

Unit-6 Distributed file Systems

Syllabus: Introduction, File service Architecture, PEER- to-PEER Systems, Napster and its Legacy, PEER-to-PEER, Middle ware, Routing Overlays

Topic 01: INTRODUCTION

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- Distributed file systems support the sharing of information in the form of files and hardware resources.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Figure 1 provides an overview of types of storage system.
- Distributed file systems support the sharing of information in the form of files and hardware resources.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Figure 1 provides an overview of types of storage system.

	<i>Sharing</i>	<i>Persis- tence</i>	<i>Distributed cache/replicas</i>	<i>Consistency maintenance</i>	<i>Example</i>
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (Ch. 18)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	✓	OceanStore(Ch. 10)

Figure 1. Storage systems and their properties

Types of consistency between copies: 1 - strict one-copy consistency
 ✓ - approximate consistency
 X - no automatic consistency

- Figure 2 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:	disk I/O and buffering

Figure 2. File system modules

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- Files contain both data and attributes.
- A typical attribute record structure is illustrated in Figure 3.

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

Figure 3. File attribute record structure

- Figure 4 summarizes the main operations on files that are available to applications in UNIX systems.

Figure 4. UNIX file system operations

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

- Distributed File system requirements
 - Related requirements in distributed file systems are:
 - ❖ Transparency
 - ❖ Concurrency
 - ❖ Replication
 - ❖ Heterogeneity
 - ❖ Fault tolerance
 - ❖ Consistency
 - ❖ Security
 - ❖ Efficiency

Topic 02: 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
 - A client module.
- The relevant modules and their relationship is shown in Figure 5.

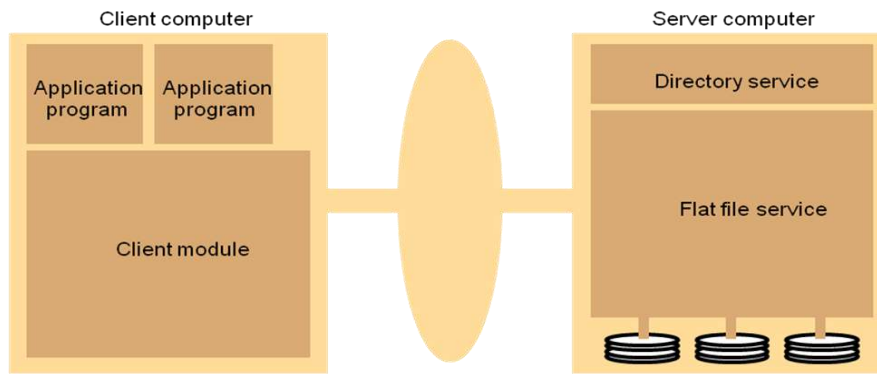


Figure 5. File service architecture

- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:
 - Flat file service:
 - ❖ Concerned with the implementation of operations on the contents of file. 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 unique among all of the files in a distributed system.
 - Directory service:
 - ❖ Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.
 - Client module:
 - ❖ It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
 - ❖ It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.
 - Flat file service interface:
 - ❖ Figure 6 contains a definition of the interface to a flat file service.

<i>Read(FileId, i, n) -> Data</i>	if $1 \leq i \leq \text{Length}(\text{File})$: Reads a sequence of up to n items from a file starting at item i and returns it in <i>Data</i> .
-throws <i>BadPosition</i>	
<i>Write(FileId, i, Data)</i>	if $1 \leq i \leq \text{Length}(\text{File}) + 1$: Write a sequence of <i>Data</i> to a file, starting at item i , extending the file if necessary.
-throws <i>BadPosition</i>	
<i>Create() -> FileId</i>	Creates a new file of length 0 and delivers a UFID for it.
<i>Delete(FileId)</i>	Removes the file from the file store.
<i>GetAttributes(FileId) -> Attr</i>	Returns the file attributes for the file.
<i>SetAttributes(FileId, Attr)</i>	Sets the file attributes (only those attributes that are not shaded in Figure 3.)

Figure 6. Flat file service operations

- Access control
 - ❖ In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
- Directory service interface
 - ❖ Figure 7 contains a definition of the RPC interface to a directory service.

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

Figure 7. Directory service operations

- 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.

➤ File Group

- ❖ A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.
 - A similar construct is used in a UNIX file system.
 - It helps with distributing the load of file serving between several servers.
 - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).

