UNIT-V

UNIT-V:

Distributed File Systems: Introduction, File Service Architecture; Peer-to-Peer Systems: Introduction, Napster and its Legacy, Peer-to-Peer Middleware, Routing Overlays. **Coordination and Agreement:** Introduction, Distributed Mutual Exclusion, Elections, Multicast Communication.

Introduction: What Is a Distributed File Systems (DFS)

A file is a collection of data organized by the user. The data within a file isn't necessarily meaningful to the operating system. Instead, a file is created by the user and meaningful to the user. It is the job of the operating system to maintain this unit, without understanding or caring why.

A *file system* is the component of an operating system that is responsible for managing files. File systems typically implement persistent storage, although volatile file systems are also possible (/proc is such an example). So what do we mean when we say *manage files*? Well, here are some examples:

- Name files in meaningful ways. The file system should allow a user to locate a file using a human-friendly name. In many cases, this is done using a hierarchical naming scheme like those we know and love in Windows, UNIX, Linux, &c.
- Access files. Create, destroy, read, write, append, truncate, keep track of position within, &c
- Physical allocation. Decide where to store things. Reduce fragmentation. Keep related things close together, &c.
- Security and protection. Ensure privacy, prevent accidental (or malicious) damage, &c.
- Resource administration. Enforce quotas, implement priorities, &c.

So what is it that a DFS does? Well, basically the same thing. The big difference isn't *what it does* but the *environment in which it lives*. A traditional file system typically has all of the users and all of the storage resident on the same machine. A distributed file system typically operates in an enviornment where the data may be spread out across many, many hosts on a network -- and the users of the system may be equally distributed.

For the purposes of our conversation, we'll assume that each node of our distributed system has a rudimentary local file system. Our goal will be to coordinate these file systems and hide their existence from the user. The user should, as often as possible, believe that they are using a local file system. This means that the naming scheme needs to hide the machine names, &c. This also means that the system should mitigate the latency imposed by network communication. And the system should reduce the increase risk of accidental and malicious damage to user data assocaited with vulnerabilities in network communication. We could actually continue this list. To do so, pick a property of a local file system and append *transparency*.

Why Would we want a DFS?

There are many reasons that we might want a DFS. Some of the more common ones are listed below:

- More storage than can fit on a single system
- More fault tolerance than can be achieved if "all of the eggs are in one basket."
- The user is "distributed" and needs to access the file system from many places

The basic model provided by **distributed file systems** is that of clients accessing **files** and directories that are provided by one or more **file** servers. ... **Note** that the view provided to different clients by the same server may be different, for example, if clients only see **files** that they are authorised to access.

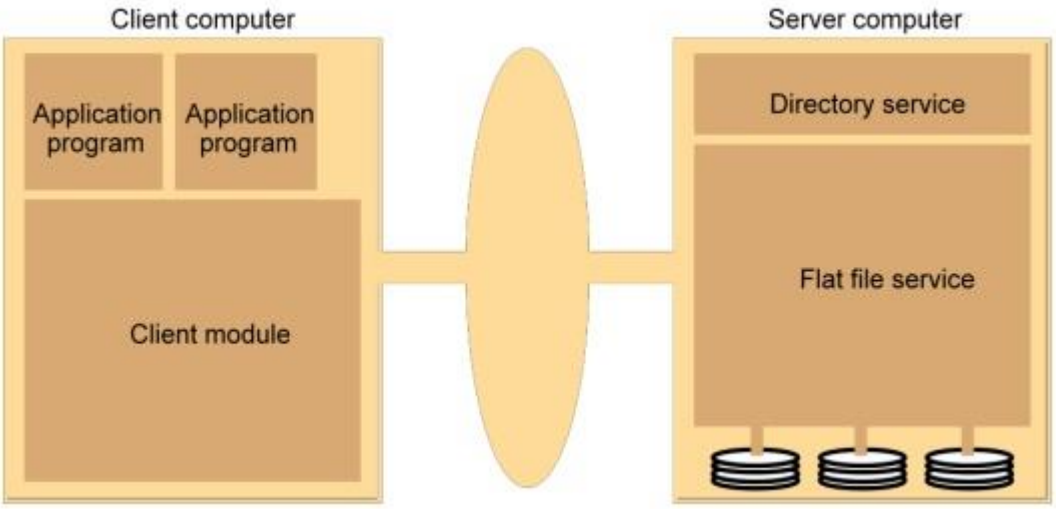
File Service Architecture

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

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

- The relevant modules and their relationship is shown in Figure 5.

