

Process Migration

References:

- Pradeep K. Sinha, “Distributed Operation Systems : Concepts and Design” , PHI.
- Milojičić, Dejan S., Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou, “Process migration”, *ACM Computing Surveys (CSUR)*, Vol. 32, No. 3, 2000, pp. 241-299. < [ppt downloaded\process migration.pdf](#) >

Process management

- Conventional(Centralied) OS:
 - deals with the mechanisms and policies for sharing the processor of the system among all processes
- Distributed operating system:
 - To make best possible use of the processing resources of the entire system by sharing them among all processes

Process management contd...

- Three concepts to achieve this goal:
 - Processor allocation
 - Deals with the process of deciding which process should be assigned to which processor
 - Process migration
 - Deals with the movement of a process from its current location to the processor to which it has been assigned
 - Threads
 - Deals with parallelism for better utilization of the processing capability of the system

Process Migration

- Process Migration:
 - Relocation of a process from its current location (*the source node*) to another node (*the destination node*).
 - A process may be migrated
 - either before it starts executing on its source node or
 - during the course of its execution.

Process Migration Contd...

- Process migration enables:
 - **dynamic load distribution**, by migrating processes from overloaded nodes to less loaded ones,
 - **fault resilience**, by migrating processes from nodes that may have experienced a partial failure,
 - **improved system administration**, by migrating processes from the nodes that are about to be shut down or otherwise made unavailable, and
 - **data access locality**, by migrating processes closer to the source of some data.

Goals

- The goals of process migration are closely tied with the type of applications that use migration.
 - **Accessing more processing power** is a goal of migration when it is used for load distribution. Migration is particularly important in the *receiver-initiated distributed scheduling algorithms*, where a lightly loaded node announces its availability and initiates process migration from an overloaded node.
 - **Resource sharing** is enabled by migration to a specific node with a special hardware device, large amounts of free memory, or some other unique resource.
 - **Fault resilience** is improved by migration from a partially failed node.

Goals Contd...

- **System administration** is simplified if long-running computations can be temporarily transferred to other machines. **For example**, an application could migrate from a node that will be shutdown, and then migrate back after the node is brought back up.
- **Mobile computing** also increases the demand for migration. Users may want to migrate running applications from a host to their mobile computer as they connect to a network.

System Requirements for Migration

- To support migration effectively, a system should provide the following types of functionality:
 - **Exporting/importing the process state-**
 - The system must provide some type of export/import interfaces that allow the process migration mechanism *to extract a process's state from the source node and import this state on the destination node.*
 - These interfaces may be provided by the underlying operating system, the programming language, or other elements of the programming environment.
 - **Naming/accessing the process and its resources-**
 - After migration, the migrated process should be accessible by the same name and mechanisms.

System Requirements for Migration

Contd..

- **Cleaning up the process's non-migratable state-**

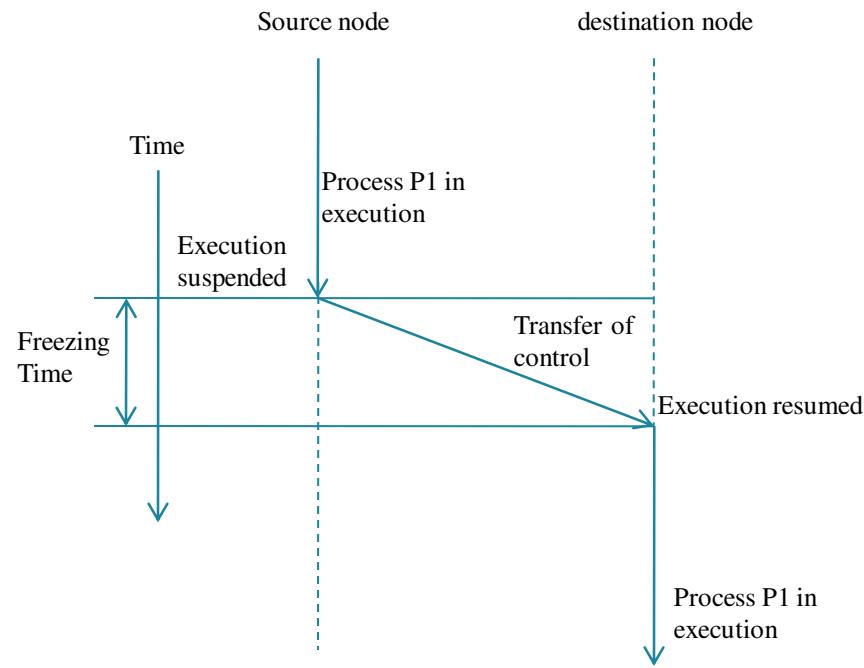
- The migrated process has associated system state that is not migratable.
- Migration must wait until the process finishes.
- If the operation can be arbitrarily long, it is typically aborted and restarted on the destination node. OR migration can wait for the completion of local file operations for limited time frame.

Process Migration contd...

- Two types:
 - Preemptive process migration
 - Process may be migrated during the course of its execution.
 - This type of process migration is **relatively expensive**, since it involves recording, migration and recreation of the process state as well as the reconstructing of any inter-process communication channels to which the migrating process is connected.
 - Non preemptive process migration
 - Process may be migrated before it starts executing on its source node.
 - This type of process migration is **relatively cheap**, since relatively little administrative overhead is involved.

Process Migration contd...

- The flow of execution of a migrating process:



Process Migration contd...

- Involves three steps:
 - Selection of a process that should be migrated
 - Selection of the destination node to which the selected process should be migrated
 - Actual transfer of the selected process to the destination node

Some desirable features

- **Minimal interference**
 - Should cause minimal interference to progress of process and system.
 - Can be done by minimizing freezing time.
 - **Freezing time:** a time period for which the execution of the process is stopped for *transferring its information* to the destination node.

Some desirable features contd...

- **Minimal residual dependencies**
 - Migrated process should not continue to depend on its previous node once it has started executing on new node.
 - Otherwise problems will occur:
 - Migrated process continue to impose a load on its previous node
 - A failure or reboot of the previous node will cause the process to fail

Some desirable features contd...

- **Efficiency**
 - *Time* required of migrating a process.
 - The *cost of locating an object*.
 - The *cost of supporting remote execution* once the process is migrated.
- **Robustness**
 - The failure of a node other than the one on which a process is currently running *should not affect the execution of that process*.

Process migration mechanisms

- Process migration involves proper handling of several **sub-activities** to meet the good process migration mechanism requirements.
- Four major **sub-activities**
 - Freezing the process on its source node and restarting it on its destination node
 - Transfer of process's address space from source to its destination node
 - Forwarding messages intended for the migrant process
 - Handling communication between cooperating processes that have been separated (placed on different nodes) as a result of process migration.

Process migration mechanisms

For preemptive process migration, the process is to take a “*snapshot*” of the process’s state on its source node and restore the snapshot on the destination node.

Steps-

- 1st step: freeing the process
 - Execution of the process is suspended and all external interactions are postponed
- 2nd step: state information is transferred to its destination node
- 3rd step: process is restarted using state information on destination node

Process migration mechanisms

- **Immediate and Delayed blocking of the process**
 - Before process can be frozen, its execution must be blocked
 - may be blocked immediately or delayed
 - if the process is not executing a system call --- can be immediately blocked
- **Interruptible and Uninterruptible States:**
 - if the process is executing a system call and at an interruptible priority waiting for a kernel event to occur, it can be immediately blocked from further execution
 - and at non-interruptible priority waiting for a kernel event to occur, it cannot be blocked, has to be delayed until the system call is complete.

Address Space Transfer Mechanisms

- **Address Space Transfer Mechanisms**
 - Information to be transferred from source node to destination node:
 - **Process's state information**
 - Consists contents of register, scheduling information, memory tables, I/O buffers, interrupt signals, I/O states, process identifier, information about I/O files
 - **Process's address space**
 - Consists code, data, stack of the program
 - Difference between the size of process's state information (few kilobytes) and address space (several megabytes)
 - Time taken to transfer address space is more
 - Possible to transfer the address space without stopping its execution
 - Not possible to resume execution until the state information is fully transferred

Address Space Transfer Mechanisms

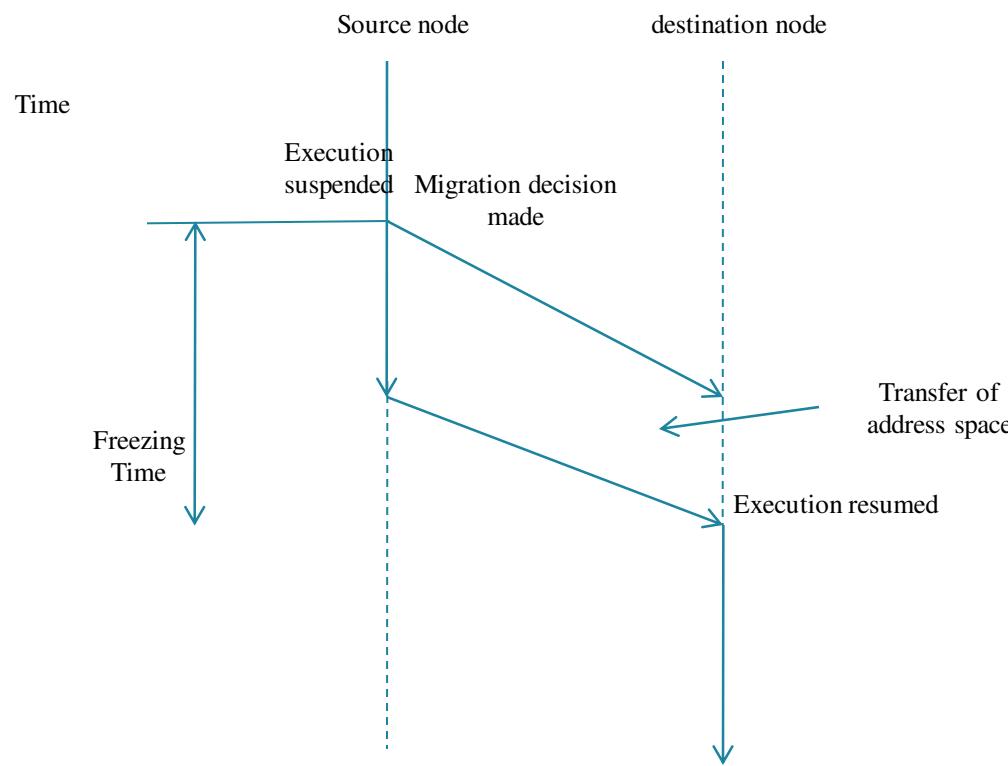
- Three methods for address space transfer
 - Total Freezing
 - Pretransferring
 - Transfer on reference

Address Space Transfer Mechanisms

- Total Freezing:
 - Process execution is stopped while its address space is being transferred
 - Simple and easy to implement
 - Disadvantages:
 - Process is suspended for a long time during migration
 - Not suitable for interactive process, the delay can occur

Address Space Transfer Mechanisms

- Total Freezing:

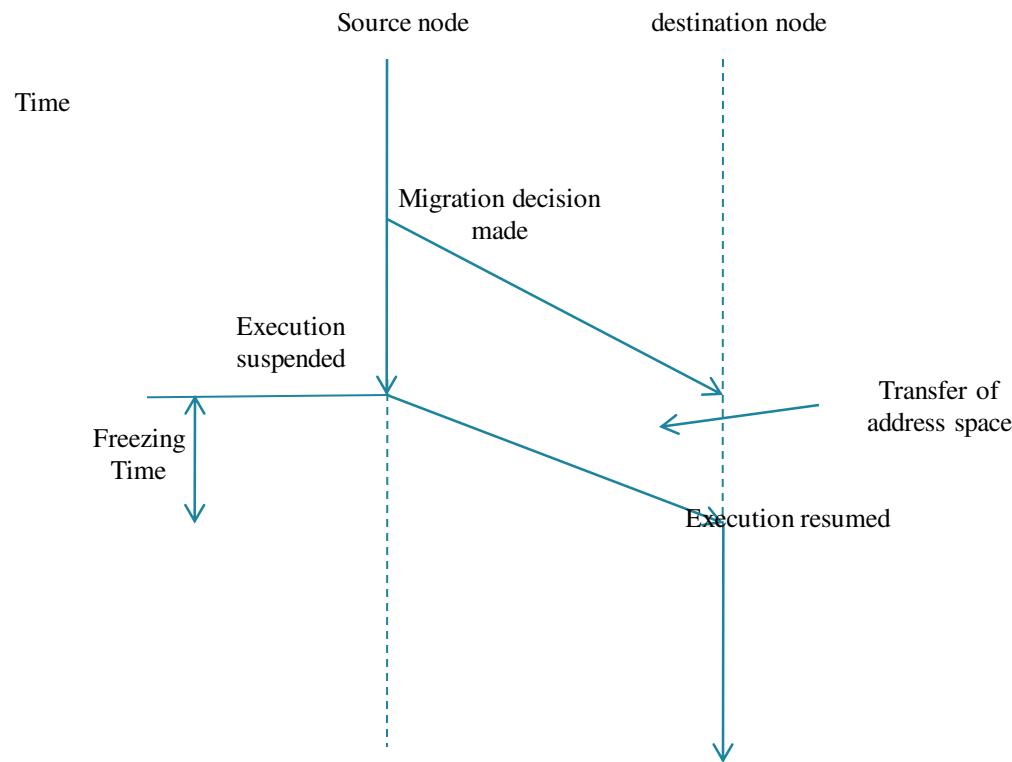


Address Space Transfer Mechanisms

- Pretransferring (precopying):
 - Address space is transferred while the process is still running on the source node.
 - Initial transfer of the complete address space followed by repeated transfers of the pages modified during previous transfer.
 - Remaining modified pages are retransferred after the process is frozen for transferring its **state information**.
 - Pretransfer operation is executed at a higher priority than all other programs on the source node.

