UNIT III
PEER TO PEER SERVICES AND FILE SYSTEMS

PEER to PEER SYSTEMS

A key problem in the design of peer-to-peer applications is providing 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.

This location problem existed in several services that predate the peer-to-peer paradigm as well. For example, Sun NFS addresses this need with the aid of a virtual file system abstraction layer at each client that accepts requests to access files stored on multiple servers in terms of virtual file references. This solution relies on a substantial amount of pre-configuration at each client and manual intervention when file distribution patterns or server provision changes. It is clearly not scalable beyond a service managed by a single organization. AFS has similar properties.

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. Like other middleware, peer-to-peer middleware should offer a simple programming interface to application programmers that is independent of the types of distributed resource that the application manipulates.

Non-functional requirements •

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

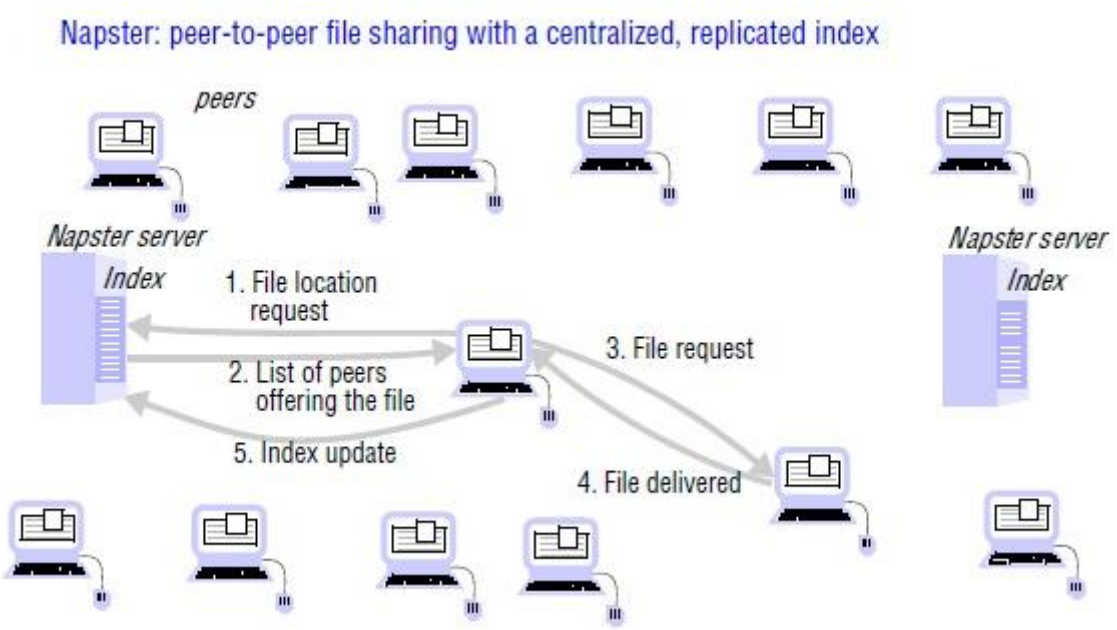
Optimization for local interactions between neighbouring peers: The 'network distance' between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. Network traffic loadings are also impacted by it. The middleware should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

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. A major challenge for peer to peer systems is to provide a dependable service despite these facts. 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.

NAPSTER AND ITS LEGACY

The first application in which a demand for a globally scalable information storage and retrieval service emerged was the downloading of digital music files. Both the need for and the feasibility of a peer-to-peer solution were first demonstrated by the Napster file sharing system [OpenNap 2001] which provided a means for users to share files.

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 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. Note that 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, fulfilling Shirky's dictum 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 (see the box below).

Anonymity for the receivers and the providers of shared data and other resources is a concern for the designers of peer-to-peer systems. In systems with many nodes, the routing of requests and results can be made sufficiently tortuous to conceal their source and the contents of files can be distributed across multiple nodes, spreading the responsibility for making them available. Mechanisms for anonymous communication that are resistant to most forms of traffic analysis are available [Goldschlag*et al.* 1999]. If files are also encrypted before they are placed on servers, the owners of the servers can plausibly deny any knowledge of the contents. But these anonymity techniques add to the cost of resource sharing, and recent work has shown that the anonymity available is weak against some attacks [Wright *et al.* 2002].

The Freenet [Clarke *et al.* 2000] and FreeHaven [Dingledine*et al.* 2000] projects are focused on providing Internet-wide file services that offer anonymity for the providers and users of the shared files. Ross Anderson has proposed the Eternity Service [Anderson 1996], a storage service that provides long-term guarantees of data availability through resistance to all sorts of accidental data loss and denial of service attacks. He bases the need for such a service on the observation that whereas publication is a permanent state for printed information – it is virtually impossible to delete material

once it has been published and distributed to a few thousand libraries in diverse organizations and jurisdictions around the world – electronic publications cannot easily achieve the same level of resistance to censorship or suppression. Anderson covers the technical and economic requirements to ensure the integrity of the store and also points out that anonymity is often an essential requirement for the persistence of information, since it provides the best defence against legal challenges, as well as illegal actions such as bribes or attacks on the originators, owners or keepers of the data.

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. This simple load distribution mechanism enabled the service to scale to meet the needs of large numbers of users.
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. Unless the access path to the data objects is distributed, object discovery and addressing are likely to become a bottleneck.
Application dependencies: Napster took advantage of the special characteristics of the application for which it was designed in other ways:

- Music files are never updated, avoiding any need to make sure all the replicas of files remain consistent after updates.
- No guarantees are required concerning the availability of individual files – if a music file is temporarily unavailable, it can be downloaded later. This reduces the requirement for dependability of individual computers and their connections to the Internet.

ROUTING OVERLAYS IN PEER TO PEER SYSTEMS

The development of middleware that meets the functional and non-functional requirements outlined in the previous section is an active area of research, and several significant middleware systems have already emerged. In this chapter we describe several of them in detail.