- Figure 5. File service architecture



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

Flat file service interface:

- Figure 6 contains a definition of the interface to a flat file service.

Figure 6. Flat file service operations

Read(FileId, i, n) -> Data -throws BadPosition	if *1≤i≤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≤i≤Length(File)+1*: Write a sequence of *Data* to a file, starting at 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 3.)

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

Figure 7. Directory service operations

Lookup(Dir, Name) -> FileId -throws NotFound	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, 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*.

¬ 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

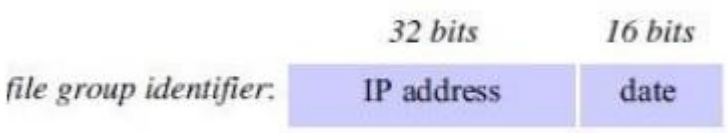
A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.

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 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.



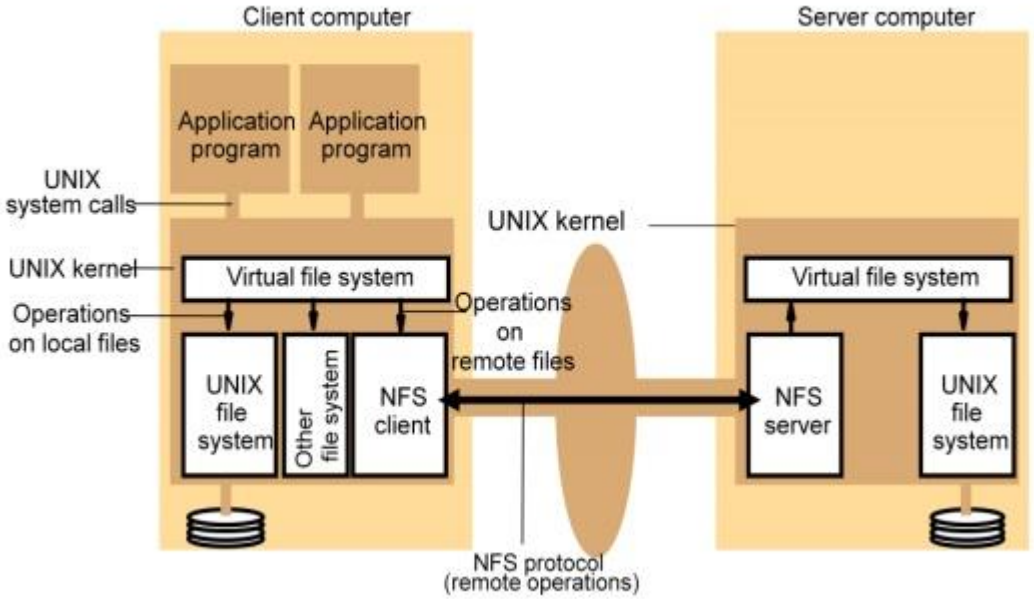
DFS: Case Studies

- NFS (Network File System)
  - Developed by Sun Microsystems (in 1985)
  - Most popular, open, and widely used.
  - NFS protocol standardized through IETF (RFC 1813)
- **AFS (Andrew File System)**
  - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)