Address Space Transfer Mechanisms

- Pretransferring (precopying):



Address Space Transfer Mechanisms

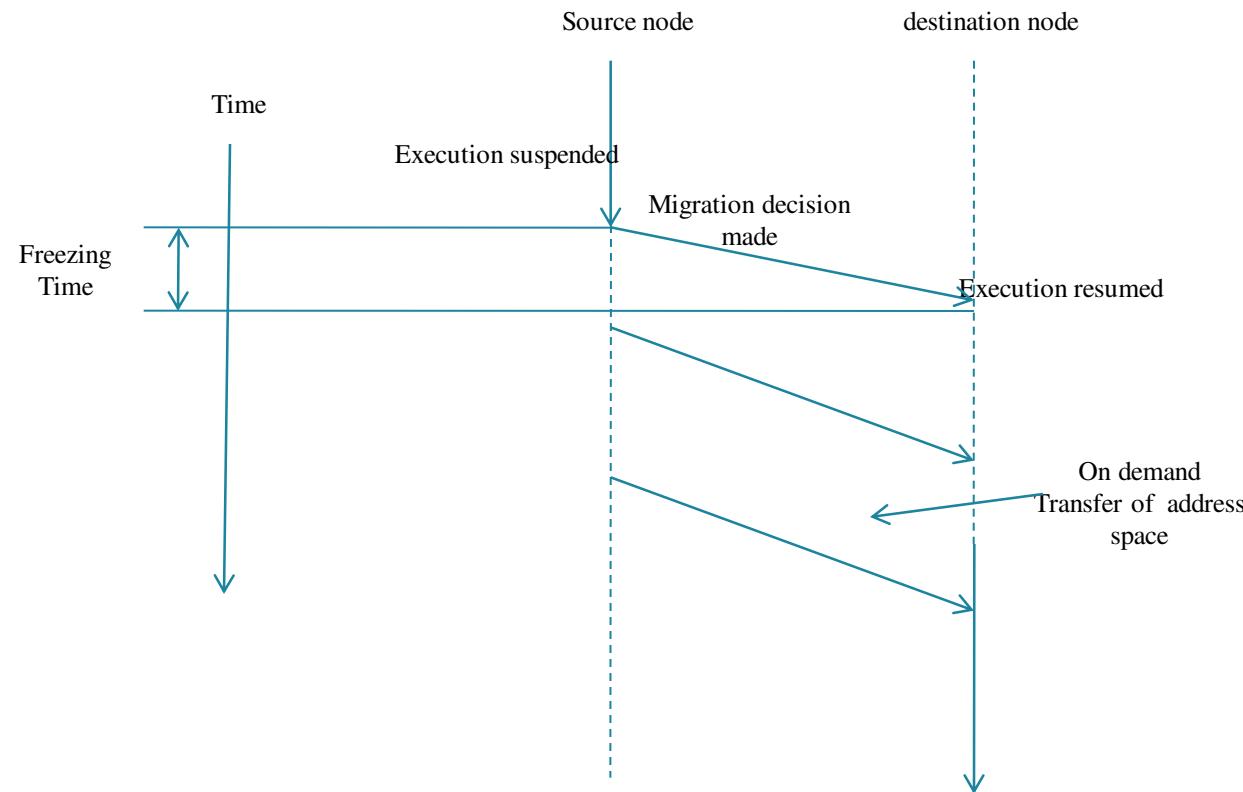
- Advantage:
 - freezing time is reduced
- Disadvantage:
 - Total time of migration is increased due to the possibility of redundant page transfers.
“Redundant pages are pages that are transferred more than once during pretransferring”

Address Space Transfer Mechanisms

- Transfer on reference
 - Based on the assumption that the process *tends* to use only a *relatively small part of their address space* while executing.
 - A page of the address space is transferred from its source node to destination node only when referenced.
 - Demand-driven copy-on-reference approach.
 - Switching time is very short and *independent of the size* of the address space.

Address Space Transfer Mechanisms

- Transfer on reference



Address Space Transfer Mechanisms

- Disadvantages:
 - Not efficient in terms of cost
 - Imposes continued load on process's source node
 - Results in failure if source node fails or reboots

Message forwarding mechanisms

- Ensures that all pending, en-route and future messages arrive at the process's new location
- **Classification of the messages to be forwarded:**
 - **Type 1:** Messages received at the source node after the process's execution has been stopped on its source node and process's execution has not yet been started on its destination node
 - **Type 2:** Message received at the source node after the process's execution has started on its destination node
 - **Type 3:** Messages that are to be sent to the migrant process from any other node after it has started executing on the destination node

Message forwarding mechanisms

- Mechanism of Resending the Message
 - Used in V-system and Amoeba.

Message Type	V-System	Amoeba
Type 1 or Type 2	<ul style="list-style-type: none">• Simply dropped the message and sender is prompted to resend it to the process's new node• Sender will retry until successful receipt of a reply	<p>Source node's kernel sends the message to sender</p> <p>Type 1: "try again later, this process is frozen"</p> <p>Type 2: "this process is unknown at this node"</p>
Type 3	Sender does a "broadcast mechanism" to find the new location of the process	

Message forwarding mechanisms

- Origin Site Mechanism
 - There is process's origin site (or home node)
 - Origin site is responsible for keeping information about the current location of all the processes created on it
 - Messages are sent to the origin site first and from there they are forwarded to the current location
 - Drawbacks:
 - not good from reliability point of view
 - Failure of origin site will disrupt the message-forwarding mechanisms
 - continuous load on migrant process's original site

Message forwarding mechanisms

- Link-Traversal Mechanism
 - Uses message queue for storing messages of type 1
 - Use of link (a forwarding address) for messages of type 2 and 3
 - Link is left at the source node pointing to the destination node
 - Link has two components:
 - Process identifier: origin node
 - last known location of the process
 - Migrated process is located by traversing a series of links

Message forwarding mechanisms

- Link-Traversal Mechanism
- Drawbacks:
 - poor efficiency
 - Several links may have to be traversed to locate a process from a node
 - poor reliability
 - If any node in the chain of links fails, the process cannot be located

Mechanisms for handling co-processes

- Important issue is the necessity to provide efficient communication between a process (parent) and its sub-processes (children)
- Two different mechanisms
 - Disallowing separation of Co-processes
 - home node or origin site concept

Mechanisms for handling co-processes

- Disallowing separation of Co-processes
 - By disallowing the migration of processes that wait for one or more of their children to complete.
 - By ensuring that when a parent process migrates, its children processes will be migrated along with it
 - Concept of logical host
 - Address spaces and their associated process's are grouped into logical host
 - Migration of a process is actually migration of logical host
- Drawback:
 - Does not allow parallelism within jobs, which is achieved by assigning various tasks of a job to the different nodes of the system and execute them simultaneously
 - Overhead is large when logical host contains several processes

Mechanisms for handling co-processes

- Home node or origin site concept
 - Complete freedom of migrating a process or its sub-processes independently and executing them on different nodes
 - All communications between a parent process and children processes take place via **home node**
- Drawback:
 - The message traffic and the communication cost increase
 - Lost communication of origin node is fails

Advantages of process migration

- Reducing average response time of processes
 - avg. response time is increase as load increase on to the node
 - Process is migrated from heavily loaded node to a ideal node
- Speeding up individual jobs
 - Execute tasks of a job concurrently
 - To migrate a job to a node having faster CPU
- Gaining higher throughput
 - In a system that does not support process migration, CPUs of all nodes are not fully utilized.
 - In a system that support process migration, CPUs of all nodes can be better utilized by using suitable load balancing policy

Advantages of process migration

- Utilizing resources effectively
 - Capabilities of the various resources such as CPU, printers, storage devices etc.
 - Depending on the nature of the process, it can be migrated to the most suitable node
- Reducing network traffic
 - Migrate the process closer to the resources
 - To migrate and cluster two or more processes which frequently communicate with each other, on the same node

Advantages of process migration

- Improving system reliability
 - Migrating critical process to more reliable node
- Improving system security
 - A sensitive process may be migrated and run on the secure node that is not directly accessible to general users thus improve the security of that process

Communication

System Architecture & Communication

- System Architectures
- Processes
- Communication in a Distributed System
- Communication Abstractions

BUILDING A DISTRIBUTED SYSTEM

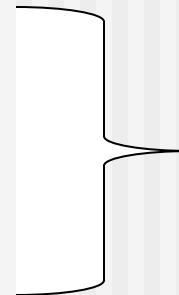
- Placing the hardware.
- Placing the software.

“Logical organization of components”

ARCHITECTURE

■ Software Architecture of Distributed System:

- It deals with how software components are organised and,
- how they work together, i.e., communicate with each other.
- Typical software architectures include:
 - *layered*,
 - *object-oriented*,
 - *data-centred*,
 - *and event-based*



Software architectures

Contd...

System Architecture:

- describe the placement of software components on physical machines
 - The realization of an architecture may be centralized (most components located on a single machine), decentralized (most machines have approximately the same functionality), or hybrid (some combination).

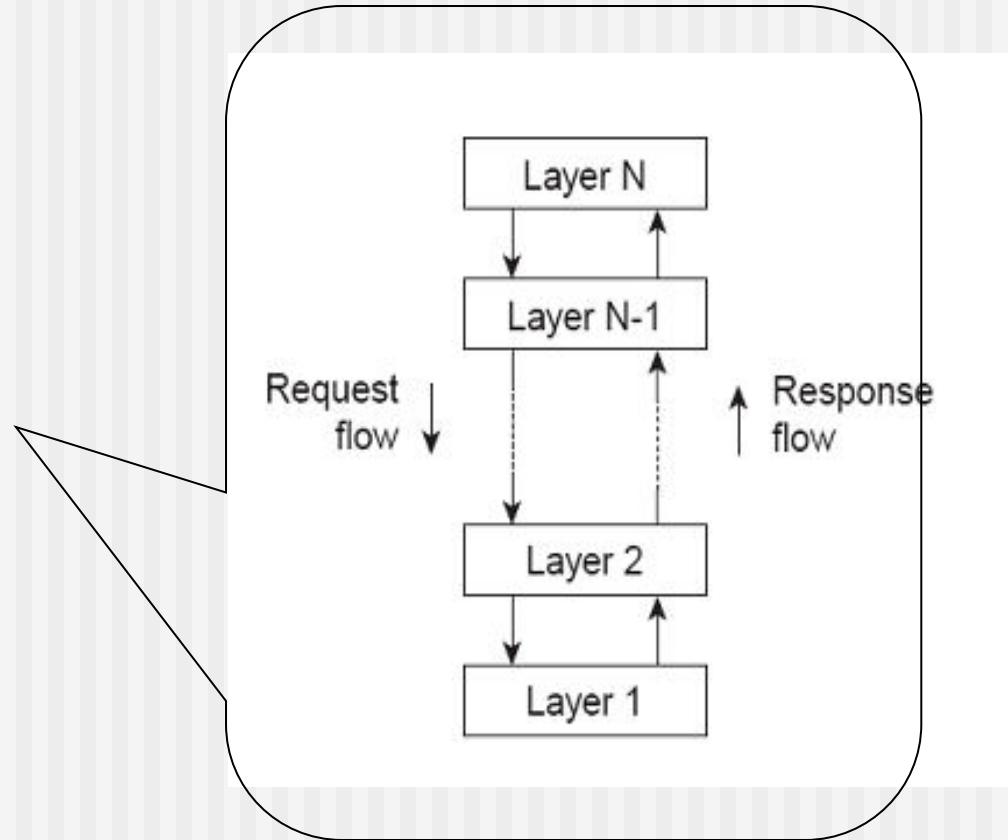
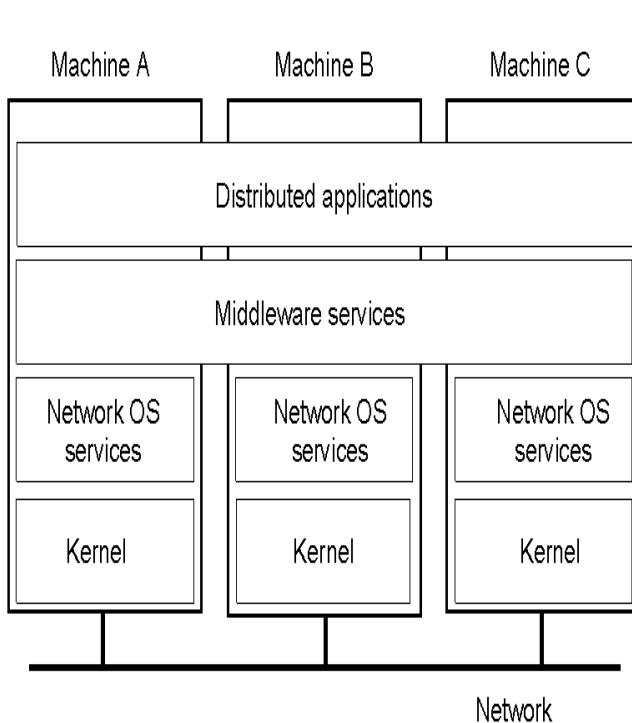
There is no single best architecture:

- The best architecture for a particular system depends on the application requirements and the environment.

