DS UNIT 3: PEER TO PEER SERVICES AND FILE SYSTEMS

Definition of a peer-to-peer system:

> 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.

→ They exploit existing naming, routing, data replication and security techniques in new ways.

→ **Applications that exploit resources available at the edges of the Internet – storage, cycles, content, human presence'**

→ Management requires knowledge of all hosts, their accessibility, (distance in number of hops), availability and performance.

Characteristic of peer-to-peer systems:

- Each computer contributes resources
- All the nodes have the same functional capabilities and responsibilities
- No centrally-administered system
- Offers a limited degree of anonymity
- Algorithm for placing and accessing the data
  - Balance workload, ensure availability
  - Without adding undue overhead

Goal of peer-to-peer system:

Load balancing: Get rid of central servers, less load on one node in the network.

Fault Tolerance: No single point of failure, if the server goes down the network can still carry on.

Efficient use of resources: There are often lots of wasted resources on network (spare file space, spare computation power ....).

Evolution of peer-to-peer systems:

1. **Napster:** founded as a pioneering peer-to-peer (P2P) file sharing Internet service that emphasized sharing audio files, typically music, encoded in MP3 format.
2. **Freenet, Gnutella, Kazaa and BitTorrent -** use a decentralized distributed data store to keep and deliver information and files.
3. **Pastry, Tapestry, CAN, Chord, Kademlia:** The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed resources on a global scale.
4. Resources are identified by globally unique identifiers (GUIDs), usually derived as a secure hash.

Benefits of P2P systems:

- ❒ Ability to exploit unused resources (storage, processing) in the host computers
- ❒ Scalability to support large numbers of clients and hosts with load balancing of network links and host computer resources
- ❒ Self-organizing properties of the middleware platforms reduces costs

Drawbacks of P2P systems:

- β Costly for the storage of mutable data compared to trusted, centralized service
- β Cannot yet guarantee anonymity to hosts

Difference between IP and peer-to-peer applications:

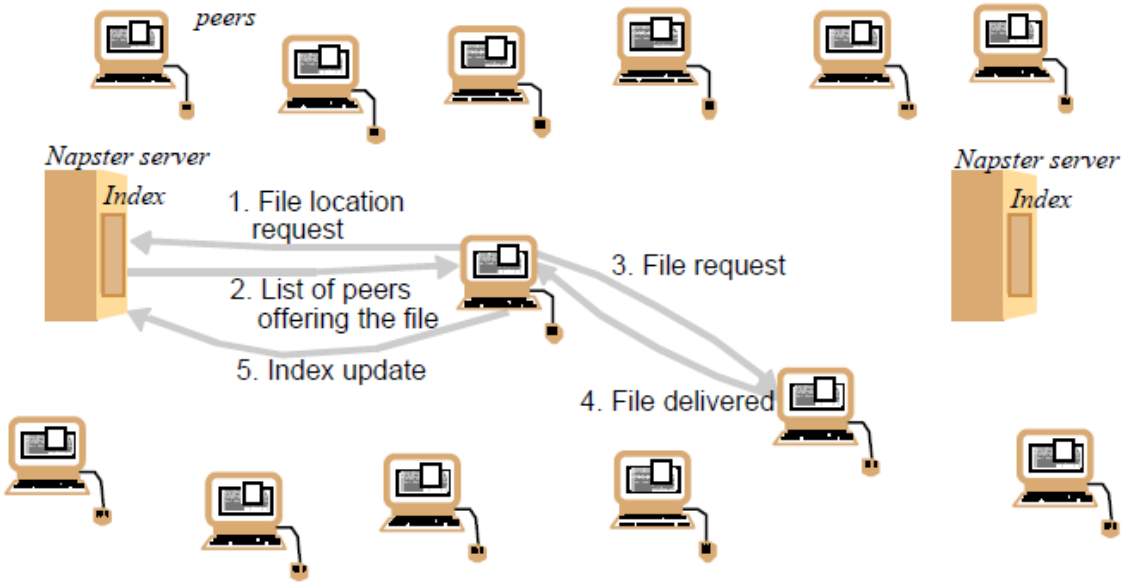
	IP	*Application-level routing overlay*
Scale	IPv4 is limited to 2^{32} addressable nodes. The IPv6 namespace is much more generous (2^{128}), but addresses in both versions are hierarchically structured and much of the space is preallocated according to administrative requirements.	Peer-to-peer systems 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.
Target identification	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
Security and anonymity	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.

3.3 NAPSTER AD 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.
- ❒ Proved feasibility of a service using hardware and data owned by ordinary Internet users
- ❒ Napster became very popular for music exchange soon after its launch in 1999. At its peak, several million users were registered and thousands were swapping music files simultaneously.

Napster's architecture and working:

- ★ Napster's architecture included **centralized indexes**, but users supplied the files, which were stored and accessed on their personal computers.
- ★ It has a Centralised server
- ★ Napster demonstrated the feasibility of building a useful large-scale service that depends almost wholly on data and computers owned by ordinary Internet users.
- ★ Napster's method of operation is illustrated by the sequence of steps shown in figure below:



- ★ **Each node registers list of files** that is has to the central server.
- ★ When a node wishes to retrieve a file, it requests from the central server a list of client nodes that have that file. **(1)**
- ★ Once the server provides the list of peers offering the file, the client picks a node from which to download the file and requests it. **(2)(3)**

- ★ The selected peer delivers the file to the requesting client. **(4).**
- ★ 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. **(5)**

Network locality concept in Napster:

To avoid swamping the computing resources of individual users 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 of Napster:

- ► Napster used a (replicated) unified index of all available music files. Unless the access path to the data objects is distributed, object discovery and addressing are likely to become a bottleneck.
- ► Single point of failure – if the centralised server fails.
- ► Client only downloads from one other client at a time.

Second generation P2P: Gnutella

- Again files are distributed across the network
- But no central server
- A node must know the IP address of at least one other Gnutella node. Clients initialised with a set of working nodes
- Each node request each node in its working set
- If a node receives a request either:
  - The file is there
  - Otherwise the request is propagated on
- Requests have a lifetime TTL (Time to live).

Problem with Gnutella:

The network has more request messages getting flooded. One solution for this problem would be to do a **random walk** from one node to next node but again, it can take a can take a long time to find the file.

3.4 PEER-TO-PEER MIDDLEWARE

Need for peer-to-peer middleware:

To provide a mechanism to enable clients to access data resources quickly and dependably 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.

Functional requirements of P2P middleware:

- Simplify construction of services across many hosts in wide network
- Add and remove resources at will
- Add and remove new hosts at will
- Interface to application programmers should be simple and independent of types of distributed resources

Non-functional requirements of P2P middleware:

→ **Global scalability:** 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:** 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 neighbouring peers:** The middleware should aim to place resources close to the nodes that access the resources the most.

→ **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.

→ **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.

- As hosts join the system, they must be integrated into the system and the load must be redistributed to exploit their resources.

- When they leave the system whether voluntarily or involuntarily, the system must detect their departure and redistribute their load and resources.

Challenge in designing P2P middleware-

1. The requirements for scalability and availability make it infeasible to maintain a database at all client nodes giving the locations of all the resources (objects) of interest.
2. Each node is made responsible for maintaining detailed knowledge of the locations of nodes and objects in a portion of the namespace as well as a general knowledge of the topology of the entire namespace as shown in the diagram below:

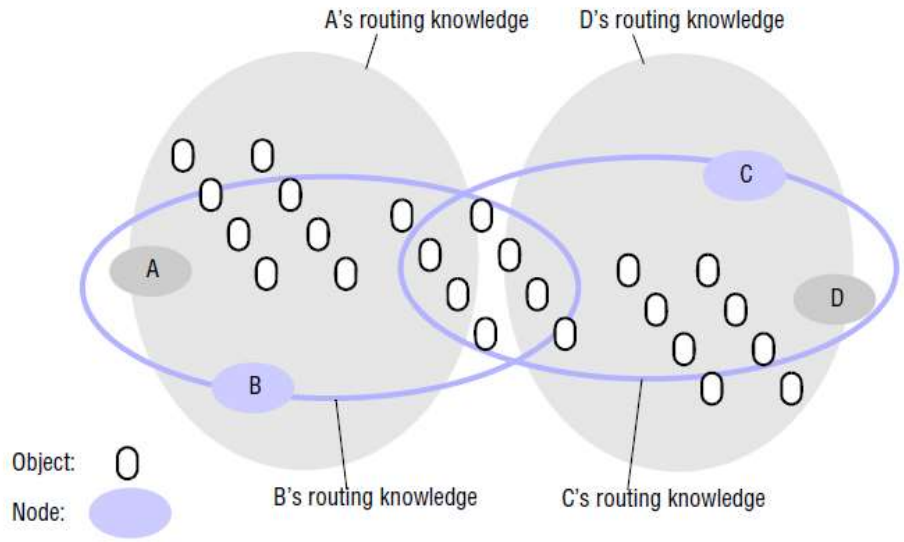


Fig: Distribution of information in a routing overlay

3.5 ROUTING OVERLAYS

Routing Overlays

- Routing Overlays
  - Responsible for locating nodes and objects
  - Implements a routing mechanism in the application layer
    - Separate from any other routing mechanisms such as IP routing
  - Ensures that any node can access any object by routing each request through a sequence of nodes
    - Exploits knowledge at each node to locate the destination

Routing Overlays

- GUIDs
  - ‘pure’ names or opaque identifiers
    - Reveal nothing about the locations of the objects
    - Building blocks for routing overlays
  - Computed from all or part of the state of the object using a function that deliver a value that is very likely to be unique. Uniqueness is then checked against all other GUIDs
  - Not human readable

Routing Overlays

- Tasks of a routing overlay
  - **Routing Request to Objects**: Client submits a request including the object GUID, routing overlay routes the request to a node at which a replica of the object resides
  - **Insertion of Objects**: A node introduces a new object by computing its GUID and announces it to the routing overlay
  - **Deletion of Objects**: Clients can remove an object
  - **Node addition and removal**: Nodes may join and leave the service

Routing Overlays

- Types of Routing Overlays
  - DHT – Distributed Hash Tables
  - DHT – GUIDs are stored based on the hash value
    - (128 bit hash value using SHA-1 algorithm)
  - DOLR – Distributed Object Location and Routing
  - DOLR is a layer over the DHT that maps GUIDs and address of nodes at which replicas of objects are located. put() and get() APIs
  - DOLR – GUIDs host address is notified using the Publish() operation

Peer-to-Peer Systems Case Study: Pastry & Tapestry

Reference: George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012

Case Studies: Pastry

- Nodes & objects assigned 128 bit GUID computed by applying Secure Hash Algorithm to node's public key or object's name or stored state.
- GUIDs randomly distributed in range 0 to 2^128 -1
- Provide no clue to value from which computed
- Clashes between GUIDs for different nodes are unlikely
- If GUID identifies node currently active, message delivered to it else to node whose GUID is numerically closest
- Delivery in O(log N) steps
- Routing uses underlying transport to transfer message to node closer to destination (may involve many IP hops)
- Pastry uses locality metric based on hop count or delay in underlying network to select appropriate neighbors when setting up routing tables