To construct a globally unique ID we use some unique attribute of the machine on which it is created, e.g. IP number, even though the file group may move subsequently.

File Group ID:



Topic 03: **PEER-TO-PEER SYSTEMS**

- Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the Internet and other networks in ever-increasing numbers.

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.
- They can be designed to offer a limited degree of anonymity to the providers and users of resources.
- A key issue for their efficient operation is the choice of an algorithm for the placement of data across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads.

Peer-to-peer middleware

The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed resources on a global scale. Several

research teams have now completed the development, evaluation and refinement of peer-to-peer middleware platforms and demonstrated or deployed them in a range of application services. The use of peer-to-peer systems for applications that demand a high level of availability for the objects stored requires careful application design to avoid situations in which all of the replicas of an object are simultaneously unavailable. There is a risk of this for objects stored on computers with the same ownership, geographic location, administration, network connectivity, country or jurisdiction

	IP	Application-level routing overlay
<i>Scale</i>	IPv4 is limited to 232 addressable nodes. The IPv6 name space is much more generous (2128), but addresses in both versions are hierarchically structured and much of the space is pre-allocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID name space is very large and flat (>2128), allowing it to be much more fully occupied.
<i>Load balancing</i>	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.
<i>Network dynamics (addition/deletion of objects/nodes)</i>	IP routing tables are updated asynchronously on a best-efforts basis with time constants on the order of 1 hour.	Routing tables can be updated synchronously or asynchronously with fractions of a second delays.
<i>Fault tolerance</i>	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.
<i>Target identification</i>	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
<i>Security and anonymity</i>	Addressing is only secure when all nodes are trusted. Anonymity for the owners of addresses is not achievable.	Security can be achieved even in environments with limited trust. A limited degree of anonymity can be provided.

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

Overlay routing versus IP routing: At first sight, routing overlays share many characteristics with the IP packet routing infrastructure that constitutes 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.

Distributed computation: The exploitation of spare computing power on end-user computers has long been a subject of interest and experiment. More recently, much larger numbers of computers have been put to use to perform several scientific calculations that require almost unlimited quantities of computing power.

Topic 04: Napster and its legacy

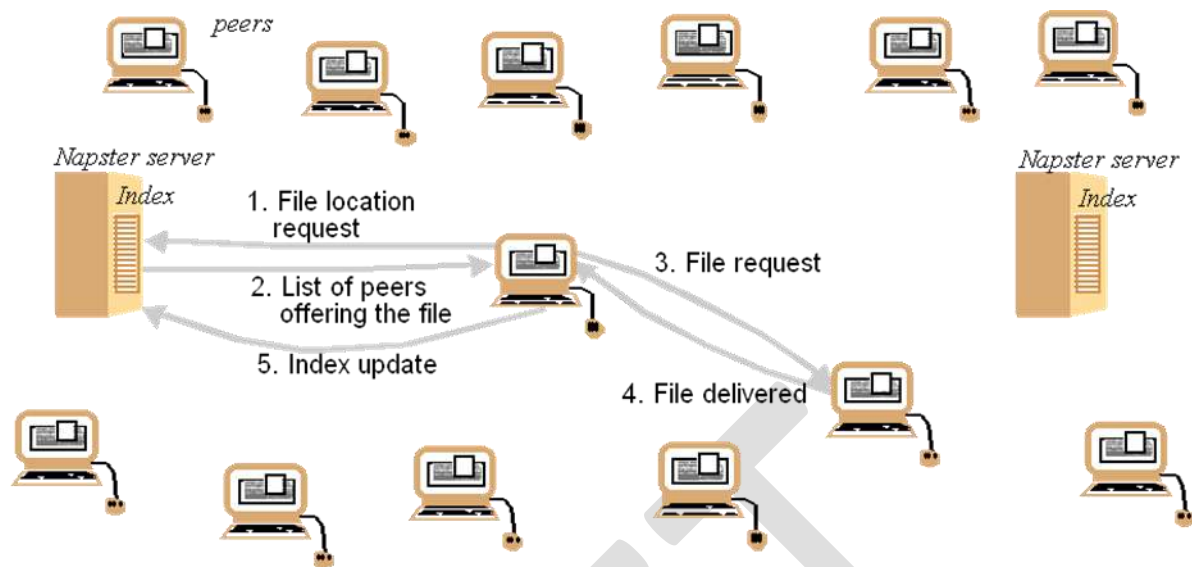


Figure : Napster: peer-to-peer file sharing with a centralized, replicated index

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 (above figure). 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 the making available of a large, widely distributed set of files to users throughout the Internet, by providing access to 'shared resources at the edges of 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.