Architectural Styles

- An **architectural style** describes a particular way to configure a collection of components and connectors.
 - **Component** - a module with well-defined interfaces; reusable, replaceable
 - **Connector** – communication link between modules

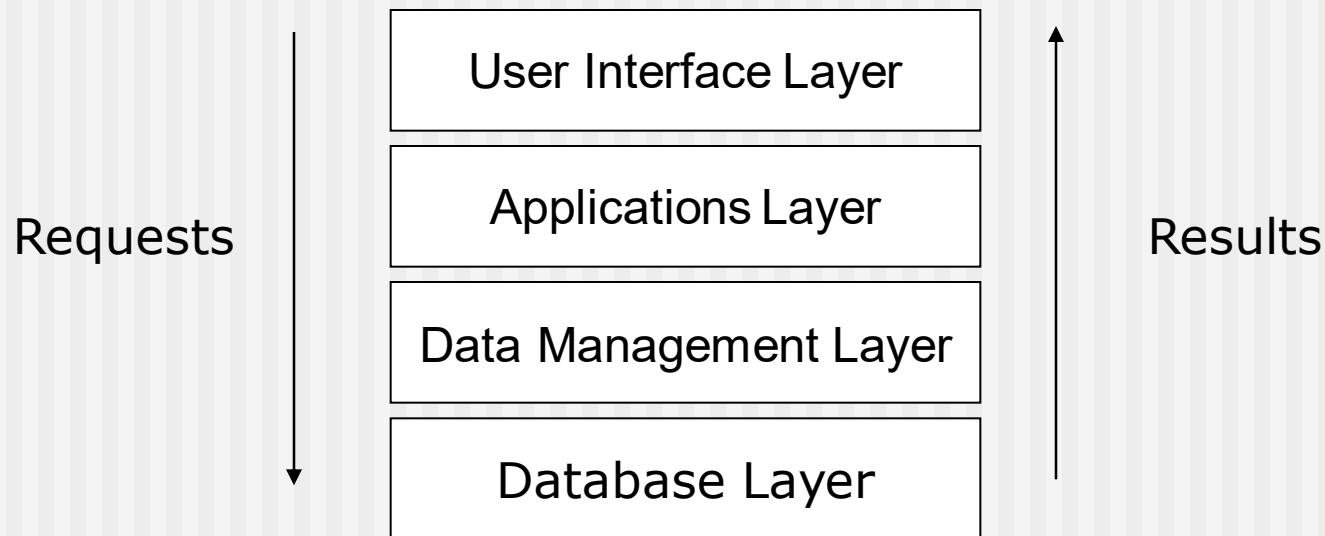
Architectural styles(1/4): Layered style



Observation: Layered style is used for client-server system.

Contd...

- Components at layer L_N are allowed to call components at underlying layers L_{N-1} , but not the other way around.

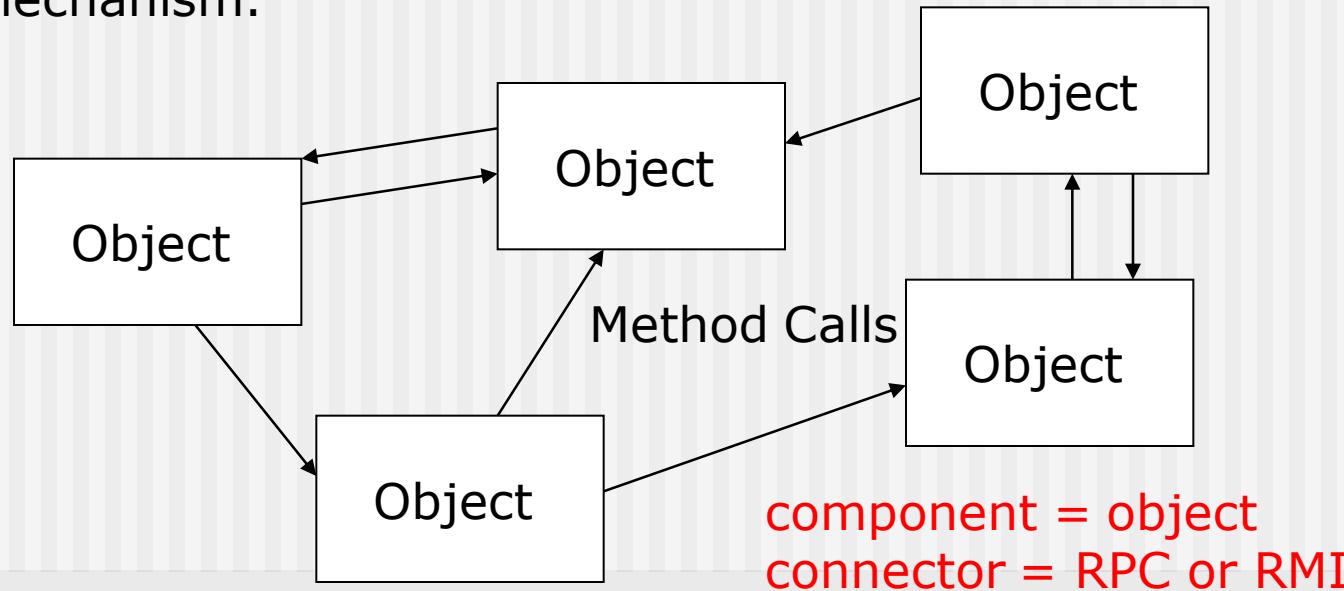


Architectural styles (2/4): object based

Basic idea: Organize into **logically different** components, and subsequently distribute those components over the various machines.

Observation: object-based style for distributed object systems.

- In essence, each object corresponds to what we have defined as a component and
- these components are connected through a (remote) procedure call mechanism.



Architectural styles (3/4): data-centered

- **Main purpose:** data access and update
- Processes interact by reading and modifying data in some shared repository (active or passive)
 - Traditional database (passive): responds to requests
 - Blackboard system (active): system updates clients when information changes.

Architectural Styles (4/4): event-based

- Processes communicate through event propagation
 - ‘Publish/Subscribe’ systems
 - Processes subscribed to events will receive them.
- Benefit is, components are loosely coupled;
 - i.e. they don’t need to explicitly refer to each other.

Contd...

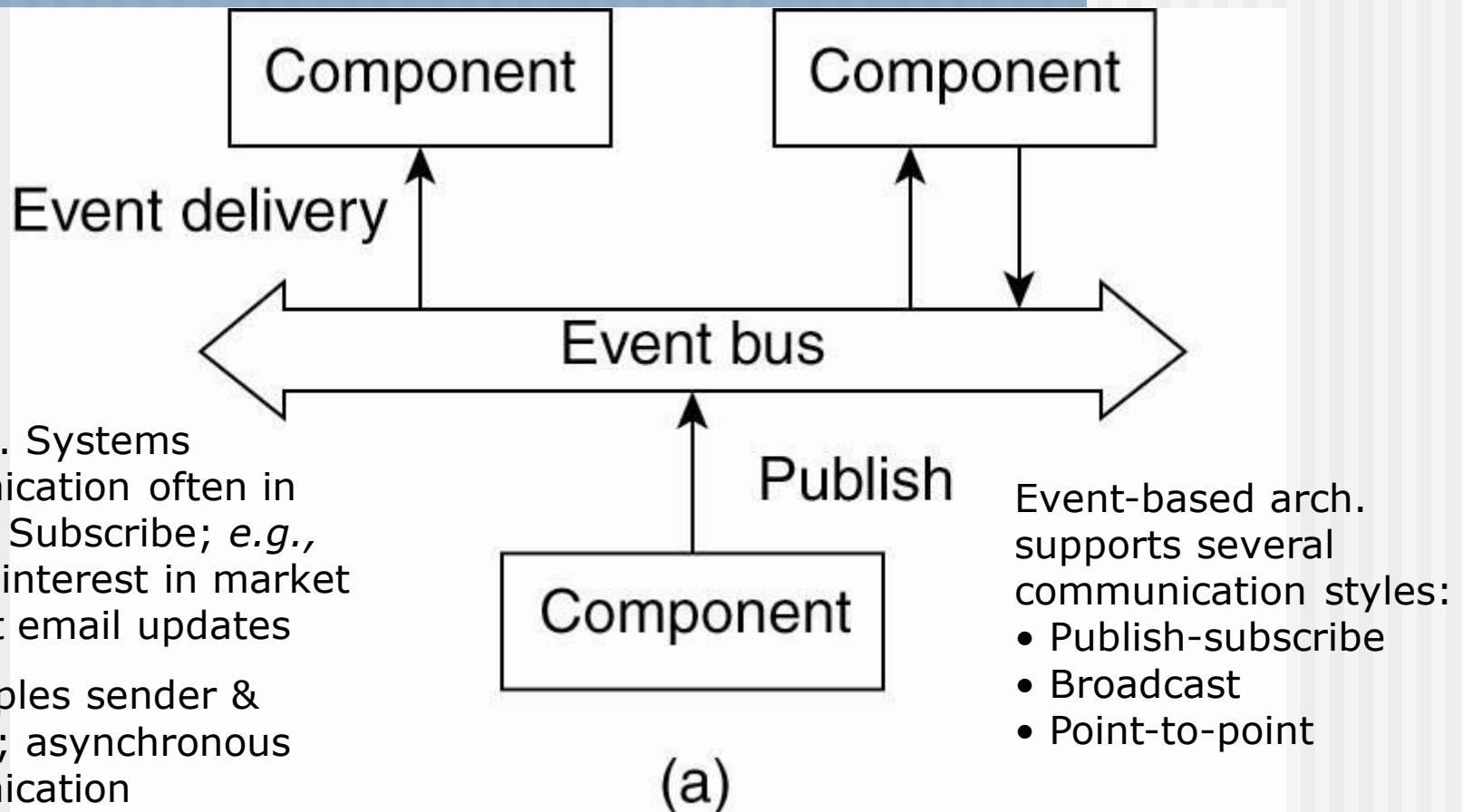
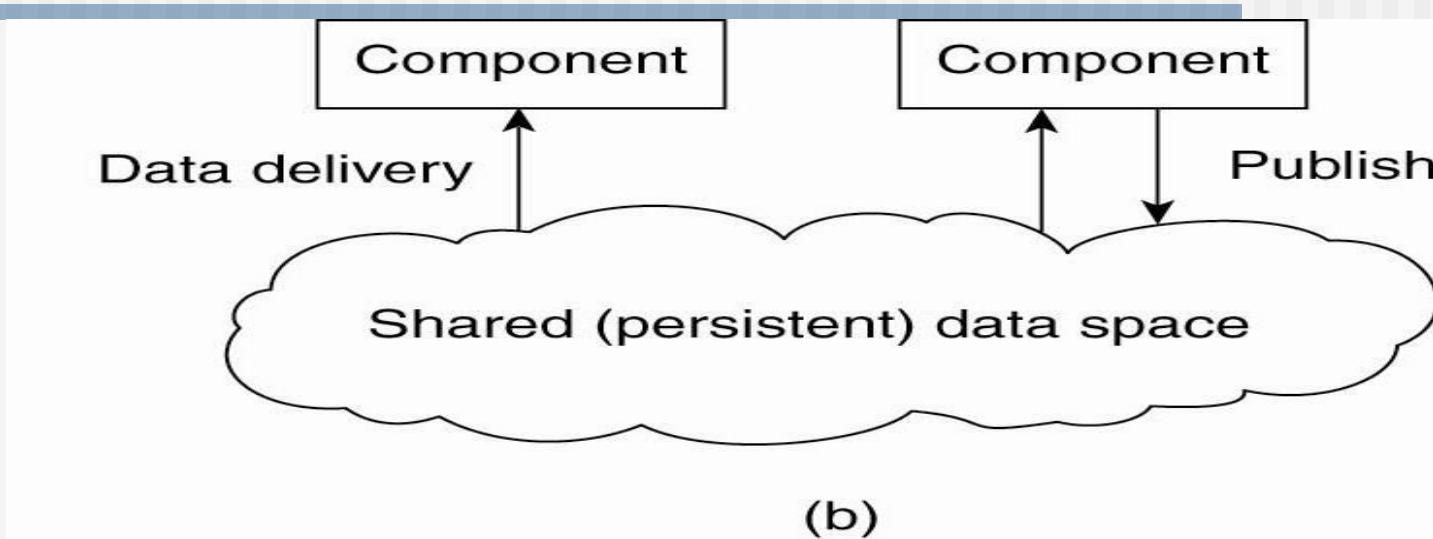


Figure (a) The event-based architectural style

Contd...



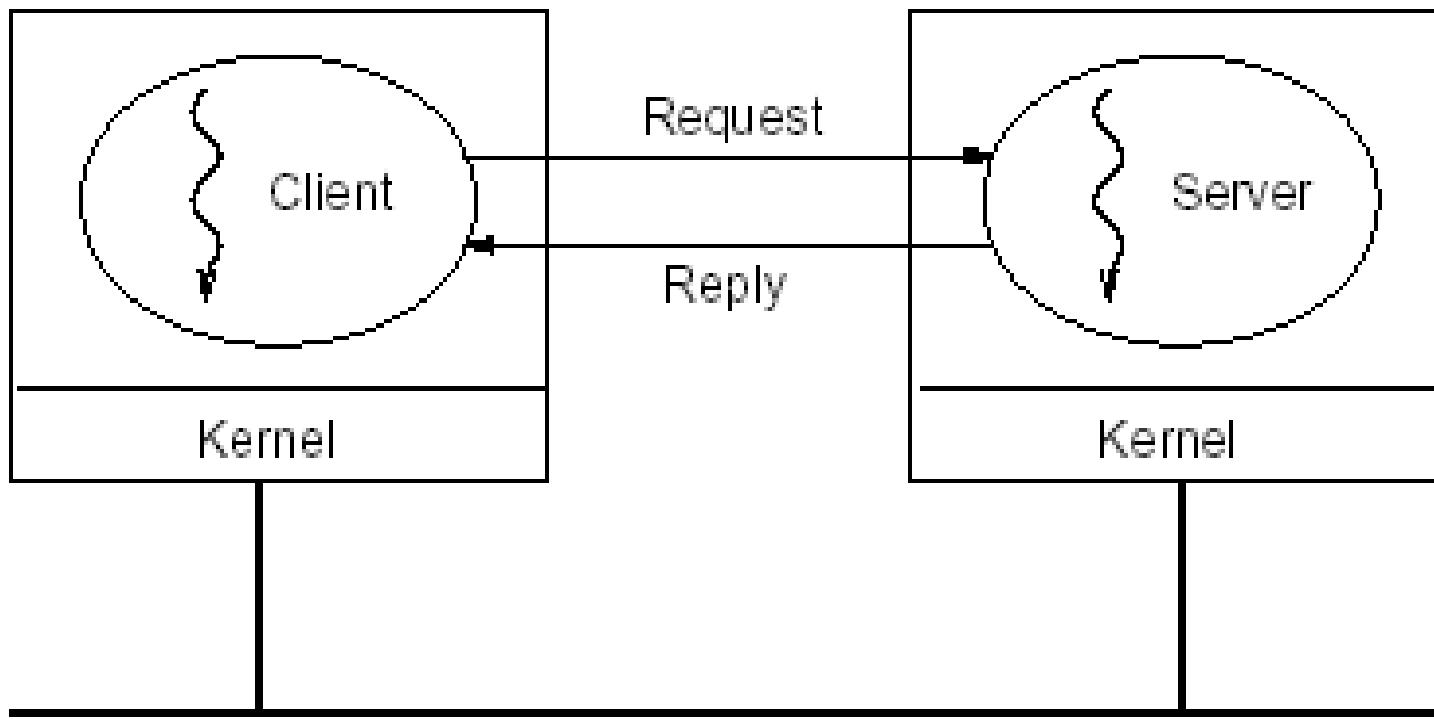
Data Centric Architecture; *e.g.*, shared distributed file systems