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. This approach to the management and location of replicated objects was first analyzed and shown to be effective for networks involving sufficiently many nodes in a groundbreaking paper by Plaxton *et al.* [1997].

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

The GUIDs used to identify nodes and objects are an example of the 'pure' names. These are also known as opaque identifiers, since they reveal nothing about the locations of the objects to which they refer.
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.
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 (such as SHA-1) 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 hashtables* (DHT). This is reflected by the simplest form of API used to access them, as shown in Figure below. With this API, the *put()* operation is used to submit a data item to be stored together with its GUID. The DHT layer takes responsibility for choosing a location for it, storing it with replicas to ensure availability) and providing access to it via the *get()* operation.

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

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

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

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

A slightly more flexible form of API is provided by a *distributed object location and routing* (DOLR) layer, as shown in Figure below. With this interface objects can be stored anywhere and the DOLR layer is responsible for maintaining a mapping between object identifiers (GUIDs) and the addresses of the nodes at which replicas of the objects are located. Objects may be replicated and stored with the same GUID at different hosts, and the routing overlay takes responsibility for routing requests to the nearest available replica.

Basic programming interface for distributed object location and routing (DOLR) as implemented by 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 the host for the object corresponding to *GUID*.

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

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

With the DHT model, a data item with GUID *X* is stored at the node whose GUID is numerically closest to *X* and at the *r* hosts whose GUIDs are next-closest to it numerically, where *r* is a replication factor chosen to ensure a very high probability of availability. With the DOLR model, locations for the replicas of data objects are decided outside the routing layer and the DOLR layer is notified of the host address of each replica using the *publish()* operation.

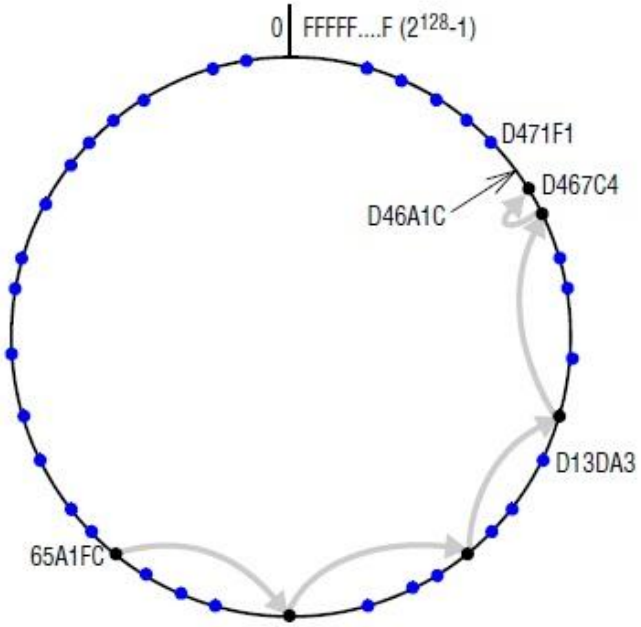
PASTRY

Pastry is a routing overlay. All the nodes and objects that can be accessed through Pastry are assigned 128-bit GUIDs. For nodes, these are computed by applying a secure hash function (such as SHA-1) to the public key with which each node is provided. For objects such as files, the GUID is computed by applying a secure hash function to the object's name or to some part of the object's stored state. The resulting GUIDs have the usual properties of secure hash values – that is, they are randomly distributed in the range 0 to 2128–1. They provide no clues as to the value from which they were computed, and clashes between GUIDs for different nodes or objects are extremely unlikely. (If a clash occurs, Pastry detects it and takes remedial action.)

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, the message is delivered to the active node whose GUID is numerically closest to it. Active nodes take responsibility for processing requests addressed to all objects in the numerical neighbourhood.

The GUID space is treated as circular: GUID 0's lower neighbour is 2^{128}–1.Figure below gives a view of active nodes distributed in this circular address space. Since every leaf set includes the GUIDs and IP addresses of the current node's immediate neighbours, a Pastry system with correct leaf sets of size at least 2 can route messages to any GUID trivially as follows: any node *A* that receives a message *M* with destination address *D* routes the message by comparing *D* with its own GUID*A* and with each of the GUIDs in its leaf set and forwarding *M* to the node amongst them that is numerically closest to *D*.

Circular routing alone is correct but inefficient *Based on Rowstron and Druschel [2001]*



The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node (2128-1). The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 (/= 4). This is a degenerate type of routing that would scale very poorly; it is not used in practice.

The Figure below shows the structure of the routing table for a specific node,

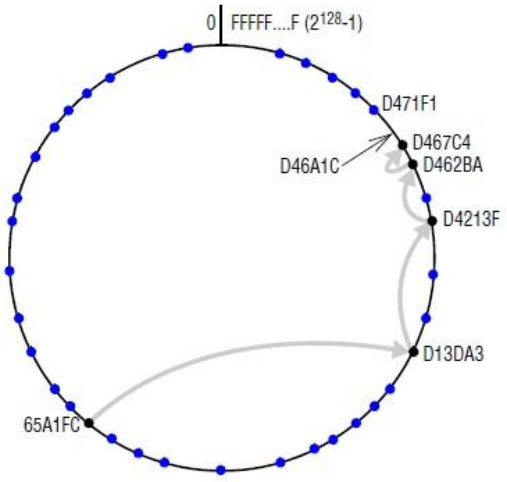
First four rows of a Pastry routing table

p =	GUID prefixes and corresponding node handles 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 that act as node handles specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded entries in the table body 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.

Figure below illustrates the actions of the routing algorithm. The routing table is structured as follows: GUIDs are viewed as hexadecimal values and the table classifies GUIDs based on their hexadecimal prefixes. The table has as many rows as there are hexadecimal digits in a GUID, so for the prototype Pastry system that we are describing, there are 128/4 = 32 rows. Any row *n* contains 15 entries – one for each possible value of the *n*th hexadecimal digit, excluding the value in the local node's GUID. Each entry in the table points to one of the potentially many nodes whose GUIDs have the relevant prefix.

