UNIT - III

PEER TO PEER SERVICES AND FILE 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 uniform service.

Peer-to-peer middleware systems are emerging that have the capacity to share computing resources, storage and data present in computers at the edge of the Internet on a global scale.

Napster and its Legacy:

Napster Architecture included centralized indexes but users supplied the files which were stored and accessed on their personal computers. Napster was shut down as a result of legal Proceeding that instituted against the operators of the Napster service by the owner of the copy sight in some of the material that was made available on it.

Peer-to-peer middleware:

A key problem in the design of peer-to-peer applications is to provide a mechanism to enable clients to access data resources quickly

and dependably wherever they are located throughout the network.

Peer-to-peer middleware systems are designed specifically to meet the need for 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.

Other important requirements include the ability to add new resources and 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 requirement.

* Global scalability
* Load Balancing

76

77

* Optimization for local interactions between neighbouring peers
* Accommodating to highly dynamic host availability
* Security of data in an environment with heterogeneous trust
* Anonymity, deniability and resilience to censorship

Routing Overlays

A distributed algorithm known as a routing overlay takes responsibility for locating node 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 object may be placed and subsequently relocated to any node in network without client involvement.

The Main task of a routing overlay in the following

1. A client wishing to invoke an application on an object submits a request including the object's GUID to routing overlay, which routes the request to a node at which a replica of the object resides.

2. 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.
3. When clients requests the removal of objects from the service the routing overlay must make them unavailable.
4. Nodes may join and leave the service.

Overlay case studies : Pastry, Tapestry

Pastry

Pastry is a routing overlay with the characteristics that we have outlined, all the nodes and objects that can be accessed through pastry are assigned 128-bit GUID's.

In a network with N participating nodes, the Pastry routing algorithm will correctly route a message addressed to any GUID in $O(\log N)$ steps. If the GUID identifies a node that is currently active, the message is delivered to that node; otherwise message is delivered to the active node whose GUID is numerically closest to it.

Routings steps involve the use of an underlying transport Protocol to transfer the message to a Pastry node that is closer to its destination.

Pastry uses a locality metric based on network distance in the underlying network to select appropriate neighbours when setting up the routing table used at each node.

Routing algorithm:

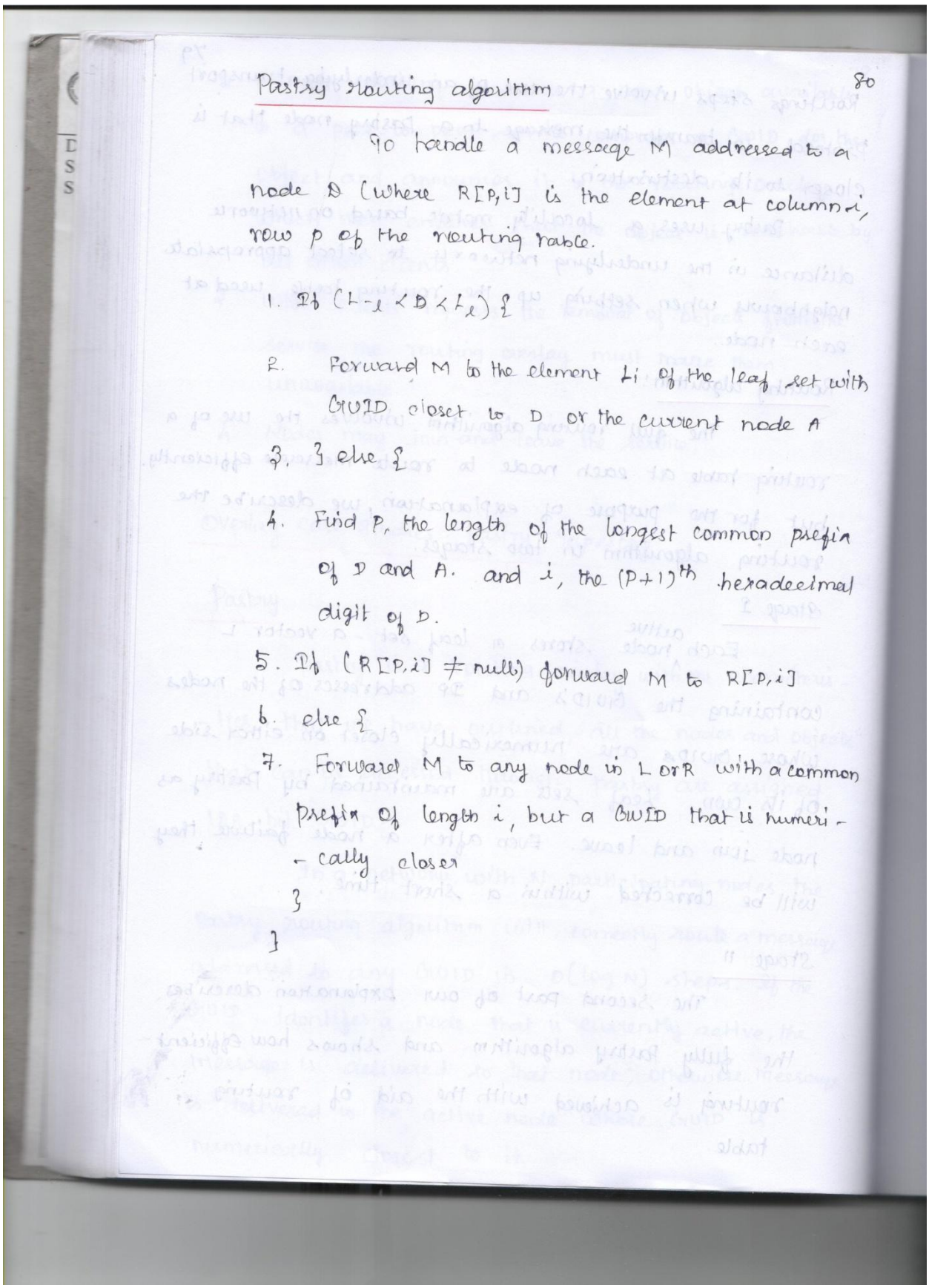
The full routing algorithm involves the use of a routing table at each node to route messages efficiently. but for the purpose of explanation, we describe the routing algorithm in two stages.

Stage I

Each active node stores a leaf set - a vector L containing the GUID's and IP addresses of the nodes whose GUIDs are numerically closest on either side of its own. Leaf sets are maintained by Pastry as node join and leave. Even after a node failure they will be corrected within a short time.

Stage II

The second part of our explanation describes the fully Pastry algorithm and shows how efficient routing is achieved with the aid of routing table.