Distributed Hash Table API in Pastry

put(GUID, data)

- The *data is stored in replicas at all nodes responsible for the object* identified by *GUID.*

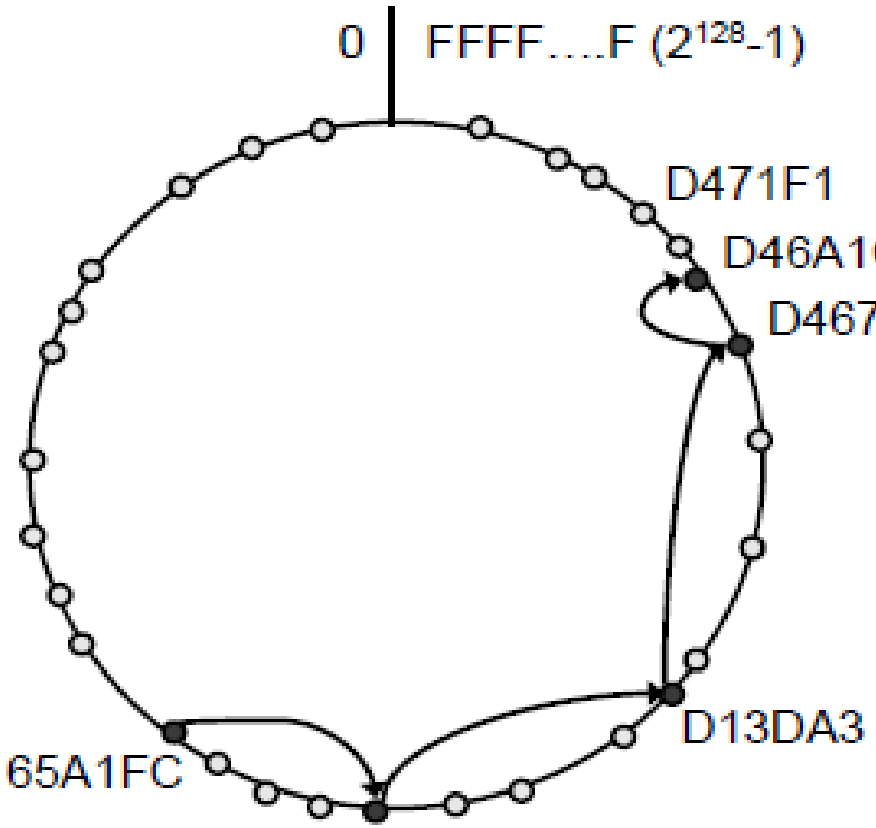
remove(GUID)

- 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 for it.

Simple Pastry Routing Algorithm



- Each node stores leaf set – vector L (of size 2l) containing GUIDs and IP addresses of l nearest nodes above and l below its GUID
- GUID space is circular: 0's neighbour is 2^{128} -1
- The dots depict live nodes.
- The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 (l = 4).
- This is a degenerate type of routing that would scale very poorly; it is not used in practice.

Simple Pastry Routing Algorithm

- Each Pastry node maintains a tree structured routing table giving GUIDs and IP addresses for a set of nodes spread throughout the entire range of 2^128 possible values with increased density of coverage for GUIDs numerically close to its own.
- For GUIDs represented as hexadecimal numbers, routing table has as many rows as hex digits in a GUID = 128/4 = 32 rows
- Any row has 15 entries - one for each possible value of the nth hex digit excluding the value in the local node's GUID
- Each entry in table points to one of the potentially *many* nodes whose GUIDs have relevant prefix

First 4 Rows of a Pastry Routing Table

p =	GUID prefixes and corresponding routeHandles *n*															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	n	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6F	6E	6F
	n	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E	65F
	n	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*		*n*	*n*	*n*	*n*	*n*
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE	65AF
	n		*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*

The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal. The n's represent [GUID, IP address] pairs specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded entries indicate that the prefix matches the current GUID up to the given value of p: the next row down or the leaf set should be examined to find a route. Although there are a maximum of 128 rows in the table, only $\log_{16} N$ rows will be populated on average in a network with N active nodes.

Pastry Routing Algorithm

if (destination is within range of our leaf set)

forward to numerically closest member

else

if (there is a longer prefix match in table)

forward to node with longest match

else

forward to node in table which

(a) has a common prefix of length p and

(b) GUID that is numerically closer.

Tapestry

- Tapestry implements a distributed hash table and routes messages to nodes based on GUIDs associated with resources using prefix routing in a manner similar to Pastry.
- Tapestry applications give additional flexibility:
- They can place replicas close (in network distance) to frequent users of resources in order to reduce latency and
- Minimize network load or to ensure tolerance of network and host failures

Distributed Object Location and Routing in Tapestry

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 as host for* the object corresponding to *GUID.*

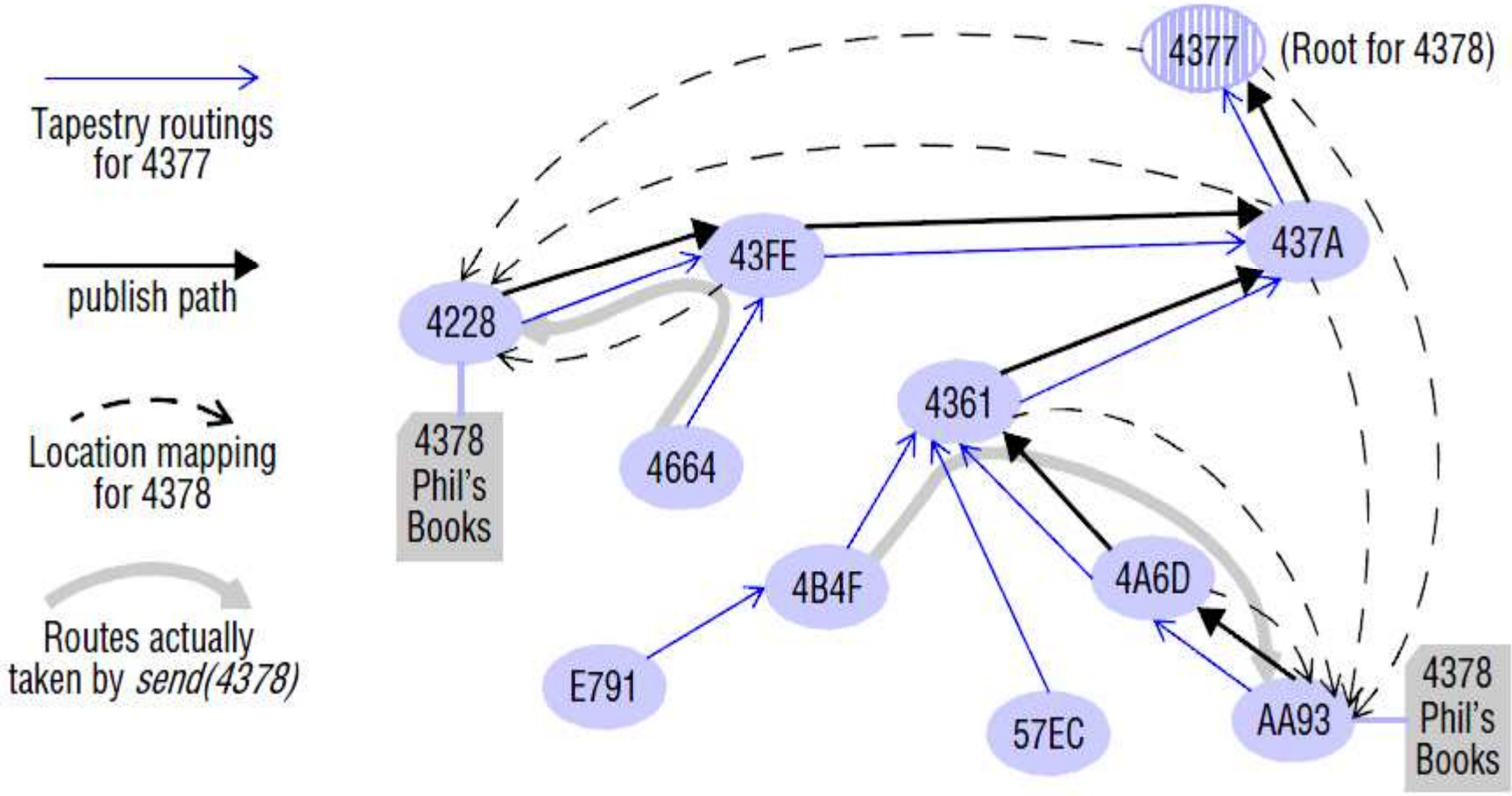
unpublish(GUID)

- 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.

Distributed Object Location and Routing in Tapestry



Structured Vs Unstructured P2P

	Structured peer-to-peer	*Unstructured peer-to-peer*
Advantages	Guaranteed to locate objects (assuming they exist) and can offer time and complexity bounds on this operation; relatively low message overhead.	Self-organizing and naturally resilient to node failure.
Disadvantages	Need to maintain often complex overlay structures, which can be difficult and costly to achieve, especially in highly dynamic environments.	Probabilistic and hence cannot offer absolute guarantees on locating objects; prone to excessive messaging overhead which can affect scalability.

Strategies for Search

- In P2P file sharing, all nodes in the network offer files to other nodes.
- Searching for a file in unstructured P2P network follows following strategies.

1. Expanded Ring Search
2. Random Walks
3. Gossiping.

Thank You

DISTRIBUTED FILE SYSTEM

INTRODUCTION

A file system is a subsystem of an operating system that performs the file management activities such as organisation, storing, retrieval, naming, sharing and protection of files.

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

Figure 12.1 Storage systems and their properties

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

Figure 12.2 File system modules

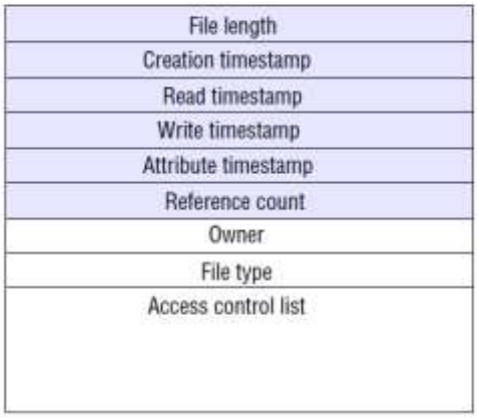
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk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

Characteristics of a file system:

1) File systems responsible for the organization, storage, retrieval, naming, sharing and protection of files.

2) Programming interface that characterizes the file abstraction.
3) Files contain both **data** and **attributes.**
4) **Metadata –** extra information stored by a file system for management of files.
5) A typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

Figure 12.3 File attribute record structure



6) A directory is a file, often of a special type, that provides a mapping from text names to internal file identifiers.

File System Operations:

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.

Figure 4. UNIX file system operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	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)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	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)	Gets the file attributes for file *name* into *buffer*.

