UNIT-6

Distributed File Systems

A distributed file system enables programs to store and access remote files exactly as local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks. The sharing of stored information is the most important aspect of distributed resource sharing. Web servers provide a restricted form of data sharing in which files stored locally, in file systems at the server on a local network, are made available to clients throughout the Internet. The design of large-scale wide area read-write file storage systems poses problems of load balancing, reliability, availability and security.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

→Characteristics of file systems:-

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout.

Files are stored on disks or other storage media. Files contain both ***data*** and ***attributes.*** The ***data*** consist of a sequence of data items, accessible by operations to read and write any portion of the sequence. The ***attributes*** are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. An attribute record structure is given in Figure below

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

File attribute Record Structure

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A ***directory*** is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the hierarchic file-naming scheme and the multi-part ***pathnames*** for files used in operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on).

The term ***metadata*** is used to refer to the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other information used by the file system.

Below Figure shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating 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

File System Modules

File system operations:-Below figure summarizes the main operations on files that are available to applications in UNIX systems.

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*.

These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/output Library or the Java file classes.

→Distributed file system requirements:-

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. The requirements are given as follows:

→Transparency
→Concurrent file updates
→File replication
→ Hardware and operating system heterogeneity
→Fault tolerance
→Consistency
→Security
→Efficiency

→Transparency: The file service is the most heavily loaded service in an intranet, so its functionality and performance are critical. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved.

***Performance transparency*:** Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

→Concurrent file updates: Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control.

→**File replication:** In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits –

i) It enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and

ii) It enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed.

→**Hardware and operating system heterogeneity:** The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers.

→**Fault tolerance:** The role of the file service in distributed systems makes it essential that the service continue to operate in case of client and server failures. The servers can be ***stateless***, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication.

→**Consistency:** Conventional file systems such as in UNIX offer ***one-copy update semantics***. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is a delay in the propagation of modifications made at one site to all of the other sites that hold copies.

→**Security:** All file systems provide access-control mechanisms based on the use of access control lists. 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 with digital signatures and encryption of data.

→**Efficiency: A** distributed file service should offer facilities that are same as those found in conventional file systems and should achieve a comparable level of performance.

→File service architecture:-

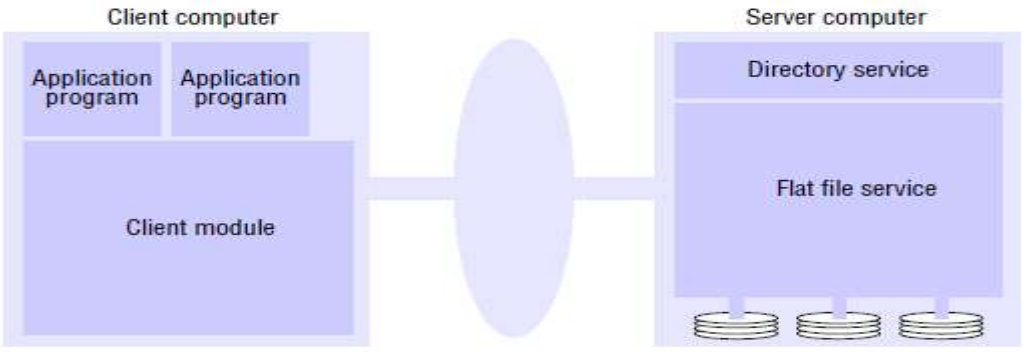
The file service is structured as three components:–

1) *Flat file service*,
2) *Directory service* and
3) *Client module*.

The modules and their relationships are shown in Figure below.

The flat file service and the directory service each provide an interface for use by client programs, and their interfaces provide a set of operations for access to files.

The client module provides a single programming interface with operations on files similar to those found in conventional file systems.



File service architecture

Flat file service: The flat file service concerns 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 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.

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 giving 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 i.e., 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.

Client module: A client module runs in each client computer, 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.

Flat file service interface: The interface to the flat file service is given below.

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).

Flat File service Operations

A ***Field*** is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested.

All of the procedures in the interface except ***Create*** throw exceptions if the ***Field*** argument contains an invalid UFID or the user doesn't have sufficient access rights.

The ***Read*** operation copies the sequence of *n* data items beginning at item *i* from the specified file into *Data*, which is then returned to the client.

The ***Write*** operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position.

Create creates a new, empty file and returns the UFID that is generated.

Delete removes the specified file.

GetAttributes **and** ***SetAttributes*** enable clients to access the attribute record.