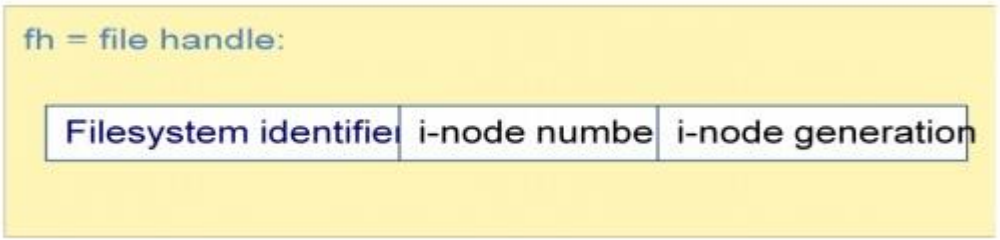
- A research project to create campus wide file system.
- Public domain implementation is available on Linux (LinuxAFS)
- It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment

NFS architecture

Figure 8 shows the architecture of Sun NFS



The file identifiers used in NFS are called file handles.



A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure 9.

Figure 9. NFS server operations (NFS Version 3 protocol, simplified)

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

- **NFS access control and authentication**

  - The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request.

- § In the local file system they are checked only on the file's access permission attribute.

- Every client request is accompanied by the userID and groupID

- § It is not shown in the Figure 8.9 because they are inserted by the RPC system.

- Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

- **Mount service**

- Mount operation:

- mount(remotehost, remotedirectory, localdirectory)

- Server maintains a table of clients who have mounted filesystems at that server.

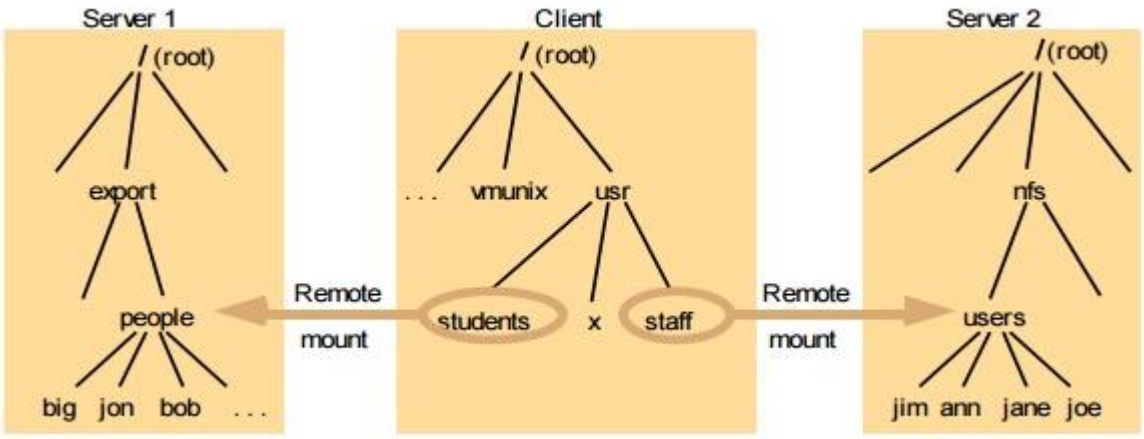
- Each client maintains a table of mounted file systems holding:

- o IP address, port number, file handle>

- Remote file systems may be hard-mounted or soft-mounted in a client computer.

- Figure 10 illustrates a Client with two remotely mounted file stores.

Figure 10. Local and remote file systems accessible on an NFS client



P2P(Peer To Peer) File Sharing

Introduction

In Computer Networking, P2P is a file sharing technology, allowing the users to access mainly the multimedia files like videos, music, e-books, games etc. The individual users in this network are referred to as **peers**. The peers request for the files from other peers by establishing TCP or UDP connections.

How P2P works(Overview)

A peer-to-peer network allows computer hardware and software to communicate without the need for a server. Unlike client-server architecture, there is no central server for processing requests in a P2P architecture. The peers directly interact with one another without the requirement of a central server.