Definition of a DFS:

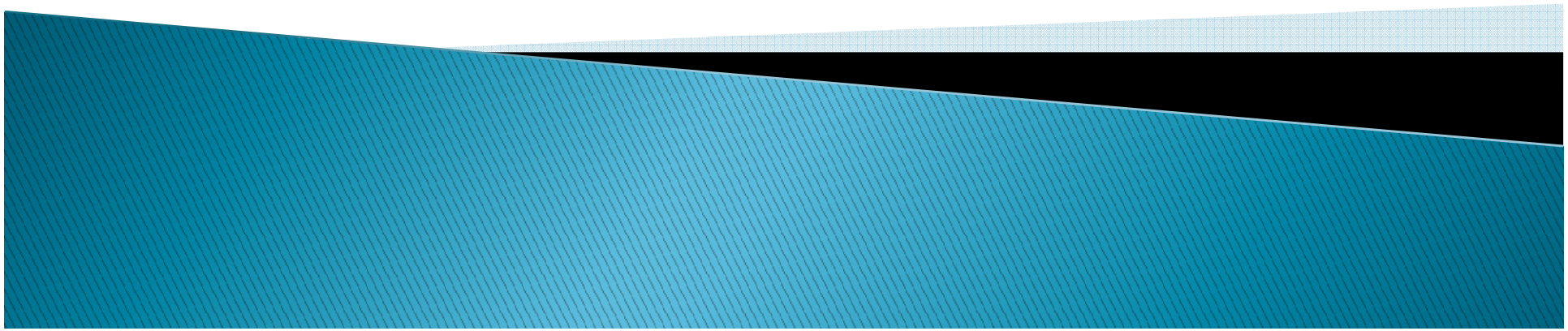
A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network i.e. **Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet.**

A distributed file system normally supports the following desirable properties:

- ★ **Remote Information Sharing:** A DFS allows a file to be transparently accessed by processes of any node of the system irrespective of the file's location.
- ★ **User mobility:** User mobility indicates that a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different times.
- ★ **Availability:** For better tolerance, files should be available for use even in the event of temporary failure of one or many nodes in the system.
- ★ **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.
- ★ **Data integrity**
- ★ **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 with digital signatures and (optionally) encryption of secret data.
- ★ **Heterogeneity**
- ★ **Transparency:** There are four types of transparency:
    - o Structure Transparency
    - o Access Transparency
    - o Naming Transparency
    - o Replication transparency
- ★ **Performance:** performance of a distributed file system equals avg amount of time needed to satisfy client request + network communication overhead
- ★ **Scalability:** Scalable growth must withstand high service load and accommodate growth of the user community

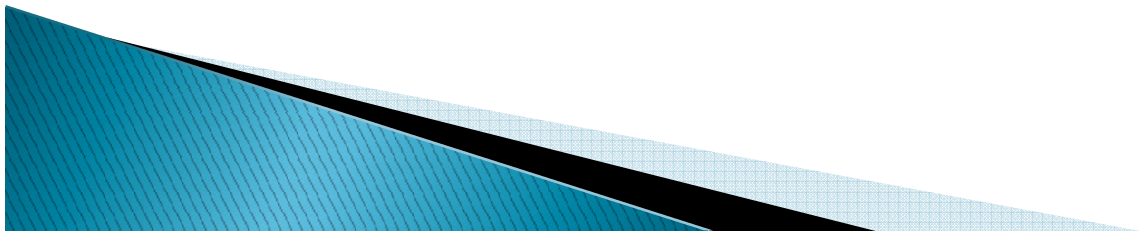
FILE SERVICE ARCHITECTURE

By
M.Roshini
IIIrd year
CSE

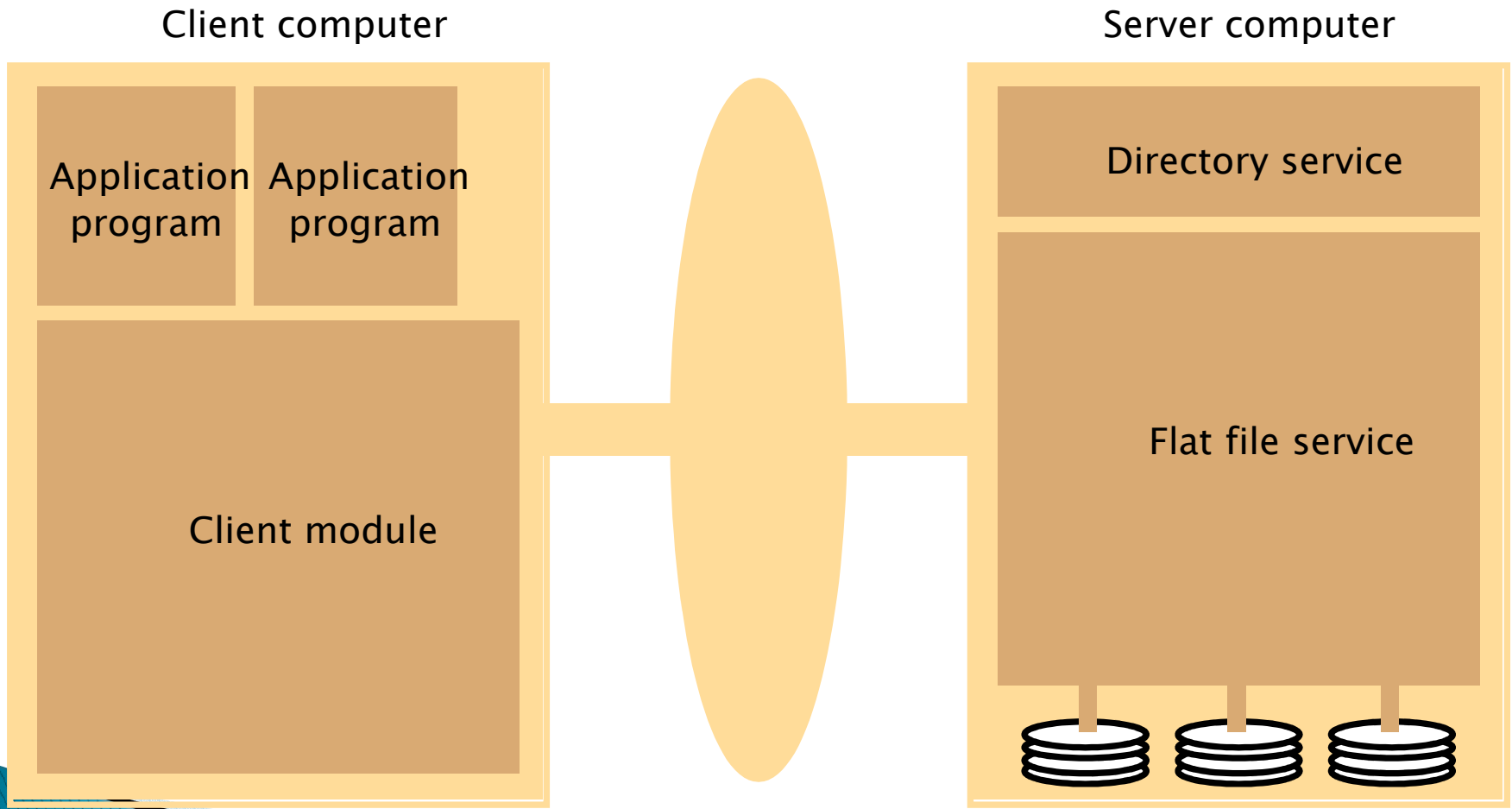


FILE SERVICE ARCHITECTURE

- 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

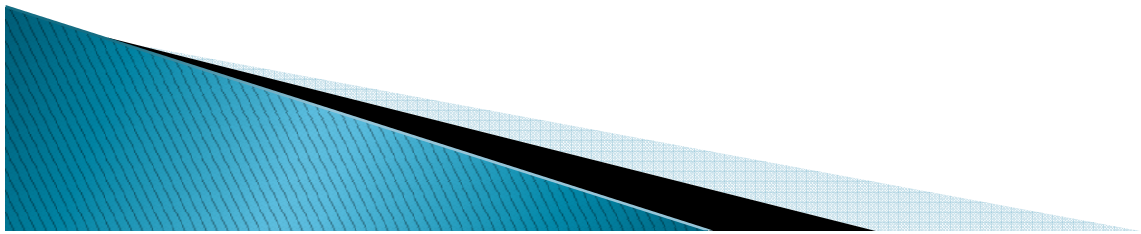


FILE SERVICE ARCHITECTURE



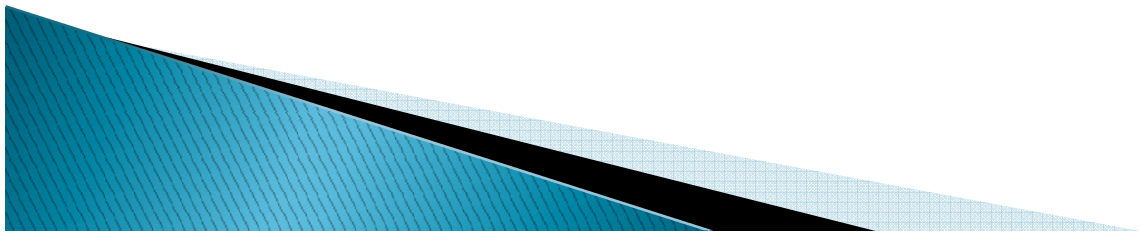
FILE SERVICE ARCHITECTURE

- 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.



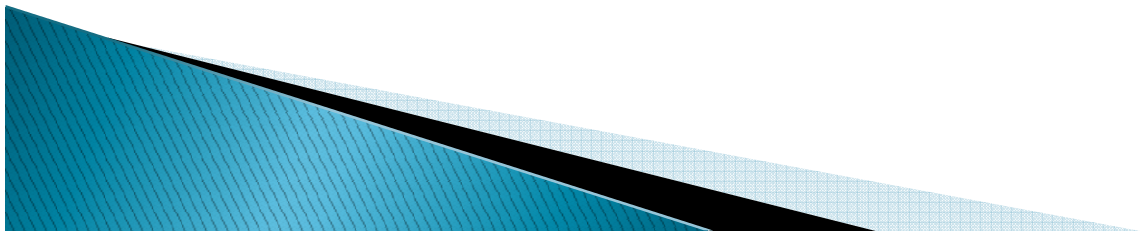
FILE SERVICE ARCHITECTURE

- 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.



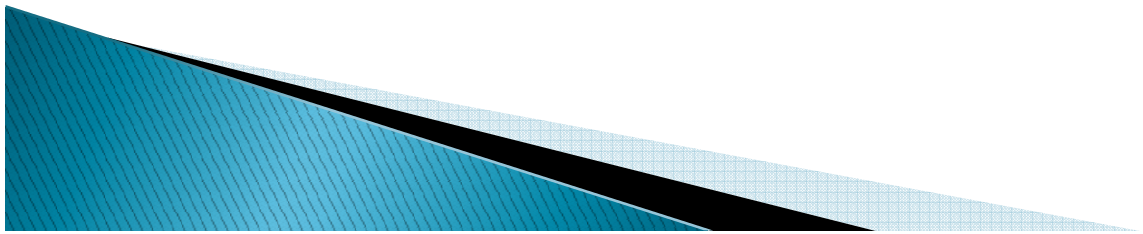
FILE SERVICE ARCHITECTURE

- 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.



