Unit – V – Distributed File Systems

6.1. 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. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disk.

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet.
- A well designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks.
- The sharing of resources is a key goal for distributed systems. The sharing of stored information is the most important aspect of distributed resource sharing.

- The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data.

- 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.

- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

	Sharing	*Persis-tence*	*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
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	✓	OceanStore

Types of Consistency:

1. Strict One copy- Slightly weaker guarantees
2. Considerably weaker guarantees

Characteristics of file systems:

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.
- Files contain both *data and attributes. The data consist of a sequence of data items that* accessible by operations to read and write any portion of the sequence.
- The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists.

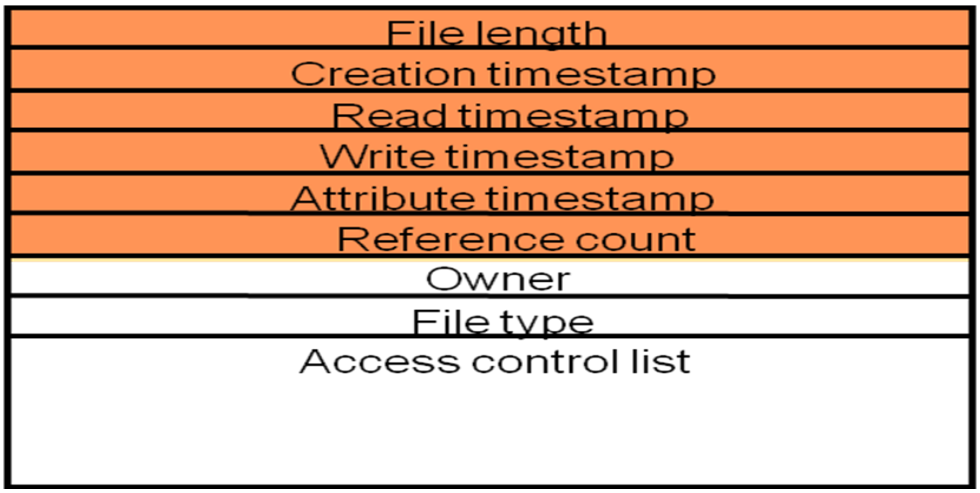
File System Module: In this, defined an abstract model for a basic distributed file service, including a set of programming interfaces.

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

Figure shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files

File attributes record structure:

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated in Figure. The shaded attributes are managed by the file system and are not normally updatable by user programs.



File system operations:

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

Distributed file system requirements:

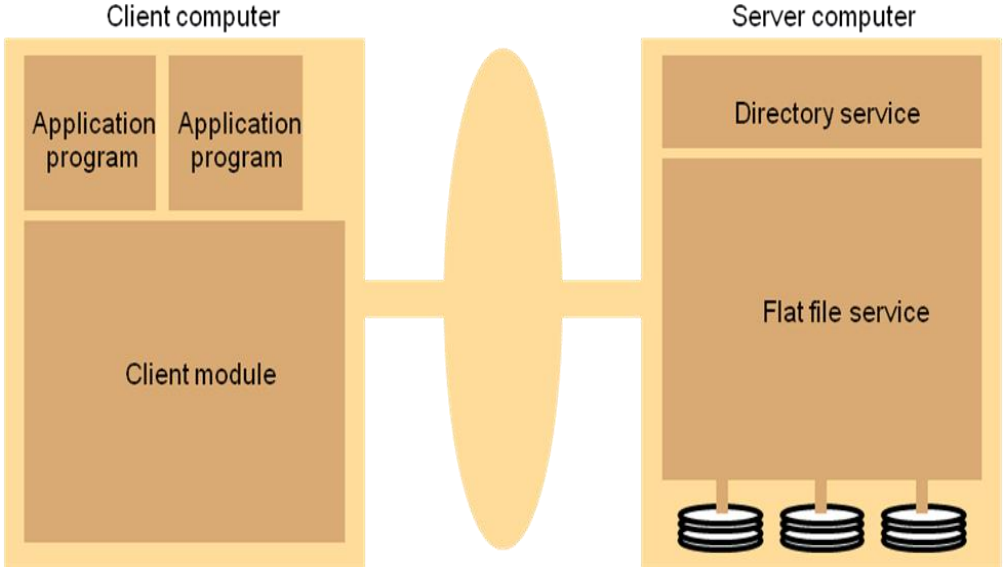
- Related requirements in distributed file systems are:
  - **T*ransparency*:** The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems. They are **Access transparency, Location transparency, Mobility transparency, Performance transparency, Scaling transparency**.
  - ***Concurrency*:** Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control.
  - ***Replication*:** A file may be represented by several copies of its contents at different locations.
  - ***Heterogeneity*:** Cclient and server software can be implemented for different operating systems and computers.

- ***Fault tolerance*:** The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures.
- ***Consistency*:** See same directory and file contents on different clients at same time.
- **S*ecurity :*** Secure communication and user authentication
- **Efficiency:** A distributed file system should be accessed high performance.

6.2. File Service Architecture: An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:

- A flat file service
- A directory service
- A client module.

- The relevant modules and their relationship is shown in Figure.



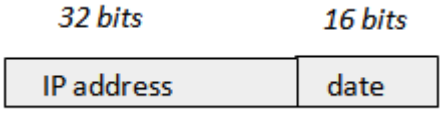
- **A flat file service** is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. With the use of UFIDs each file has a unique among all of the files in a distributed system.
  - Table contains a definition of the interface to a flat file service.

Read(FileId, i, n) ->	: Reads a sequence of data upto n items from a file starting at item *i* and returns it in *Data.*
Write(FileId, i, Data)	: Write a sequence of *Data* to a file, starting item i, extending the file if necessary.
Create() -> FileId	Creates a new file of length0 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 (i.e. File attribute record structure.)

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

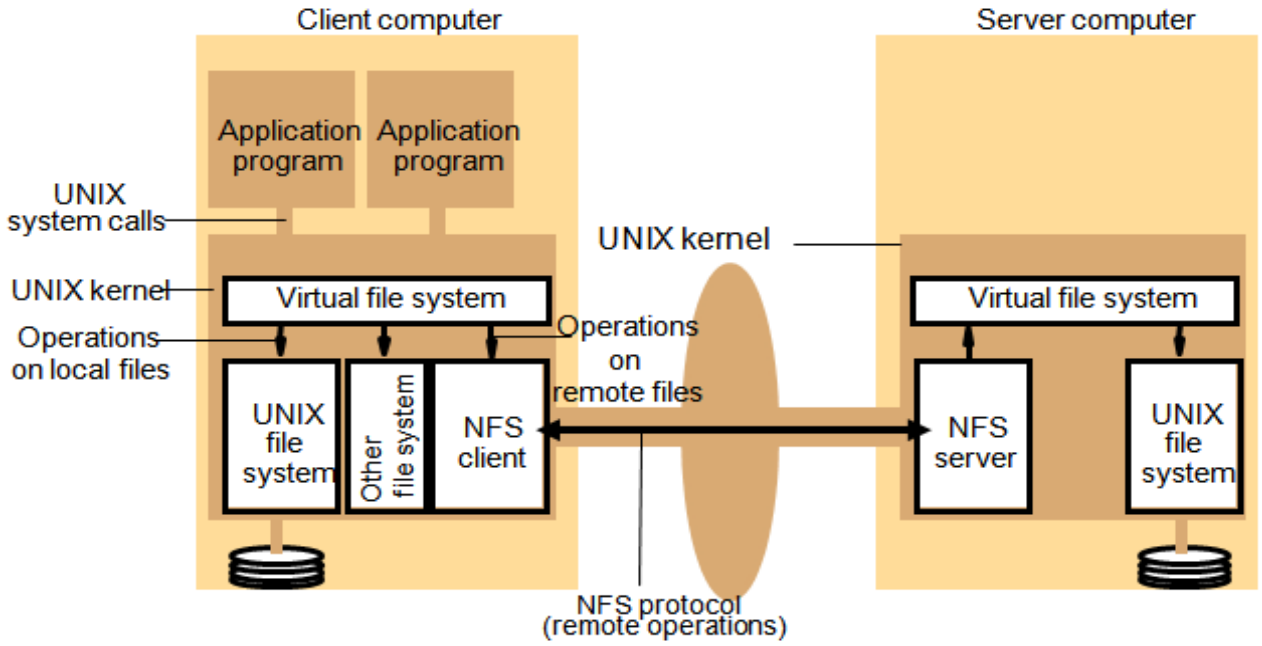
Lookup(Dir, Name) -> FileId -throws NotFound	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, File) -throws NameDuplicate	If *Name* is not in the directory, adds(*Name,File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name)	If *Name* is in the directory, the entry containing *Name* is removed 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*.