Now, when one peer makes a request, it is possible that multiple peers have the copy of that requested object. Now the problem is how to get the IP addresses of all those peers. This is decided by the underlying architecture supported by the P2P systems. By means of one of these methods, the client peer can get to know about all the peers which have the requested object/file and the file transfer takes place directly between these two peers.

Three such Architectures exist:

1. Centralized Directory
2. Query Flooding
3. Exploiting Heterogeneity

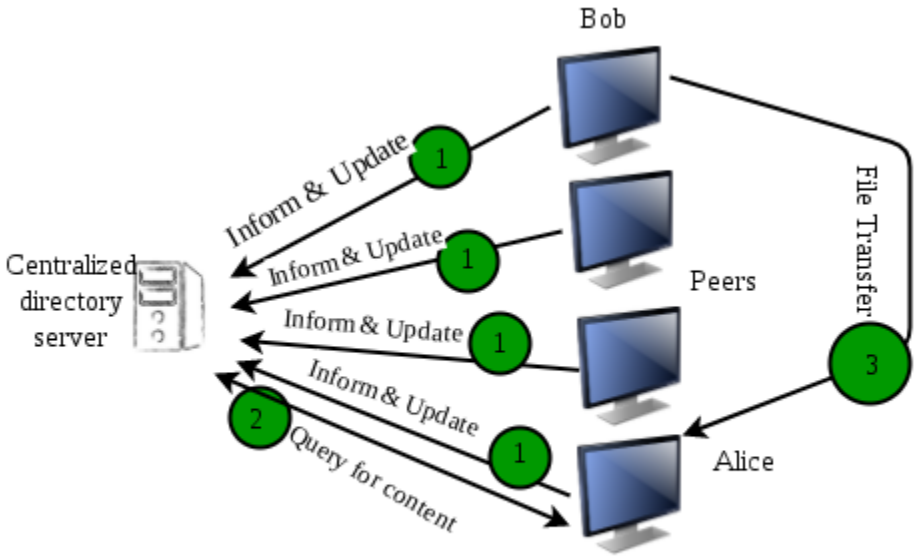
1. Centralized Directory

- It is somewhat similar to client server architecture in the sense that it maintains a huge central server to provide directory service.
- All the peers inform this central server of their IP address and the files they are making available for sharing.
- The server queries the peers at regular intervals to make sure if the peers are still connected or not.
- So basically this server maintains a huge database regarding which file is present at which IP addresses.

Working

- Now whenever a requesting peer comes in, it sends its query to the server.
- Since the server has all the information of its peers, so it returns the IP addresses of all the peers having the requested file to the peer.
- Now the file transfer takes place between these two peers.

The first system which made use of this method was **Napster**, for the purpose of Mp3 distribution.



P2P paradigm with a centralised directory

The major problem with such an architecture is that there is a single point of failure. If the server crashes, the whole P2P network crashes. Also, since all of the processing is to be done by a single server so a huge amount of database has to be maintained and regularly updated.