FILE SERVICE ARCHITECTURE

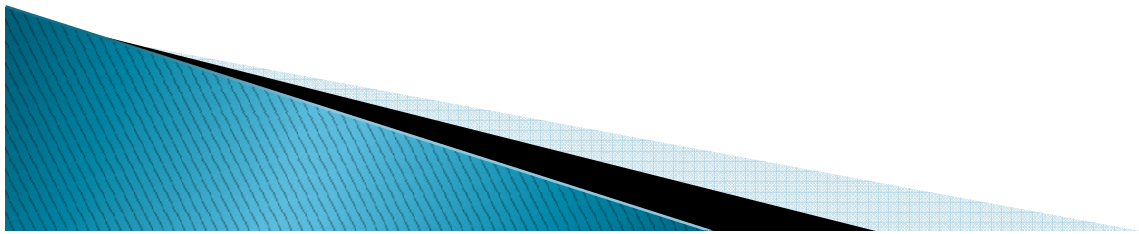
- Flat file service interface:
  - It contains a definition of the interface to a flat file service



FILE SERVICE ARCHITECTURE

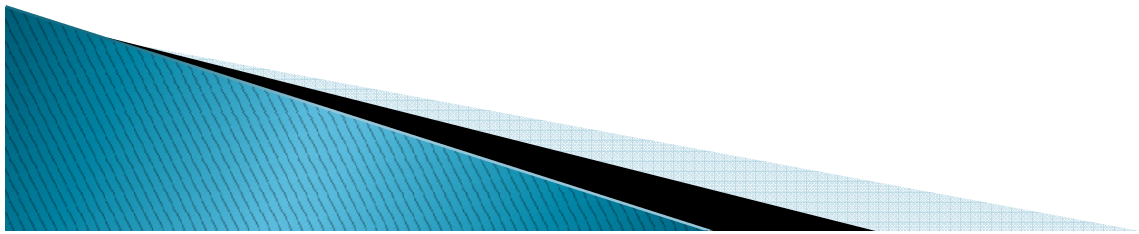
Figure 12.6 Flat file service operations

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).



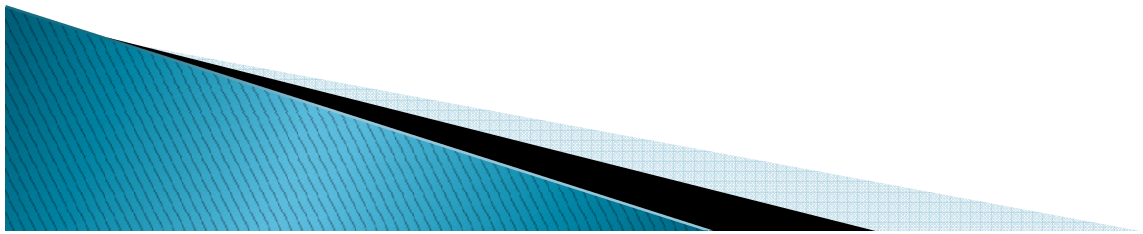
SPECIFICATIONS

- ‘i’ –specify the position in the file.
- 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.
- Write operation-copies the sequence of n data items in Data into the specified file beginning at item i,replacing the previous content of the file.



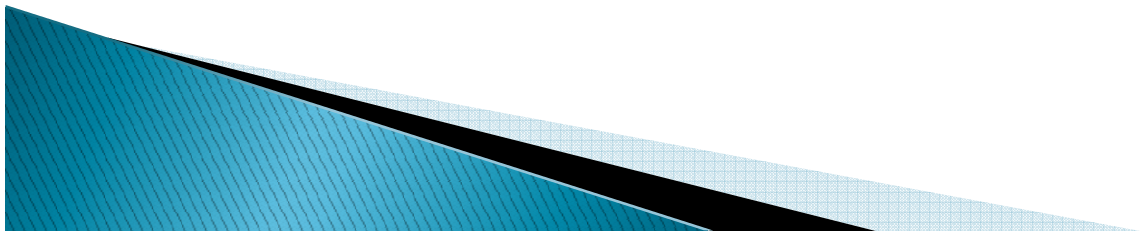
REASONS FOR DIFFERENCE BETWEEN FLAT FILE SERVICE INTERFACE & UNIX FILE SYSTEM INTERFACE

- Repeatable operations-client can repeat the calls in which they receive no reply.
- Stateless servers-restarted after a failure and resume operation without any need for client and server to restore any state.



FILE SERVICE ARCHITECTURE

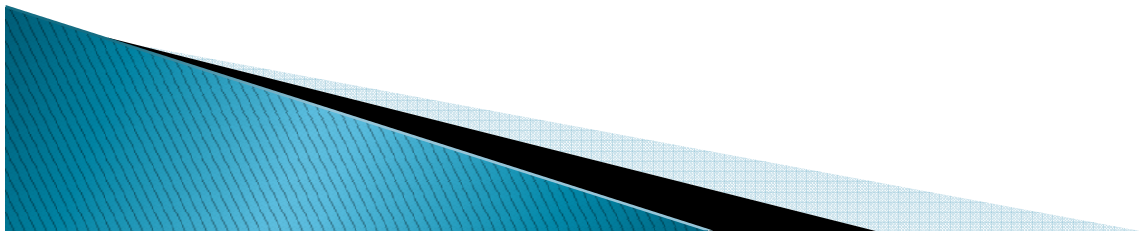
- Access control
  - In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
- Directory service interface
  - The following figure contains a definition of the RPC interface to a directory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



DIRECTORY FILE OPERATIONS

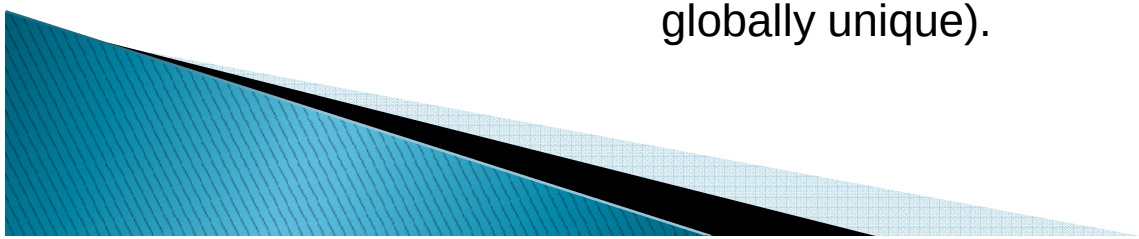
Figure 12.7 Directory service operations

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*.



FILE SERVICE ARCHITECTURE

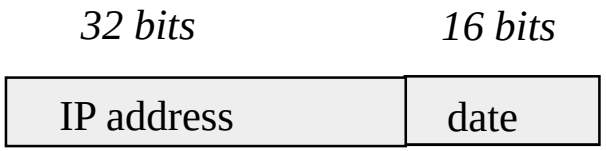
- Hierarchic file system
  - A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.
  - Each directory holds the names of the files and other directories that are accessible from it.
- File Group
  - A file group is a collection of 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).



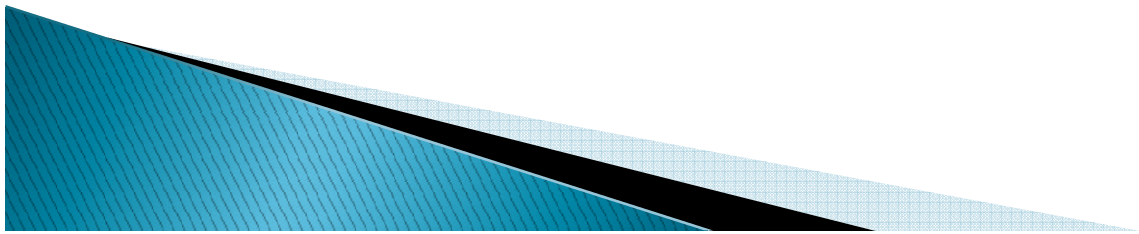
FILE SERVICE ARCHITECTURE

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:



THANK YOU



ANDREW FILE SYSTEM

- The **Andrew File System** (**AFS**) is a distributed file system which uses a set of trusted servers to present a homogeneous, location-transparent file name space to all the client workstations.
- It was developed by Carnegie Mellon University as part of the Andrew Project.
- AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS servers hold 'local' UNIX files, but the filing system in the servers is NFS-based, so files are referenced by NFS-style file handles rather than i-node numbers.

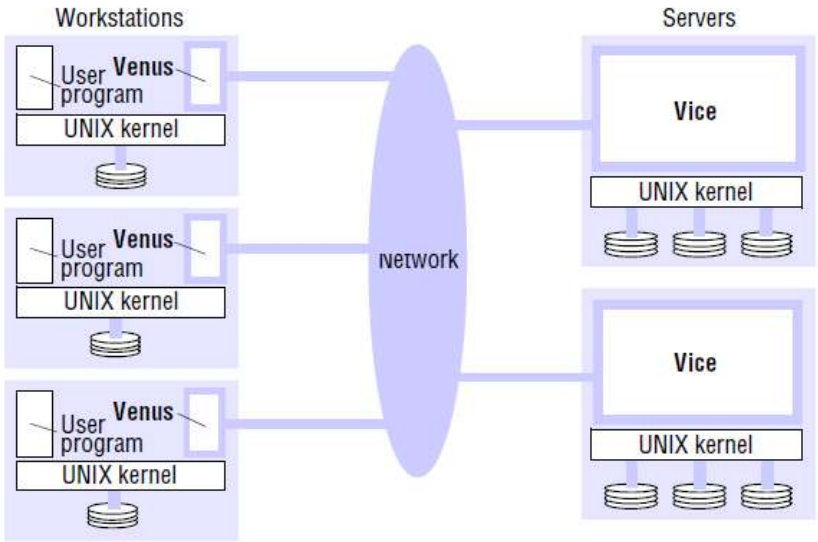
Design characteristics of AFS:

Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole-file caching: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible.

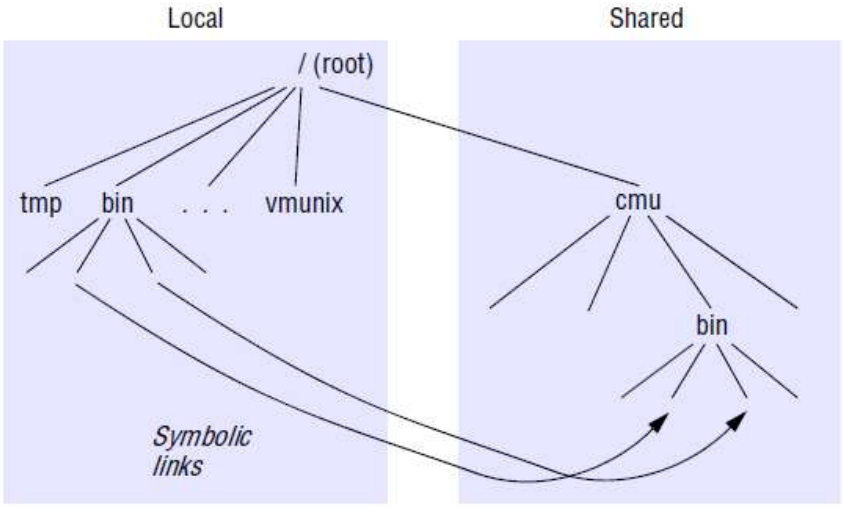
AFS architecture:

1) AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*.