- **Client module** 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.
  - A client model also holds information about the network locations of flat-file and directory server processes and achieves better performance through implementation of a cache of recently used file blocks at the client.
- Access control: In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
- **Directory service interface**: Figure 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.
- **Hierarchic file system :**
  - A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.
- **File Group :**
  - A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.
    - A similar construct is used in a UNIX file system.
    - It helps with distributing the load of file serving between several servers.
    - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).
  - To construct a globally unique ID we use some unique attribute of the machine on which it is created, e.g. IP number, even though the file group may move subsequently.



Examples for File Service architecture are NFS, AFS

6.2.1: Network File System (NFS) architecture:



The NFS protocol was originally developed for use in networks of UNIX systems. 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.

- The file identifiers used in NFS are called file handles. In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* 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

Sun Network File System

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

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 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.

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.

NFS server operations

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) → *statu*	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,nam, 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 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 the file fh.

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.

NFS server interface : A simplified representation of the RPC interface provided by NFS version 3 servers. The NFS file access operations *read*, *write*, *getattr* and *setattr* are almost identical to the *Read*, *Write*, *GetAttributes* and *SetAttributes* operations defined for our flat file service model. 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.

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.

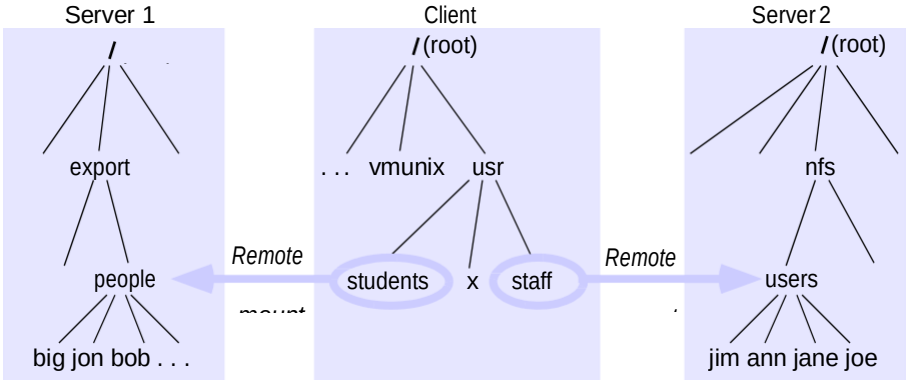


Figure: Local and remote file system accessible on an NFS client

Figure 12.10 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.

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. The *lookup* operation looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes.

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 user- level 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 NFS Protocol write operation offers two points

1. Data in *write* operations received from clients is stored in the memory cache at the server *and* written to disk before a reply is sent to the client. This is called *write- through* caching. The client can be sure that the data is stored persistently as soon as the reply has been received.
2. Data in *write* operations is stored only in the memory cache. It will be written to disk when a *commit* operation is received for the relevant file. The client can be sure that the data is persistently stored only when a reply to a *commit* operation for the relevant file has been received. Standard NFS clients use this mode of operation, issuing a *commit* whenever a file that was open for writing is closed.

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.

Each data or metadata item in the cache is tagged with two timestamps:

Tc is the time when the cache entry was last validated.

Tm is the time when the block was last modified at the server.

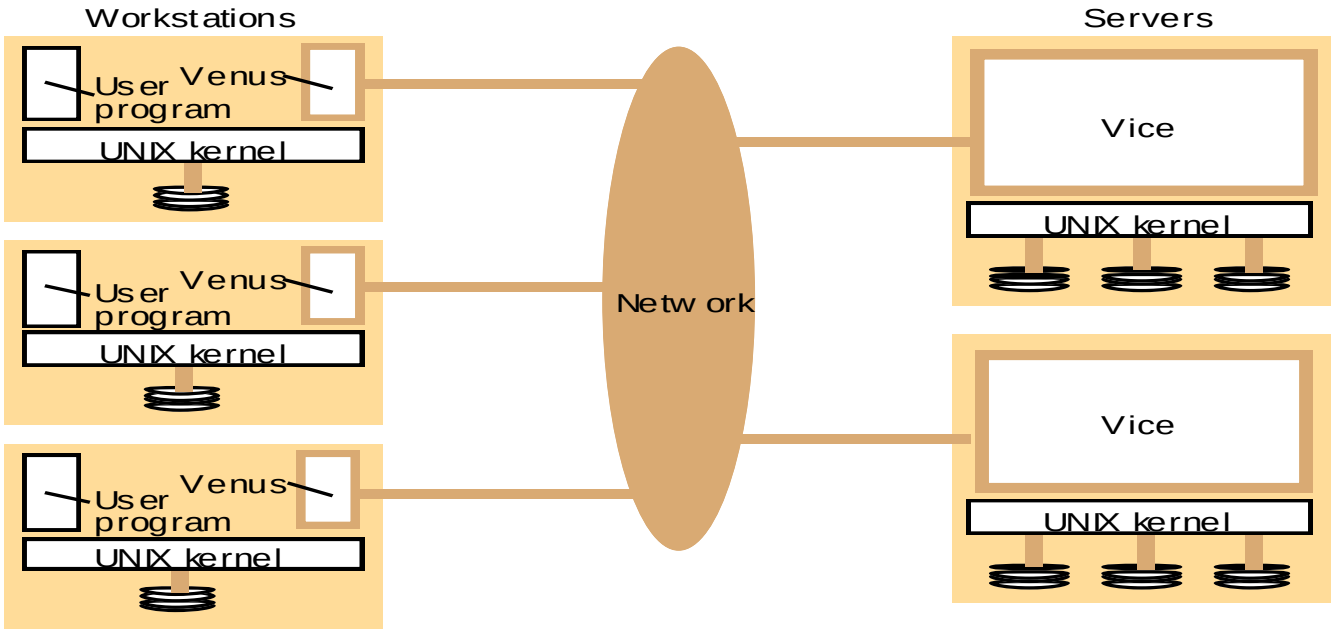
$(T-Tc<t) \vee (Tm_{client} = Tm_{server})$

6.2.2: AFS architecture:

- Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus.
- Figure 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.
- 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 the 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.