The interface to flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

Repeatable operations:- With the exception of ***Create,*** clients may repeat calls to which they receive no reply. Repeated execution of ***Create*** produces a different new file for each call.

Stateless servers:- The interface is suitable for implementation by *stateless* servers. Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

Directory service interface:-

The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service. The interface to the directory service is given as follows:

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*.

Directory Service Operations

- The ***Lookup*** operation in the directory service performs a single *Name UFID* translation.

- ***AddName*** adds an entry to a directory and increments the reference count field in the file's attribute record.

- ***UnName*** removes an entry from a directory and decrements the reference count.

- ***GetNames*** operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names.

Access control:-In distributed implementations, access rights checks have to be performed at the server. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Two alternative approaches can be adopted:

• 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 hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname*. The root has a distinguished name, and each file or directory has a name in a directory.

The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. 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. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

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.

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.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, 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

→PEER-TO-PEER SYSTEMS:-

Peer-to-peer systems represent a paradigm for the construction of distributed systems and applications in which data and computational resources are contributed by many hosts on the Internet, all of which participate in the provision of a uniform service. Their emergence is a consequence of the rapid growth of the Internet, including many millions of computers and similar numbers of users requiring access to shared resources.

A key problem for peer-to-peer systems is the placement of data objects across many hosts and provision for access to them in a manner that balances the workload and ensures availability without adding overheads.

The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for separately managed servers and their associated infrastructure.

Traditional client-server systems manage and provide access to resources such as files, web pages or other information objects located on a single server computer or a small cluster of tightly coupled servers. With such centralized designs, few decisions are required about the placement of the resources or the management of server hardware resources.

Peer-to-peer systems provide access to information resources located on computers throughout a network (whether it be the Internet or a corporate network). Algorithms for the placement and retrieval of information objects are a key aspect of the system design. The aim is to deliver a service that is fully decentralized and self-organizing, dynamically balancing the storage and processing loads between all the participating computers as computers join and leave the service.

Peer-to-peer systems share these characteristics:-

• Their design ensures that each user contributes resources to the system.

• Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.

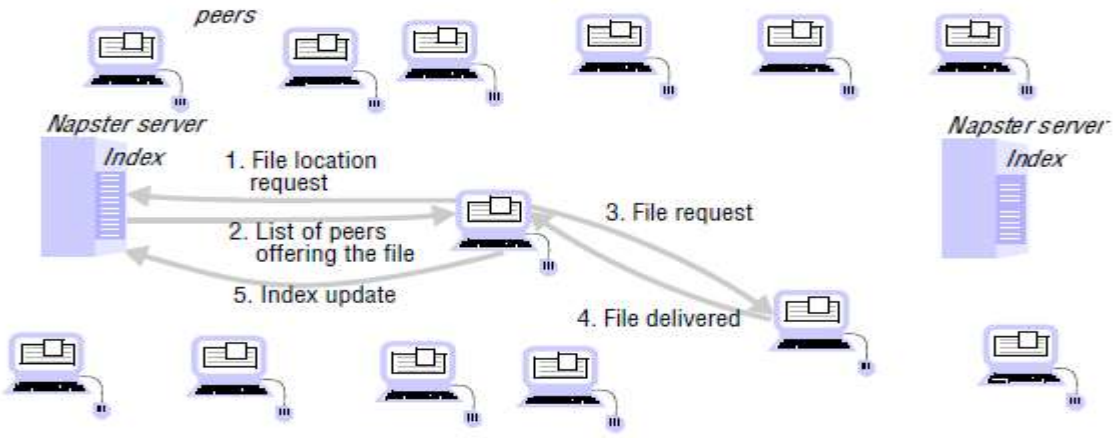
• Their correct operation does not depend on the existence of any centrally administered systems.

Three generations of peer-to-peer system and application development can be identified.

→The **first generation** was launched by the **Napster music exchange service.**
→A **second generation** of file sharing applications offering greater scalability, anonymity and fault tolerance quickly followed including **Freenet Gnutella, Kazaa and Bit Torrent.**
→The **Third Generation is Peer-to-peer middleware** such as **pasetry, tapestry.**

→Napster and its legacy:-

The first application in which a demand for a globally scalable information storage and retrieval service emerged was the downloading of digital music files. Napster file sharing system provided a means for users to share files. Napster became very popular for music exchange soon after its launch in 1999. Several million users were registered and thousands were swapping music files simultaneously. Napster's architecture included centralized indexes, but users supplied the files, which were stored and accessed on their personal computers. Napster's method of operation is illustrated by the sequence of steps shown in Figure below.