2) **Vice** is the name given to the server software that runs as a user-level UNIX process in each server computer.
3) **Venus** is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.
4) The files available to user processes running on workstations are either local or shared.
5) Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes.
6) Shared files are stored on servers, and copies of them are cached on the local disks of workstations.
7) The name space seen by user processes is illustrated in Figure 12.12. It is a conventional UNIX directory hierarchy, with a specific sub tree (called ***cmu***) containing all of the shared files

Figure 12.12 File name space seen by clients of AFS



8)
9) Local files are used only for temporary files (/tmp) and processes that are essential for workstation start-up.
10) A flat file service is implemented by the Vice servers, and the hierarchic directory structure required by UNIX user programs is implemented by the set of Venus processes in the workstations.
11) Each file and directory in the shared file space is identified by a unique, 96-bit file identifier (fid) similar to a UFID.

32 bits	*32 bits*	*32 bits*
Volume number	File handle	Uniquifier

12) One of the file partitions on the local disk of each workstation is used as a cache, holding the cached copies of files from the shared space.

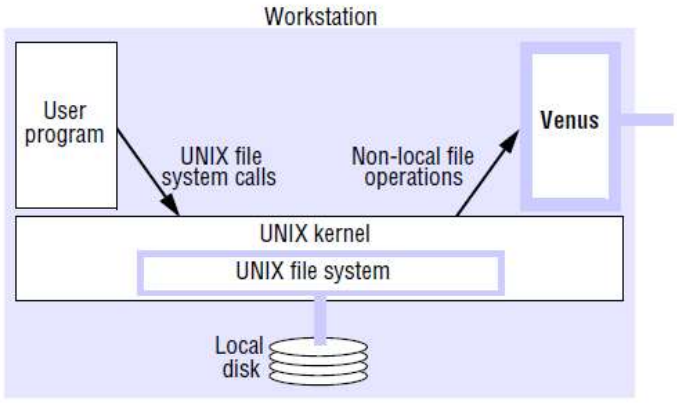
13) Venus manages the cache removing the least recently used files when a new file is acquired from a server to make the required space if the partition is full.

Figure 12.14 Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	←	
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

The **callback promise** mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after updating it.

Figure 12.13 System call interception in AFS



Cache consistency of AFS:

a) When Vice supplies a copy of a file to a Venus process it also provides a *callback promise* – a token issued by the Vice server that is the custodian of the

file, guaranteeing that it will notify the Venus process when any other client modifies the file.

b) Callback promises are stored with the cached files on the workstation disks and have two states: ***valid* or *cancelled*.**
c) When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a *callback* to each – a callback is a remote procedure call from a server to a Venus process.
d) When the Venus process receives a callback, it sets the *callback promise* token for the relevant file to *cancelled*.
e) Whenever Venus handles an open on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked.
f) If its value is cancelled, then a fresh copy of the file must be fetched from the Vice server.
g) If the token is valid, then the cached copy can be opened and used without reference to Vice.

Update semantics of AFS:

- The goal of this cache-consistency mechanism is to achieve the best approximation to one-copy file semantics.
- The results of each *write* to a file be distributed to all sites holding the file in their cache before any further accesses can occur.
- For a client *C* operating on a file *F* whose custodian is a server *S*, the following guarantees of currency for the copies of *F* are maintained:
  - after a successful *open*: *latest*(*F, S*)
  - after a failed *open*: *failure*(*S*)
  - after a successful *close*: *updated*(*F, S*)
  - after a failed *close*: *failure*(*S*)
- where *latest*(*F, S*) denotes a guarantee that the current value of *F* at *C* is the same as the value at *S*, *failure*(*S*) denotes that the *open* or *close* operation has not been performed at *S* (and the failure can be detected by *C*), and *updated*(*F, S*) denotes that *C*'s value of *F* has been successfully propagated to *S*.

Security in AFS:

- AFS makes use of **Kerberos** to authenticate users
- AFS uses access control lists(ACL) to restrict access to file directories

Distributed File Systems

G.K.PARINITHA

File Model

- Different File systems use different conceptual models.
- Unstructured and structured files.
- Mutable and Immutable files.

Unstructured and Structured Files

- In the unstructured model, file is an unstructured sequence of data. There is no substructure known to the file server. File content is also an uninterpreted sequence of bytes

- The interpretation of the meaning and structure of the data stored in the files is up to the application.
- Kernel does not interpret the content or structure of the file.

Unstructured and Structured Files

- In structured files, the file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different sizes.

- Record is the smallest unit in structured file. Read and write operation is performed on set of records.

Unstructured and Structured Files

- Structured files are of two types: index record and non-index record file.
- Index Record file: Records have one or more key fields. Key is used to access the records. File is maintained by B-tree or any other suitable data structures like hash tables and indices.

- Non-indexed record file: File record is accessed by specifying its position within the file

Unstructured and Structured Files

Structured file format

Mutable and Immutable Files

- Mutable and Immutable files are based on the modifiability criteria.

- A mutable file can be updated or extended. This means you can change, add or remove elements of a collection.
- Most existing operating systems use the mutable file model. An update performed on a file overwrites its old contents to produce the new contents.

Mutable and Immutable Files

- An immutable file is one that, once created, cannot be changed.
- Immutable files are easy to cache and to replicate across servers since their contents are guaranteed to remain unchanged.

- In the immutable model, rather than updating the same file, a new version of file is created each time a change is made to the file contents and the old version is retained unchanged.

Mutable and Immutable Files

- Immutable file model eliminates all problems associated with mutable versions. It suffers from two problems:

- Increased use of disk space

- Increased disk allocation activity

File Accessing Model

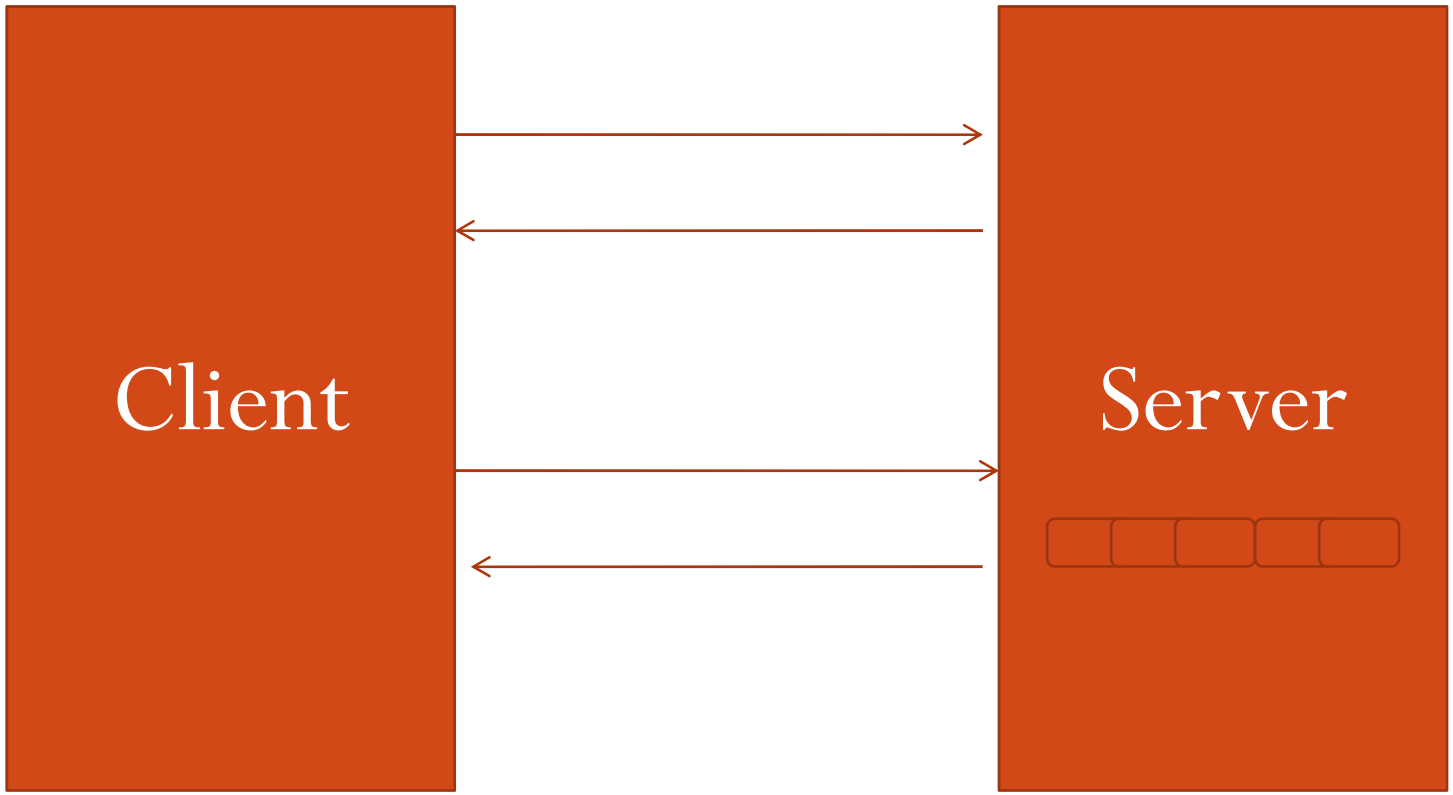
- ACCESSING REMOTE FILES:
- Accessing remote files are of two types
  - Remote service model
  - Data caching model

- In Remote service model, the client submits requests to server; all processing is done on server. File never moves from server. Problem is server bottleneck

File Accessing Model

- In Data Caching Model, it uses locality feature to reduce network traffic.
- For open file operation, transfer entire file to client and for close operation, it transfers entire file to server.
- Client works on the file locally.

File Accessing Model



Request from client to access remote file

File stays on the server

File Accessing Model

- This takes advantage of the locality feature of the found in file accesses.
- A replacement policy such as LRU is used to keep the cache size bounded.
- It is simple and efficient if working on entire file.
- Problem is it needs local disk space.

Unit of Data Transfer

- Transfer levels are file, block, byte and record.
- The unit of data transfer when a request is satisfied by a server can be:
  - A whole file
  - A number of block of a file
  - A specific number of bytes
  - A number of records.

Transfer Level : File

- Whole File is moved to the client side or server side before performing operation.
- It is simple and involves less communication overhead
- Immune to server.
- A client required to have large storage space.
- If small fraction of a file is required, moving whole file is wasteful.

Transfer Level :Block

- File transfer between client and server takes place in blocks.
- Also called page level transfer model.
- A client not required to have large storage.
- Used in diskless workstation
- No need for copying entire file.
- Network traffic overhead.