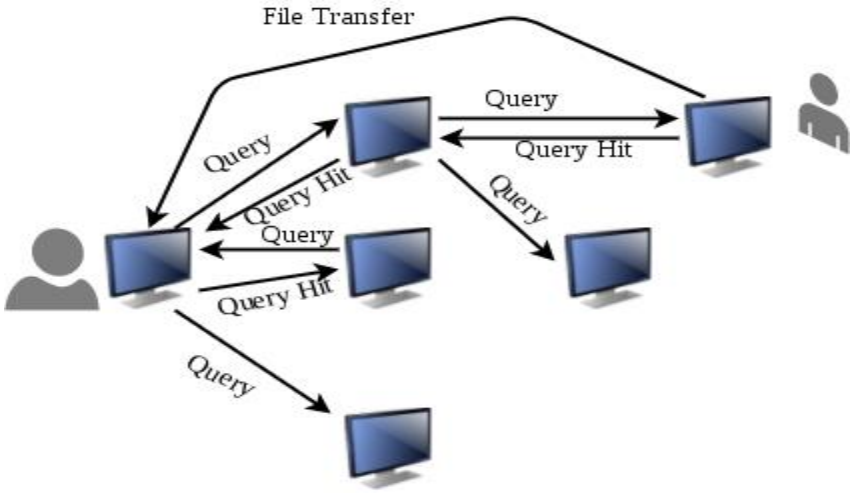
2. Query Flooding

- Unlike the centralized approach, this method makes use of distributed systems.
- In this, the peers are supposed to be connected into an overlay network. It means if a connection/path exists from one peer to other, it is a part of this overlay network.
- In this overlay network, peers are called as nodes and the connection between peers is called an edge between the nodes, thus resulting in a graph-like structure.

Working

- Now when one peer requests for some file, this request is sent to all its neighboring nodes i.e. to all nodes which are connected to this node. If those nodes don't have the required file, they pass on the query to their neighbors and so on. This is called as query flooding.
- When the peer with requested file is found (referred to as query hit), the query flooding stops and it sends back the file name and file size to the client, thus following the reverse path.
- If there are multiple query hits, the client selects from one of these peers.

Gnutella was the first decentralized peer to peer network.



This method also has some disadvantages like, the query has to be sent to all the neighboring peers unless a match is found. This increases traffic in the network.

3. Exploiting heterogeneity

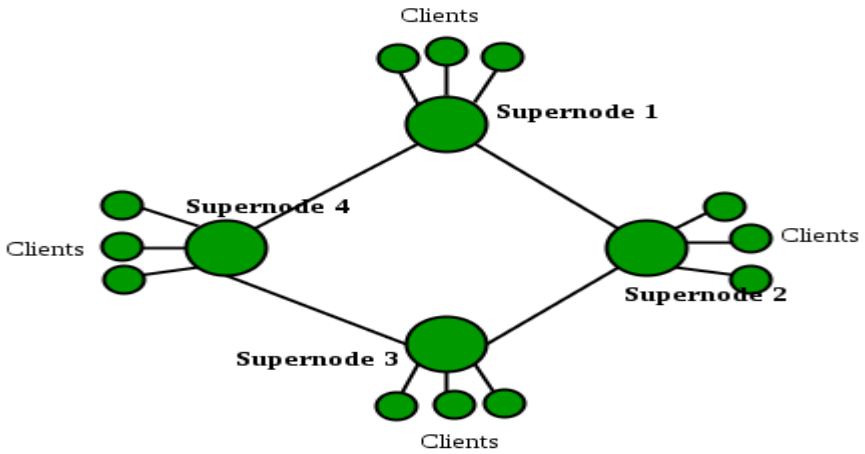
- This P2P architecture makes use of both the above discussed systems.
- It resembles a distributed system like Gnutella because there is no central server for query processing.
- But unlike Gnutella, it does not treat all its peers equally. The peers with higher bandwidth and network connectivity are at a higher priority and are called as **group leaders/super nodes**. The rest of the peers are assigned to these super nodes.
- These super nodes are interconnected and the peers under these super nodes inform their respective leaders about their connectivity, IP address and the files available for sharing.

KaZaA technology is such an example which makes use of Napster and Gnutella both.

Thus, the individual group leaders along with their child peers form a Napster-like structure. These group leaders then interconnect among themselves to resemble a Gnutella-like structure.

Working

- This structure can process the queries in two ways.
- The first one is that the super nodes could contact other super nodes and merge their databases with its own database. Thus, this super node now has information of a large number of peers.
- Another approach is that when a query comes in, it is forwarded to the neighboring super nodes until a match is found, just like in Gnutella. Thus query flooding exists but with limited scope as each super node has many child peers. Hence, such a system exploits the heterogeneity of the peers by designating some of them as group leaders/super nodes and others as their child peers.



Coordination and Agreement

Overview

We start by addressing the question of why process need to coordinate their actions and agree on values in various scenarios.