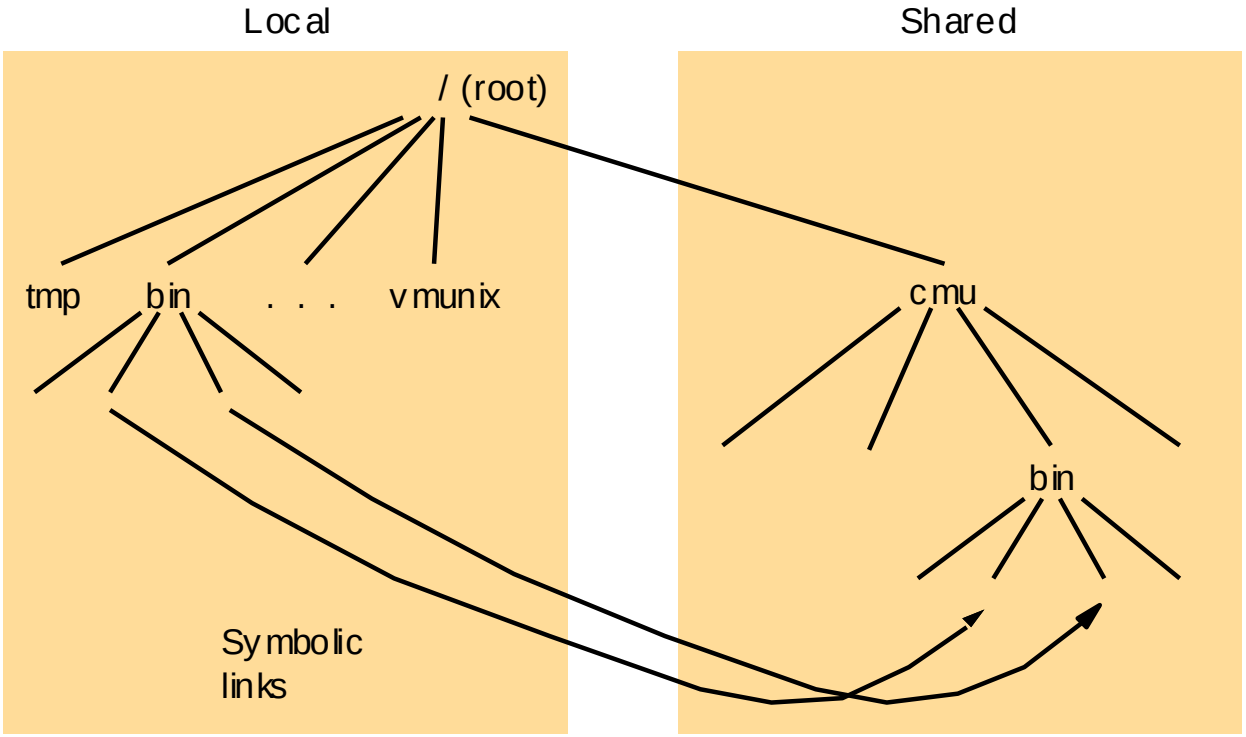
Implementation



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

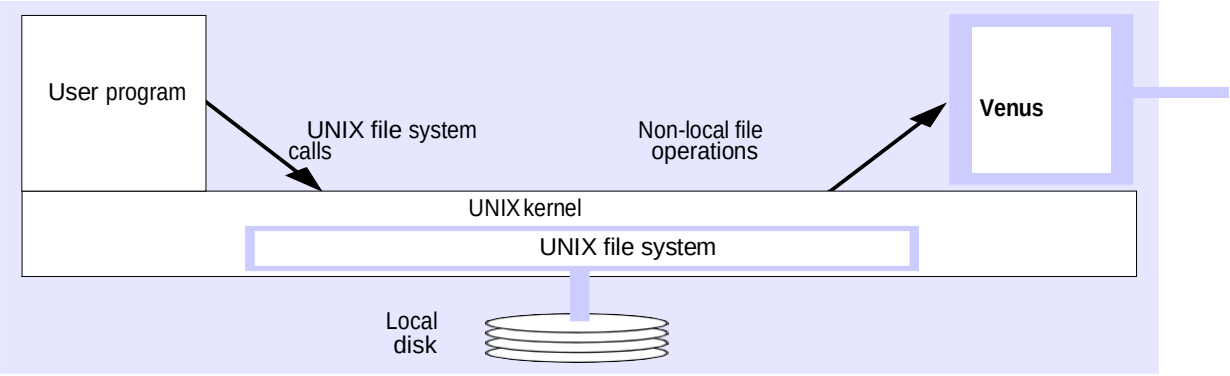
Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

- The name space seen by user processes is illustrated in Figure.



- It is a conventional UNIX directory hierarchy, with a specific sub-tree (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.

System call interception in AFS



Files are grouped into *volumes* for ease of location and movement. Volumes are generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS. For example, each user's personal files are generally located in a separate volume. Other volumes are allocated for system binaries, documentation and library code.

The representation of *fid*s includes the volume number for the volume containing the file (*cf.* the *file group identifier* in UFIDs), an NFS file handle identifying the file within the volume (*cf.* the *file number* in UFIDs) and a *uniquifier* to ensure that file identifiers are not reused:

File Identifier(fid's)

32 bits	32 bits	32 bits
Volume number	File handle	Uniquifier

Other Aspects

AFS introduces several other interesting design developments and refinements that we outline here, together with a summary of performance evaluation results:

UNIX kernel modifications • We have noted that the Vice server is a user-level process running in the server computer and the server host is dedicated to the provision of an

AFS service. The UNIX kernel in AFS hosts is altered so that Vice can perform file operations in terms of file handles instead of the conventional UNIX file descriptors. This is the only kernel modification required by AFS, and it is necessary if Vice is not to maintain any client state (such as file descriptors).

Location database • Each server contains a copy of a fully replicated location database giving a mapping of volume names to servers. Temporary inaccuracies in this database may occur when a volume is moved, but they are harmless because forwarding information is left behind in the server from which the volume is moved.

Threads • The implementations of Vice and Venus make use of a non-preemptive threads package to enable requests to be processed concurrently at both the client (where several user processes may have file access requests in progress concurrently) and the server. In the client, the tables describing the contents of the cache and the volume database are held in memory that is shared between the Venus threads.

Read-only replicas • Volumes containing files that are frequently read but rarely modified, such as the UNIX */bin* and */usr/bin* directories of system commands and */man* directory of manual pages, can be replicated as read-only volumes at several servers. When this is done, there is only one read-write replica and all updates are directed to it. The propagation of the changes to the read-only replicas is performed

after the update by an explicit operational procedure. Entries in the location database for volumes that are replicated in this way are one-to-many, and the server for each client request is selected on the bases of server loads and accessibility.

Bulk transfers • AFS transfers files between clients and servers in 64-kilobyte chunks. The use of such a large packet size is an important aid to performance, minimizing the effect of network latency. Thus the design of AFS enables the use of the network to be optimized.

Partial file caching • The need to transfer the entire contents of files to clients even when the application requirement is to read only a small portion of the file is an obvious source of inefficiency. Version 3 of AFS removed this requirement, allowing file data to be transferred and cached in 64-kbyte blocks while still retaining the consistency semantics and other features of the AFS protocol.