Transfer Level :Byte

- File transfer between client and server takes place in bytes
- Flexibility maximized
- Difficult cache management to handle the variable length data

Transfer Level : Record

- File transfer between client and server takes place in records.
- Handling structured and indexed files.
- More network traffic.
- More overhead to re-construct a file

File Sharing Semantics

- Files are shared between number of users.
- File protection, naming and sharing is issue for file sharing.
- Directory structure may allow files to be shared by users.
- Sharing must be done through a protection scheme.
- Consistency semantics is related with file sharing on the network. If there is a difference in the content of file, then it creates a problem.

File Sharing Semantics

- Unix Semantics:
- It enforces an absolute time ordering on all operations and ensures that every read operation on a file sees the effects of all previous write operations
- It implements:
  - Writes to an open file visible to other users of the same open file
  - Sharing file pointer to allow multiple users of the same open file

File Sharing Semantics

- A file is coupled with a single physical image that is associated as special resource.
- If there is a conflict for single then it causes delays in user processes.
- Centralized systems use UNIX semantics.

File Sharing Semantics

- Session Semantics:

- Andrew File systems implemented complex remote file sharing semantics:
  - Writes to a file by an user is not visible to other users.
  - Once the file is closed, the changes are visible only to new sessions.

File Sharing Semantics

- In this semantics, a file can be associated with multiple views.
- Almost no constraints are imposed on scheduling accesses.
- No user is delayed in reading or writing the personal copy of the file
- AFS file systems may be accessible by systems around the world.
- Access control is maintained through complicated access control lists, which may grant access to the entire world or to specifically named remote environments.

File Sharing Semantics

- Immutable shared file semantics:
- Under this system, when a file is declared as shared by its creator, it becomes immutable and the name cannot be re-used for any other resource.
- Hence it becomes read-only, and shared access is simple
- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has two key properties. Its name may not be reused and its contents may not be altered.

File Sharing Semantics

- Transaction-like semantics:
- All changes occur atomically. Begin transaction, perform operations and end transaction.
- Partial modifications made to the shared data by a transaction will not be visible to other concurrently executing transactions until the transaction ends.
- The final file content is the same as if all the transactions were run in some sequential order.

THANK YOU

Naming: Identifiers, Addresses, Name Resolution – Name Space Implementation

What is a name?

A name in a distributed system is a string of bits or characters that is used to refer to an entity.

Need for Naming:

- ❒ In a distributed system, names are used to refer to a wide variety of resources such as- Computers, services, remote objects, and files, as well as users.
- ❒ Naming is fundamental issue in DS design as it facilitates communication and resource sharing.
- ❒ A name in the form of URL is needed to access a specific web page.
- ❒ Processes cannot share particular resources managed by a computer system unless they can name them consistently.
- ❒ We need to resolve a description by means of attributes to an entity adhering to that description.

Naming services:

1. The **naming facility** of a distributed operating system enables users and programs to assign character-string names to objects and subsequently use these names to refer to those objects.
2. The **locating facility**, which is an integral part of the naming facility, maps an object's name to the object's location in a distributed system.
3. The naming and locating facilities jointly form a **naming system**.

Benefits of Naming services:

→ Resource localization
→ Uniform naming
→ Device independent address

Identifiers and Addresses:

An address is thus just a special kind of name: it refers to an access point of an entity.

A true identifier is a name that has the following properties:

1. An identifier refers to at most one entity.

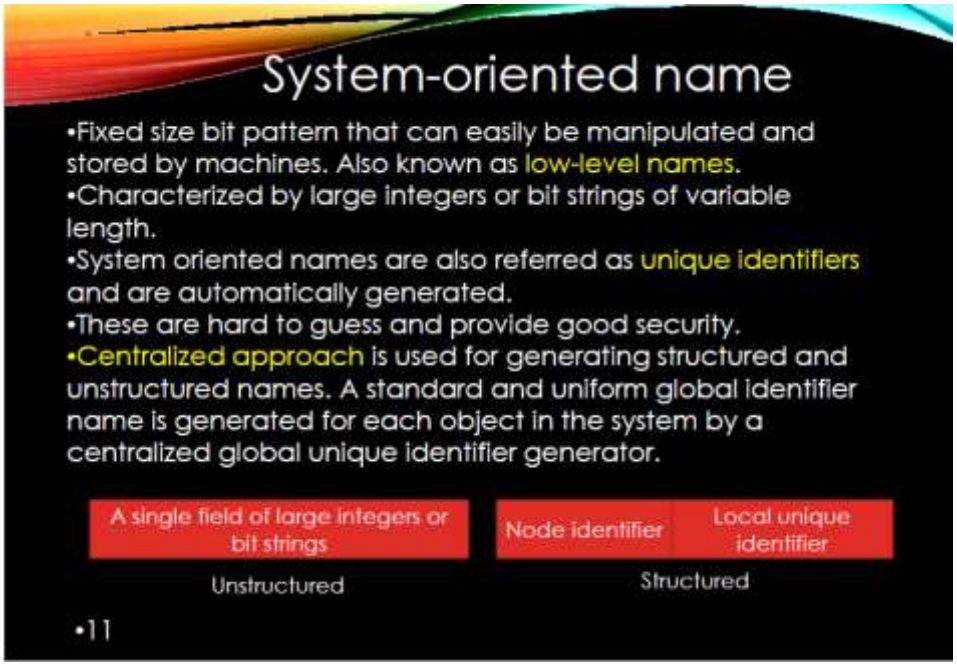
2. Each entity is referred to by at most one identifier.

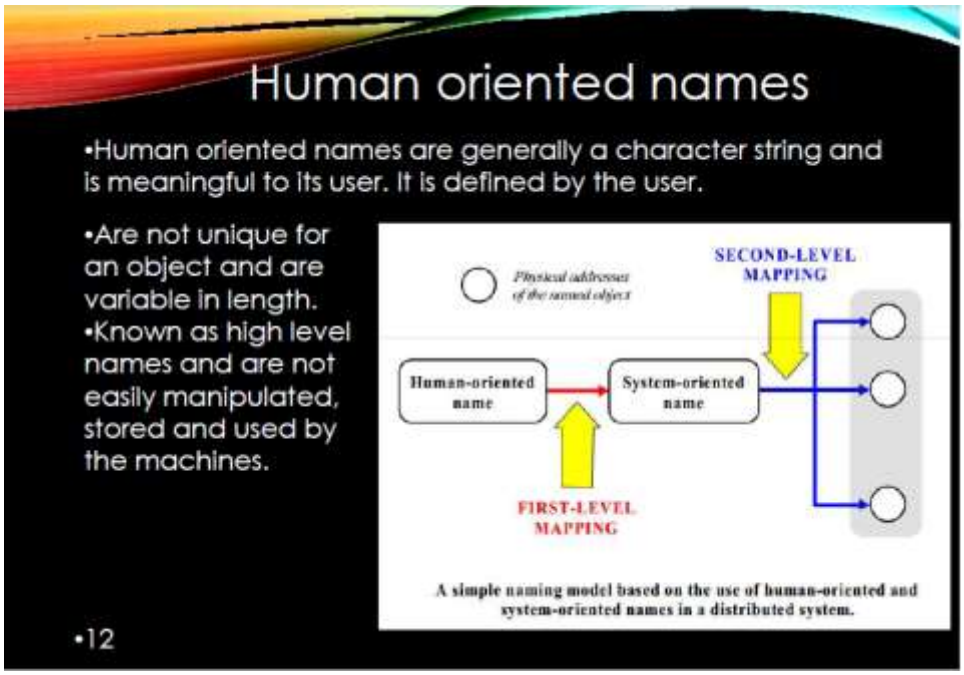
3. An identifier always refers to the same entity

A name is also called an **identifier** because it is used to denote or identify an object.

> If an address can be reassigned to a different entity, we cannot use an address as an identifier.

Types of names:



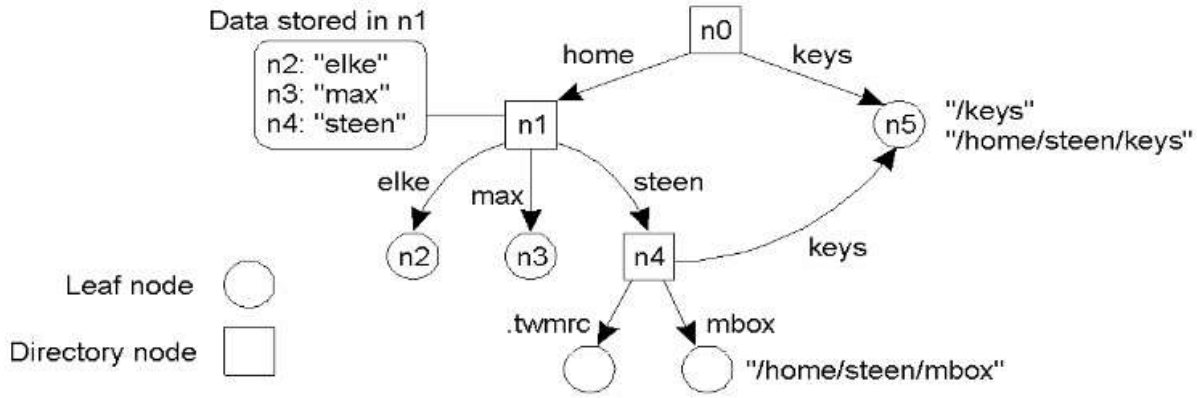


Identifier to refer two different processes:

For example, assume two processes each refer to an entity by means of an identifier. To check if the processes are referring to the same entity, it is sufficient to test if the two identifiers are equal.

NAME SPACES

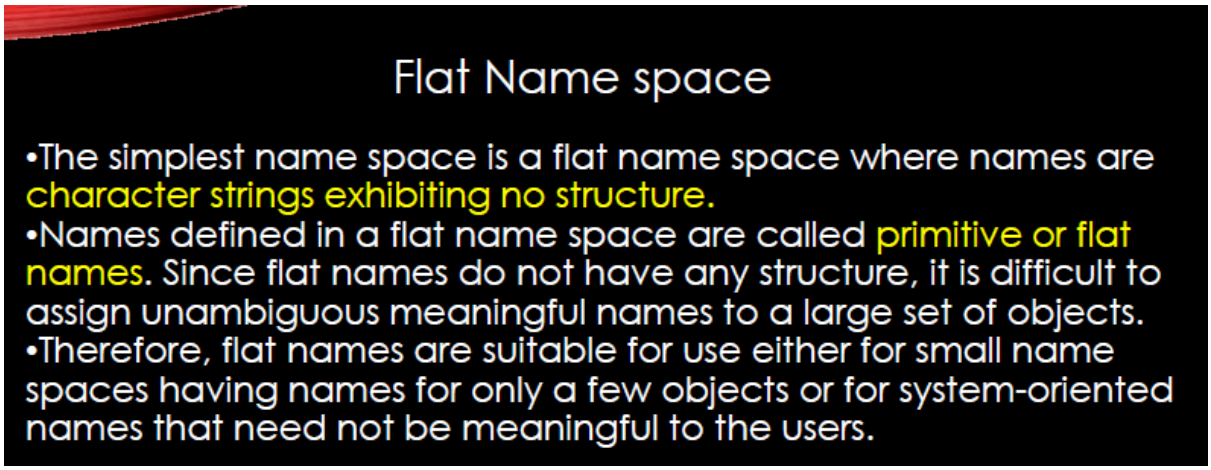
- A name space is an organization mechanism for a group of names.
- Name spaces are typically represented as a directed graph.