1. Consider a mission critical application that requires several computers to communicate and decide whether to proceed with or abort a mission. Clearly, all must come to agreement about the fate of the mission.
2. Consider the Berkeley algorithm for time synchronization. One of the participate computers serves as the coordinator. Suppose that coordinator fails. The remaining computers must elect a new coordinator.
3. Broadcast networks like Ethernet and wireless must agree on which nodes can send at any given time. If they do not agree, the result is a collision and no message is transmitted successfully.
4. Like other broadcast networks, sensor networks face the challenging of agreeing which nodes will send at any given time. In addition, many sensor network algorithms require that nodes elect coordinators that take on a server-like responsibility. Choosing these nodes is particularly challenging in sensor networks because of the battery constraints of the nodes.
5. Many applications, such as banking, require that nodes coordinate their access of a shared resource. For example, a bank balance should only be accessed and updated by one computer at a time.

Failure Assumptions and Detection

Coordination in a synchronous system with no failures is comparatively easy. We'll look at some algorithms targeted toward this environment. However, if a system is asynchronous, meaning that messages may be delayed an indefinite amount of time, or failures may occur, then coordination and agreement become much more challenging.

A *correct process* "is one that exhibits no failures at any point in the execution under consideration." If a process fails, it can fail in one of two ways: a crash failure or a byzantine failure. A crash failure implies that a node stops working and does not

respond to any messages. A byzantine failure implies that a node exhibits arbitrary behavior. For example, it may continue to function but send incorrect values.

Failure Detection

One possible algorithm for detecting failures is as follows:

- Every *t* seconds, each process sends an "I am alive" message to all other processes.
- Process *p* knows that process *q* is either *unsuspected*, *suspected*, or *failed*.
- If *p* sees *q*'s message, it sets *q*'s status to unsuspected.

This seems ok if there are no failures. What happens if a failure occurs? In this case, *q* will not send a message. In a synchronous system, *p* waits for *d* seconds (where *d* is the maximum delay in message delivery) and if it does not hear from *q* then it knows that *q* has failed. In an asynchronous system, *q* can be suspected of failure after a timeout, but there is no guarantee that a failure has occurred.

Mutual Exclusion

The first set of coordination algorithms we'll consider deal with mutual exclusion. How can we ensure that two (or more) processes do not access a shared resource simultaneously? This problem comes up in the OS domain and is addressed by negotiating with shared objects (locks). In a distributed system, nodes must negotiate via message passing.

Each of the following algorithms attempt to ensure the following:

- Safety: At most one process may execute in the critical section (CS) at a time.
- Liveness: Requests to enter and exit the critical section eventually succeed.
- Causal ordering: If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

Central Server

The first algorithm uses a central server to manage access to the shared resource. To enter a critical section, a process sends a request to the server. The server behaves as follows:

- If no one is in a critical section, the server returns a token. When the process exits the critical section, the token is returned to the server.
- If someone already has the token, the request is queued.

Requests are serviced in FIFO order.

If no failures occur, this algorithm ensures safety and liveness. However, ordering is not preserved (**why?**). The central server is also a bottleneck and a single point of failure.

Token Ring

The token ring algorithm arranges processes in a logical ring. A token is passed clockwise around the ring. When a process receives the token it can enter its critical section. If it does not need to enter a critical section, it immediately passes the token to the next process.

This algorithm also achieves safety and liveness, but not ordering, in the case when no failures occur. However, a significant amount of bandwidth is used because the token is passed continuously even when no process needs to enter a CS.