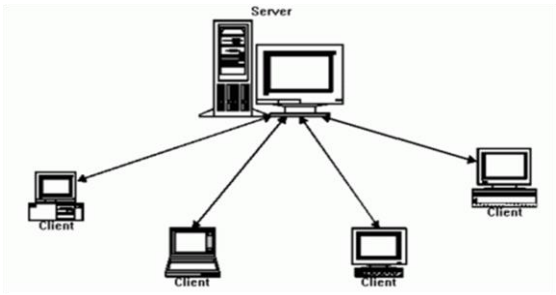
Performance • The primary goal of AFS is scalability, so its performance with large numbers of users is of particular interest. Howard *et al.* [1988] give details of extensive comparative performance measurements, which were undertaken using a specially developed *AFS benchmark* that has subsequently been widely used for the evaluation of distributed file systems. Not surprisingly, whole-file caching and the callback protocol led to dramatically reduced loads on the servers. Satyanarayanan [1989a] states that a server load of 40% was measured with 18 client nodes running a standard benchmark, against a load of 100% for NFS running the same benchmark. Satyanarayanan attributes much of the performance advantage of AFS to the reduction in server load deriving from the use of callbacks to notify clients of updates to files, compared with the timeout mechanism used in NFS for checking the validity of pages cached at clients.

Wide area support: • Version 3 of AFS supports multiple administrative cells, each with its own servers, clients, system administrators and users. Each cell is a completely autonomous environment, but a federation of cells can cooperate in presenting users with a uniform, seamless file name space. The resulting system was widely deployed by the Transarc Corporation, and a detailed survey of the resulting performance usage patterns was published [Spasojevic and Satyanarayanan 1996]. The system was installed on over 1000 servers at over 150 sites. The survey showed cache hit ratios in the range of 96 – 98% for accesses to a sample of 32,000 file volumes holding 200 Gbytes of data.

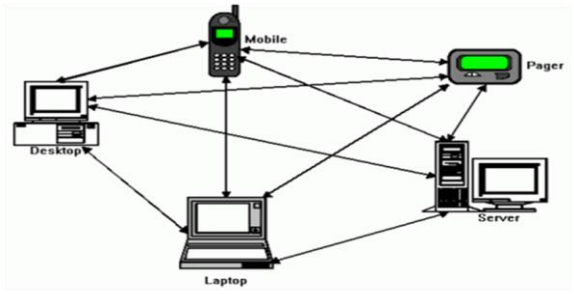
6.3. Peer-to-Peer Systems: Peer to peer is an approach to computer networking where all computers share equivalent responsibility for processing data. Peer-to-peer networking (also known simply as *peer networking*) differs from client-server networking, where certain devices have responsibility for providing or "serving" data and other devices consume or otherwise act as "clients" of those servers.

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

- Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the Internet and other networks in ever-increasing numbers.
- An alternative to the client/server model of distributed computing is the peer-to-peer model.
- Client/server is naturally hierarchical, with resources centralized on a limited number of servers.
- In peer-to-peer networks, both resources and control are widely distributed among nodes that are theoretically equals. (A node with more information, better information, or more power may be "*more equal,*" but that is a function of the node, not the network controllers.)
- A key feature of peer-to-peer networks is decentralization. This has many implications. Robustness, availability of information and fault-tolerance tends to come from redundancy and shared responsibility instead of planning, organization and the investment of a controlling authority.
- Peer-to-peer applications provide better communication for 'the applications which exploit resources available at the edges of the Internet such as storage, cycles, content, human presence'. Each type of resource sharing mentioned in that definition is already represented by distributed applications available for most types of personal computer.



The Client/Server Model



The Peer-to-Peer Model

Characteristics:

1. Each user contributes resources to the system.
2. All the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
3. Correct operation does not depend on the existence of any centrally administered systems.
4. Limited degree of anonymity to the providers and users of resources.
5. Efficient operation is the choice of an algorithm.

Advantages of Peer-to-peer networking over Client –Server networking are :-

1) It is easy to install and so is the configuration of computers on this network,
2) All the resources and contents are shared by all the peers, unlike server-client Architecture where Server shares all the contents and resources.
3) P2P is more reliable as central dependency is eliminated. Failure of one peer doesn't affect the functioning of other peers. In case of Client –Server network, if server goes down whole network gets affected.
4) There is no need for full-time System Administrator. Every user is the administrator of his machine. User can control their shared resources.
5) The over-all cost of building and maintaining this type of network is comparatively very less.

Disadvantages (drawbacks) of Peer to peer architecture over Client Server are:-

1) In this network, the whole system is decentralized thus it is difficult to administer. That is one person cannot determine the whole accessibility setting of whole network.

2) Security in this system is very less viruses, spywares, trojans, etc malwares can easily transmitted over this P-2-P architecture.

3) Data recovery or backup is very difficult. Each computer should have its own back-up system.

4) Lot of movies, music and other copyrighted files are transferred using this type of file transfer. P2P is the technology used in torrents.

Note: Peer to peer networks are good to connect small number (around 10) of computer and places where high level of security is not required. In case of business network where sensitive data can be present this type of architecture is not advisable or preferred.

Applications:

- Theory
  - Dynamic discovery of information
  - Better utilization of bandwidth, processor, storage, and other resources
  - Each user contributes resources to network
- Practice examples
  - Sharing browser cache over 100Mbps lines
  - Disk mirroring using spare capacity
  - Deep search beyond the web

6.3.1.Peer-to-peer middleware: The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed resources on a global scale. Several research teams have now completed the development, evaluation and refinement of peer-to-peer middleware platforms and demonstrated or deployed them in a range of application services.

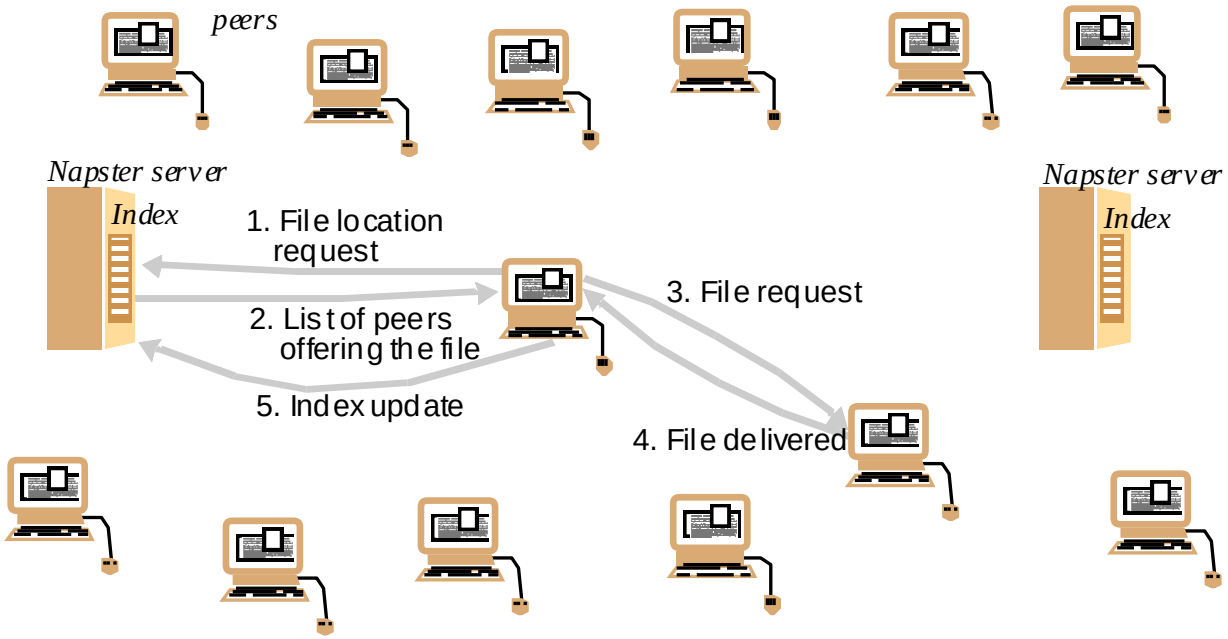
6.3.2. Routing Overlays: Routing overlays share many characteristics with the IP packet routing infrastructure that constitutes the primary communication mechanism of the Internet. It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems.