Types of name spaces:

The various types of name spaces are:

1. Flat Name space
2. Partitioned Name Space
3. Hierarchical Name Space



PARTITIONED NAME SPACE

When there is a need to assign unambiguous meaningful names to a large set of objects, a naming convention that partitions the name space into disjoint classes is normally used.

•Each partition of a partitioned name space is called a domain of the name space.

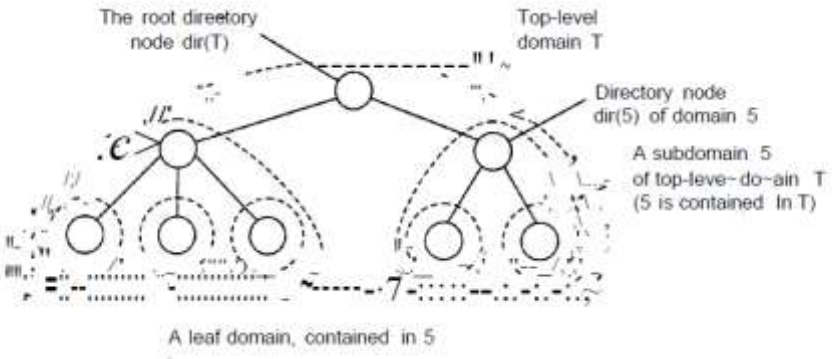
•A name defined in a domain is called a simple name. In a partitioned name space, all objects cannot be uniquely identified by simple names, and hence compound names are used for the purpose of unique identification.

Hierarchical Name space:

Hierarchical name space

•A commonly used type of partitioned name space is the hierarchical name space, in which the name space is partitioned into multiple levels and is structured as an inverted tree.
•Each node of the name space tree corresponds to a domain of the name space. In this type of name space, the number of levels may be either fixed (Eg. Grapevine, Xerox Clearinghouse) or arbitrary (Eg.DARPA).
•For instance, telephone numbers fully expanded to include country and area codes form a four-level hierarchical name space and network addresses in computer networks form a three-level hierarchical name space where the three levels are for network number, node number, and socket number.

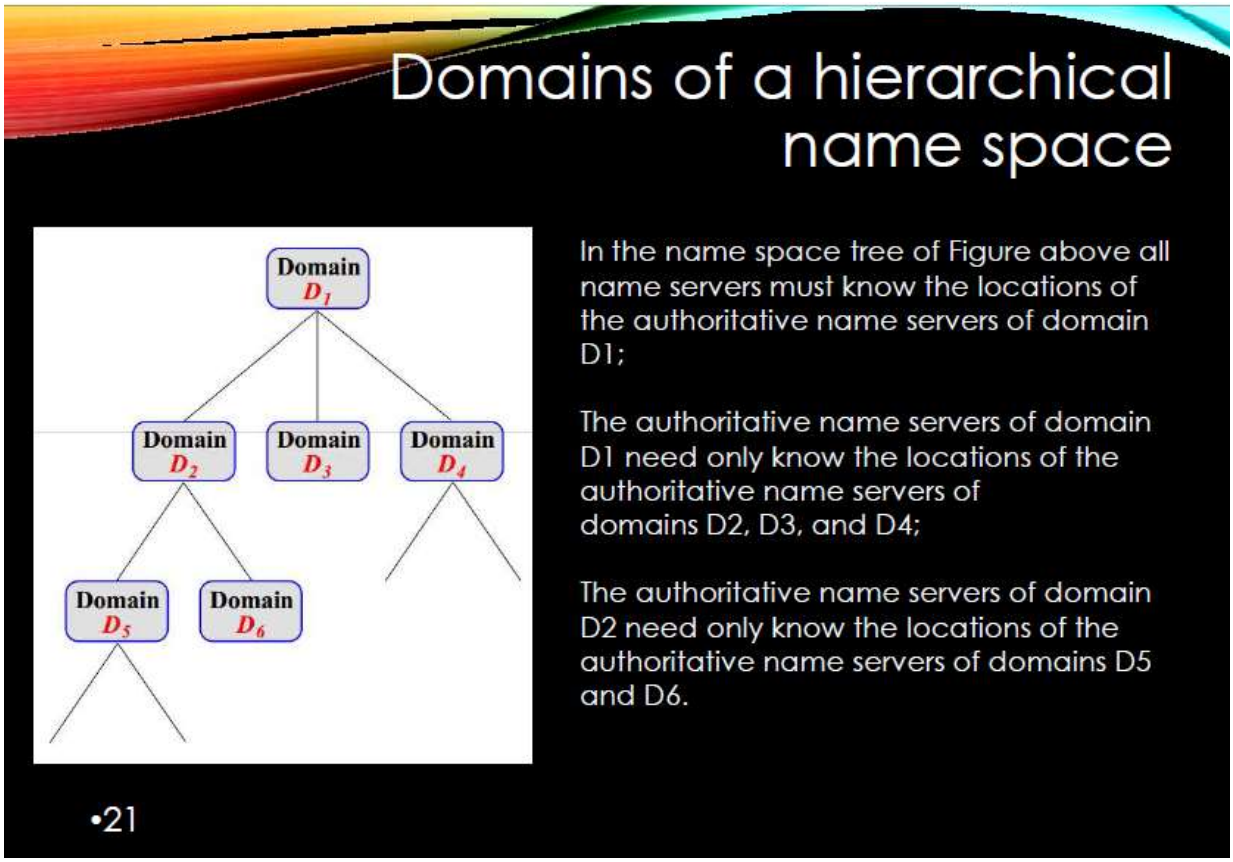
Each domain *D* has an associated directory node *dirt D)* that keeps track of the entities in that domain. This leads to a tree of directory nodes. The directory node of the top-level domain, caUed the root (directory) node, knows about all entities. This general organization of a network into domains and directory nodes is illustrated in Fig. 5-5.



.

Name servers:

- Name spaces are managed by name servers.
- A name server is a process that maintains information about named objects and provides facilities that enable users to access that information.
- The name servers that store the information about an object are called the authoritative name servers of that object.
- In a hierarchical name space, it is sufficient that each name server store only enough information to locate the authoritative name servers for the root domain of the name tree.



Name resolution:

→ The mapping of names into the objects they represent is the responsibility of the name resolution mechanism.

→ The major operation that a name service supports is to resolve names.

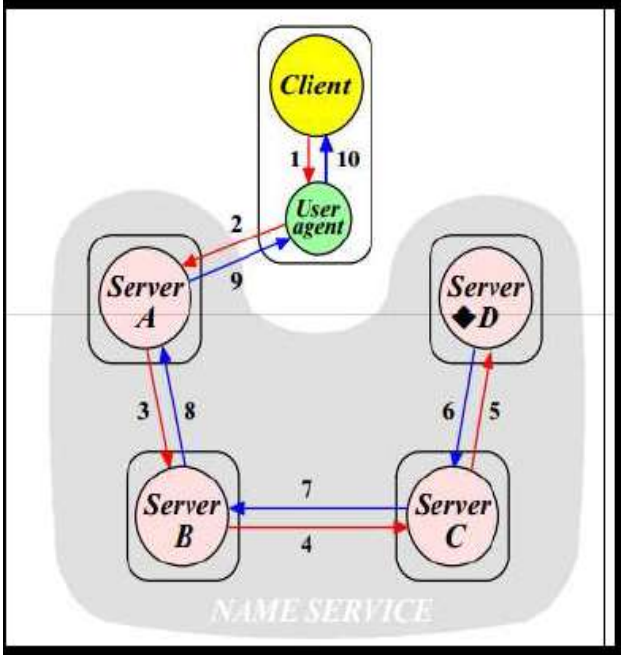
→ The two types of resolution are:

- ★ Iterative name resolution
- ★ Recursive name resolution

Recursive resolution:

Recursive resolution

- In this method, the name agent forwards the name resolution request to the name server that stores the first context needed to start the resolution of the given name.
- After this, the name servers that store the contexts of the given pathname are recursively activated one after another until the authority attribute of the named object is extracted from the context corresponding to the last component name of the pathname.
- The last name server returns the authority attribute to its previous name server, which then returns it to its own previous name server, and so on.
- Finally, the fast name server that received the request from the name agent returns the authority attribute to the name agent.



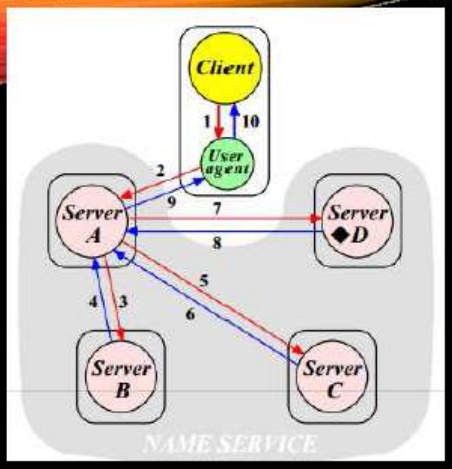
As an example, if the name */a/b/c/d* is to be resolved, the name agent sends it to the name server (say SA) of the root context (/) and waits for a reply.

Then SA searches for the component name *a* in the root context, extracts the corresponding binding information, sends the remaining pathname *b/c/d* to the name server (say SB) of the next context (/a), and waits for a reply. Then SB extracts from context */a* the binding information corresponding to the component name *b*, sends the remaining pathname c to the name server (say SC) of the next context (/a/b), and waits fora reply. Then SC extracts from context /a/b the authority attribute corresponding to the component name c and returns it to S B,. which in turn returns it to SA, and finally S, returns it to the name agent.

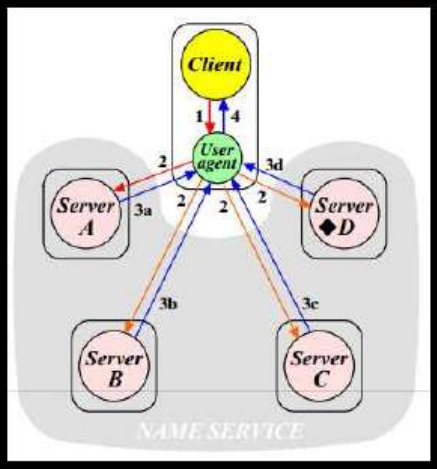
Iterative resolution:

ITERATIVE RESOLUTION

•In this method, name servers do not call each other directly. Rather, the name agent retains control over the resolution process and one by one calls each of the servers involved in the resolution process.
•As in the recursive process, the name agent first sends the name to be resolved to the name server that stores the first context needed to start the resolution of the given name.
•The server resolves as many components of the name as possible. If the name is completely resolved, the authority attribute of the named object is returned by the server to the name agent.
•Otherwise, the server returns to the name agent the unresolved portion of the name along with the location information of another name server that the name agent should contact next.