Pastry routing example *Based on Rowstron and Druschel [2001]*



Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing table the message can be delivered in ~ $\log_{16}(N)$ hops.

The routing process at any node *A* uses the information in its routing table *R*and leaf set *L* to handle each request from an application and each incoming message from another node according to the algorithm shown in Figure below.

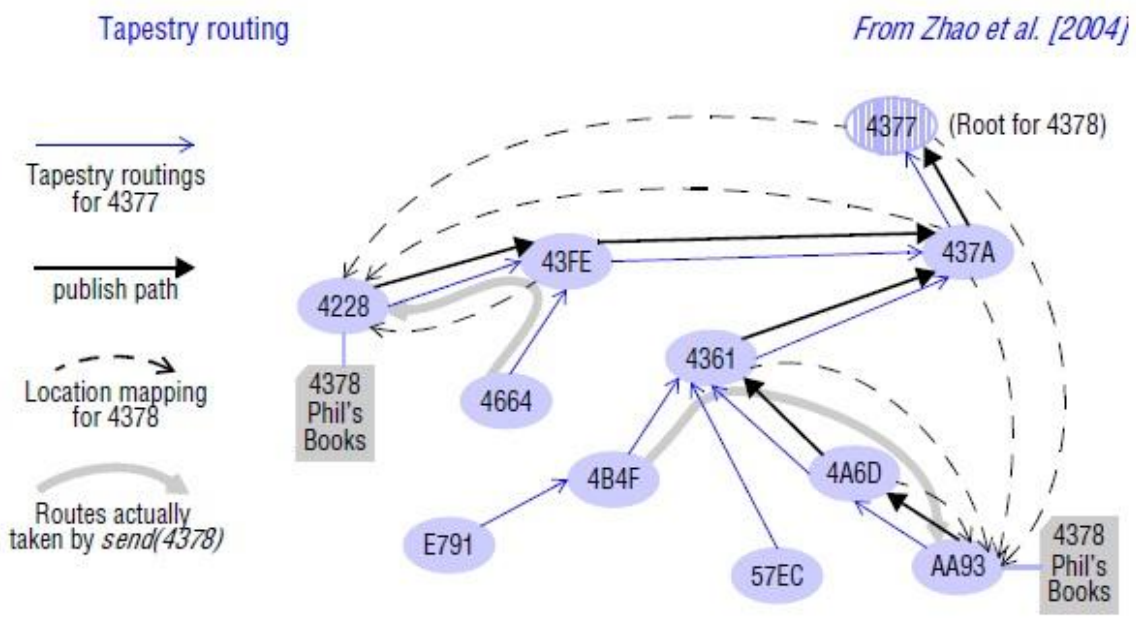
Pastry's routing algorithm

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* ($L_{-l} < D < L_l$) { // the destination is within the leaf set or is the current node.
2. Forward M to the element L_i of the leaf set with GUID closest to D or the current node A.
3. } *else* { // use the routing table to despatch M to a node with a closer GUID
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* ($R[p,i] \neq null$) forward M to $R[p,i]$ // route M to a node with a longer common prefix.
6. else { // there is no entry in the routing table.
7. Forward M to any node in L or R with a common prefix of length p but a 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. But Tapestry's API conceals the distributed hash table from applications behind a DOLR interface. Nodes that hold resources use the *publish(GUID)* primitive to make them known to Tapestry, and 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.
In Tapestry 160-bit identifiers are used to refer both to objects and to the nodes that perform routing actions. Identifiers are either *NodeIds*, which refer to computers that perform routing operations, or *GUIDs*, which refer to the objects. For any resource with GUID *G* there is a unique root node with GUID *RG* that is numerically closest to
G. Hosts *H* holding replicas of *G* periodically invoke *publish*(*G*) to ensure that newly arrived hosts become aware of the existence of *G*. On each invocation of *publish*(*G*) a publish message is routed from the invoker towards node *RG*. On receipt of a publish message *RG* enters (*G*, *IPH*), the mapping between *G* and the sending host's IP address in its routing table, and each node along the publication path caches the same mapping. This process is illustrated in Figure. When nodes hold multiple (*G*, *IP*) mappings for the same GUID, they are sorted by the network distance (round-trip time) to the IP address. For replicated objects this results in the selection of the nearest available replica of the object as the destination for subsequent messages sent to the object.

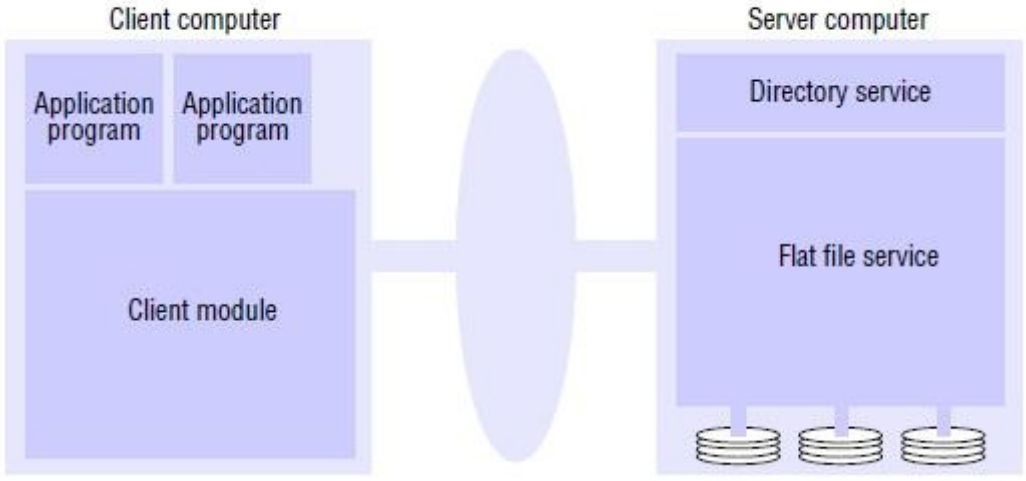


Replicas of the file *Phil's Books* (G=4378) are hosted at nodes 4228 and AA93. Node 4377 is the root node for object 4378. The Tapestry routings shown are some of the entries in routing tables. The publish paths show routes followed by the publish messages laying down cached location mappings for object 4378. The location mappings are subsequently used to route messages sent to 4378.