Multicast and Logical Clocks

Each process has a unique identifier and maintains a logical clock. A process can be in one of three states: released, waiting, or held. When a process wants to enter a CS it does the following:

- sets its state to waiting
- sends a message to all other processes containing its ID and timestamp
- once all other processes respond, it can enter the CS

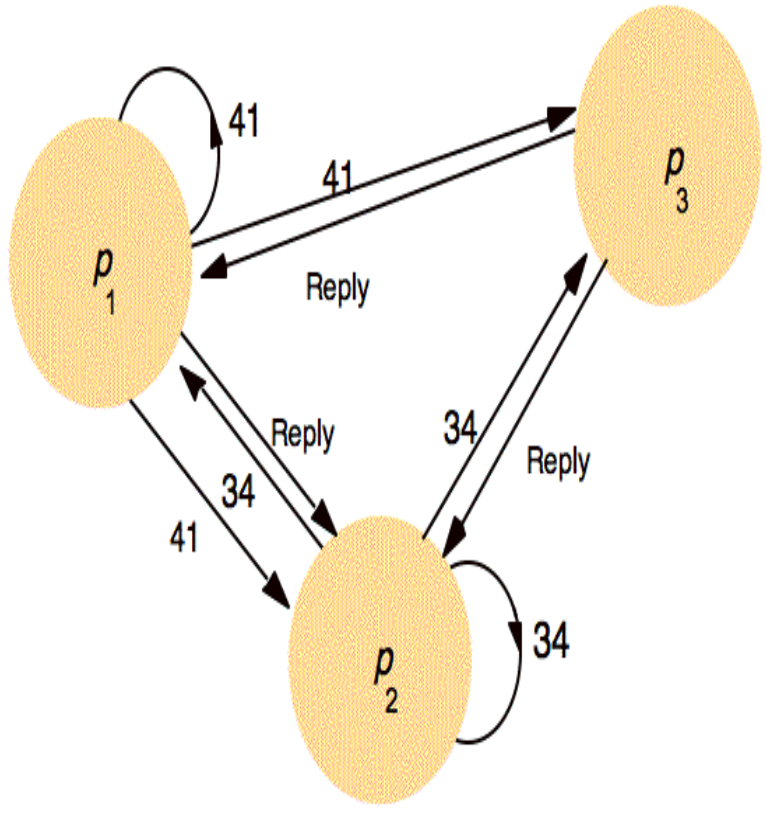
When a message is received from another process, it does the following:

- if the receiver process state is held, the message is queued
- if the receiver process state is waiting and the timestamp of the message is after the local timestamp, the message is queued (if the timestamps are the same, the process ID is used to order messages)
- else - reply immediately

When a process exits a CS, it does the following:

- sets its state to released
- replies to queued requests

Figure 12.5
Multicast synchronization



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

This algorithm provides safety, liveness, and ordering. However, it cannot deal with failure and has problems of scale.

None of the algorithms discussed are appropriate for a system in which failures may occur. In order to handle this situation, we would need to first detect that a failure has occurred and then reorganize the processes (e.g., form a new token ring) and reinitialize appropriate state (e.g., create a new token).

Election

An election algorithm determines which process will play the role of coordinator or server. All processes need to agree on the selected process. Any process can start an election, for example if it notices that the previous coordinator has failed. The requirements of an election algorithm are as follows:

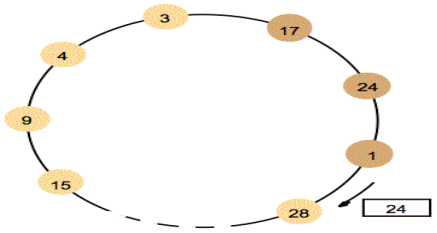
- Safety: Only one process is chosen -- the one with the largest identifying value. The value could be load, uptime, a random number, etc.
- Liveness: All process eventually choose a winner or crash.

Ring-based

Processes are arranged in a logical ring. A process starts an election by placing its ID and value in a message and sending the message to its neighbor. When a message is received, a process does the following:

- If the value is greater that its own, it saves the ID and forwards the value to its neighbor.
- Else if its own value is greater and the it has not yet participated in the election, it replaces the ID with its own, the value with its own, and forwards the message.
- Else if it has already participated it discards the message.
- If a process receives its own ID and value, it knows it has been elected. It then sends an elected message to its neighbor.
- When an elected message is received, it is forwarded to the next neighbor.