Combination of data-centered and event based architectures

Processes communicate asynchronously

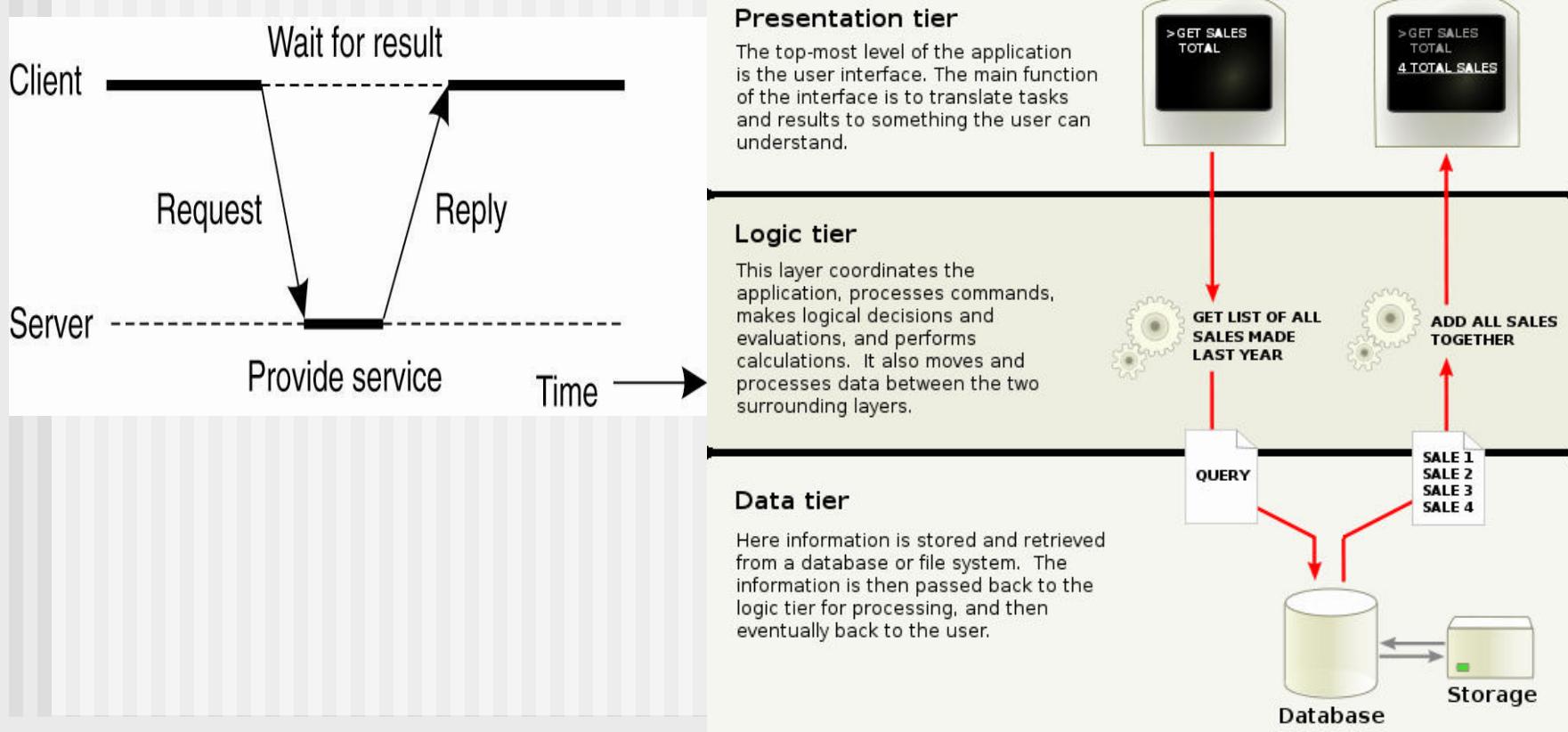
Figure (b) The shared data-space architectural style.

CLIENT-SERVER



Client-Server from another perspective

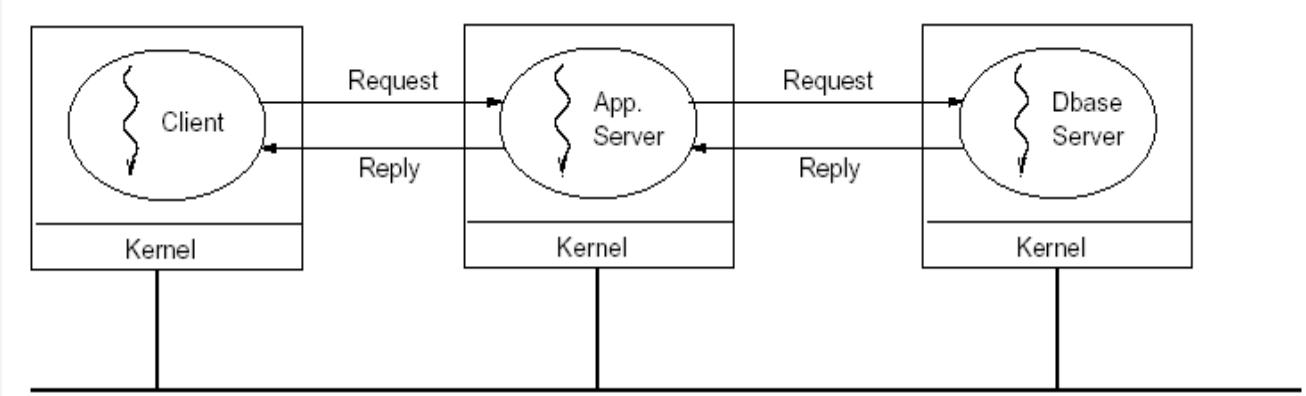
- A typical client-server application can be decomposed into three logical parts:
the interface part, the application logic part, and the data part.
- Implementations of the client-server architecture vary with regards to how the parts are separated over the client and server roles.



VERTICAL DISTRIBUTION (MULTI-TIER)

splitting up a server's functionality over multiple computers

The vertical distribution, or multi-tier, architecture distributes the traditional server functionality over **multiple servers**. A client request is sent to the first server.

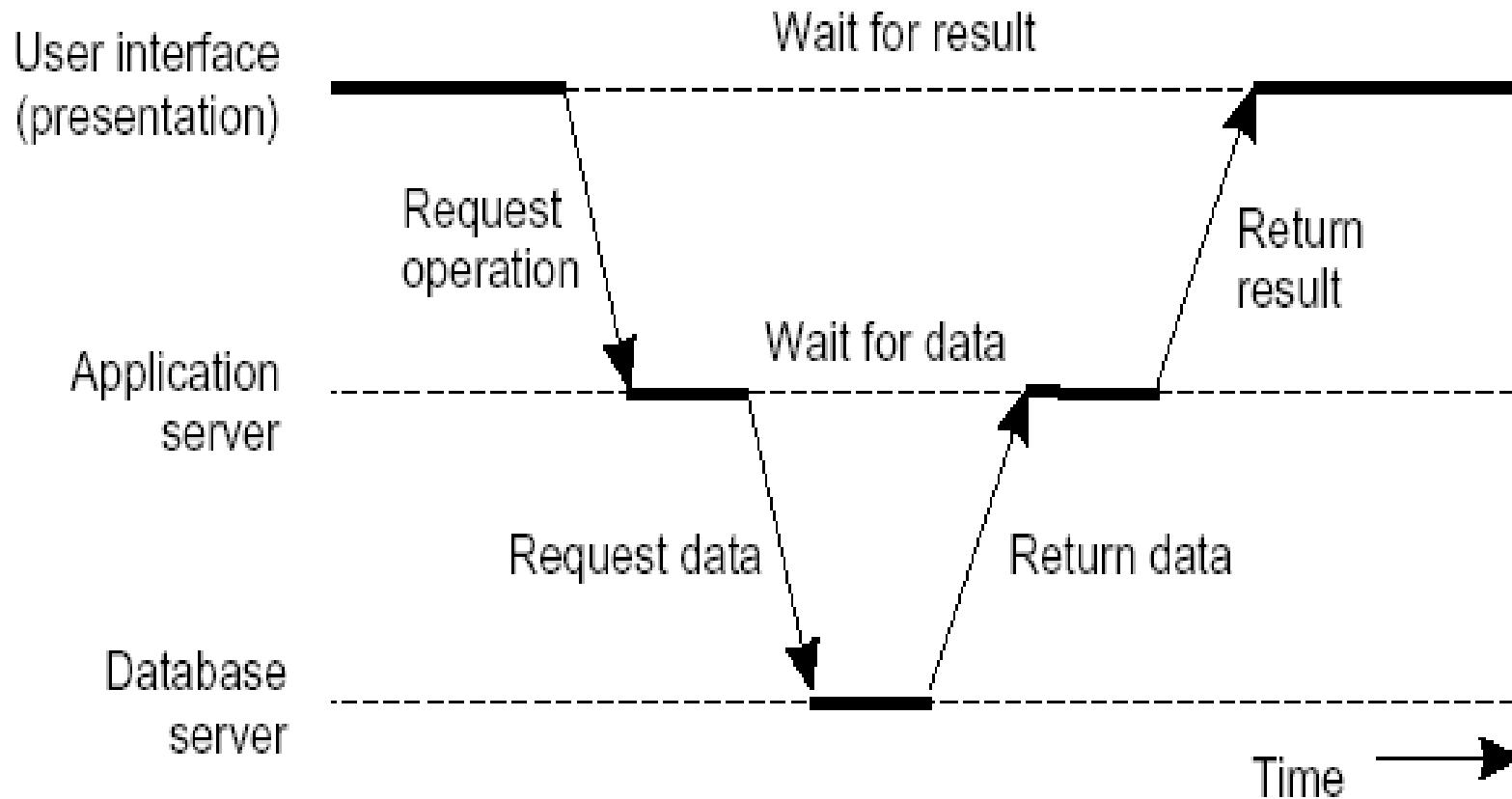


Three layers' of functionality:

- User interface
- Processing/Application logic
- Data
- Splitting up the server functionality in this way is beneficial to a system's **scalability as well as its flexibility**.
- **Scalability** is improved because the processing load on each individual server is reduced, and the whole system can therefore accommodate more users.

Logically different components on different machines

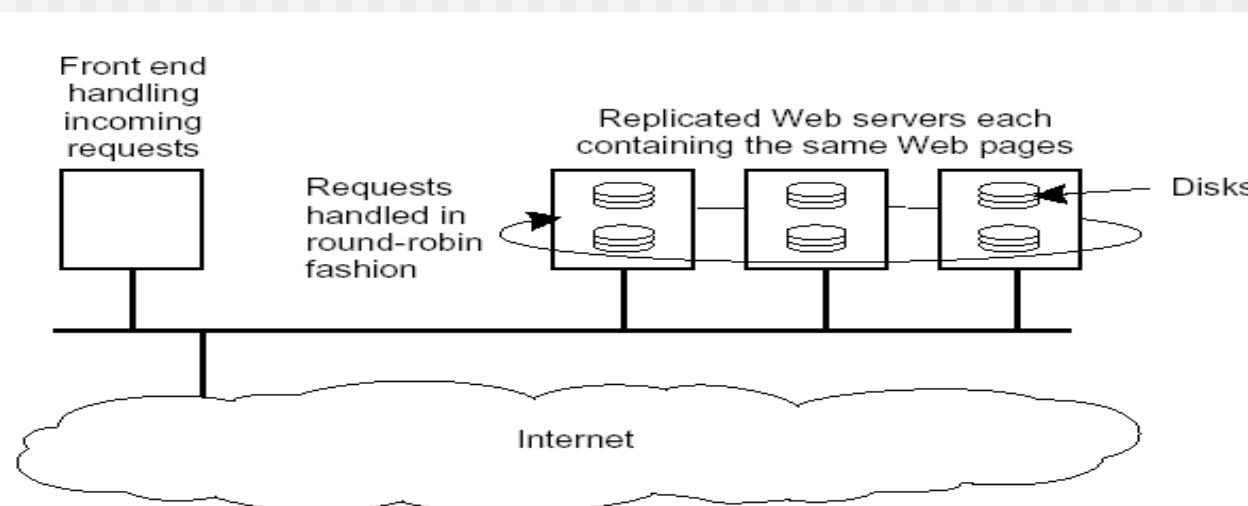
Vertical Distribution from another perspective