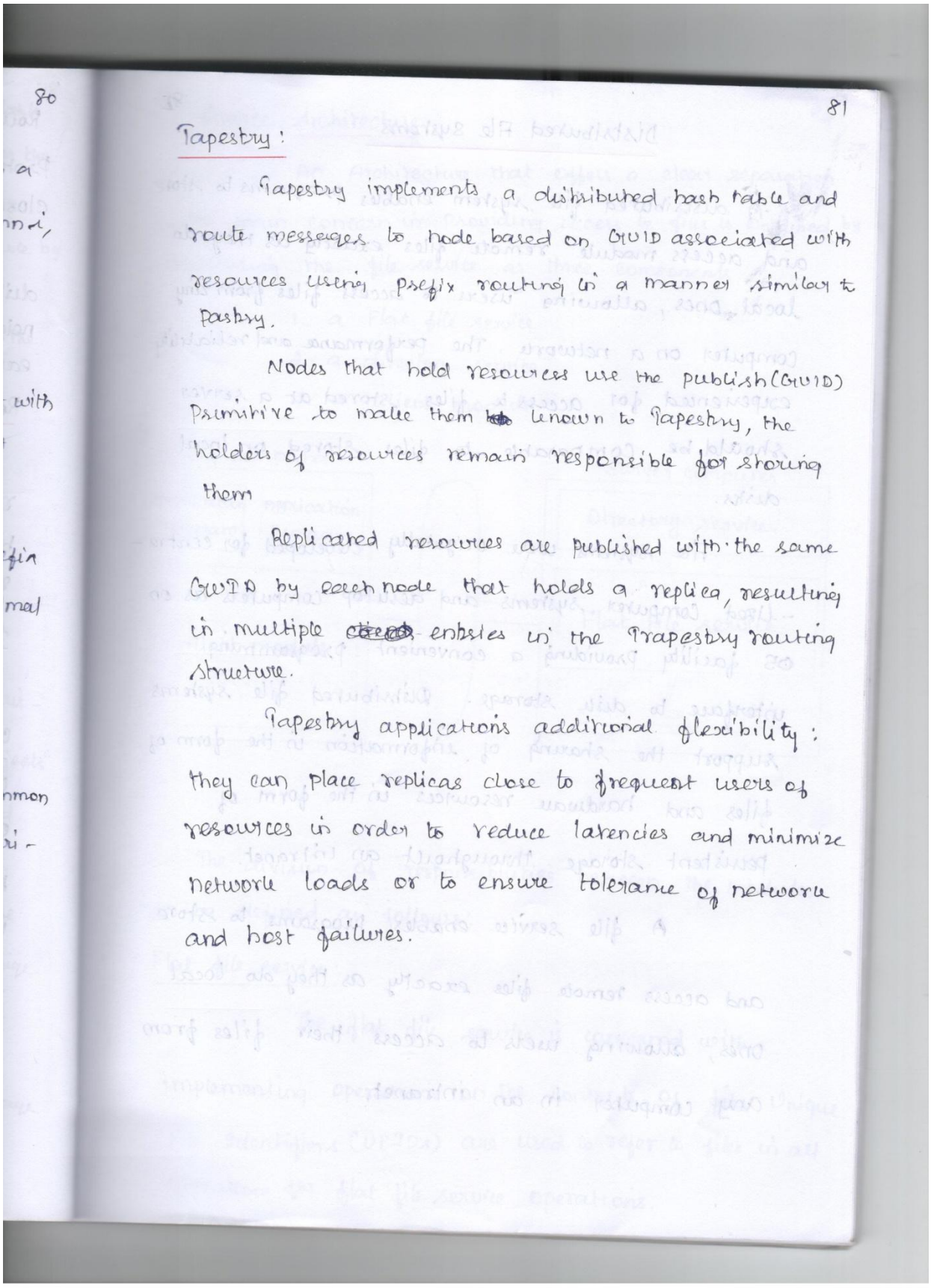
Pastry routing algorithm 80

To handle a message M addressed to a node D (where R[P,i] is the element at column i, row P of the routing table.

1. If ($L_{-l} < D < L_l$) {
2. Forward M to the element Li of the leaf set with GUID closest to D or the current node A
3. } else {
4. Find P, the length of the longest common prefix of D and A. and i, the $(P+1)^{th}$ hexadecimal digit of D.
5. If (R[P,i] ≠ null) forward M to R[P,i]
6. else {
7. Forward M to any node in L or R with a common prefix of length i, but a GUID that is numeri- - cally closer

}

}



80

81

Tapestry:

Tapestry implements a distributed hash table and route messages to node based on GUID associated with resources using prefix routing in a manner similar to Pastry.

Nodes that hold resources use the publish(GUID) primitive to make them known to Tapestry, the holders of resources remain responsible for storing them

Replicated resources are published with the same GUID by each node that holds a replica, resulting in multiple entries in the Tapestry routing structure.

Tapestry applications additional flexibility: they can place replicas close to frequent users of resources in order to reduce latencies and minimize network loads or to ensure tolerance of network and host failures.

Distributed File Systems

A distributed file system enables programs to store and access module remote files exactly as they do 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 files stored on local disks.

File systems were originally developed for centra-lized computer systems and desktop computers as an OS 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 in the form of persistent storage throughout an intranet.