6.4. Napster and its legacy: Downloading the digital music files is the first application which globally scalable storage and retrieval in service. Both need and feasibility of a peer-to-peer solution was first demonstrated by the Napster file sharing system 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 and providing access to 'shared resources at the edges of the Internet'.

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



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.

6.5. Peer-to-peer middleware:

Peer-to-Peer Middleware is to provide mechanism to access data resources anywhere in network. A key problem in Peer-to-Peer applications is to provide a way for clients to access data resources efficiently. Similar needs in client/server technology led to solutions like NFS. However, NFS relies on pre-configuration and is not scalable enough for peer-to-peer.

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

Peer clients need to locate and communicate with any available resource, even though resources may be widely distributed and configuration may be dynamic, constantly adding and removing resources and connections.

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 :
  - 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 :
  - **Global Scalability**: peer-to-peer applications are to exploit the hardware resources of very large numbers of hosts connected to the Internet. So it that access millions of objects on tens of thousands or hundreds of thousands of hosts.
  - **Load Balancing**: The performance of any system is designed to exploit a large number of computers depends upon the balanced distribution of workload across them. This will be achieved by a random placement of resources together with the use of replicas of heavily used resources.
  - **Optimization for local interactions between neighboring peers:** The 'network distance' between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. The

middleware should aim to place resources close to the nodes that access them the most.

- **Accommodation 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.
- ***Security of data in an environment with heterogeneous trust*:** Security of data in an environment simplify construction of services across many hosts in wide network
- Anonymity, deniability and resistance to restrict.

6.6. Routing Overlays: Routing overlays share many characteristics with the IP packet routing infrastructure that constitutes the primary communication mechanism of the Internet. It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems. This is shown figure (table).

- A routing overlay is a distributed algorithm for a middleware layer responsible for routing requests from any client to a host that holds the object to which the request is addressed.
- 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.

- Global User IDs (GUID) also known as opaque identifiers are used as names, but do not contain location information.
- 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.

Figure Distinctions between IP and overlay routing for 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.

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 submit 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.

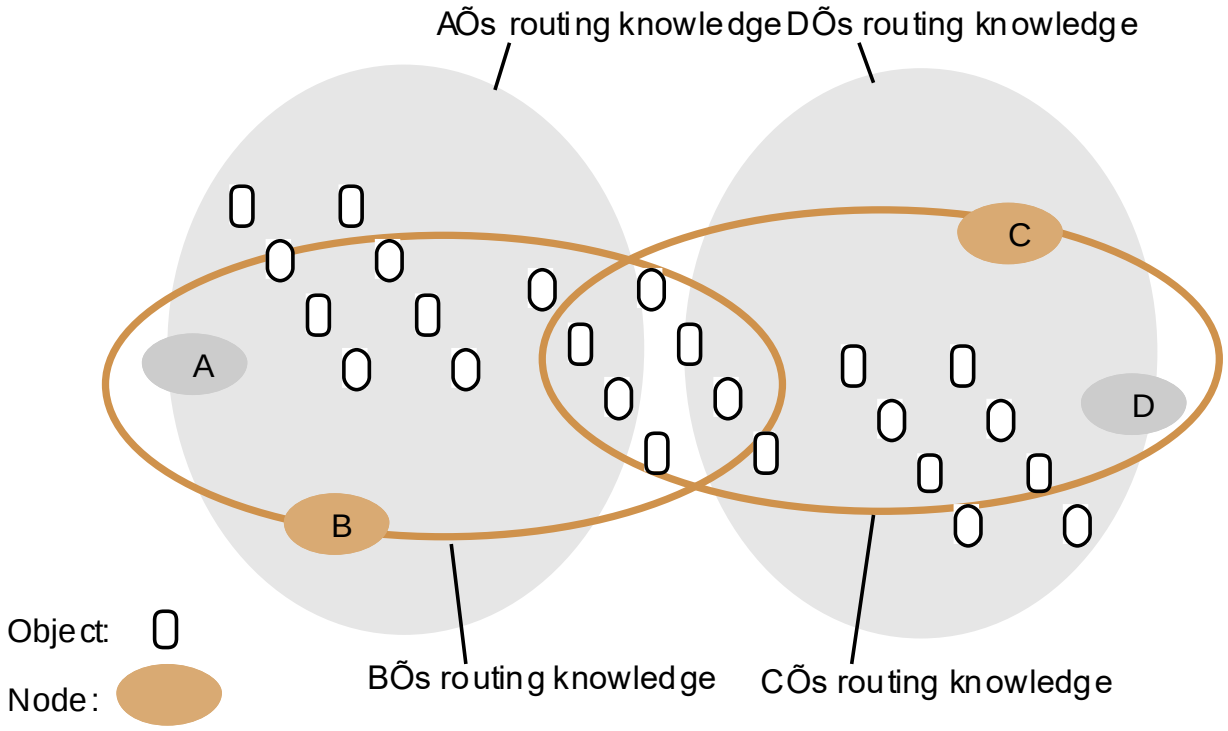


Figure: **Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry**

***p**ut(GUID, data)*

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

remove(GUID)

Deletes all references to *GUID* and the associated data.

value = get(GUID)

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

Figure: **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])

The 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.

The interfaces in Figures 1 & 2 are based on a set of abstract to show that most peer-to-peer routing overlay implementations developed to date provide very similar functionality.

Coordination and Agreement

7.1. Introduction: Collection of algorithms are used to provide a communication with the help of peer-to-peer process in distributed systems. In view of that,

- A set of processes to coordinate their actions or to agree on one or more values.

- For example, a complex piece of machinery such as a spaceship is essential that the computers need control the operation for proceeding or has been aborted. In this case, the computers must coordinate their actions correctly with respect to shared resources (the spaceship's sensors and actuators). The computers must be able to proceeding their process if there is no fixed master-slave relationship between the components.

- The reason for avoiding fixed master-slave relationships is that we often require our systems to keep working correctly even if failures occur, so we need to avoid single points of failure, such as fixed masters.

- The distributed system can using either asynchronous or synchronous for communication. In an asynchronous system we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, on the time taken to execute each step of a process, and on clock drift rates. The synchronous assumptions allow us to use timeouts to detect process crashes.
- A failure model is another important in distributed system. This begin by considering some algorithms that tolerate no failures and progress through benign failures before exploring how to tolerate arbitrary failures. Along the way, we encounter a fundamental result in the theory of distributed systems: even under surprisingly benign failure conditions, it is impossible to guarantee

in an asynchronous system that a collection of processes can agree on a shared value – for example, for all of a spaceship's controlling processes to agree 'mission proceed' or 'mission abort'.