HORIZONTAL DISTRIBUTION

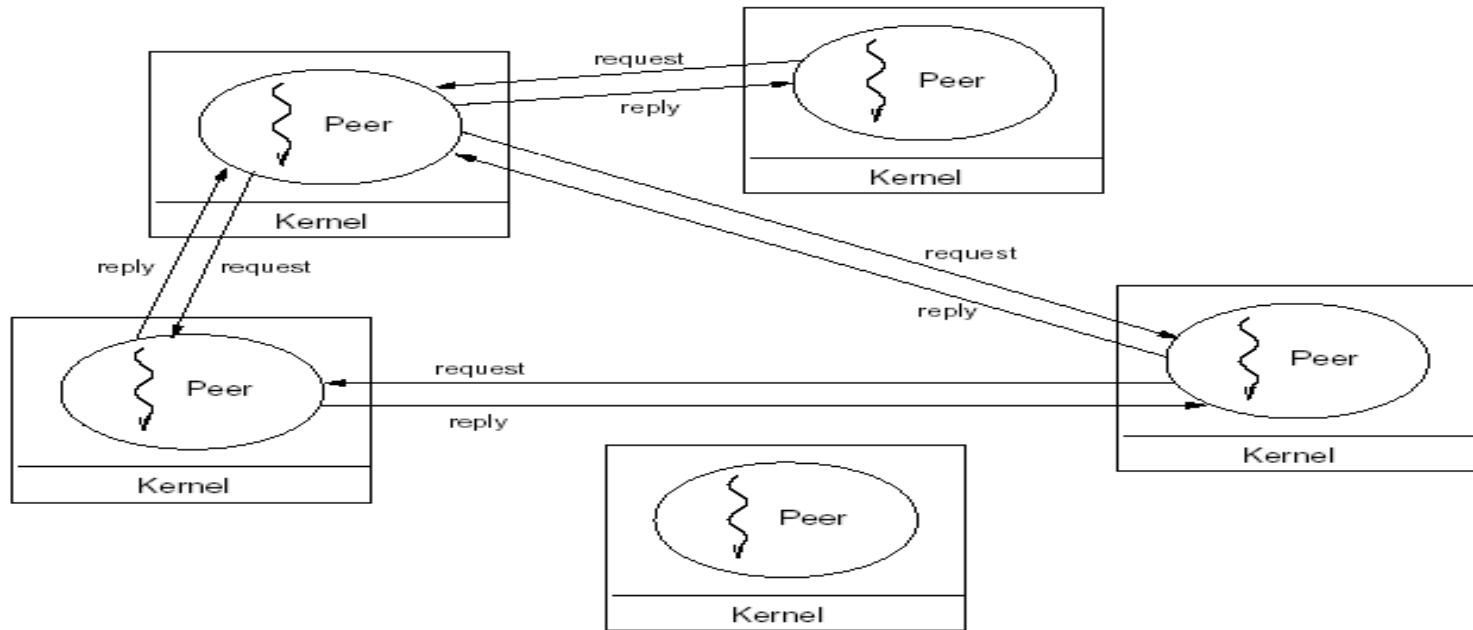
replicating a server's functionality over multiple computers

- In this case, each server machine contains a complete copy of all hosted Web pages and client requests are passed on to the servers in a round robin fashion.
- The horizontal distribution architecture is generally used to **improve scalability** (by reducing the load on individual servers) and **reliability** (by providing redundancy).



Logically equivalent components replicated on different machines

PEER TO PEER COMMUNICATION ARCHITECTURE



- All processes have client and server roles.
- With the potentially huge number of participating nodes in a peer to peer network, **it becomes practically impossible for a node to keep track of all other nodes in the system and the information they offer.**

OVERLAY NETWORKS

- How do peers keep track of all other peers?
 - **Overlay networks** connect nodes in the P2P system
 - Nodes in the overlay use their own addressing system for storing and retrieving data in the system
 - Nodes can route requests to locations that may not be known by the requester.
- Overlay Network
1. Unstructured
 2. Structured

UNSTRUCTURED OVERLAY

- Data stored at random nodes.
- Exchange partial views with neighbours to update.
- Partial view: which means that a node partial view consists of a random list of other nodes.

STRUCTURED OVERLAY

- The node's neighbours is determined according to a **specific structure**.
- Nodes have identifier and range.
- Node is responsible for data that falls in its range.
- Search is routed to appropriate node.

Comparison

- Structured networks typically guarantee that if an object is in the network it will be located in a bounded amount of time.
- Unstructured networks offer no guarantees.
 - For example, some will only forward search requests to a specific number of hops

HYBRID ARCHITECTURES

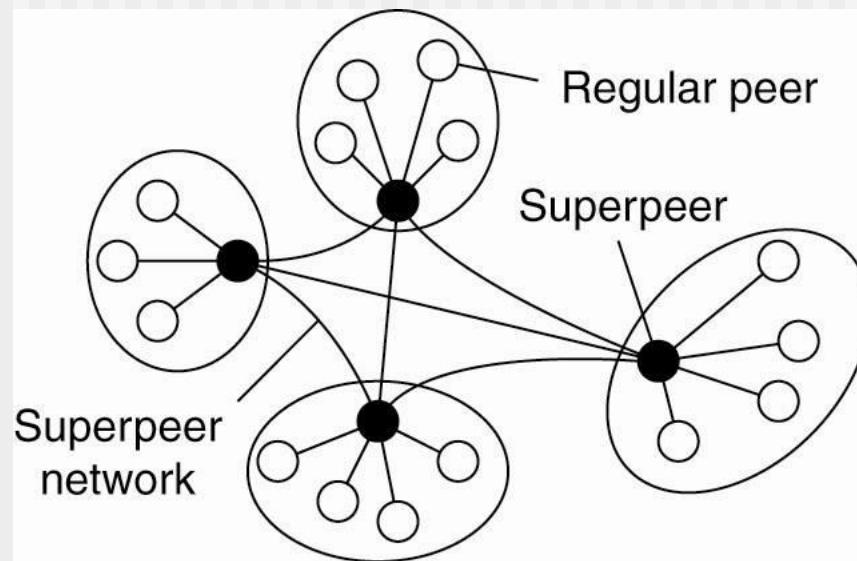
■ Combination of architectures:

Examples:

- Super peer networks
- Collaborative distributed systems

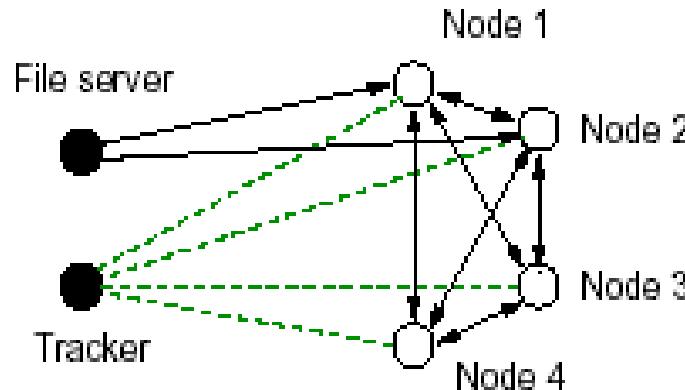
Superpeer Networks

- In this architecture a few superpeers form a peer to peer network, while the regular peers are clients to a superpeer.
- Superpeers managing the index of the regular peers
- Act as servers to regular peer nodes, peers to other superpeers
- Improve scalability



Collaborative Distributed Systems

- In **collaborative distributed systems**, peers typically support each other to deliver content in a peer to peer like architecture, while they use a client server architecture for the initial setup of the network.
- Nodes requesting to download a file from a server first contact the server to get the location of a tracker.
- Tracker keeps track of **active nodes** that have chunks of file.
- Using information from the tracker, clients can download the file in chunks from multiple sites in the network.
- Nodes must then offer downloaded chunks to other nodes and are registered with the tracker, so that the other nodes can find them.



PROCESSES AND THREADS

- A *process*, in the simplest terms, is an executing program.
- One or more threads run in the context of the process.
- Sometimes threads are also referred to as *lightweight processes* because they take up less operating system resources than regular processes.
- A *thread* is the basic unit to which the operating system allocates processor time.
- A thread can execute any part of the process code, including parts currently being executed by another thread.

PROCESSES AND THREADS

- Some systems provide only a *process model*, while others provide only a *thread model*. More common are systems that provide both threads and processes. In this case each process can contain multiple threads, which means that the **threads can only freely access the memory of other threads in the same process**.
- A *server process* in a distributed system typically receives many requests for work from various clients, and it is important to provide quick responses to those clients. In particular, a server should not refuse to do work for one client because it is blocked while doing work for another client. This is a typical result of implementing a **server as a single-threaded process**.
- Alternatives to this are to implement the **server using multiple threads**, one of which acts as a dispatcher, and the others acting as workers.

STATEFUL VS STATELESS SERVERS

■ Stateful:

- Keeps persistent information about clients
- Improved performance
- Expensive recovery
- Must track clients

■ Stateless:

- Does not keep state of clients
- Soft state design: limited client state
- Can change own state without informing clients
- Increased communication

COMMUNICATION

In distributed system processes running on separate computers cannot directly access each other's memory.

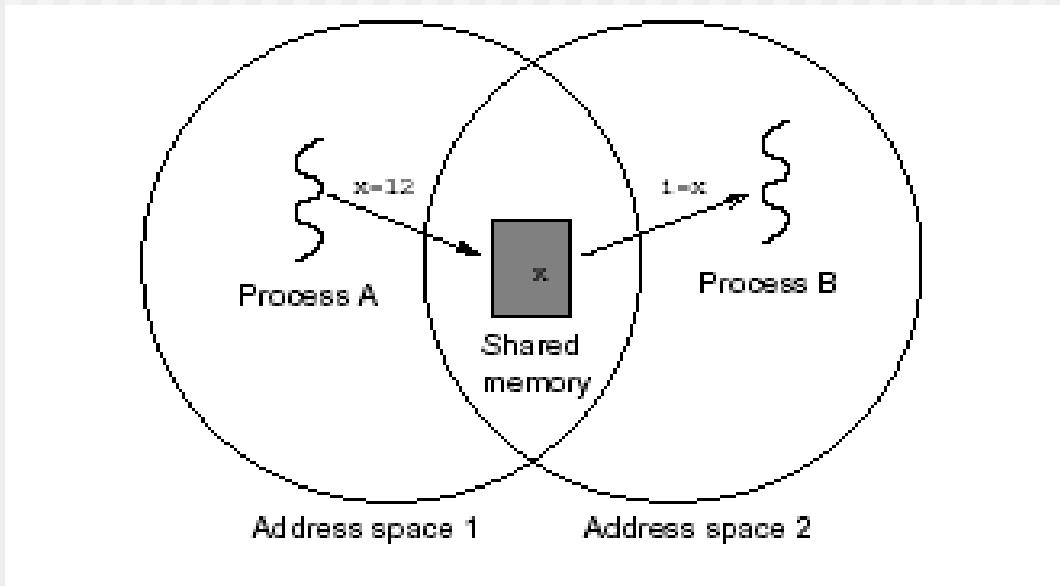
■ Two approaches to communication:

Shared memory-processes must have access to some form of shared memory (i.e., they must be threads, they must be processes that can share memory, or they must have access to a shared resource, such as a file)

In DSM the processes on separate computers all have access to the same virtual address space. The memory pages that make up this address space actually reside on separate computers.

Message passing- processes communicate by sending each other messages

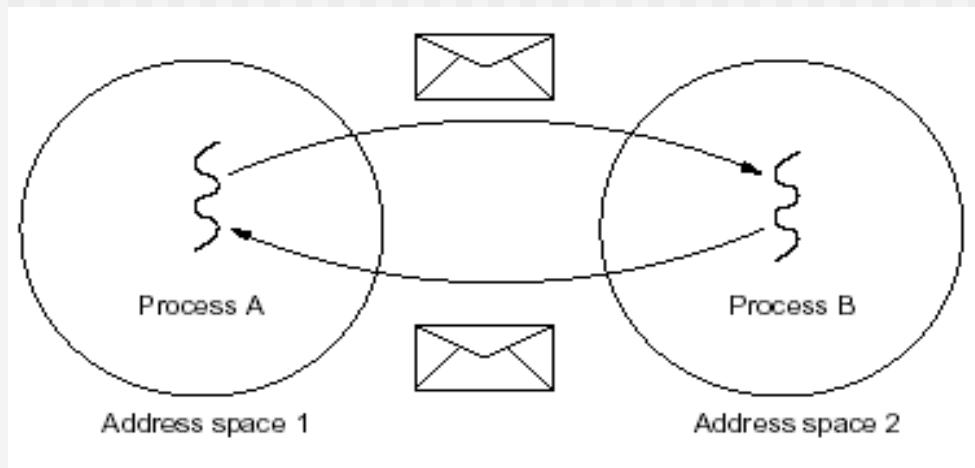
Shared Memory



■ Shared Memory:

- There is no way to physically share memory
- Distributed Shared Memory

Message Passing



■ Message Passing:

- Over the network
- Introduces higher chances of failure
- Heterogeneity introduces possible incompatibilities

COMMUNICATION MODES

- Data oriented vs control oriented communication
- Synchronous vs asynchronous communication
- Transient vs persistent communication

Data-Oriented vs Control-Oriented Communication

- **Data-oriented communication**

- Facilitates data exchange between threads

- **Control-oriented communication**

- Associates a transfer of control with every data transfer
 - remote procedure call (RPC) & remote method invocation (RMI)

- **Observation:**

- Hardware and OSes often provide data-oriented communication
 - Higher-level infrastructure often provides control-oriented communication – middleware

Synchronous vs Asynchronous Communication

■ Synchronous:

- Sender blocks until message received
 - Often sender blocked until message is processed and a reply received
- **Sender and receiver must be active at the same time**
- Client-Server generally uses synchronous communication

■ Asynchronous:

- Sender continues execution after sending message (does not block waiting for reply)
- **Message may be queued if receiver not active**
- Message may be processed later at receiver's convenience

When is Synchronous suitable? Asynchronous?