Napster: peer-to-peer file sharing with a centralized, replicated index

In step 5 clients are expected to add their own music files to the pool of shared resources by transmitting a link to the Napster indexing service for each available file. Thus the motivation for Napster and the key to its success was making available of a large, widely distributed set of files to users throughout the Internet.

Napster was shut down as a result of legal proceedings instituted against the operators of the Napster service by the owners of the copyright in some of the material (i.e., digitally encoded music) that was made available on it.

Limitations: Napster used a (replicated) unified index of all available music files. The requirement for consistency between the replicas was not strong, so this did not affect performance, but for many applications it would constitute a limitation. Unless the access path to the data objects is distributed, object discovery and addressing are likely to become a bottleneck.

Application dependencies: Napster took advantage of the special characteristics of the application for which it was designed in other ways:

• Music files are never updated, avoiding any need to make sure all the replicas of files remain consistent after updates.

• No guarantees are required concerning the availability of individual files – if a music file is temporarily unavailable, it can be downloaded later. This reduces the requirement for dependability of individual computers and their connections to the Internet.

→Peer-to-peer middleware:-

The third generation is characterized by the emergence of middleware layers for the management of distributed resources on a global scale. Several research teams have completed the development, evaluation and refinement of peer-to-peer middleware platforms and deployed them in a range of application services. The best-known and most fully developed examples include Pastry, Tapestry, CAN, Chord and Kademlia.

Resources are identified by globally unique identifiers (GUIDs), usually derived as a secure hash from the resource's state. The use of a secure hash makes a resource 'self certifying' i.e., clients receiving a resource can check the validity of the hash.

A key problem in the design of peer-to-peer applications is providing a mechanism to enable clients to access data resources quickly wherever they are located throughout the network. Napster maintained a unified index of available files for this purpose, giving the network addresses of their hosts. Second-generation peer-to-peer file storage systems such as Gnutella and Freenet employ partitioned and distributed indexes, but the algorithms used are specific to each system.

Peer-to-peer middleware systems are designed specifically to meet the need for the automatic placement and location of the distributed objects managed by peer-to-peer systems and applications.

Functional requirements: The function of the peer-to-peer middleware is to simplify the construction of services that are implemented across many hosts in a widely distributed network. To achieve this it must enable clients to locate and communicate with any individual resource made available to a service, even though the resources are widely distributed amongst the hosts.

Other important requirements include the ability to add new resources and to remove them and to add hosts to the service and remove them. Peer-to-peer middleware should offer a simple programming interface to application programmers that are independent of the types of distributed resource that the application manipulates.

Non-functional requirements: To perform effectively, peer-to-peer middleware must also address the following non-functional requirements.

- ***Global scalability***: Peer-to-peer middleware must be designed to support applications that access millions of objects on hundreds of thousands of hosts.

- ***Load balancing*****:** The performance of any system designed to exploit a large number of computers depends upon the balanced distribution of workload across them. This will be achieved by a random placement of resources together with the use of replicas of heavily used resources.

- ***Optimization for local interactions between neighboring peers*****:** The 'network distance' between nodes that interact has impact on the latency of individual interactions, such as client requests for access to resources. Network traffic loads are also impacted by it. The middleware should aim to place resources close to the nodes that access them the most.

- ***Accommodating to highly dynamic host availability*****:** Peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time. As hosts join the system, they must be integrated into the system and the load must be redistributed. When they leave the system whether voluntarily or involuntarily, the system must detect their departure and redistribute their load and resources.

- ***Security of data in an environment with heterogeneous trust*****:** In global-scale systems with participating hosts of diverse ownership, trust must be built up by the use of authentication and encryption mechanisms to ensure the integrity and privacy of information.

→Routing overlays:-

In peer-to-peer systems a distributed algorithm known as a *routing overlay* takes responsibility for locating nodes and objects. It is responsible for routing requests from any client

to a host that holds the object to which the request is addressed. The objects of interest may be placed at and relocated to any node in the network without client involvement. It is termed an overlay since it implements a routing mechanism in the application layer that is separate from any other routing mechanisms deployed at the network level such as IP routing.

The routing overlay ensures that any node can access any object by routing each request through a sequence of nodes, exploiting knowledge at each of them to locate the destination object. Peer-to-peer systems usually store multiple replicas of objects to ensure availability. In that case, the routing overlay maintains knowledge of the location of all the available replicas and delivers requests to the nearest 'live' node (i.e. one that has not failed) that has a copy of the relevant object.

The main task of a routing overlay is:

> ***Routing of requests to objects*:** A client wishing to invoke an operation on an object submits a request including the object's GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.