Coordination and agreement related to group communication is the ability to multicast a message to a group is a very useful communication paradigm, with applications from locating resources to coordinating the updates to replicated data. It examines multicast reliability and ordering semantics, and gives algorithms to achieve the variations.

7.1.1 Failure assumptions and failure detectors:

Failure assumptions: The fundamental network components may suffer failures by a reliable communication protocol – for example, by retransmitting missing or corrupted messages. Also for the sake of simplicity, we assume that no process failure implies a threat to the other processes' ability to communicate. This means that none of the processes depends upon another to forward messages.

In any particular interval of time, communication between some processes may succeed while communication between others is delayed. For example, the failure of a router between two networks may mean that a collection of four processes is split into two pairs, such that intra-pair communication is possible over their respective networks; but inter-pair communication is not possible while the router has failed. This is known as a *network partition.*

Failure detectors: A *failure detector* is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes. The object local to each process is called a *local failure detector*. We outline how to implement failure detectors shortly, but first we concentrate on some of the properties of failure detectors.

A failure 'detector' is categorized into two types, they are

1. *Unreliable failure detectors* and 2. *Reliable failure detectors.*

An ***unreliable failure*** detector may produce one of two values to identity the process: *Unsuspected* or *Suspected*. Both of these results are May or may not accurately reflect whether the process has actually failed.

A result of *Unsuspected* signifies that the detector has recently received evidence suggesting that the process has not failed. For example, a message was recently received from it. But of course, the process may have failed since then.

A result of *Suspected* signifies that the failure detector has some indication that the process may have failed. For example, it may be that no message from the process has been received for more than a nominal maximum length of silence (even in an asynchronous system, practical upper bounds can be used as hints).

A ***reliable failure detector*** is one that is always accurate in detecting a process's failure. A result of *Failed* means that the detector has determined that the process has crashed.

Thus, a failure detector may sometimes give different responses to different processes, since communication conditions vary from process to process.

7.2 Distributed mutual exclusion:

A collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is the *critical section* problem in the domain of operating systems. In a distributed system, a solution to *distributed mutual exclusion*: one that is based solely on message passing.

In some cases shared resources are managed by servers that also provide mechanisms for mutual exclusion. But in some practical cases, a separate mechanism for mutual exclusion is required.

7.2.1. Algorithms for mutual exclusion

Consider a system of *N* processes *pi*, $i = 1, 2, \ldots . N$, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.

Assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

enter() // enter critical section – block if necessary
resourceAccesses() // access shared resources in critical section.

exit() // leave critical section – other processes may now enter

Our essential requirements for mutual exclusion are as follows:

ME1: (safety) At most one process may execute in the critical section (CS) at a time.

ME2: (liveness) Requests to enter and exit the critical section eventually succeed.

Condition ME2 implies freedom from both ***deadlock*** and ***starvation***. A *deadlock* would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section by their mutual interdependence. But even without a deadlock, a poor algorithm might lead to *starvation*.

The absence of starvation is a *fairness* condition. Another fairness issue is the order in which processes enter the critical section. It is not possible to order entry to the critical section by the times that the processes requested it, because of the absence of global clocks. But a useful fairness requirement that is sometimes made makes use of the happened-before ordering between messages that request entry to the critical section.

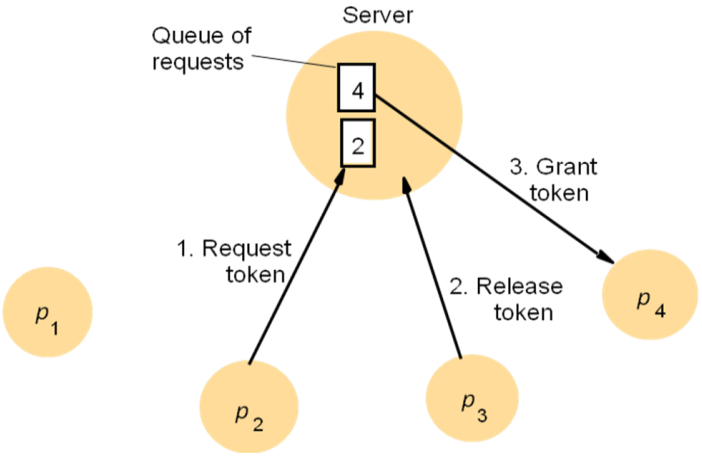
The performance of algorithms for mutual exclusion according to the following criteria:

- **Bandwidth** consumption, which is proportional to the number of messages, sent in each entry and exit operations.
- The **client delay** incurred by a process at each entry and exit operation.
- **Throughput** of the system. Rate at which the collection of processes as a whole can access the critical section. Measure the effect using the **synchronization delay** between one process exiting the critical section and the next process entering it; the shorter the delay is, the greater the throughput.

7.2.2. The central server algorithm: The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section.

- A process sends a request message to server and awaits a reply from it.
- If a reply constitutes a token signifying the permission to enter the critical section.
- If no other process has the token at the time of the request, then the server replied immediately with the token.
- If token is currently held by other processes, the server does not reply but queues the request.
- Client on exiting the critical section, a message is sent to server, giving it back the token.

Figure shows the Central Server algorithm that managing a mutual exclusion token for a set of processes.

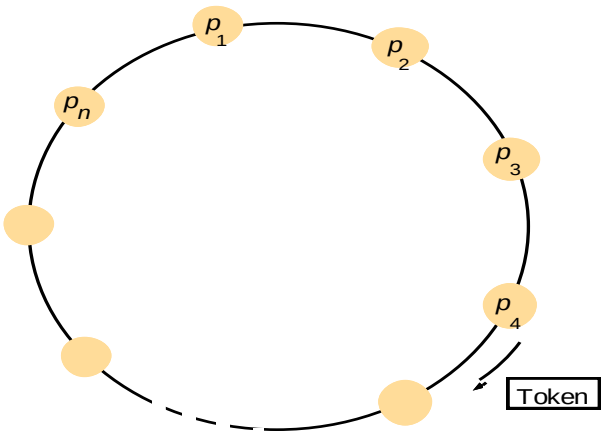


- ME1: safety
- ME2: liveness
- Are satisfied but not
- ME3: ordering
- **Bandwidth**: entering takes two messages(request followed by a grant), **delayed** by the round-trip time; exiting takes one release message, and does **not delay** the exiting process.
- **Throughput** is measured by synchronization delay, the round-trip of a release message and grant message.

7.2.3. Ring-based Algorithm:

- Simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.
- Each process pi has a communication channel to the next process in the ring, p(i+1)/mod N.
- The unique token is in the form of a message passed from process to process in a single direction clockwise.
- If a process does not require entering the CS when it receives the token, then it immediately forwards the token to its neighbor.
- A process requires the token waits until it receives it, but retains it.
- To exit the critical section, the process sends the token on to its neighbor.

A ring of processes transferring a mutual exclusion token is shown in fig.



ME1: safety

ME2: liveness

Are satisfied but not

ME3: ordering

Bandwidth: continuously consumes the bandwidth except when a process is inside the CS. Exit only requires one message

Delay: experienced by process is zero message(just received token) to N messages(just pass the token).

Throughput: synchronization delay between one exit and next entry is anywhere from 1 to N message transmission.