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* and a *client module*. The relevant modules and their relationships are shown in Figure below.

File service architecture



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 requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is

unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure below contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId*is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId*argument contains an invalid UFID or the user doesn't have sufficient access rights. These exceptions are omitted from the definition for clarity.

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).

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary.

Create creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.

GetAttributes and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the
flat file service itself.

Directory service interface • Figure below contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

Directory service operations

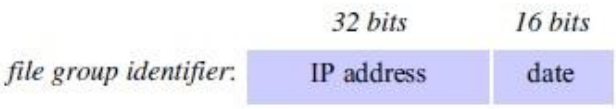
Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and 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*.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:



Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

ANDREW FILE SYSTEM

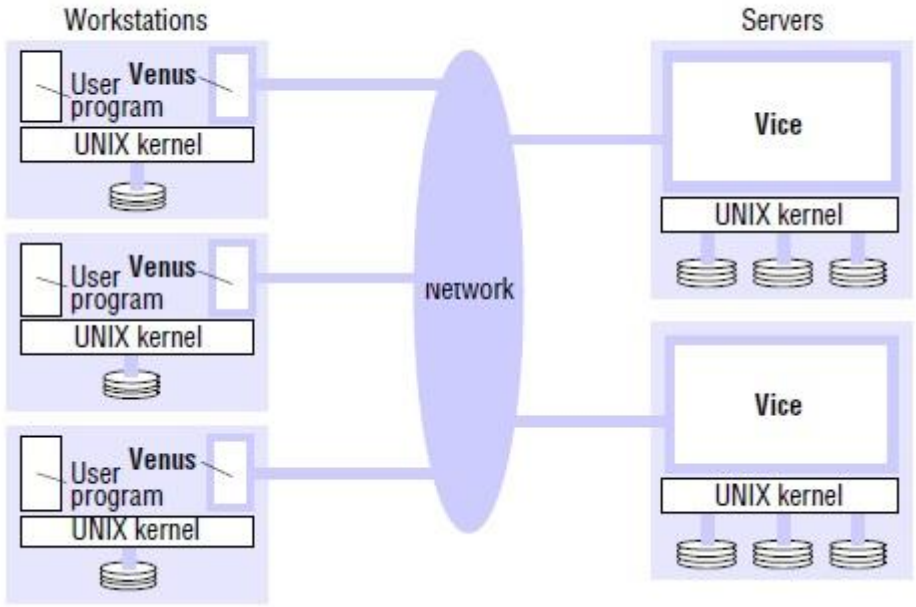
AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

- *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.

Implementation

AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*. Figure below shows the distribution of Vice and Venus processes. 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 corresponds to the client module in our abstract model.

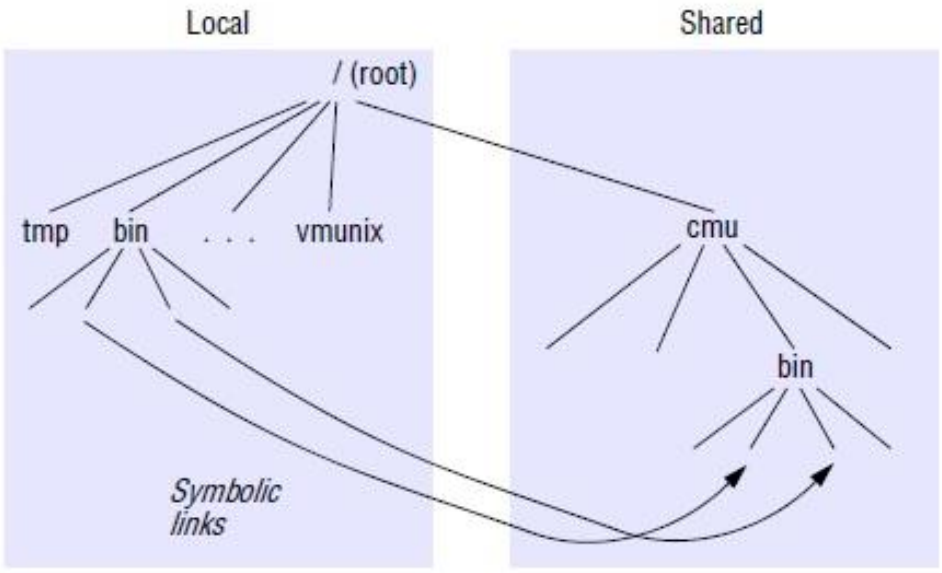
Distribution of processes in the Andrew File System



The files available to user processes running on workstations are either *local* or *shared*. Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes. Shared files are stored on servers, and copies of them are cached on the local disks of workstations. The name space seen by user processes is illustrated in Figure below. It is a conventional UNIX directory hierarchy, with a specific subtree (called *cmu*) containing all of the shared files. This splitting of the file name space into local and shared files leads to some loss of location transparency, but this is hardly noticeable to users other than system administrators. Local

files are used only for temporary files (*/tmp*) and processes that are essential for workstation startup. Other standard UNIX files (such as those normally found in */bin*, */lib* and so on) are implemented as symbolic links from local directories to files held in the shared space. Users' directories are in the shared space, enabling users to access their files from any workstation.

File name space seen by clients of AFS



The UNIX kernel in each workstation and server is a modified version of BSD UNIX. The modifications are designed to intercept *open*, *close* and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer (illustrated in Figure below).