Transient vs Persistent Communication

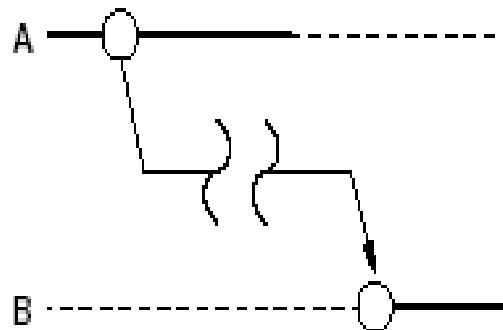
■ Transient:

- Message discarded if cannot be delivered to receiver immediately
- a message will only be delivered if a receiver is active. If there is no active receiver process (i.e., no one interested in or able to receive messages) then an undeliverable message will simply be dropped.
- Example: HTTP request

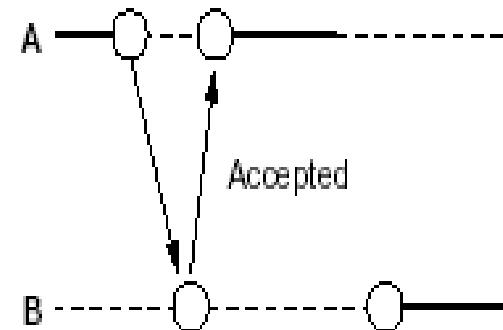
■ Persistent

- A message will be stored in the system until it can be delivered to the intended recipient.
- Example: email

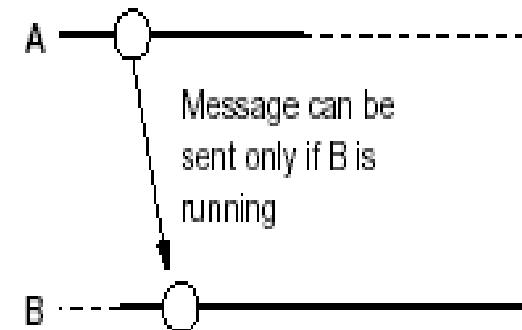
Possible combinations of synchronous/asynchronous and transient/persistent communication



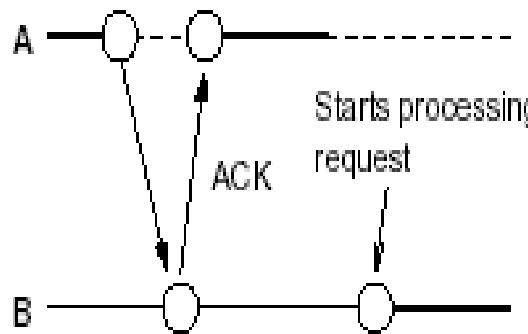
Persistent Asynchronous



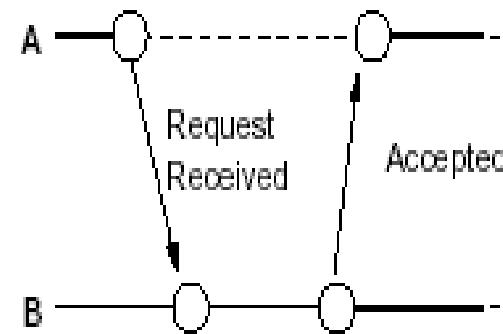
Persistent Synchronous



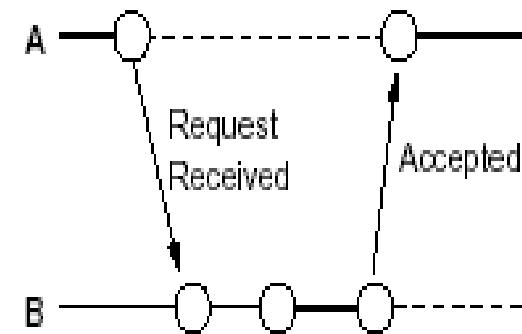
Transient Asynchronous



Transient Synchronous
(Receipt Based)



Transient Synchronous
(Delivery Based)



Transient Synchronous
(Response Based)

COMMUNICATION ABSTRACTIONS

- Number of communication abstractions that make writing distributed applications easier.

Provided by higher level APIs:

- Remote Procedure Call (RPC)
- Remote Method Invocation (RMI)
- Message-Oriented Communication
- Group Communication
- Streams

REMOTE PROCEDURE CALL (RPC)

- **Idea:** allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network).
 - Synchronous - based on blocking messages
 - Message-passing details hidden from application
 - Procedure call parameters used to transmit data
 - Client calls local “stub” which does messaging and marshaling

REMOTE METHOD INVOCATION (RMI)

- The transition from Remote Procedure Call (RPC) to Remote Method Invocation (RMI) is a transition from the server metaphor to the object metaphor.

Why is this important?

- Using RPC: programmers must explicitly **specify the server** on which they want to perform the call.
- Using RMI: programmers invoke methods on remote objects.
- More natural resource management and error handling

MESSAGE-ORIENTED COMMUNICATION:

MESSAGE-ORIENTED MIDDLEWARE (MOM)

- Message-oriented communication is provided by message-oriented middleware (MOM).

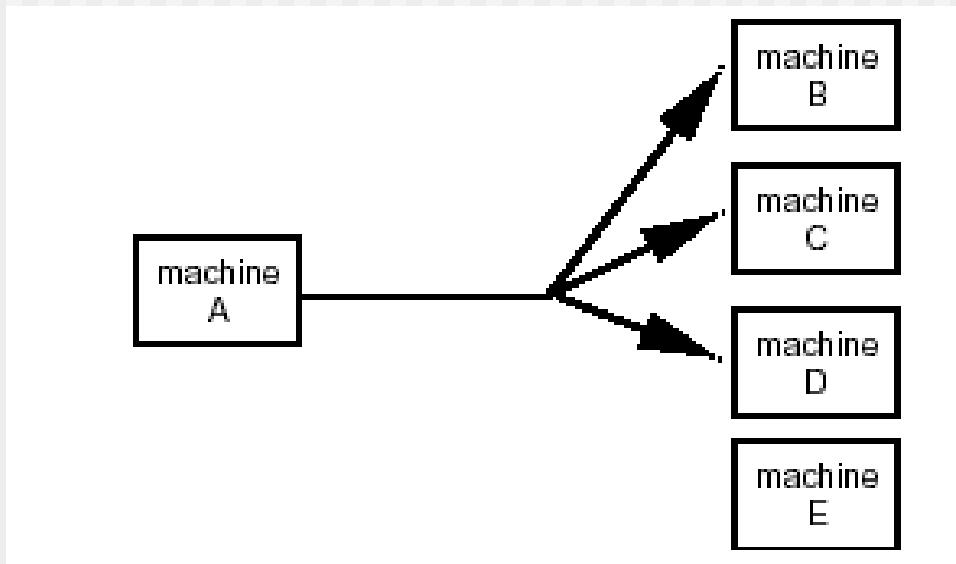
Middleware support for message passing:-

- Asynchronous or Synchronous communication
- Persistent or Transient messages
- Various combinations of the above

What more does it provide than send()/receive()?

- Persistent communication (Message queues)
- Hides implementation details
- Marshaling (packing of function parameters into a message packet)

GROUP COMMUNICATION



- Sender performs a single send()

GROUP COMMUNICATION.....

- Two kinds of group communication:
 - Broadcast (messgae sent to everyone)
 - Multicast (message sent to specific group)
- Used for:
 - Replication of services
 - Replication of data
 - Service discovery
- Issues:
 - Reliability
 - Ordering
- Example:
 - IP multicast

STREAMS

- Support for Continuous Media
 - 1. Between applications
 - 2. Between devices
 - 3. Data represented as single stream rather than discrete chunks

SUMMARY

- Architectures
 - Client-server (and multi-tier)
 - Peer to peer
 - Hybrid architectures
- Processes in distributed systems
- Communication modes (data oriented vs control oriented, synchronous vs asynchronous, transient vs persistent)
- Communication Abstractions
 - RPC and RMI
 - Message-oriented communication
 - Group communication (multicast)
 - Streams

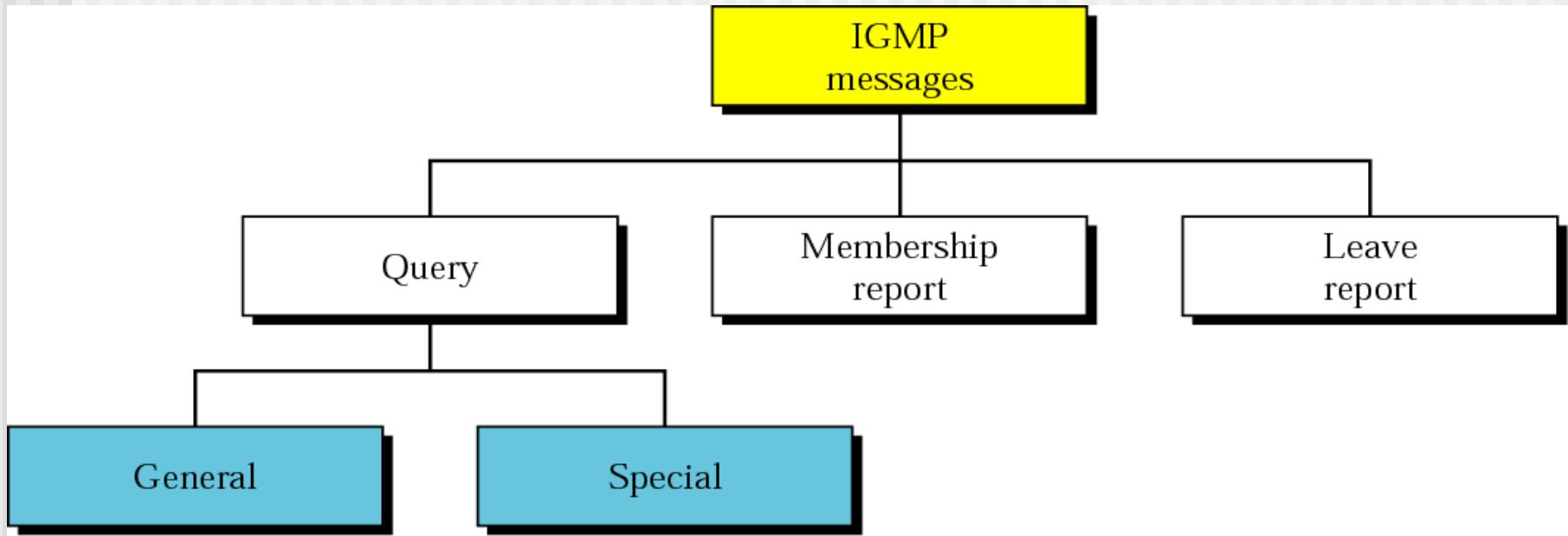
Appendix: Internet Group Management Protocol

- *IP protocol can be involved in two types of communication: unicasting and multicasting. Multicasting has many applications. For example, multiple stockbrokers can simultaneously be informed of changes in a stock price.*
- *IGMP is a protocol that manages group membership. The IGMP protocol gives the multicast routers information about the membership status of hosts (routers) connected to the network. .*

Note:

IGMP is a group management protocol. It helps a multicast router create and update a list of loyal members related to each router interface.

IGMP message types



Synchronization

Synchronization and Coordination

- Distributed Algorithms
- Time and Clocks
- Global State
- Concurrency Control
- Distributed Transactions

Synchronization

- How processes cooperate and synchronize with each other?
- Simultaneously access a shared resources , such as printer,
- Agree on the ordering of events, such as whether message $m1$ from process P was sent before or after message $m2$ from process Q .

DISTRIBUTED ALGORITHMS

Algorithms that are intended to work in a distributed environment.

- Used to achieve tasks such as:
 - Communication
 - Accessing resources
 - Allocating resources
 - etc.
- Synchronization and coordination linked to distributed algorithms
 - Achieved using distributed algorithms

SYNCHRONOUS Vs ASYNCHRONOUS DISTRIBUTED SYSTEMS

Timing model of a distributed system

■ Affected by:

- Execution speed/time of processes
- Communication delay
- Clocks & clock drift
- (Partial) failure

Synchronous Distributed System:

Time variance is bounded

- Execution: bounded execution speed and time
- Communication: bounded transmission delay
- Clocks: bounded clock drift (and differences in clocks)

■ Effect:

- Easier to design distributed algorithms
- Very restrictive requirements
 - Limits concurrent use of network
 - Require precise clocks and synchronization

Asynchronous Distributed System

Time variance is not bounded

- Execution: different steps can have varying duration
 - Communication: transmission delays vary widely
 - Clocks: arbitrary clock drift
- Effect:
- Allows no assumption about time intervals
 - Most asynch DS problems hard to solve
 - Solution for asynch DS is also a solution for synch DS
 - Most real distributed systems are hybrid synch and asynch.

EVALUATING DISTRIBUTED ALGORITHMS

■ General Properties:

- Performance
 - number of messages exchanged
 - response/wait time
 - delay
 - throughput
 - complexity:
- Efficiency
 - resource usage: memory, CPU, etc.
- Scalability
- Reliability
 - number of points of failure (low is good)

SYNCHRONIZATION AND COORDINATION

- making sure that processes **doing the right thing at the right time.**
- allowing processes to **synchronize and coordinate their actions**

- Two fundamental issues:
 - Coordination (the right thing)
 - Synchronization (the right time)

COORDINATION

- Coordination refers to coordinating the actions of separate processes relative to each other and **allowing them to agree on global state.**

“Coordinate actions and agree on values.”

- **Coordinate Actions:**

- What actions will occur
- Who will perform actions

- **Agree on Values:**

- Agree on global value
- Agree on environment
- Agree on state

Cont ...

- Examples of coordination include ensuring that processes agree on –
 - what actions will be performed?
 - who will be performing actions?
 - and the state of the system?

SYNCHRONIZATION

- Synchronization is coordination with respect to time, and refers to the ordering of events and execution of instructions in time.
“Ordering of all actions”

- Total ordering of events
(whether message m1 from process P was sent before or after message m2 from process Q)
- Ordering of access to resources

Cont ...

- Examples of synchronization include,
- Ordering distributed events and ensuring that a process performs an action at a particular time.

MAIN ISSUES

- **Time and Clocks:** synchronizing clocks and using time in distributed algorithms
- **Global State:** how to acquire knowledge of the system's global state
- **Concurrency Control:** coordinating concurrent access to resources
- **Coordination:** when do processes need to coordinate and how do they do it

TIME AND CLOCKS

TIME AND CLOCKS

- It is often important to know **when** events occurred and in **what** order they occurred.
- In a non-distributed system dealing with time is trivial as there is a single shared clock, where all processes see the same time.

Cont ...

- In a distributed system, on the other hand, each computer has its own clock.
- Because no clock is perfect each of these clocks has its own skew which causes clocks on different computers to drift and eventually become out of sync.