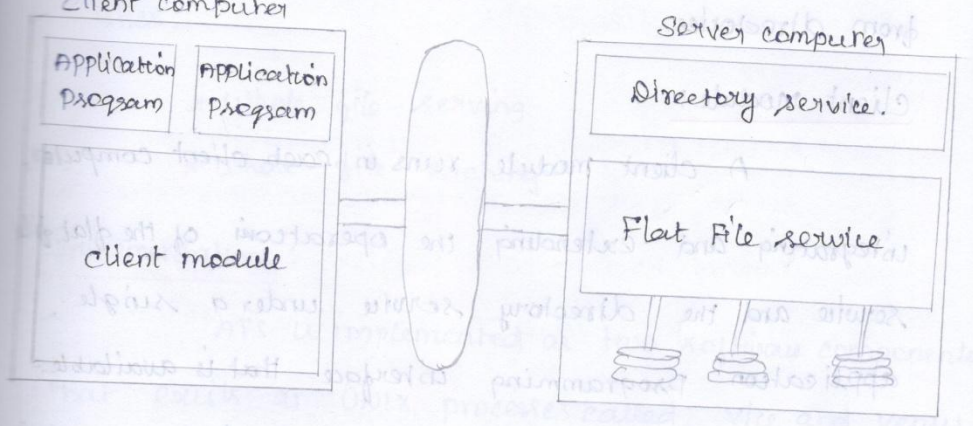
A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet.

ems to store
s they do
from any
d reliability
server
n local
for centra-
as on
ing
systems
form of
store
cal
from

File Service Architecture

An Architecture that offers a clear separation of the main concern in providing access to files is obtained by structuring the file service as three components

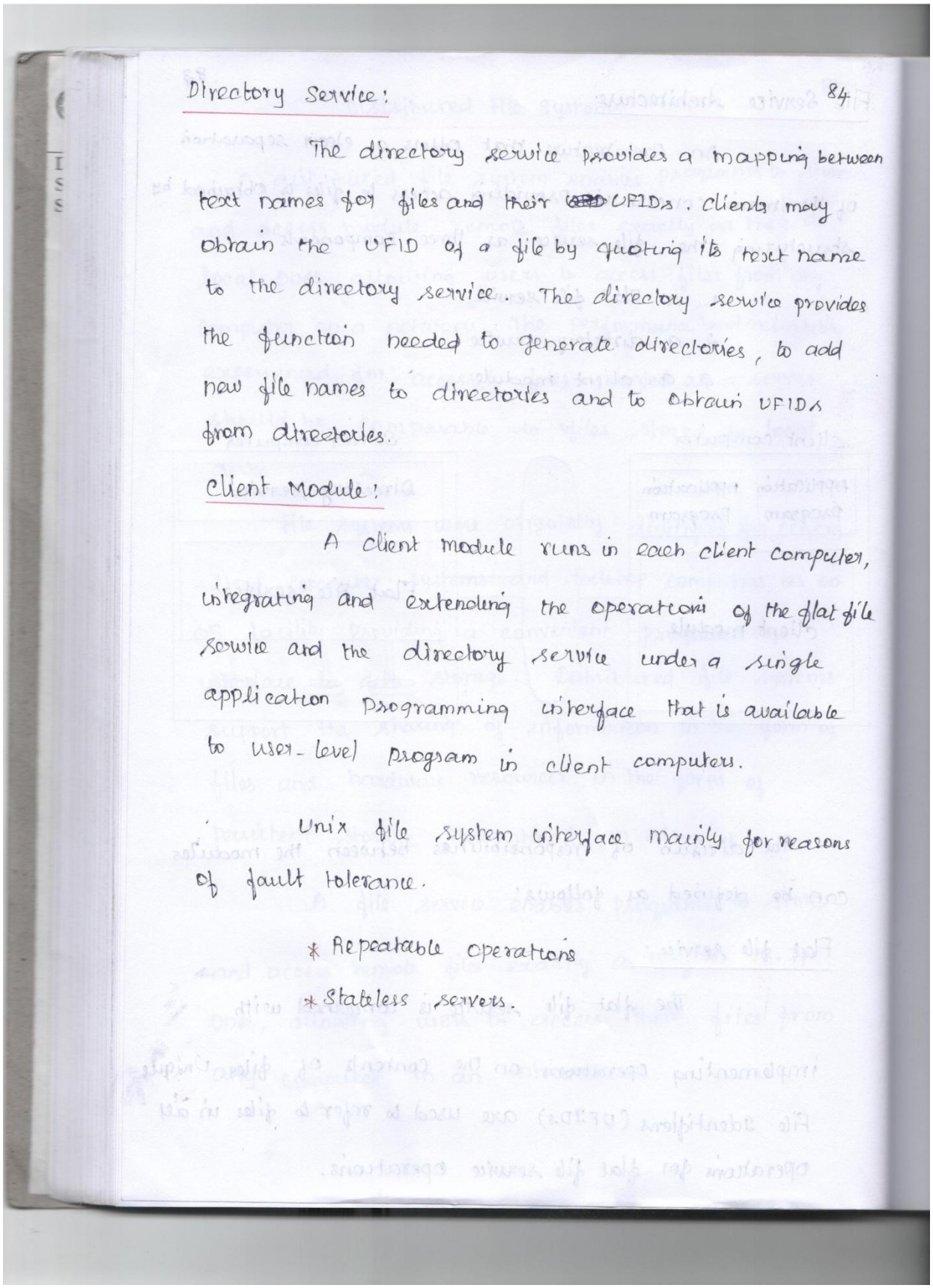
1. a Flat file service
2. a directory service
3. a client module.