Parallel iterative resolution protocol



Sequential iterative resolution protocol

To continue the name resolution. The name agent sends a name resolution request along with the unresolved portion of the name to the next name server. The process continues until the name agent receives the authority attribute of the named object.

NAME CACHES AND LDAP

- K.SRIRAGHAV – 3rd year CSE

Need for name caches

- In UNIX systems, the system call frequency for name-mapping operations (open, stat, lstat) constitute over 50% of the file system calls.
- Researchers also observed that in a large distributed system, a substantial portion of the network traffic is naming related.
- Hence there is a need for client to cache the result of a name resolution operation for a while, rather than repeating it every time the value is needed.

What is name cache?

- Caching name servers (*DNS caches*) store DNS query results for a period of time determined in the configuration (time-to-live) of each domain-name record.
- **Advantages:**
- Name caches improve the efficiency of the DNS by **reducing DNS traffic** across the distributed system
- Reducing load on authoritative name-servers, particularly root name-servers.
- Because they can answer questions more quickly, they also increase the performance of end-user applications that use the DNS.

Characteristics of name caches

- **High degree of locality of name lookup:**
  - The property of “locality of reference” has been observed in program execution , database access and file access.
  - Due to this locality feature, a reasonable size name cache , used for caching recently used information, can provide excellent hit ratios.
- **Slow update of name information database:**
  - Naming data has a high read-to-modify ratio.
  - This behaviour implies that the cost of maintaining the consistency of cached data is significantly low.

Characteristics of name caches - [contd]

- **On-use consistency of cached information:**
  - Name cache consistency can be maintained by detecting and discarding stale cache entries on use.
  - With On-use consistency checking, there is no need to invalidate all related cache entries when a naming data update occurs .

Types of name caches

- **Directory cache:**
  - Each entry consists of a directory page.
  - Used in systems that use the iterative method of name resolution.
  - All recently used directory pages that are brought to the client node during name resolution are cached for a while.
  - Ex : LOCUS
  - **Disadv:** For only one useful entry of a directory, an entire page of a directory blocks a large area of precious cache space.

Types of name caches-[contd]

- **Prefix cache:**
  - Used in naming systems that use the zone-based context distribution mechanism.
  - Each entry consists of a name prefix and the corresponding zone identifier.
  - Ex: Sprite , V-System
  - **Disadv** - Not suitable to use with structure-free context distribution approach.

Types of name caches - [contd]

- **Full-name cache:**
  - Each entry consists of an object's full pathname and the identifier and location of its authoritative name server.
  - So, requests for accessing an object whose name is available in local cache can be directly sent to authoritative name server
  - Ex : Galaxy
  - **Adv** - Used with any naming system
  - **Disadv** - Larger in size than prefix name caches.

Approaches for name cache implementation

- **Cache per process:**
  - Separate name cache is maintained for each process.
  - Each cache is maintained in the corresponding process's address space.
  - Therefore, accessing of cached information is fast and no memory area of the OS is occupied by the name caches.
  - Very process must create its own name cache from scratch.
  - Hence if the processes are short-lived, caches will have short lifetimes.
  - Hit ratio will be less for process-oriented caches due to start-up misses.
  - Ex. Sprite and V-System

Approaches for name cache implementation - s[contd]

- **A single cache for all processes of a node**
  - In this approach, a single name cache is maintained at each node for all the processes of that node.
  - When compared to the process-oriented name caches, these caches are larger in size.
  - Accessing of information is slow
  - However, cached information in a single name cache is long lived – therefore no start-up- misses.
  - So higher average hit ratio as compared to process-oriented caches.
  - Ex : Locus and Galaxy

Multicache consistency

- When a data update , related name cache entries become stale and must be invalidated or updated properly.
- Two methods for Multicache consistency are :
  - Immediate invalidate
  - On-use update
- **On-use update :**
  - Most commonly used method for maintaining name consistency.
  - No attempt is made to invalidate all related cache entries when a naming update occurs.
  - When a client uses a stale cached data, it is informed by the naming system that data being used is either incorrectly specified.
  - On receiving a negative reply, necessary steps are taken to obtain the updated data.

Immediate invalidate

First approach

- Whenever a naming update operation is done, an invalidate message identifying the data to be invalidated is broadcast to all the nodes in the system
- Each node's name cache are then examined for the presence of updated data
- If it is present, the corresponding cache entry is invalidated.

Second approach

- The storage node of naming data keeps a list of nodes on which the data is cached.
- When a storage node receives a request for a naming data update, only the nodes in the corresponding list are notified to invalidate.

LDAP

Lightweight Directory Access Protocol

What is LDAP ? What is directory?

- LDAP is an application protocol for querying and modifying directory services
  - Runs over TCP/IP
- **Directory**
  - A set of objects with similar attributes
    - Organized in a logical and hierarchical manner
  - Example:
    - Telephone directory
      - Series of names (either of persons or organizations)
      - Organized alphabetically
      - Each name has an address and phone number
  - LDAP is often used by other services for authentication
  - **LDAP directory tree is drawn.**

LDAP – Attribute Naming

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20
WWW_Server	—	130.37.20.20

Figure 5-22. A simple example of an LDAP directory entry using LDAP naming conventions.

Origin and influences

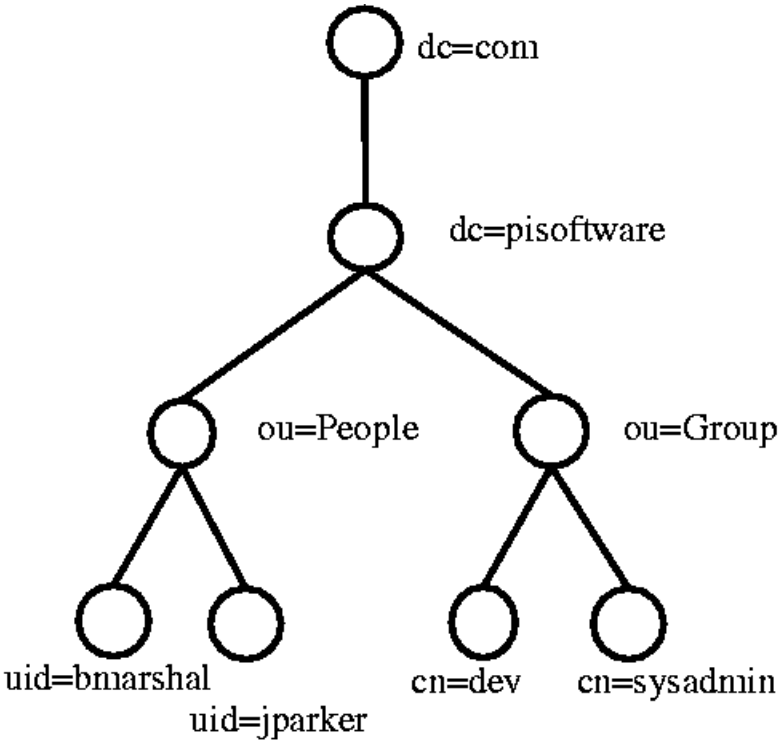
- X.500 directory services
  - Traditionally accessed via the X.500 Directory Access Protocol (DAP)
  - Required the Open Systems Interconnection (OSI) protocol stack
- LDAP originally intended to be a "lightweight" alternative protocol for accessing X.500 directory services
  - Through the simpler (and now widespread) TCP/IP protocol stack

Directory structure

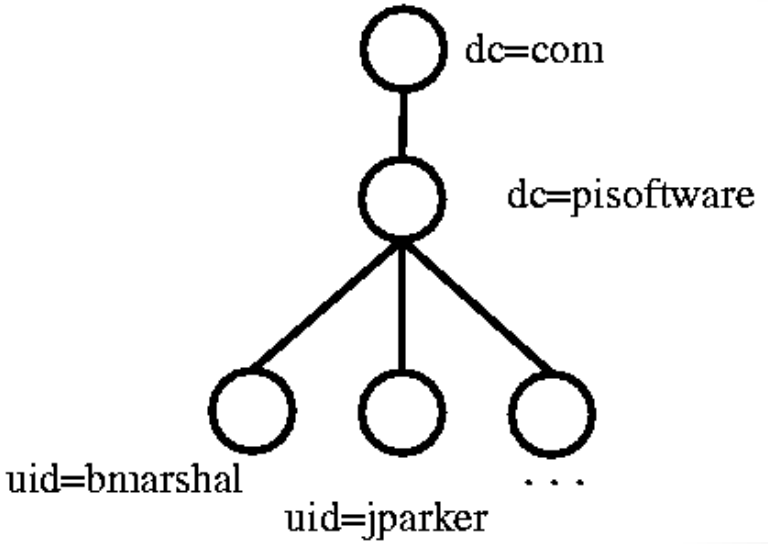
- Protocol accesses LDAP directories
  - Follows the 1993 edition of the X.500 model:
    - directory is a tree of directory entries
    - Entry consists of a set of attributes
    - An attribute has
      - a name
      - an *attribute type* or *attribute description*
      - one or more values
    - Attributes are defined in a *schema*
    - Each entry has a unique identifier:
      - *Distinguished Name* (DN)
        - Consists of its *Relative Distinguished Name* (RDN) constructed from some attribute(s) in the entry
        - Followed by the parent entry's DN
    - Think of the DN as a full filename and the RDN as a relative filename in a folder

LDAP Data Structure

Hierarchical



Flat



dc: domain component
ou: organizational unit

Protocol overview

- Client starts an LDAP session by connecting to an LDAP server
  - Default on TCP port 389
- Client sends operation requests to the server
  - Server sends responses in turn
- With some exceptions the client need not wait for a response before sending the next request
  - Server may send the responses in any order

Protocol overview

- The client may request the following operations:
  - Start TLS
    - Optionally protect the connection with Transport Layer Security (TLS), to have a more secure connection
  - Bind - authenticate and specify LDAP protocol version
  - Search - search for and/or retrieve directory entries
  - Compare - test if a named entry contains a given attribute value
  - Add a new entry
  - Delete an entry
  - Modify an entry
  - Modify Distinguished Name (DN) - move or rename an entry
  - Abandon - abort a previous request
  - Extended Operation - generic operation used to define other

LDAP BACKENDS

- THE BASIC DAEMON PROCESS THAT RUNS ON THE LDAP SERVER CALLED SLAPD COMES WITH THREE DIFFERENT BACKEND DATABASES
- WE ASSUME THAT IN OUR CASE WE USE LDBM THE MOST USED ONE

LDIF

- LDIF STANDS FOR LDAP DATA INTERCHANGE FORMAT
- DIRECTORY ENTRIES IN LDAP ARE IN THE FORM OF LDIF

LDIF FORMAT

- BASIC FORM OF LDIF :
  #COMMENT
  DN: <DISTINGUSHED NAME>
  <ATTRDESC>: <ATTRVALUE>
  <ATTRDESC>: <ATTRVALUE>
  .....
- EXAMPLE :
  DN: UID=ALAKESH DC=IIT DC=EDU

Refer

1. **Authors :**Wien-Verteilte_Systeme_VO_(Göschka)_-_Tannenbaum

Book name:

distributed_systems_principles_and_paradigms_2nd_edition

Page (237/705)