System call interception in AFS

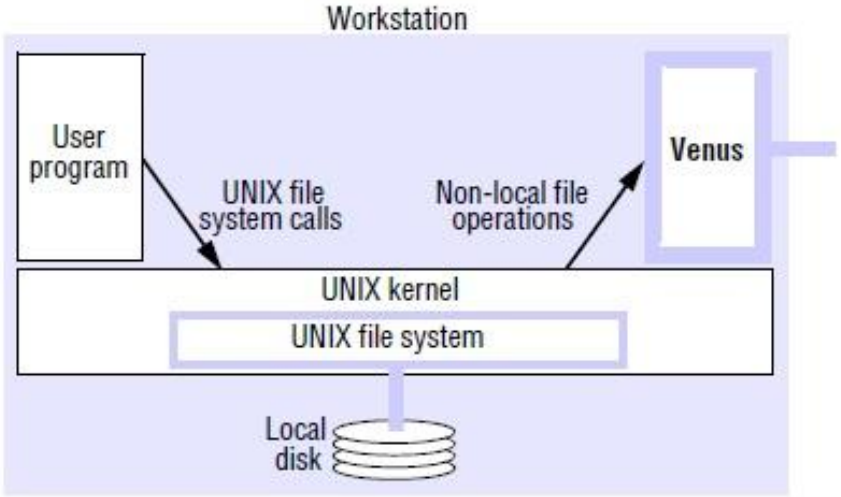


Figure below describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls mentioned in our outline scenario above. The *callback promise* mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after updating it.

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.

Cache consistency

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. Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*. 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. When the Venus process receives a callback, it sets the *callback promise* token for the relevant file to *cancelled*.

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. If its value is *cancelled*, then a fresh copy of the file must be fetched from the Vice server, but if the token is *valid*, then the cached copy can be opened and used without reference to Vice. Figure below shows the RPC calls provided by AFS servers for operations on files.

The main components of the Vice service interface

Fetch(fid) → *attr, data*	Returns the attributes (status) and, optionally, the contents of the file identified by *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() → *fid*	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs the server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	Call made by a Vice server to a Venus process; cancels the callback promise on the relevant file.

Note: Directory and administrative operations (*Rename*, *Link*, *Makedir*, *Removedir*, *GetTime*, *CheckToken* and so on) are not shown.

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. This is not practicable in large-scale systems; instead, the callback promise mechanism maintains a well-defined approximation to one-copy semantics.

For AFS-1, the update semantics can be formally stated in very simple terms. 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*)

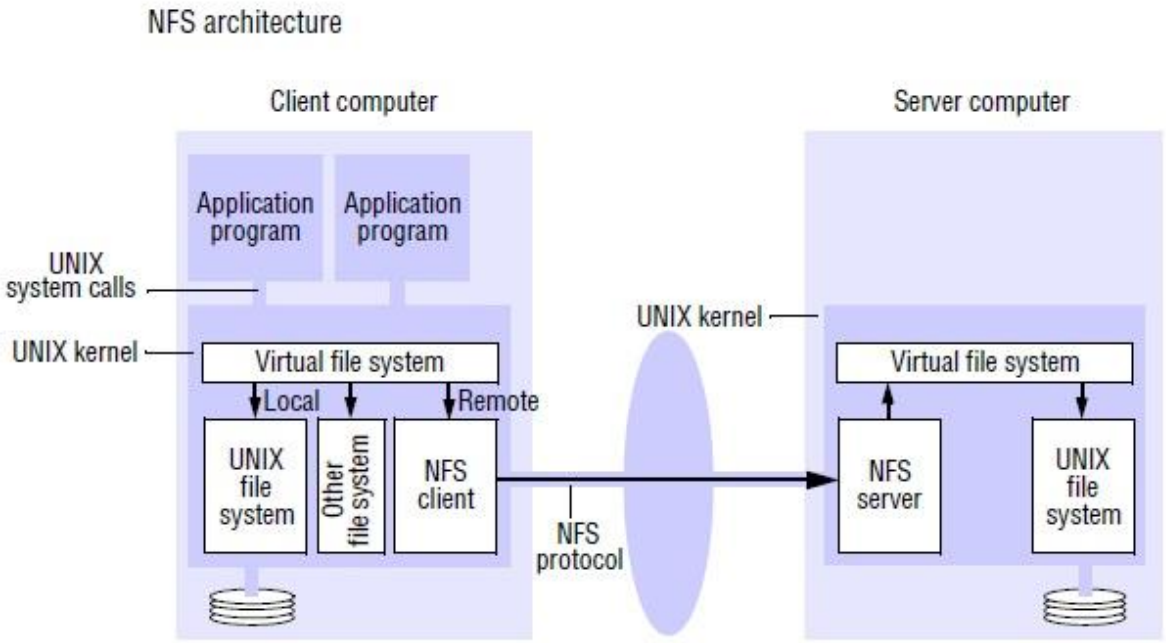
SUN NETWORK FILE SYSTEM

The Figure shows the architecture of Sun NFS. It follows the abstract model defined in the preceding section. All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol (version 3).

The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be

acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.



Virtual file system • The Figure above makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).

The file identifiers used in NFS are called *file handles*. A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file. In UNIX implementations of NFS, the file handle is derived from the file's *i-nodenumber* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):

File handle:	Filesystem identifier	i-node number of file	i-node generation number

NFS adopts the UNIX mountable filesystem as the unit of file grouping defined in the preceding section. The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system). The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed. In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused (for example, in a UNIX *creat* system call). The client obtains the first file handle for a remote file system when it mounts it. File handles are passed from server to client in the results of *lookup*, *create* and *mkdir* operations and from client to server in the argument lists of all server operations.

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But

unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:

- User programs can access files via UNIX system calls without recompilation or reloading;
- A single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);
- The encryption key used to authenticate user IDs passed to the server (see below) can be retained in the kernel, preventing impersonation by user-level clients.

NFS server interface • A simplified representation of the RPC interface provided by NFS version 3 servers (defined in RFC 1813 [Callaghan *et al.* 1995]) is shown in Figure below.