CLOCKS

■ Computer Clocks:

- Oscillates at known frequency

■ Clock Skew:

- Different computers run at slightly different rates
- Clocks get out of sync
- Maximum drift rate

■ Timestamps:

- Used to denote at which time an event occurred

PHYSICAL CLOCKS

■ Based on actual time:

- Physical clocks keep track of physical time.
- In distributed systems that based on actual time it is necessary to keep individual computer clocks synchronized.
- The clocks can be synchronized to global time (external synchronization), or to each other (internal synchronization).

PHYSICAL CLOCKS

■ Examples:

- Performing events at an exact time (turn lights on/off, lock/unlock gates)
- sorting of events (for security, for profiling)
- Tracking (tracking a moving object with separate cameras)

SYNCHRONIZING PHYSICAL CLOCKS

■ External Synchronization:

- Clocks synchronize to an external time source
- Synchronize with UTC every δ seconds

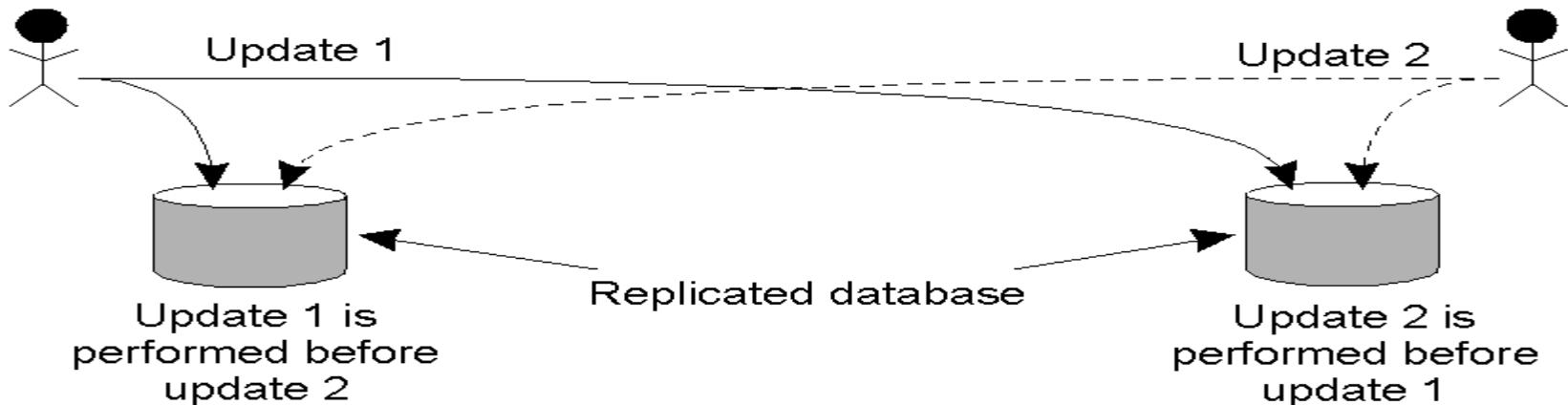
■ Internal Synchronization:

- Clocks synchronize locally
- Only synchronized with each other

■ Time Server:

- Server that has the correct time
- Server that calculates the correct time

What can go wrong?



- We need to guarantee that concurrent updates on a replicated database are seen in same order everywhere. This requires a **totally-ordered multicast**.
- Update 1: adds 100 rupees to an account, (initial value= 1000/-)
- Update 2: adds 1% interest to the same account.
- In absence of proper synchronization: replica 1=1111, replica 2= 1110
- Clock reading might be different for different computers!
- Inconsistent state
- Can use Lamport's to totally order

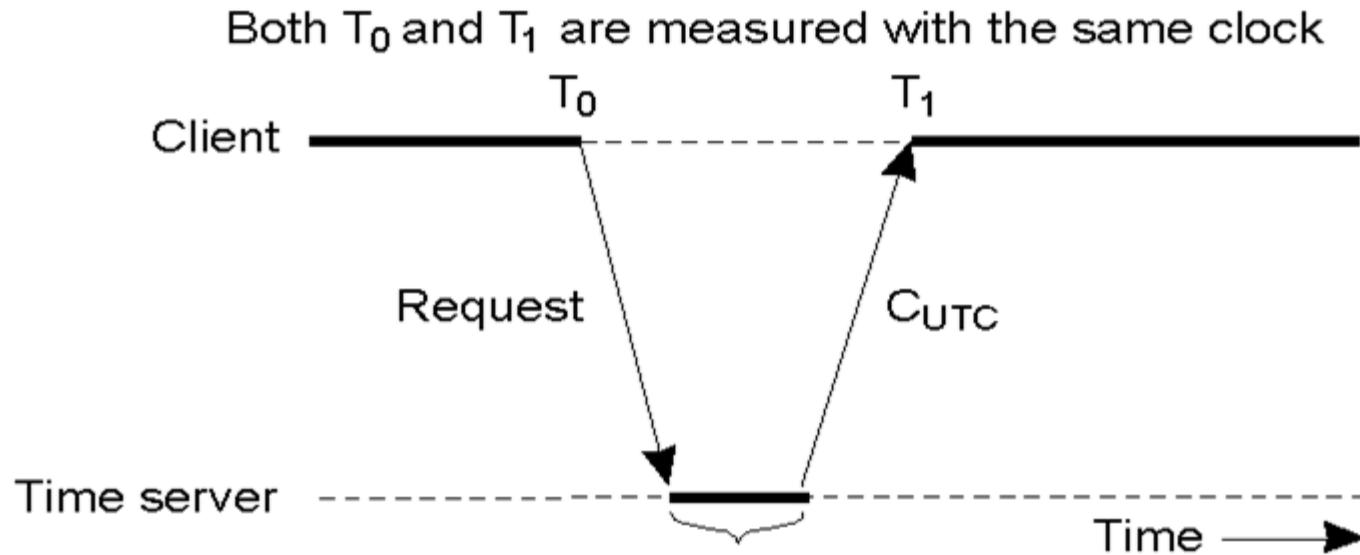
Computer Clock

- We need to measure time accurately:
 - to know the time an event occurred at a computer
- Algorithms for clock synchronization useful for
 - concurrency control based on timestamp ordering
 - authenticity of requests e.g. in Kerberos etc
- Each computer in a DS has its own internal clock

Clock Synchronization

- There exists a *time server* receiving signals from a UTC source
 - Cristian's algorithm
- There is no UTC source available
 - Berkley's algorithm
- Exact time *does not matter!*
 - Lamport's algorithm

Clock Sync. Algorithm: Cristian's

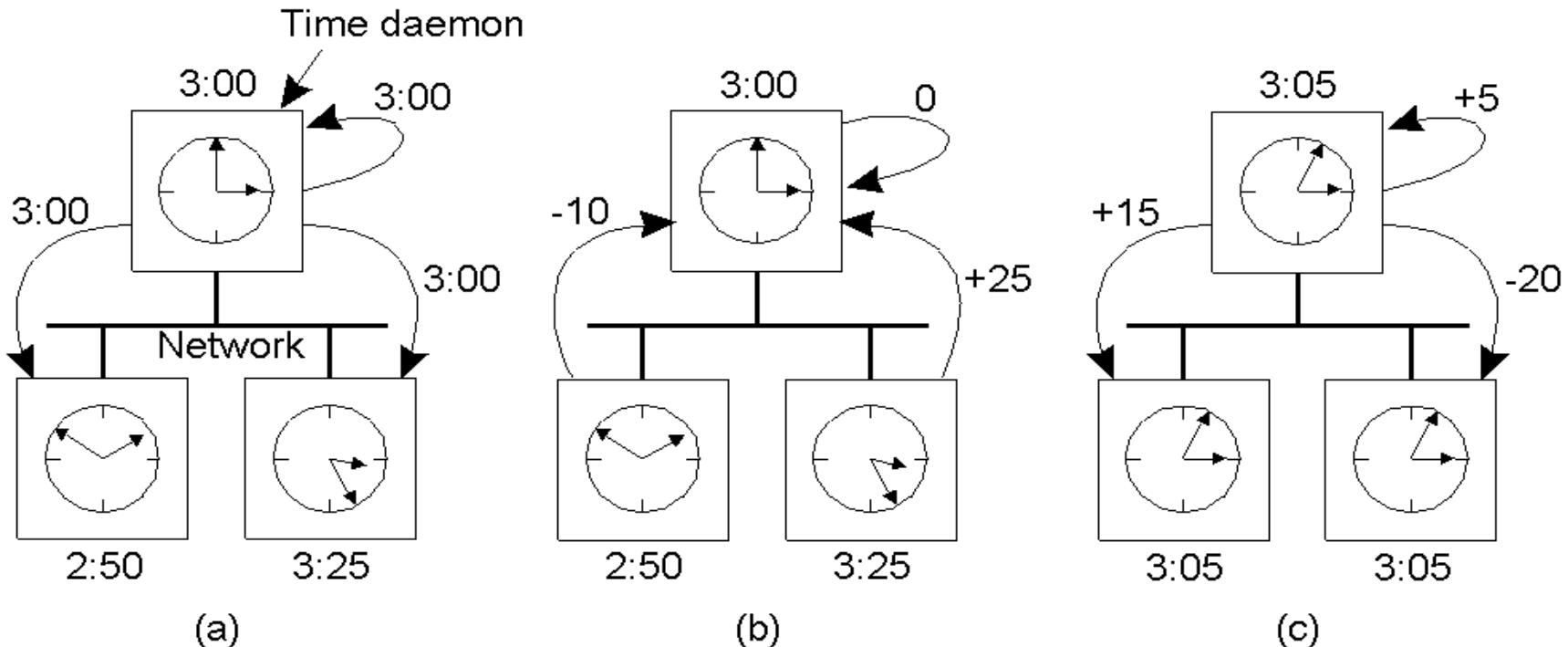


- Cristian's algorithm requires clients to periodically synchronize with a central time server (typically a server with a UTC receiver)
- Every computer periodically asks the “time server” for the current time
- The server responds ASAP with the current time C_{UTC}
- The client sets its clock to C_{UTC}

Berkeley Algorithm

- An algorithm for internal synchronization of a group of computers
- A *master* polls to collect clock values from the others (*slaves*)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average
- It sends the required adjustment to the slaves
- If master fails, can elect a new master to take over

The Berkeley Algorithm



- a) The time daemon asks all the other machines for their clock values
- b) The machines answer
- c) Takes an average & tells everyone how to adjust their clock

Berkeley Algorithm

Computers	Clock Reading
A (daemon)	3:00
B (left)	2:50
C (right)	3:25

Computers	Ahead/Behind
A (daemon)	0:00
B (left)	-0:10
C (right)	+0:25

Berkeley Algorithm

- Average time: 3:05

Computers	Needed Adjustment
A (daemon)	+0:05
B (left)	+0:15
C (right)	-0:20

LOGICAL CLOCKS

- **The relative ordering of Events is more important than actual physical time in many applications**
 - In a single process the ordering of events (e.g., state changes) is trivial
 - In a distributed system, however, besides local ordering of events, all processes must also agree on ordering of **causally related events** (e.g., sending and receiving of a single message)
- **Local ordering:**
 - System consists of N processes p_i
 - If p_i observes e before e' , we have $e \rightarrow e'$
- **Global ordering:**
 - Lamport's happened before relation \rightarrow
 - Smallest relation, such that
 1. $e \rightarrow e'$
 2. For every message m , send(m) \rightarrow receive(m)
 3. Transitivity: $e \rightarrow e'$ and $e' \rightarrow e''$ implies $e \rightarrow e''$

ORDERING EVENTS

- Event ordering linked with concept of *causality*:
 - Saying that event a happened before event b is same as saying that event a casually affects event b
 - If events a and b happen on processes that do not exchange any data, their exact ordering is not important

Relation “has happened before”

- Smallest relation satisfying the three conditions:
 - If a and b are events in the same process and a comes before b , then $a \rightarrow b$
 - If a is the sending of a message by a process and b its receipt by another process then
 $a \rightarrow b$
 - If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Relation “has happened before”

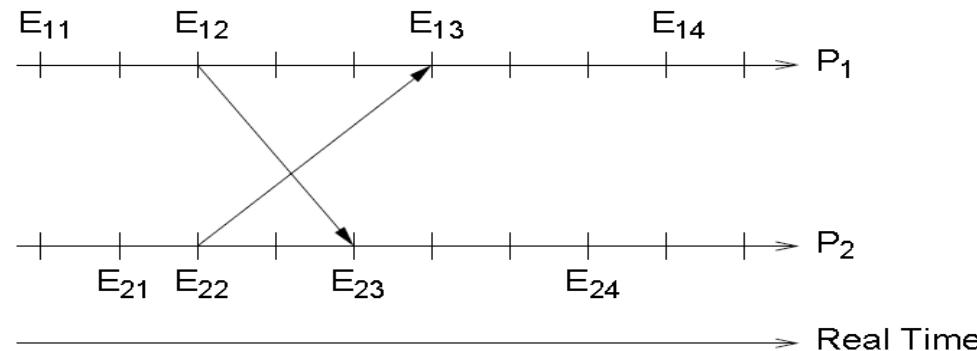
- We cannot always order events: relation “has happened” before is only a *preorder*
- If a did not happen before b , it cannot causally affect b .

