UNIT-3

DISTRIBUTED FILE SYSTEM

What is DFS?

- A Distributed File System (DFS) enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network.
- The resources on a particular machine are local to itself. Resources on other machines are remote.
- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.

Characteristics of (non-distributed) file systems

File systems are mainly responsible for organization, storage, retrieval, naming, sharing and protection of files.

They provide a programming interface that characterizes the file abstraction about the details of storage allocation & layout

- **Data and Attributes**
  - Data : Sequence of data items used for operations
  - Attributes :Single record holding information such as length of the file, timestamp, owner's identity, access control list
- **Directory:** mapping from text names to internal file identifiers.
- **Metadata** : Information stored by file system that is required for file management.
  - EX: File attributes, directories & all other persistent information

File attribute record structure

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

Layered Module structure for the implementation of a non-distributed File System

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk I/O and buffering

UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

Distributed file system requirements

- **1.Transparency:** Some aspects of Distributed system are hidden from user.
  - **Access**: Client pgms can be unaware of distribution of files. Same set of operations are provided for access to remote as well as local files.
  - **Location** : Client program should see a uniform name space.
  - **Mobility** : Client programs need not change their tables when files are moved to any other location.
  - **Performance** : Client program should continue to perform satisfactorily while the load on the service varies.
  - **Scaling** : The service can be expanded to deal with wide range of load and network size.

- **2.Concurrent file updates** : Changes to the file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file.(Concurrency Control)
- **3.File replication** : A file may be represented by several copies of its contents at different locations.

  Benefits:

  1. Load balancing to enhance the scalability of the service.
  2. Enhances the fault tolerance.
- **4.Hardware and OS heterogeneity** : The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers.

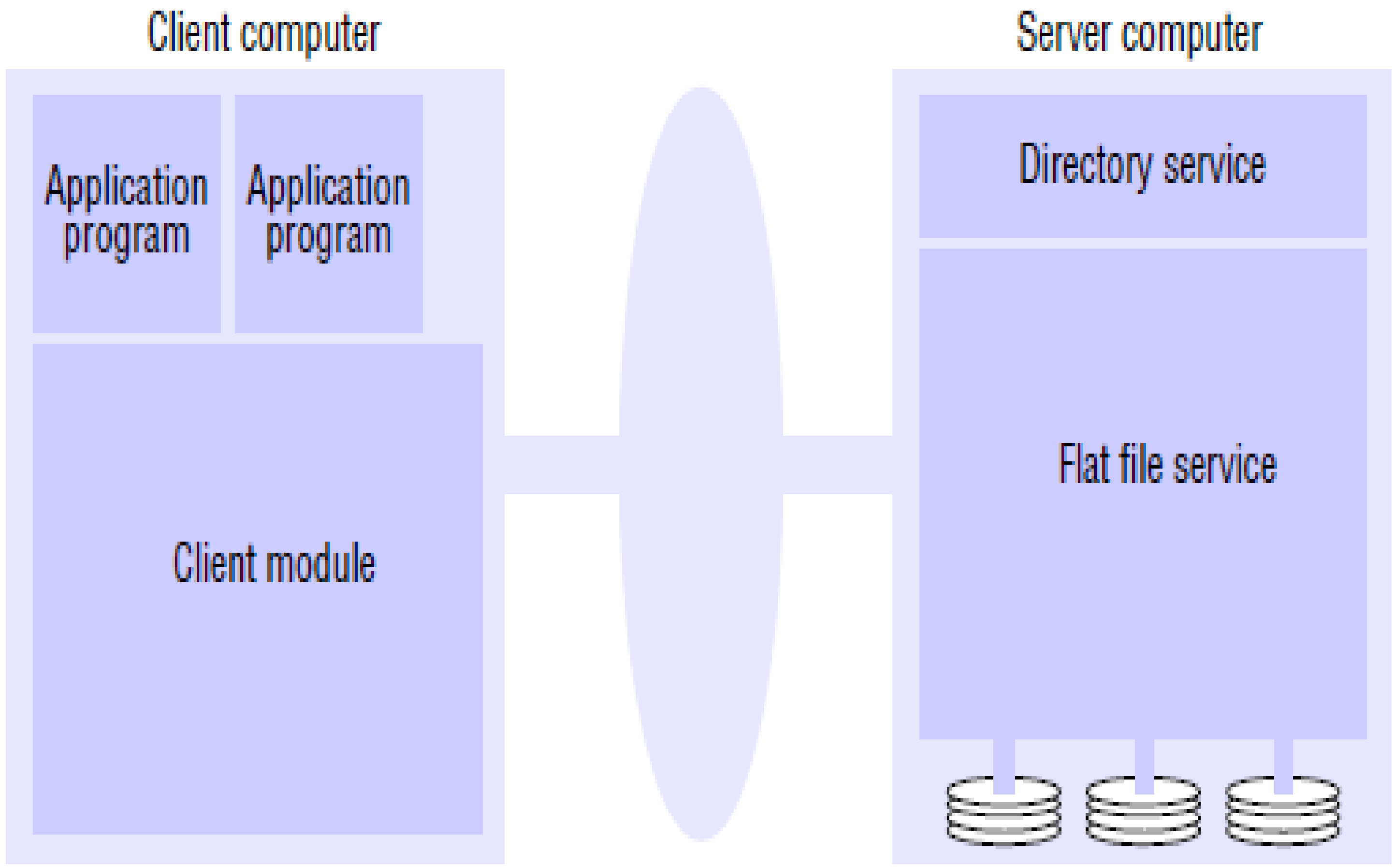
- **5.Fault tolerance** : Avoiding failure
  - To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics.

- **6.Consistency** : Maintaining the consistency between multiple copies of file.
- **7.Security**: In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages.
  - Digital signatures and encryption of secret data is used.
- **8.Efficiency** : A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance.

File Service architecture

- File service is structured as three components,
  - A flat file service
  - A directory service
  - A client module

File service architecture



- **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.
  - UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files.
  - 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.

Flat file service interface

Read(FileId, i, n) → Data — throws *BadPosition*	If $1 \leq i \leq Length(File)$: 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 $1 \leq i \leq Length(File)+1$: 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).

- **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.
  - Directory files are stored in files of the flat file service.

Directory service interface

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*.

- **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.
  - The client module also holds information about the network locations of the flat file server and directory server processes.
  - Achieves satisfactory performance through the implementation of a cache of recently used file blocks at the client.

- **Access control**
  - In the UNIX file system, the user's access rights are checked against the access *mode (read or write) requested in the open call* and the file is opened only if the user has the necessary rights.
  - In distributed implementations, access rights checks have to be performed at the server.
    - Issue: if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless.

- Two alternative approaches used are:
  - An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability, which is returned to the client for submission with subsequent requests.
  - A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

- **Hierarchic file system**
  - A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree.
  - The root of the tree is a directory with a 'well-known' UFID.
  - A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.
  - In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories.

- **File groups**
  - A *file group is a collection of files located on a given server.*
  - *A server may* hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs.
  - In a distributed file system that supports file groups, the representation of UFIDs includes a **file group identifier** component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.
  - Whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:

	32 bits	16 bits
file group identifier:	IP address	date