NFS server operations (NFS version 3 protocol, simplified)

lookup(dirfh, name) → *fh, attr*	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → *newfh, attr*	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → *status*	Removes file *name* from directory *dirfh*.
getattr(fh) → *attr*	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) → *attr*	Sets the attributes (mode, user ID, group ID, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → *attr, data*	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) → *attr*	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → *status*	Changes the name of file *name* in directory *dirfh* to *toname* in directory *todirfh*.
link(newdirfh, newname, fh) → *status*	Creates an entry *newname* in the directory *newdirfh* that refers to the file *or* directory *fh*.
symlink(newdirfh, newname, string) → *status*	Creates an entry *newname* in the directory *newdirfh* of type *symbolic link* with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) → *string*	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) → *newfh, attr*	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) → *status*	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) → *entries*	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) → *fsstats*	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

The file and directory operations are integrated in a single service; the creation and insertion of file names in directories is performed by a single *create* operation, which takes the text name of the new file and the file handle for the target directory as arguments. The other NFS operations on directories are *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* and *statfs*. They resemble their UNIX counterparts with the exception of *readdir*, which provides a representation independent method for reading the contents of directories, and *statfs*, which gives the status information on remote file systems.

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested. The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes. These additional parameters are not shown in our overview of the NFS protocol in Figure; they are supplied automatically by the RPC system.

In its simplest form, there is a security loophole in this access-control mechanism. An NFS server provides a conventional RPC interface at a well-known port on each host and any process can behave as a client, sending requests to the server to access or update a file. The client can modify the RPC calls to include the user ID of any user, impersonating the user without their knowledge or permission. This security loophole has been closed by the use of an option in the RPC protocol for the DES encryption of the user's authentication information. More recently, Kerberos has been integrated with Sun NFS to provide a stronger and more comprehensive solution to the problems of user authentication and security.

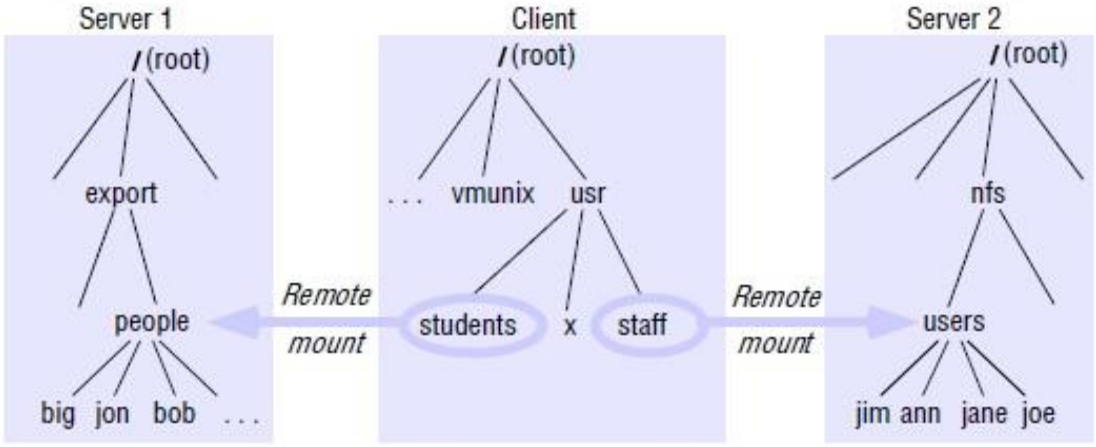
Pathname translation • UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process whenever the *open*, *creat*or *stat* system calls are used. In NFS, pathnames cannot be translated at a server, because the name may cross a 'mount point' at the client – directories holding different parts of a multi-part name may reside in filesystems at different servers. So pathnames are parsed, and their translation is performed in an iterative manner by the client. Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate *lookup* request to the remote server.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for remote mounting. An access list is associated with each filesystem name indicating which hosts are permitted to mount the filesystem.

Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted. The remote directory may be any subtree of the required remote filesystem, enabling clients to mount any part of the remote filesystem. The modified *mount* command communicates with the mount service process on the remote host using a *mount protocol*. This is an RPC protocol and includes an operation that takes a directory pathname and returns the file handle of the specified directory if the client has access permission for the relevant filesystem. The location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client.

Figure below illustrates a *Client* with two remotely mounted file stores. The nodes *people* and *users* in filesystems at *Server 1* and *Server 2* are mounted over nodes *students* and *staff* in *Client*'s local file store. The meaning of this is that programs running at *Client* can access files at *Server 1* and *Server 2* by using pathnames such as */usr/students/jon*and */usr/staff/ann*.

Local and remote filesystems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.

Automounter • The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client. The original implementation of the automounter ran as a userlevel UNIX process in each client computer. Later versions (called *autofs*) were implemented in the kernel for Solaris and Linux.

Server caching • Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance.
In conventional UNIX systems, file pages, directories and file attributes that have been read from disk are retained in a main memory *buffer cache* until the buffer space is required for other pages. If a process then issues a read or a write request for a page that is already in the cache, it can be satisfied without another disk access. *Read-ahead* anticipates read accesses and fetches the pages following those that have most recently been read, and *delayed-write* optimizes writes: when a page has been altered (by a write request), its new contents are written to disk only when the buffer page is required for another page. To guard against loss of data in a system crash, the UNIX *sync* operation flushes altered pages to disk every 30 seconds. These caching techniques work in a conventional UNIX environment because all read and write requests issued by userlevel processes pass through a single cache that is implemented in the UNIX kernel space. The cache is always kept up-to-date, and file accesses cannot bypass the cache.

Client caching • The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

NAME SERVICES

A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming *contexts*: individual subsets of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name.

Name spaces

A *name space* is the collection of all valid names recognized by a particular service. The service will attempt to look up a valid name, even though that name may prove not to correspond to any object – i.e., to be *unbound*. Name spaces require a syntactic definition to separate valid names from invalid names. For example, '...' is not acceptable as the DNS name of a computer, whereas *www.cdk99.net* is valid (even though it is unbound).