Lessons learned from Napster • Napster demonstrated the feasibility of building a useful large-scale service that depends almost wholly on data and computers owned by ordinary Internet users. To avoid swamping the computing resources of individual users (for example, the first user to offer a chart-topping song) and their network connections, Napster took account of network locality – the number of hops between the client and the server – when allocating a server to a client requesting a song.

Limitations: Napster used a (replicated) unified index of all available music files. For the application in question, the requirement for consistency between the replicas was not strong, so this did not hamper performance, but for many applications it would constitute a limitation.

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

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

Functional requirements: The function of 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 at will and to add hosts to the service and remove them.

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

- ***Global scalability:*** One of the aims of peer-to-peer applications is to exploit the hardware resources of very large numbers of hosts connected to the Internet. Peer-to-peer middleware must therefore be designed to support applications that access millions of objects on tens of thousands or 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. For the systems we are considering, 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 a substantial impact on the latency of individual interactions, such as client requests for access to resources. Network traffic loadings are also impacted by it.
- ***Accommodating to highly dynamic host availability:*** Most peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time. The hosts and network segments used in peer-to-peer systems are not owned or managed by any single authority; neither their reliability nor their continuous

participation in the provision of a service is guaranteed.

- **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.
- **Anonymity, deniability and resistance to censorship:** Anonymity for the holders and recipients of data is a legitimate concern in many situations demanding resistance to censorship. A related requirement is that the hosts that hold data should be able to plausibly deny responsibility for holding or supplying it.

Routing overlays

- In peer-to-peer systems a distributed algorithm known as a *routing overlay* takes responsibility for locating nodes and objects.
- The name denotes the fact that the middleware takes the form of a layer that 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 subsequently 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 quite separate from any other routing mechanisms deployed at the network level such as IP routing.
- 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 that has a copy of the relevant object.

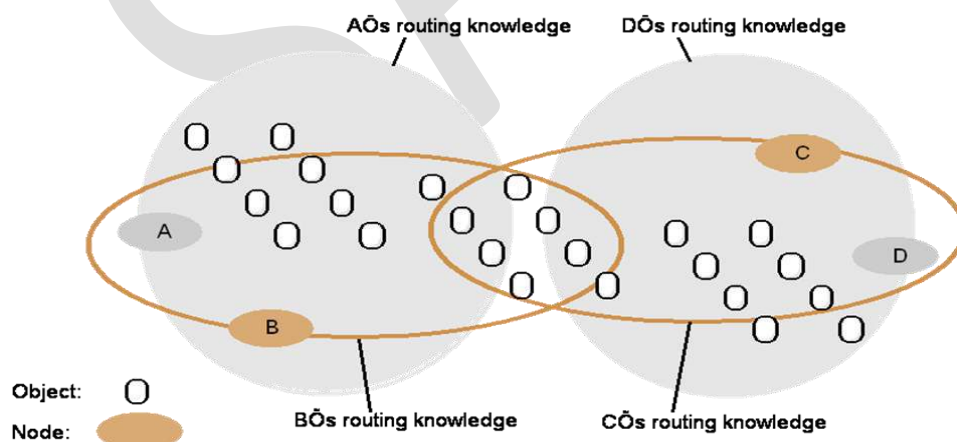


Fig: Distribution of information in a routing overlay

The main task of a routing overlay is the following:

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 to assume some of the responsibilities of other nodes. When a node leaves (either voluntarily or as a result of a system or network fault), its responsibilities are distributed amongst the other nodes.

put(GUID, data)

The *data* is stored 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)

The data associated with *GUID* is retrieved from one of the nodes responsible it.

Figure 6.4 : Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

An object's GUID is computed from all or part of the state of the object using a function that delivers a value that is, with very high probability, unique. 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 6.4

With this API, the `put()` operation is used to submit a data item to be stored together with its GUID. A slightly more flexible form of API is provided by a distributed object location and routing (DOLR) layer, as shown in Figure 6.5. 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. Objects may be replicated and stored with the same GUID at different hosts, and the routing overlay takes responsibility for routing requests.

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*])

Figure 6.5 Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

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.