Ricart and Agrawala's algorithm:

On initialization

state := RELEASED;

To enter the section

state := WANTED;

Multicast *request* to all processes; request processing deferred here

T := request's timestamp;

Wait until (number of replies received = (*N* – 1));

state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$

if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))

then

queue *request* from p_i without replying;

else

reply immediately to p_i;

end if

To exit the critical section

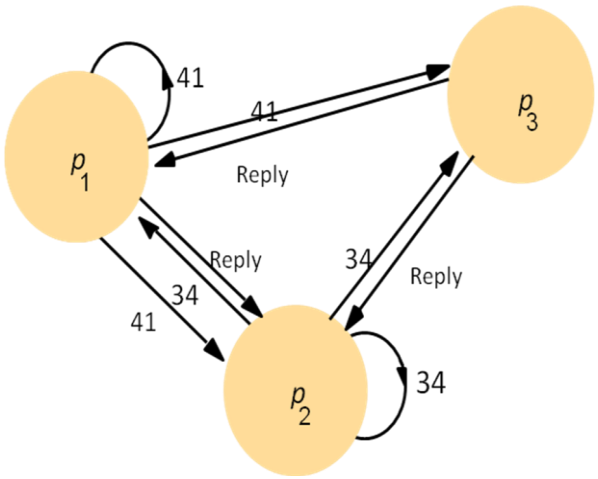
state := RELEASED;

reply to any queued requests;

7.2.4.An algorithm using multicast and logical clocks: Ricart and Agrawala developed an algorithm to implement

- Mutual exclusion between N peer processes based upon multicast.
- Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.
- The condition under which a process replies to a request are designed to ensure that conditions ME1, ME2 and ME3 are met.
- Each process pi keeps a Lamport clock. Message requesting entry are of the form<T, pi>.
- Each process records its state of either RELEASE, WANTED or HELD in a variable state.
  - If a process requests entry and all other processes is RELEASED, then all processes reply immediately.
  - If some process is in state HELD, then that process will not reply until it is finished.
  - If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.
  - If two or more processes request entry at the same time, then whichever bears the lowest timestamp will be the first to collect N-1 replies.

To illustrate the algorithm, consider a situation involving three processes, *p*1 , *p*2 and *p*3 , shown in Figure.



→P1 and P2 request CS concurrently. The timestamp of P1 is 41 and for P2 is 34. When P3 receives their requests, it replies immediately. When P2 receives P1's request, it finds its own request has the lower timestamp, and so does not reply, holding P1 request in queue. However, P1 will reply. P2 will enter CS. After P2 finishes, P2 reply P1 and P1 will enter CS.

→Granting entry takes 2(N-1) messages, N-1 to multicast request and N-1 replies.

→**Bandwidth** consumption is high.

→**Client delay** is again 1 round trip time

→**Synchronization delay** is one message transmission time.

7.2.6. Maekawa's voting algorithm: In 1985, Maekawa observed that in order for a process to enter a critical section,

- It is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate.

Maekawa associated a *voting* set Vi associated with each process pi. i.e., *pi* ($i = 1, 2, \ldots N$),

$$V_i \subseteq \{p_1, p_2, \ldots, p_N\}$$
$$such\ that\ for\ all\ i, j = 1,2,\ldots N:$$
$$p_i \in V_i$$
$$V_i \cap V_j \neq \varnothing$$
$$|V_i| = K$$

- There is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.
- The optimal solution to minimizes K is K~sqrt(N) and M=K.

Maekawa's algorithm is shown below to obtain entry to the critical section, a process *pi* sends *request* messages to all *K* members of *Vi* (including itself). *pi* cannot enter the critical section until it has received all *K reply* messages. When a process *pj* in *Vi* receives *pi* 's *request* message, it sends a *reply* message immediately.

On initialization
    state := RELEASED;
    voted := FALSE;
For p_i *to enter the critical section*
    state := WANTED;
    Multicast *request* to all processes in V_i;
    Wait until (number of replies received = *K*);
    state := HELD;
On receipt of a request from p_i *at* p_j
    if (*state* = HELD *or voted* = TRUE)

then
queue *request* from p_i without replying;
else
send *reply* to p_i;
voted := TRUE;
end if

For p_i to exit the critical section
state := RELEASED;
Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_j
if (queue of requests is non-empty)
then
remove head of queue – from p_k, say;
send *reply* to p_k;
voted := TRUE;
else
voted := FALSE;
end if

- ME1 is met. If two processes can enter CS at the same time, the processes in the intersection of two voting sets would have to vote for both. The algorithm will only allow a process to make at most one vote between successive receipts off a release message.
- Deadlock prone. For example, p1, p2 and p3 with V1={p1,p2}, V2={p2, p3}, V3={p3,p1}. If three processes concurrently request entry to the CS, then it is possible for p1 to reply to itself and hold off p2; for p2 rely to itself and hold off p3; for p3 to reply to itself and hold off p1. Each process has received one out of two replies, and none can proceed.
- **Bandwidth** utilization is 2sqrt(N) messages per entry to CS and sqrt(N) per exit.
- **Client delay** is the same as Ricart and Agrawala's algorithm, one round-trip time.
- **Synchronization delay** is one round-trip time.

Fault tolerance: The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

→ What happens when messages are lost?

→ What happens when a process crashes?

→ None of the algorithm that we have described would tolerate the loss of messages if the channels were unreliable.

- The ring-based algorithm cannot tolerate any single process crash failure.
- Maekawa's algirithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required.
- The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token.
- The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process by taking it to grant all requests implicitly.

7.3. Elections: Algorithm to choose a unique process to play a particular role is called an election algorithm. E.g. central server for mutual exclusion, one process will be elected as the server. Everybody must agree. If the server wishes to retire, then another election is required to choose a replacement.

→ Requirements:

- E1(safety): a participant pi has *electedi* = ⊥ or *electedi* = *P*,
  Where P is chosen as the non-crashed process at the end of run with the largest

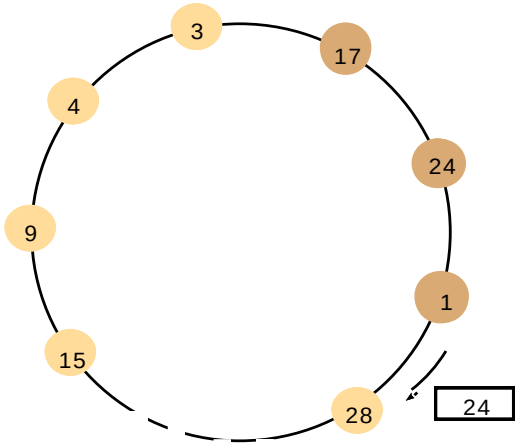
identifier.

- E2(liveness): All processes Pi participate in election process and eventually set *electedi* ‡ ⊥ – or crash.

7.3.1. A ring based election algorithm:

- All processes arranged in a logical ring.
- Each process has a communication channel to the next process.
- All messages are sent clockwise around the ring.
- Assume that no failures occur, and system is asynchronous.
- Goal is to elect a single process coordinator which has the largest identifier.

A ring-based election in progress:



1. **Initially**, every process is marked as non-participant. Any process can begin an election.
2. The **starting** process marks itself as participant and place its identifier in a message to its neighbour.
3. A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
4. If arrived identifier is **smaller** and receiver is not a participant, substitute its own identifier in the message and forward if. It does not forward the message if it is already a participant.
5. On forwarding of any case, the process marks itself as a participant.
6. If the received identifier is that of the receiver itself, then this process' s identifier must be the greatest, and it becomes the **coordinator**.
7. The coordinator marks itself as non-participant set elected_i and sends an **elected** message to its neighbour enclosing its ID.
8. When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.
9. E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.
10. E2 is also met due to the guaranteed traversals of the ring.
11. Tolerate no failure makes ring algorithm of limited practical use.

