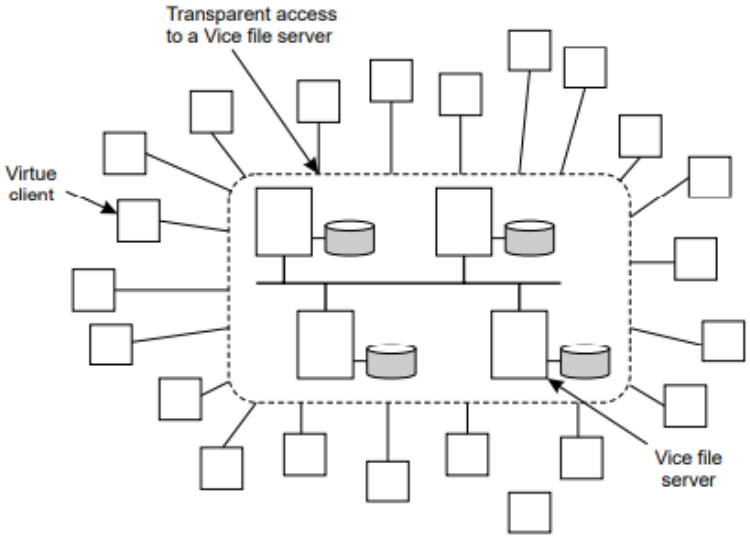
Coda design goals:

- Scalability
- Constant data availability
- Transparency

- Security
- Consistency

CODA-Architecture:

- All the nodes are partitioned into two groups.
- One group consists of a relatively small number of dedicated Vice file servers, which are centrally administered.
- The other group consists of a very much larger collection of Virtue workstations that give users and processes access to the file system, as shown in Fig. .
- Every Virtue workstation hosts a user-level process called Venus.
- A Venus process is responsible for providing access to the files that are maintained by the Vice file servers.
- In Coda, Venus is also responsible for allowing the client to continue operation even if access to the file servers is (temporarily) impossible.
- The important issue is that Venus runs as a user-level process.
- Again, there is a separate Virtual File System (VFS) layer that intercepts all calls from client applications, and forwards these calls either to the local file system or to Venus.



CODA Architecture

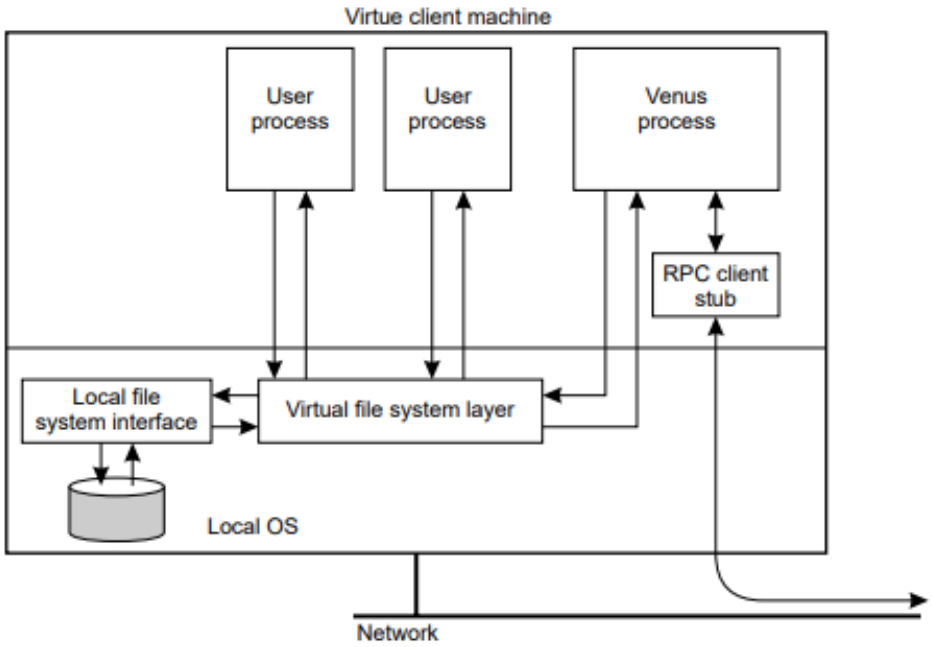


Figure 10-2. The internal organization of a Virtue workstation.

CODA-Implementation:

- RPC2 offers reliable RPCs on top of the (unreliable) UDP protocol.
- Each time a remote procedure is called, the RPC2 client code starts a new thread that sends an invocation request to the server and subsequently blocks until it receives an answer.
- As request processing may take an arbitrary time to complete, the server regularly sends back messages to the client to let it know it is still working on the request.
- If the server dies, sooner or later this thread will notice that the messages have ceased and report back failure to the calling application.
- An interesting aspect of RPC2 is its support for side effects.
- A side effect is a mechanism by which the client and server can communicate using an application-specific protocol.