But the routing overlay must also perform some other tasks:

> ***Insertion of objects*:** A node wishing to make a new object available to a peer-to-peer service computes a GUID for the object and announces it to the routing overlay, which then ensures that the object is reachable by all other clients.
>
> ***Deletion of objects*:** When clients request the removal of objects from the service the routing overlay must make them unavailable.
>
> ***Node addition and removal*:** Nodes (i.e., computers) may join and leave the service. When a node joins the service, the routing overlay arranges for it some of the responsibilities of other nodes. When a node leaves, its responsibilities are distributed among the other nodes.

Uniqueness is verified by searching for another object with the same GUID. A hash function is used to generate the GUID from the object's value. Because these randomly distributed identifiers are used to determine the placement of objects and to retrieve them, overlay routing systems are sometimes described as ***distributed hash tables*** **(DHT).** This is reflected by the simplest form of API used to access them, as shown in Figure below.

With this API, the ***put()*** operation is used to submit a data item to be stored together with its GUID. The DHT layer takes responsibility for choosing a location for it, storing it (with replicas to ensure availability) and providing access to it through the ***get()*** operation.

put(GUID, data)
Stores *data* in replicas at all nodes responsible for the object identified by *GUID*.

remove(GUID)
Deletes all references to *GUID* and the associated data.

value = get(GUID)
Retrieves the data associated with *GUID* from one of the nodes responsible for it.

Interface for a distributed hash table (DHT) as implemented by the Pastry

A slightly more flexible form of API is provided by a ***distributed object location and routing* (DOLR) layer**, as shown in Figure below.

With this interface objects can be stored anywhere and the DOLR layer is responsible for maintaining a mapping between object identifiers (GUIDs) and the addresses of the nodes at which replicas of the objects are located.

publish(GUID)
GUID can be computed from the object (or some part of it, e.g., its name). This function makes the node performing a *publish* operation the host for the object corresponding to *GUID*.

unpublish(GUID)
Makes the object corresponding to *GUID* inaccessible.

sendToObj(msg, GUID, [n])
Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter *[n]*, if present, requests the delivery of the same message to *n* replicas of the object.

Interface for distributed object location and routing (DOLR) as implemented by Tapestry

GUIDs are not human-readable, so client applications must obtain the GUIDs for resources of interest through some form of indexing service using human-readable names or search requests. Generally, these indexes are also stored in a peer-to-peer manner to overcome the weaknesses of centralized indexes evidenced by Napster. In **BitTorrent** a web index search leads to a file containing details of the desired resource, including its GUID and the URL of a ***tracker*** (a host that holds an up-to-date list of network addresses for providers willing to supply the file).

→**Overlay routing versus IP routing:** Routing overlays share many characteristics with the IP packet routing infrastructure, the primary communication mechanism of the Internet.

It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems. The answer lies in several distinctions that are identified in Figure below.

	IP	*Application-level routing overlay*
Scale	IPv4 is limited to 2^{32} addressable nodes. The IPv6 namespace is much more generous (2^{128}), but addresses in both versions are hierarchically structured and much of the space is preallocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID namespace is very large and flat (>2^{128}), allowing it to be much more fully occupied.
Load balancing	Loads on routers are determined by network topology and associated traffic patterns.	Object locations can be randomized and hence traffic patterns are divorced from the network topology.
Network dynamics (addition/deletion of objects/nodes)	IP routing tables are updated asynchronously on a best-effort basis with time constants on the order of 1 hour.	Routing tables can be updated synchronously or asynchronously with fractions-of-a-second delays.
Fault tolerance	Redundancy is designed into the IP network by its managers, ensuring tolerance of a single router or network connectivity failure. *n*-fold replication is costly.	Routes and object references can be replicated *n*-fold, ensuring tolerance of *n* failures of nodes or connections.

Distinctions between IP and overlay routing for peer-to-peer applications

→Case Studies:-

SUN NFS:

Sun Microsystems's Network File System(NFS) has been widely adopted since its introduction in 1985. The design and development of NFS was undertaken by Sun Microsystems in 1984. The design and implementation of NFS have achieved success both technically and commercially.

NFS provides access to remote files for client programs. The client-server relationship is symmetrical i.e. each computer in an NFS network can act as both a client and a server. Any computer can be a server, exporting some of its files and a client, accessing files on other machines.

To adopt NFS as a standard, the definitions of the key interfaces were placed in the public domain. The source code for a reference implementation was made available to other computer vendors under license.

Andrew File System:

The design of Andrew File System was made to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.AFS was initially implemented on servers running UNIX and later it was made available in public domain versions.