Names may have an internal structure that represents their position in a hierarchic name space such as pathnames in a file system, or in an organizational hierarchy such as Internet domain names; or they may be chosen from a flat set of numeric or symbolic identifiers. One important advantage of a hierarchy is that it makes large name spaces more manageable. Each part of a hierarchic name is resolved relative to a separate context of relatively small size, and the same name may be used with different meanings in different contexts, to suit different situations of use. In the case of file systems, each directory represents a context. Thus */etc/passwd*is a hierarchic name with two components. The first, 'etc', is resolved relative to the context '/', or root, and the second part, 'passwd', is relative to the context '/*etc*'. The name */oldetc/passwd*can have a different meaning because its second component is resolved in a different context. Similarly, the same name */etc/passwd* may resolve to different files in the contexts of two different computers.

The DNS name space has a hierarchic structure: a domain name consists of one or more strings called *name components* or *labels*, separated by the delimiter '.'. There is no delimiter at the beginning or end of a domain name, although the root of the DNS name space is sometimes referred to as '.' for administrative purposes. The name components are non-null printable strings that do not contain '.'. In general, a *prefix* of a name is an initial section of the name that contains only zero or more entire components. For example, in DNS *www* and *www.cdk5* are both prefixes of *www.cdk5.net*. DNS names are not case-sensitive, so *www.cdk5.net* and *WWW.CDK5.NET* have the same meaning.

Aliases • An *alias* is a name defined to denote the same information as another name, similar to a symbolic link between file path names. Aliases allow more convenient names to be substituted for relatively complicated ones, and allow alternative names to be used by different people for the same entity. An example is the common use of URL shorteners, often used in Twitter posts and other situations where space is at a premium. For example, using web redirection, *http://bit.ly/ctqjvH* refers to *http://cdk5.net/additional/rmi/programCode/ShapeListClient.java*. As another example, the DNS allows aliases in which one domain name is defined to stand for another. Aliases are often used to specify the names of machines that run a web server or an FTP server. For example, the name *www.cdk5.net* is an alias for *cdk5.net*. This has the advantage that clients can use either name for the web server, and if the web server is moved to another computer, only the entry for *cdk5.net* needs to be updated in the DNS database.

Naming domains • A *naming domain* is a name space for which there exists a single overall administrative authority responsible for assigning names within it. This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task.

Name resolution

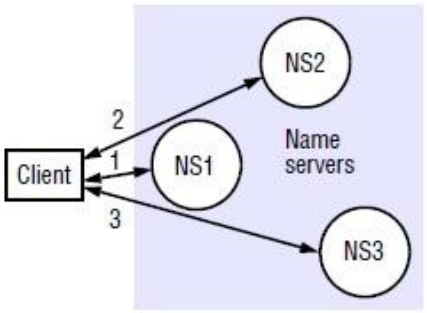
For the common case of hierarchic name spaces, name resolution is an iterative or recursive process whereby a name is repeatedly presented to naming contexts in order to look up the attributes to which it refers. A naming context either maps a given name onto a set of primitive attributes (such as those of a user) directly, or maps it onto a further naming context and a derived name to be presented to that context. To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output.

Name servers and navigation • Any name service, such as DNS, that stores a very large database and is used by a large population will not store all of its naming information on a single server computer. Such a server would be a bottleneck and a critical point of failure. Any heavily used name services should use replication to achieve high availability. We shall see that DNS specifies that each subset of its database is replicated in at least two failure-independent servers.

One navigation model that DNS supports is known as *iterative navigation* (see Figure below). To resolve a name, a client presents the name to the local name server, which attempts to resolve it. If the local name server has the name, it returns the result immediately. If it does not, it will suggest another

server that will be able to help. Resolution proceeds at the new server, with further navigation as necessary until the name is located or is discovered to be unbound.

Iterative navigation

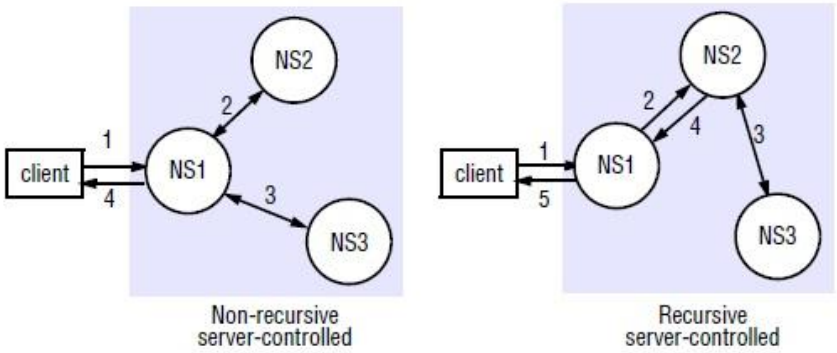


A client iteratively contacts name servers NS1–NS3 in order to resolve a name

In *multicast navigation*, a client multicasts the name to be resolved and the required object type to the group of name servers. Only the server that holds the named attributes responds to the request.

Another alternative to the iterative navigation model is one in which a name server coordinates the resolution of the name and passes the result back to the user agent. Ma [1992] distinguishes *non-recursive* and *recursive server-controlled navigation*. Under non-recursive server-controlled navigation, any name server may be chosen by the client. This server communicates by multicast or iteratively with its peers in the style described above, as though it were a client. Under recursive server-controlled navigation, the client once more contacts a single server. If this server does not store the name, the server contacts a peer storing a (larger) prefix of the name, which in turn attempts to resolve it. This procedure continues recursively until the name is resolved.

Non-recursive and recursive server-controlled navigation



A name server NS1 communicates with other name servers on behalf of a client

DOMAIN NAME SYSTEM

The Domain Name System is a name service design whose main naming database is used across the Internet. It was devised principally by Mockapetris and specified in RFC1034 [Mockapetris 1987] and RFC 1035. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them [Harrenstien*et al.* 1985].

This original scheme was soon seen to suffer from three major shortcomings:

- It did not scale to large numbers of computers.
- Local organizations wished to administer their own naming systems.
- A general name service was needed – not one that serves only for looking up computer addresses.