Figure 12.7
A ring-based election in progress



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

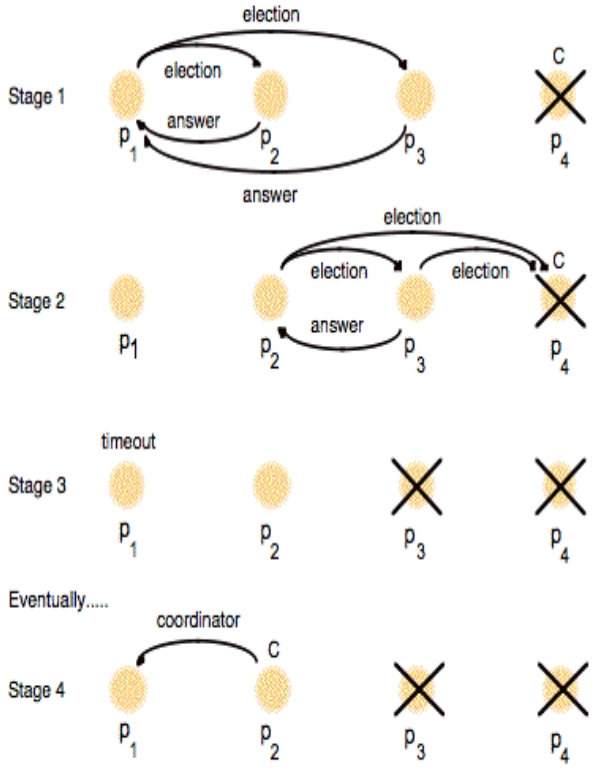
Safety is guaranteed - only one value can be largest and make it all the way through the ring. Liveness is guaranteed if there are no failures. However, the algorithm does not work if there are failures.

Bully

The bully algorithm can deal with crash failures, but not communication failures. When a process notices that the coordinator has failed, it sends an election message to all higher-numbered processes. If no one replies, it declares itself the coordinator and sends a new coordinator message to all processes. If someone replies, it does nothing else. When a process receives an election message from a lower-numbered process it returns a reply and starts an election. This algorithm guarantees safety and liveness and can deal with crash failures.

Figure 12.8
The bully algorithm

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



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

Consensus

All of the previous algorithms are examples of the consensus problem: how can we get all processes to agree on a state? Here, we look at when the consensus problem is solvable.

The system model considers a collection of processes p_i (i = 1, 2, ..., N). Communication is reliable, but processes may fail. Failures may be crash failures or byzantine failures.

The goals of consensus are as follows:

- Termination: Every correct process eventually decides on a value.
- Agreement: All processes agree on a value.
- Integrity: If all correct processes propose the same value, that value is the one selected.

We consider the Byzantine Generals problem. A set of generals must agree on whether to attack or retreat. Commanders can be treacherous (faulty). This is similar to consensus, but differs in that a single process proposes a value that the others must agree on. The requirements are:

- Termination: All correct processes eventually decide on a value.
- Agreement: All correct processes agree on a value.
- Integrity: If the commander is correct, all correct processes agree on what the commander proposed.

If communication is unreliable, consensus is impossible. Remember the blue army discussion from the second lecture period. With reliable communication, we can solve consensus in a synchronous system with crash failures.

We can solve Byzantine Generals in a synchronous system as long as less than 1/3 of the processes fail. The commander sends the command to all of the generals and each general sends the command to all other generals. If each correct process chooses the majority of all commands, the requirements are met. Note that the requirements do not specify that the processes must detect that the commander is fault.

It is impossible to guarantee consensus in an asynchronous system, even in the presence of 1 crash failure. That means that we can design systems that reach consensus most of the time, but cannot guarantee that they will reach consensus

every time. Techniques for reaching consensus in an asynchronous system include the following:

- Masking faults - Hide failures by using persistent storage to store state and restarting processes when they crash.
- Failure detectors - Treat an unresponsive process (that may still be alive) as failed.
- Randomization - Use randomized behavior to confuse byzantine processes.