Note:

- The election was started by process 17.
- The highest process identifier encountered so far is 24.
- Participant processes are shown darkened

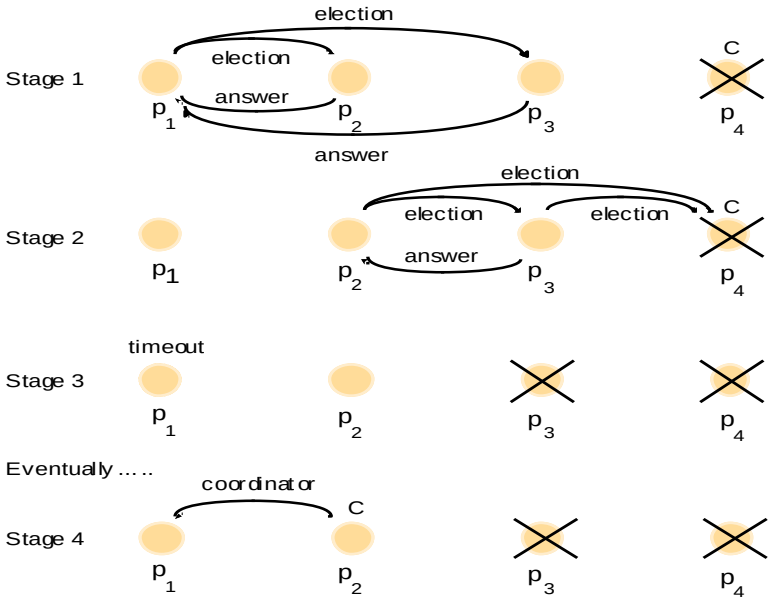
→ If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election.

→ The elected message is sent N times. Making **3N-1 messages in all**.

- **Turnaround time** is also 3N-1 sequential message transmission time

7.3.2. The bully algorithm: The bully algorithm

- Allows processes to crash during an election, although it assumes that message delivery between processes is reliable.
- Assume system is synchronous to use timeouts to detect a process failure.
- Assume each process knows which processes have higher identifiers and that it can communicate with all such processes.
- Three types of messages:
    - **Election** is sent to announce an election message. A process begins an election when it notices, through **timeouts**, that the coordinator has failed. T=2Ttrans+Tprocess From the time of sending
    - **Answer** is sent in response to an election message.
    - **Coordinator** is sent to announce the identity of the elected process.



The election of coordinator p_2, after the failure of p_4 and then p_3

- ➢ The process begins a election by sending an election message to these processes that have a higher ID and awaits an answer in response. If none arrives within time T, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers. Otherwise, it waits a further time T' for coordinator message to arrive. If none, begins another election.
- ➢ If a process receives a coordinator message, it sets its variable elected_i to be the coordinator ID.
- ➢ If a process receives an election message, it send back an answer message and begins another election unless it has begun one already.

➔ E1 may be broken if timeout is not accurate or replacement. (suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator)

➔ E2 is clearly met by the assumption of reliable transmission.

➔ **Best case** the process with the second highest ID notices the coordinator's failure. Then it can immediately elect itself and send N-2 coordinator messages.

➔ The bully algorithm requires O(N^2) messages in the **worst case** - that is, when the process with the least ID first detects the coordinator's failure. For then N-1 processes altogether begin election, each sending messages to processes with higher ID.

7.4 Multicast communication: A *multicast operation* is an operation that sends a single message from one process to each of the members of a group of processes. In such a way, the membership of the group is transparent to the sender. There is a range of possibilities in the desired behavior of a multicast. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. ***Fault tolerance based on replicated services*:** A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.
2. ***Discovering services in spontaneous networking***: It defines service discovery in the context of spontaneous networking. Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
3. ***Better performance through replicated data*:** Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value are multicast to the processes managing the replicas.
4. ***Propagation of event notifications*:** Multicast to a group may be used to notify processes when something happens. For example, in Face book, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

7.4.1. IP multicast – An implementation of multicast communication

For group communication, different group communication protocols were used. In addition to these IP multicast is provided which presents Java's API to it through the ***Multicast Socket*** class.

IP multicast: *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers i.e., ports belonging to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group.

A *multicast group* is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4.

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send data grams to a multicast group without being a member.

An application program performs multicasts by sending UDP data grams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group.

At the IP level, a computer belongs to a multicast group when one or more of its processes have sockets that belong to that group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

- ***Multicast routers*:** IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single data grams to routers on other networks, where they are again multicast to local members.
- ***Multicast address allocation***: Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the ***Internet Assigned Numbers Authority*** (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171. This document defines a partitioning of this address space into a number of blocks, including:

  → Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.

  → Internet Control Block (224.0.1.0 to 224.0.1.225).

  → Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.

  → Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

Multicast addresses may be permanent or temporary. Permanent groups exist even when there are no members i.e., their addresses are assigned by IANA and span the various blocks mentioned above.

For example, 224.0.1.1 in the Internet block is reserved for the Network Time Protocol (NTP), as discussed in Chapter 14, and the range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project.

7.4.2. Failure model for multicast data grams • Data grams multicast over IP multicast have the same failure characteristics as UDP data grams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. This can be called ***unreliable*** multicast, because it does not guarantee that a message will be delivered to any member of a group.

7.4.3.Java API to IP multicast : The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port or any free local port.

A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
public static void main(String args[]){
// args give message contents & destination multicast group (e.g. "228.5.6.7")
MulticastSocket s =null;
try {
InetAddress group = InetAddress.getByName(args[1]);
s = new MulticastSocket(6789);
s.joinGroup(group);
byte [] m = args[0].getBytes();
DatagramPacket messageOut = □
new DatagramPacket(m, m.length, group, 6789);
s.send(messageOut);
byte[] buffer = new byte[1000];
for(int i=0; i< 3; i++) { // get messages from others in group
DatagramPacket messageIn = □
new DatagramPacket(buffer, buffer.length);
s.receive(messageIn);
System.out.println("Received:" +new String(messageIn.getData()));
}
s.leaveGroup(group);
} catch (SocketException e){System.out.println("Socket: " + e.getMessage());
} catch (IOException e){System.out.println("IO: " + e.getMessage());
} finally { if(s != null) s.close();}
}
}
```

- → In the example, the arguments to the *main* method specify a message to be multicast and the multicast address of a group (for example, *"228.5.6.7").*
- → After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789.
- → After that, it attempts to receive three multicast messages from its peers via its socket, which also belongs to the group on the same port.
- → When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.
- → The Java API allows the TTL to be set for a multicast socket by means of the *setTimeToLive* method. The default is 1, allowing the multicast to propagate only on the local network.

Reliability and ordering: Now consider the effect of the failure semantics of IP multicast on the four examples of the use of replication.

1. *Fault tolerance based on replicated services*: Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request; it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.

2. *Discovering services in spontaneous networking*: One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond.

An occasional lost request is not an issue when discovering services. In fact, Jini uses IP multicast in its protocol for discovering services.

3. *Better performance through replicated data*: Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.

4. *Propagation of event notifications*: The particular application determines the qualities required of multicast. For example, the Jini lookup services use IP multicast to announce their existence.

→ These examples suggest that some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them.

→ The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.