The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes – and what we have referred to in this chapter as naming domains are called simply *domains* in the DNS. In principle, however, any type of object can be named, and its architecture gives scope for a variety of implementations. Organizations and departments within them can manage their own naming data. Millions of names are bound by the Internet DNS, and lookups are made against it from around the world. Any name can be resolved by any client. This is achieved by hierarchical partitioning of the name database, by replication of the naming data, and by caching.

Domain names • The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in wide spread use, and that is the one used for naming across the Internet. The Internet DNS namespace is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

com– Commercial organizations
edu– Universities and other educational institutions
gov– US governmental agencies
mil– US military organizations
net– Major network support centres
org– Organizations not mentioned above
int– International organizations

In addition, every country has its own domains:

us– United States
uk– United Kingdom
fr– France

DNS queries • The Internet DNS is primarily used for simple host name resolution and for looking up electronic mail hosts, as follows:

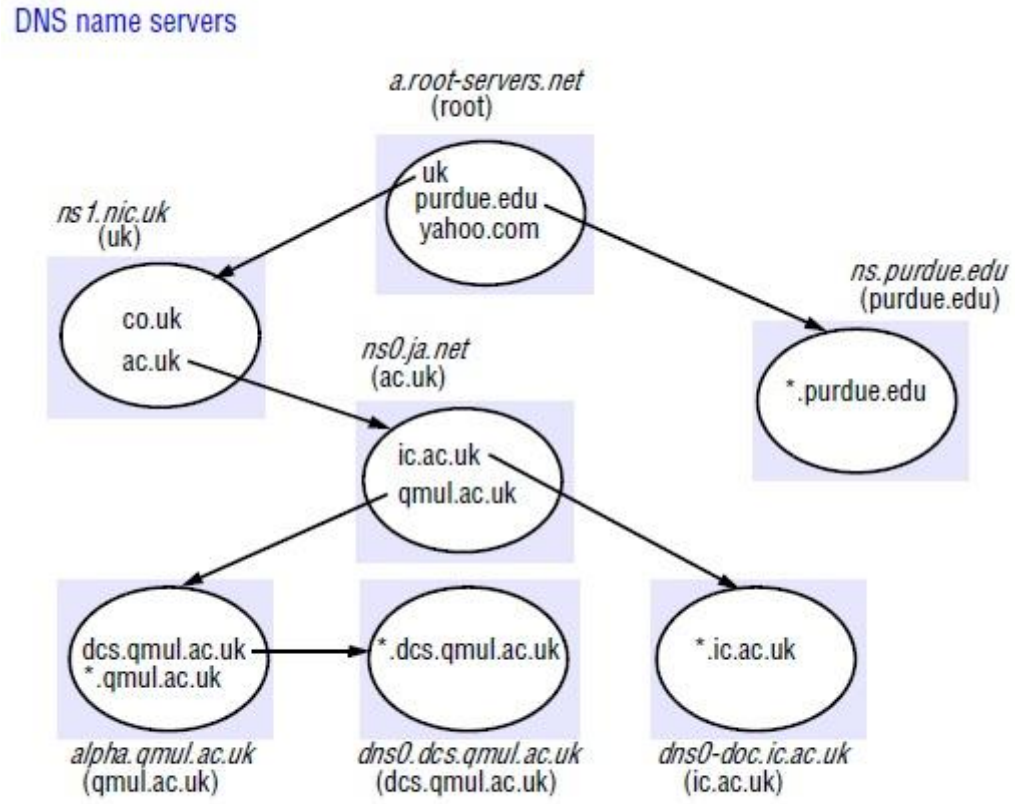
Host name resolution: In general, applications use the DNS to resolve host names into IP addresses. For example, when a web browser is given a URL containing the domain name *www.dcs.qmul.ac.uk*, it makes a DNS enquiry and obtains the corresponding IP address. As was pointed out in Chapter 4, browsers then use HTTP to communicate with the web server at the given IP address, using a reserved port number if none is specified in the URL. FTP and SMTP services work in a similar way; for example, an FTP program may be given the domain name *ftp.dcs.qmul.ac.uk*and can make a DNS enquiry to get its IP address and then use TCP to communicate with it at the reserved port number.

Mail host location: Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains. For example, when the address *tom@dcs.rnx.ac.uk* is to be resolved, the DNS is queried with the address *dcs.rnx.ac.uk* and the type designation 'mail'. It returns a list of domain names of hosts that can accept mail for *dcs.rnx.ac.uk*, if such exist (and, optionally, the corresponding IP addresses). The DNS may return more than one domain name so that the mail software can try alternatives if the main mail host is unreachable for some reason. The DNS returns an integer preference value for each mail host, indicating the order in which the mail hosts should be tried.

DNS name servers • The problems of scale are treated by a combination of partitioning the naming database and replicating and caching parts of it close to the points of need. The DNS database is distributed across a logical network of servers. Each server holds part of the naming database – primarily data for the local domain. Queries concerning computers in the local domain are satisfied by servers within that domain. However, each server records the domain names and addresses of other name servers, so that queries pertaining to objects outside the domain can be satisfied.

Figure below shows the arrangement of some of the DNS database as it stood in the year 2001. This example is equally valid today even if some of the data has altered as systems have been reconfigured over time. Note that, in practice, root servers such as*a.root-servers.net* hold entries for several levels of domain, as well as entries for first level domain names. This is to reduce the number of navigation steps required to resolve domain names. Root name servers hold authoritative entries for the name servers for the top-level domains. They are also authoritative name servers for the generic top-level domains, such as *com* and *edu*. However, the root name servers are not name servers for the

country domains. For example, the *uk* domain has a collection of name servers, one of which is called *ns1.nic.net*. These name servers know the name servers for the second-level domains in the United Kingdom such as *ac.uk* and *co.uk*. The name servers for the domain *ac.uk* know the name servers for all of the university domains in the country, such as *qmul.ac.uk* or *ic.ac.uk*. In some cases, a university domain delegates some of its responsibilities to a sub domain, such as *dcs.qmul.ac.uk*.



Name server names are in italics, and the corresponding domains are in parentheses. Arrows denote name server entries