The division of responsibilities between the modules can be defined as follows:

Flat file service:

The flat file service is concerned with implementing operations on the contents of files. Unique File Identifiers (UFIDs) are used to refer to files in all operations for flat file service operations.



Directory Service: 84

The directory service provides a mapping between text names for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the function needed to generate directories, to add new file names to directories and to obtain UFIDs from directories.

Client Module:

A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level program in client computers.

Unix file system interface mainly for reasons of fault tolerance.

* Repeatable operations
* Stateless servers.

Case Study: The Andrew File System

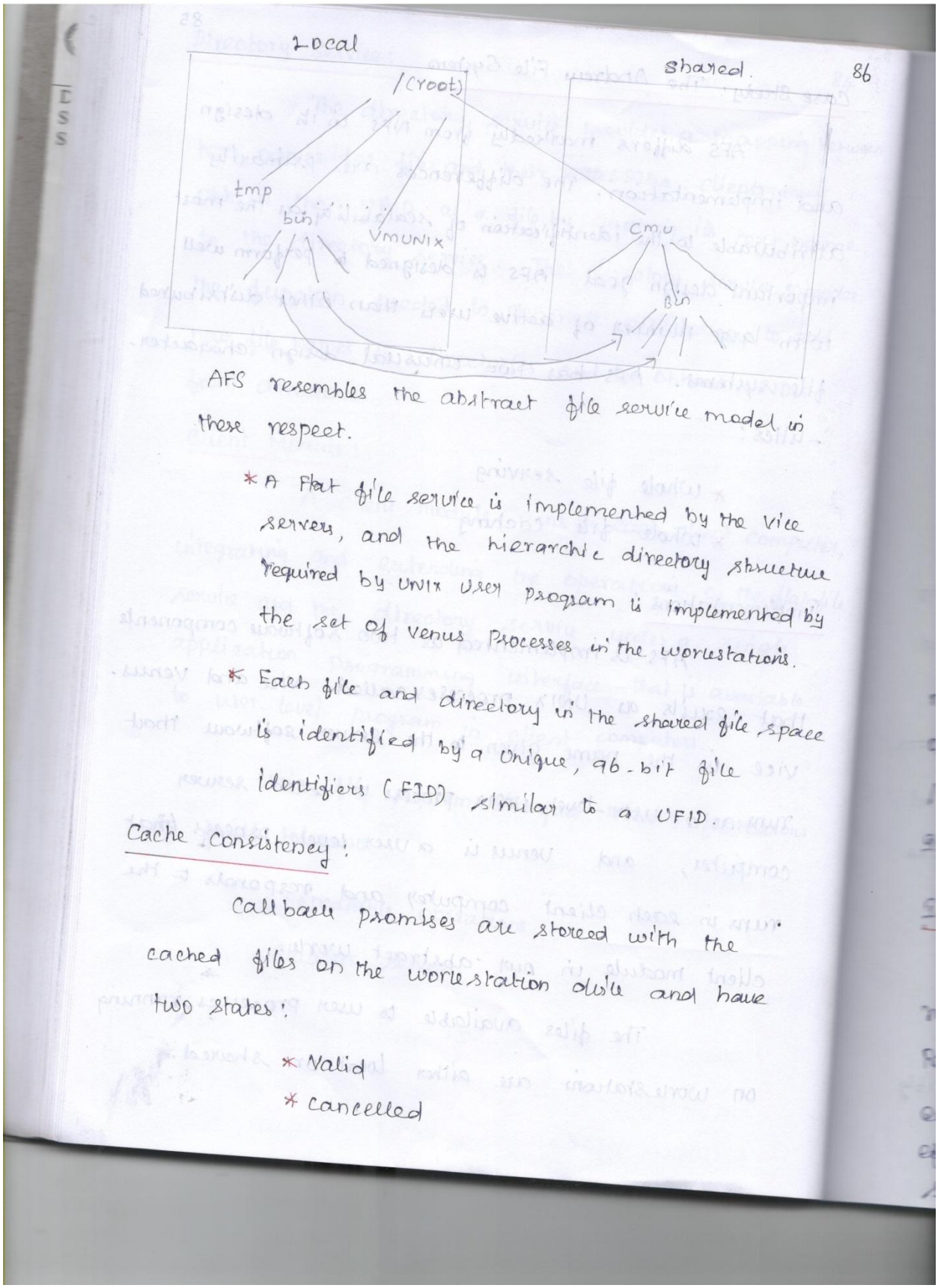
AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with large number of active users than other distributed file systems. AFS has two unusual design characteristics:

* Whole file serving
* Whole file caching

Implementations:

AFS is implemented as two software components that exists as UNIX processes called Vice and Venus. Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user level process that runs in each client computer and responds to the client module in our abstract work.

The files available to user processes running on workstations are either local or shared.



AFS resembles the abstract file service model in these respect.

* A flat file service is implemented by the vice servers, and the hierarchic directory structure required by UNIX user program is implemented by the set of venus processes in the workstations.
* Each file and directory in the shared file space is identified by a unique, 96-bit file identifiers (FID) similar to a UFID.

Cache Consistency:

Callback promises are stored with the cached files on the workstation disk and have two states:

* Valid
* Cancelled

86

87

Update Semantics

The goal of this cache consistency mechanism is to achieve the best approximation to one copy file semantics that is practicable without serious performance degradation. A strict implementation of one-copy semantics for UNIX file access primitives would require that 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.

Naming

Any process that requires access to a specific resource must process a name or an identifier for it. Examples of human readable names are file names such as /etc/password, URL such as http://www.cdk4.net and Internet domain names, such as www.cdk4.net.

Identifiers:

The term identifier is sometimes used to refer to names that are interpreted only by programs. Remote object references and NFS file handles are examples of identifiers. Identifiers are chosen for the efficiency with which they can be looked up and stored by software.

Name resolution :

Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names. More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name. The process of looking up a name is called name resolution.

Name space implementation :

A name space forms that heart of naming service, that is a service that allows users and processes to add, remove and look up names. A naming service is implemented by name servers. If a distributed system is restricted to a local Area Network, it is often feasible to implement a naming service by means of only a single name server.

Name space Distribution :

Name spaces for a large scale, possibly worldwide distributed systems, are usually organized hierarchically.

The global layer is formed by highest-level nodes, (ie) the root node and other directory nodes logically close to the root, namely its children. Nodes in the global layer are often characterized by their stability, in the sense that directory tables are rarely changed. Such nodes may represent organizations or group of organizations, for which names are stored in the name space.