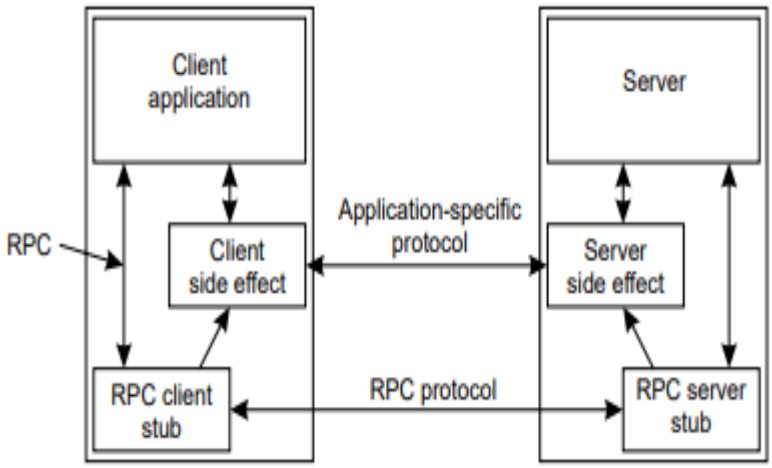


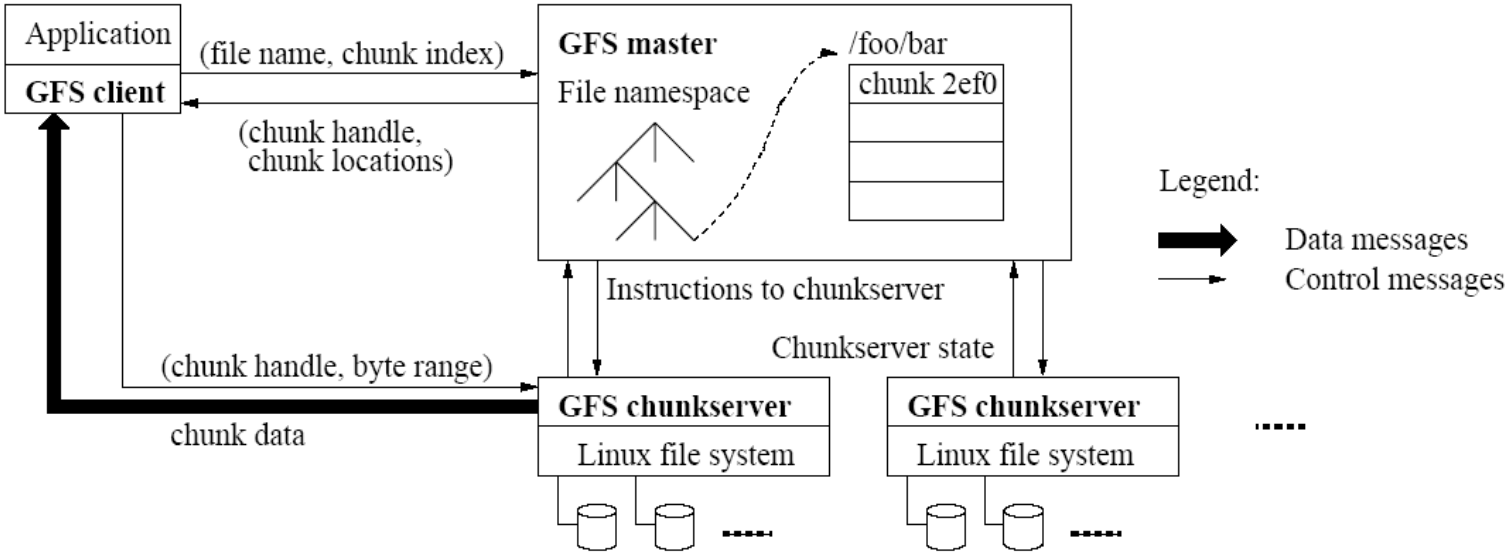
Figure 10-3. Side effects in Coda's RPC2 system.

The GOOGLE file system:

Definition: Google File System (GFS) is a scalable distributed file system (DFS) created by Google Inc. and developed to accommodate Google's expanding data processing requirements. GFS provides fault tolerance,

reliability, scalability, availability and performance to large networks and connected nodes. GFS is made up of several storage systems built from low-cost commodity hardware components. It is optimized to accomodate Google's different data use and storage needs, such as its search engine, which generates huge amounts of data that must be stored. The Google File System capitalized on the strength of off-the-shelf servers while minimizing hardware weaknesses. GFS is also known as GoogleFS.

GFS Architecture:



2. Other nodes act as the chunk servers for storing data.
3. The file system namespace and locking facilities are managed by master.
4. The master periodically communicates with the chunk servers to collect management information and give instruction to chunk servers to do work such as load balancing or fail recovery.
5. With a single master, many complicated distributed algorithms can be avoided and the design of the system can be simplified.
6. The single GFS master could be the performance bottleneck and single point of failure.
7. To mitigate this, Google uses a shadow master to replicate all the data on the master and the design guarantees that all data operations are transferred between the master and the clients and they can be cached for future use.
8. With the current quality of commodity servers, the single master can handle a cluster more than 1000 nodes.

The features of Google file system are as follows:

1. GFS was designed for high fault tolerance.
2. Master and chunk servers can be restarted in a few seconds and with such a fast recovery capability, the window of time in which data is unavailable can be greatly reduced.
3. Each chunk is replicated at least three places and can tolerate at least two data crashes for a single chunk of data.
4. The shadow master handles the failure of the GFS master.
5. For data integrity, GFS makes checksums on every 64KB block in each chunk.
6. GFS can achieve the goals of high availability, high performance and implementation.
7. It demonstrates how to support large scale processing workloads on commodity hardware designed to tolerate frequent component failures optimized for huge files that are mostly appended and read.