Item	Global	Administration - Nodal	Managerial.
Geographical scale of network	world wide	organization	Department
Total no. of nodes	Few	many	Vast number
Responsiveness to lookups	seconds	milliseconds	Immediate
update propagation	Lazy	Immediate	Immediate
No. of replicas	Many	None or few	None
Is client-side caching applied ?	Yes	Yes	Sometimes.

Name caches :

Readers might have observed that name resolution operations not likely to be especially cheap. Based on the measurements made by few researchers in the past, it has been found that the OS that provide a flexible, hierarchical name space, the system overhead in name resolution operation is considerably large.

This is mainly due to the following characteristics of name service related activities.

1. High degree of locality of name lookup
2. Slow update of name information database
3. On-use consistency of cached information is possible.

LDAP (Lightweight Directory Access Protocol)

A common approach to tackling distributed directory services is to combine structured naming with attribute based naming. This approach has been widely adopted, for example in Microsoft Directory Service and other systems.

Many of these systems use, or rely on the lightweight directory access protocol commonly referred simply as LDAP. The LDAP directory service has been derived from OSI's X.500 directory service.

An LDAP directory service consists of a number of records, usually referred to as directory entries. A directory entry is comparable to a resource record in DNS. Each record

90

91

is made up of a collection of (attribute, value) pairs, where each attribute has an associated type. A distinction is made between single valued attributes and multi-valued attributes.

As an example, a simple directory entry identifying the network addresses of some general servers, were as follows.

Attribute	Abbr	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
Organization Unit	OU	Comp. Sc
Common Name	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

In our example, we have used a naming convention described in the LDAP standards, which applies to the first five attributes. The attributes Organization and Organization Unit describe, respectively, the organization and the department associated with the data that are stored in the record.

DIB

The collection of all directory entries in an LDAP directory service is called a Directory Information Base (DIB). An important aspects of a DIB is that each record is uniquely named so that it can be looked up. Such a globally unique name appears as a sequence of naming attributes in each record.

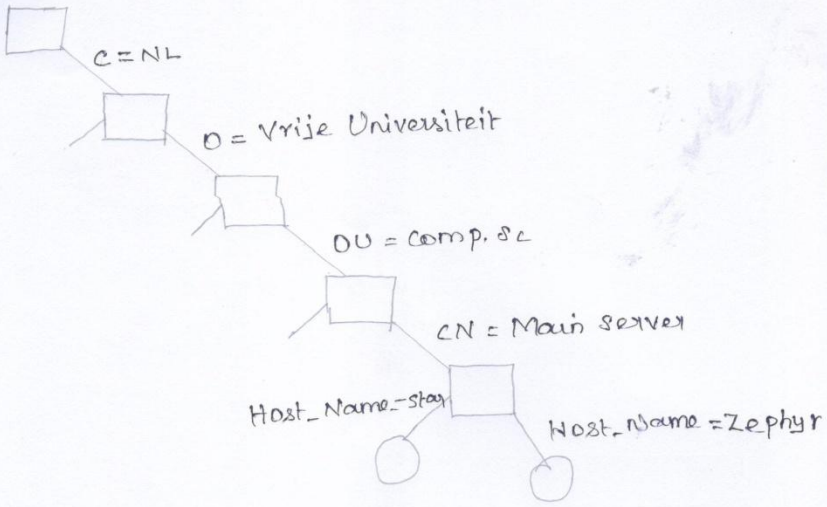
DIT

Directory Information Tree (DIT). It is a essentially forms the naming graph of an LDAP directory service in which each node represents a directory entry. In addition a node may also act as a directory in the traditional sense, in that there may be several children for which the node act as a parent.

Directory usually assumes there is a global index server that can be searched first. The index will indicate which LDAP domains need to be searched for further.

92

93



A node in an LDAP naming graph can thus simultaneously represent a directory in the traditional sense, as well as an LDAP record. This distinction is supported by two different lookup operations. The read operation is used to read a single record given its path name in the DIT.

LDAP typically represents a standardized way of supporting attribute-based naming. Other recent directory services following this more traditional approach have been developed as well, notably in the context of grid computing and web services. One specific example is the universal directory and discovery integration or simply UDDI.