LOGICAL CLOCKS

The relation \rightarrow is a partial order:

- If $a \rightarrow b$, then a causally affects b
- We consider unordered events to be concurrent:

Example: $a \not\rightarrow b$ and $b \not\rightarrow a$ implies $a \parallel b$



- Causally related: $E_{11} \rightarrow E_{12}, E_{13}, E_{14}, E_{23}, E_{24}, \dots$
 $E_{21} \rightarrow E_{22}, E_{23}, E_{24}, E_{13}, E_{14}, \dots$
- Concurrent: $E_{11} \parallel E_{21}, E_{12} \parallel E_{22}, E_{13} \parallel E_{23}, E_{11} \parallel E_{22}, E_{13} \parallel E_{24}, E_{14} \parallel E_{23}, \dots$

Lamport Algorithm

- Each process increments local clock between any two successive events
- Message contains a timestamp
- Upon receiving a message, if received timestamp is ahead, receiver fast forward its clock to be one more than sending time

Lamport Algorithm

Lamport's logical clocks:

- Software counter to locally compute the happened-before relation →
- Each process p_i maintains a **logical clock** L_i
- **Lamport timestamp:**
 - $L_i(e)$: timestamp of event e at p_i
 - $L(e)$: timestamp of event e at process it occurred at

Implementation:

- ① Before timestamping a local event p_i executes $L_i := L_i + 1$
- ② Whenever a message m is sent from p_i to p_j :
 - p_i executes $L_i := L_i + 1$ and sends L_i with m
 - p_j receives L_i with m and executes $L_j := \max(L_j, L_i) + 1$
($\text{receive}(m)$ is annotated with the new L_j)

Properties:

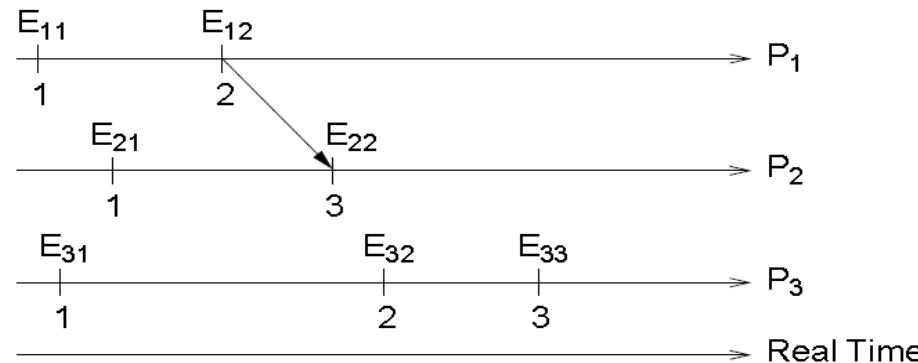
- $a \rightarrow b$ implies $L(a) < L(b)$
- $L(a) < L(b)$ does not necessarily imply $a \rightarrow b$

VECTOR CLOCKS

VECTOR CLOCKS

Main shortcoming of Lamport's clocks:

- $L(a) < L(b)$ does not imply $a \rightarrow b$
- We cannot deduce causal dependencies from time stamps:



- We have $L_1(E_{11}) < L_3(E_{33})$, but $E_{11} \not\rightarrow E_{33}$
- Why?
 - Clocks advance independently or via messages
 - There is no history as to where advances come from

VECTOR CLOCKS

Vector clocks:

- At each process, maintain a clock for every other process
- I.e., each clock V_i is a vector of size N
- $V_i[j]$ contains i 's knowledge about j 's clock

Implementation:

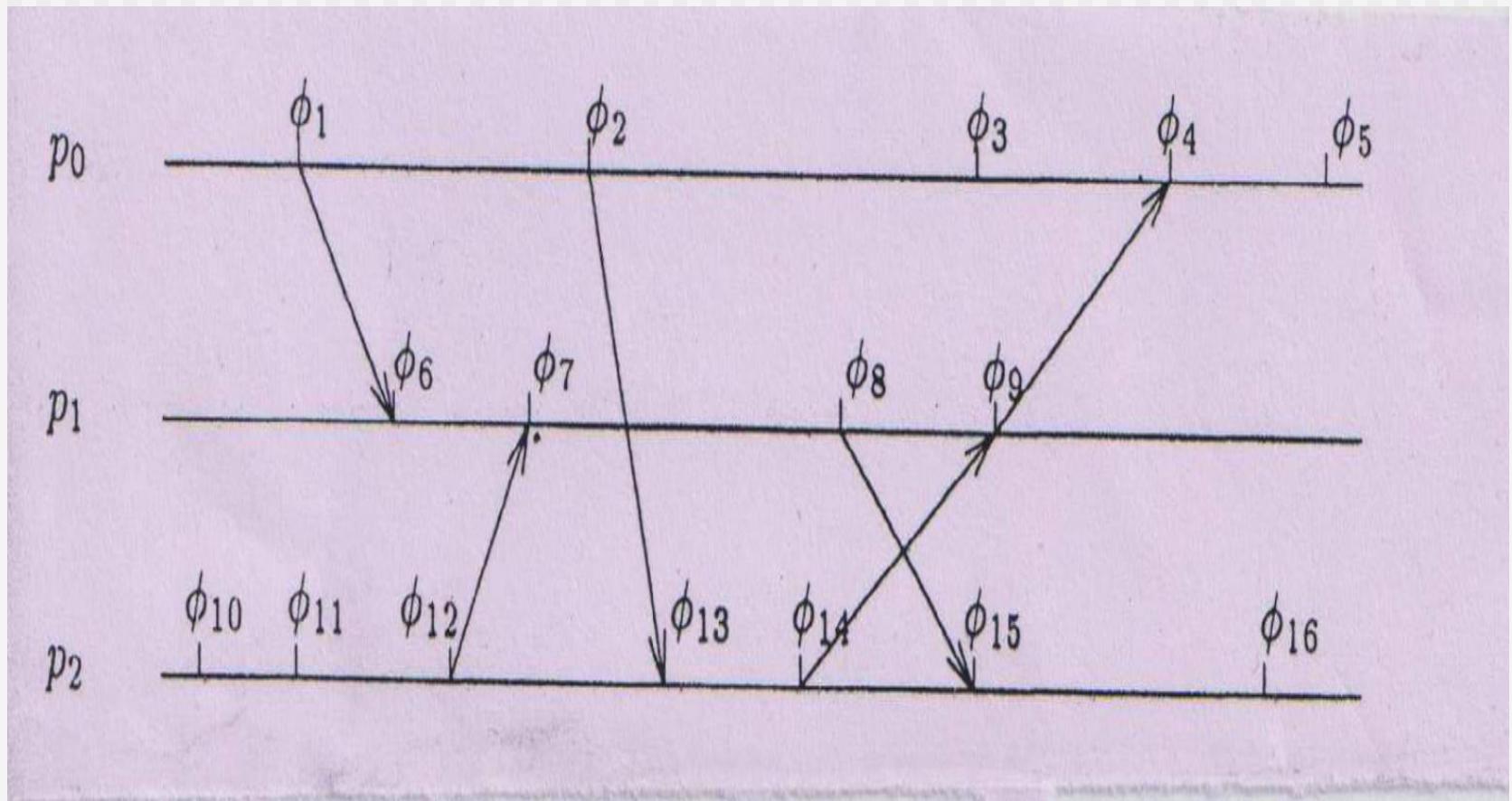
- ① Initially, $V_i[j] := 0$ for $i, j \in \{1, \dots, N\}$
- ② Before p_i timestamps an event, $V_i[i] := V_i[i] + 1$
- ③ Whenever a message m is sent from p_i to p_j :
 - p_i executes $V_i[i] := V_i[i] + 1$ and sends V_i with m
 - p_j receives V_i with m and merges the vector clocks V_i and V_j :

$$V_j[k] := \begin{cases} \max(V_j[k], V_i[k]) + 1 & , \text{if } j = k \\ \max(V_j[k], V_i[k]) & , \text{otherwise} \end{cases}$$

This last part ensures that everything that subsequently happens at p_j is now causally related to everything that previously happened at p_i .

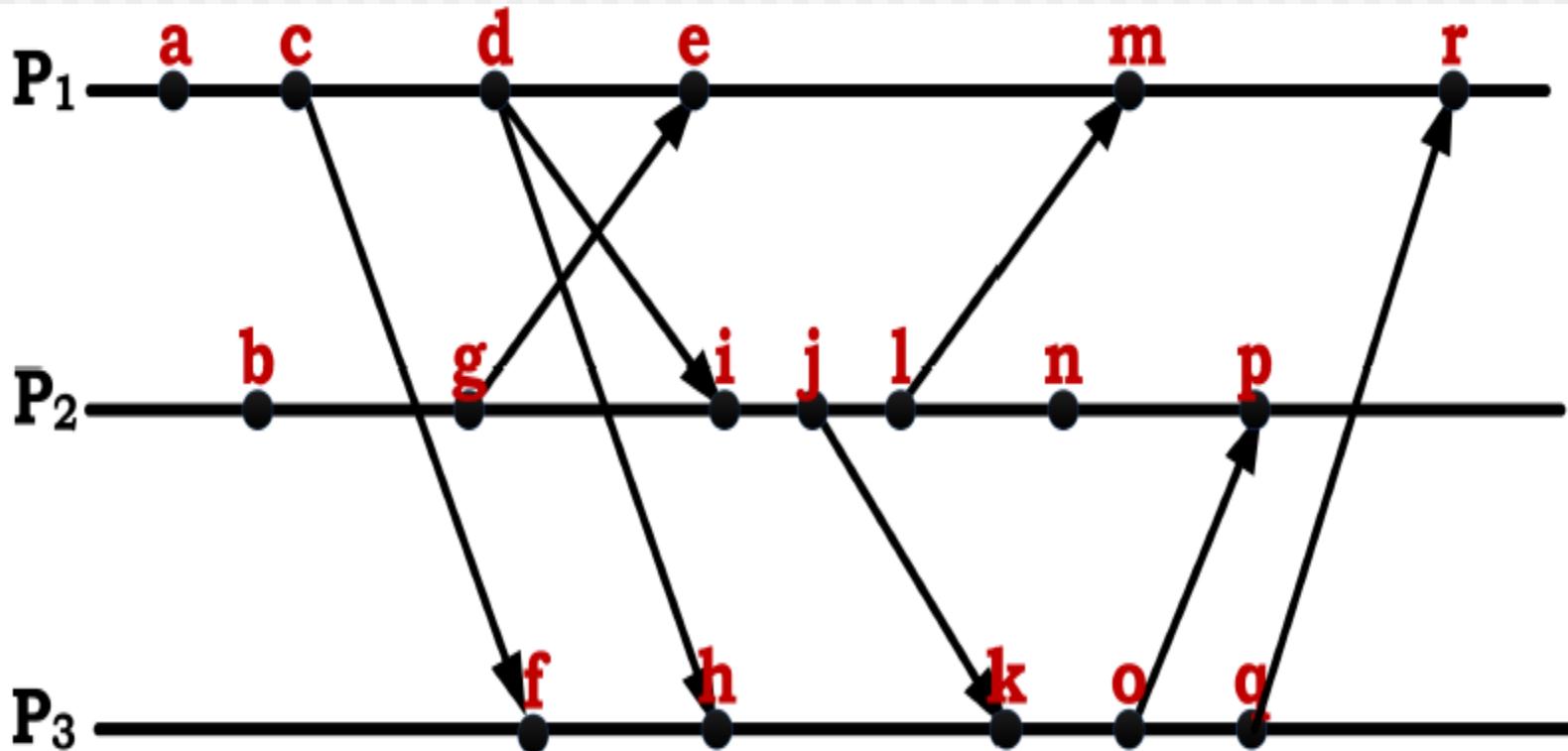
Tutorial- Lamport and Vector Clock

- For the figure below, write down the lamport clock and vector clock value at each event.



Tutorial- Lamport and Vector Clock

For the figure below, write down the lamport clock and vector clock value at each event.



GLOBAL STATE

Determining Global States

■ Global State

*“The global state of a distributed computation is the set of **local states** of all individual processes involved in the computation plus the state of the communication channels.”*

Determining Global States

- Knowing the global state of distributed system may be useful for many reasons.

For example:

- How to know that system is dealing with deadlock or distributed computation has correctly terminated?

GLOBAL STATE

Determining global properties: often difficult, but essential for some applications.

➤ **Distributed garbage collection:**

Do any references exist to a given object?: Collects remote server objects that are no longer referenced by any client in the network

.

➤ **Distributed deadlock detection:**

Do processes wait in a cycle for each other?

➤ **Distributed algorithm termination detection:**

Did a set of processes finish all activity? (Consider messages in transit)

All these properties are stable: once they occur, they do not disappear without outside intervention.

Analysis of GLOBAL STATE (Distributed Snapshot)

- A distributed snapshot reflects a state in which the distributed system might have been.
- **A snapshot reflects a consistent global state**
- **Example:** if we have recorded that a process Q has received a message from another process P,
- then we should also recorded that process P has actually sent that message
- Otherwise, a snapshot will contain the recording of messages that have been received but never sent,
- Which indicate the inconsistent global state
- The reverse condition (P has sent a message that Q has not yet received) is allowed

More on States

■ process state

- memory state + register state + open files + kernel buffers + ...

Or

- application specific info like transactions completed, functions executed etc.,

■ channel state

- “*Messages in transit*” i.e. those messages that have been sent but not yet received

Why global state determination is difficult in Distributed Systems?

- **Distributed State :**

Have to collect information that is spread across several machines!!

- **Only Local knowledge :**

A process in the computation does not know the state of other processes.

Difficulties due to Non Determinism

■ Deterministic Computation

- At **any point** in computation there is at most **one event** that can **happen next**.

■ Non-Deterministic Computation

- At **any point** in computation there can be **more than one event** that can **happen next**.

Deterministic Computation Example

A Variant of producer-consumer example

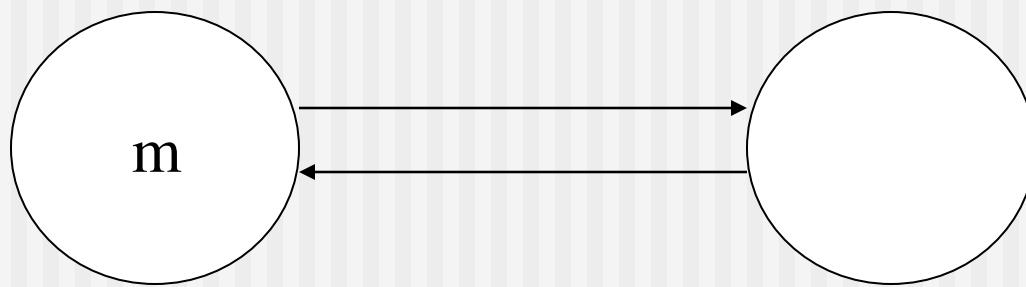
- Producer code:

```
while (1)
{
    produce m;
    send m;
    wait for ack;
}
```

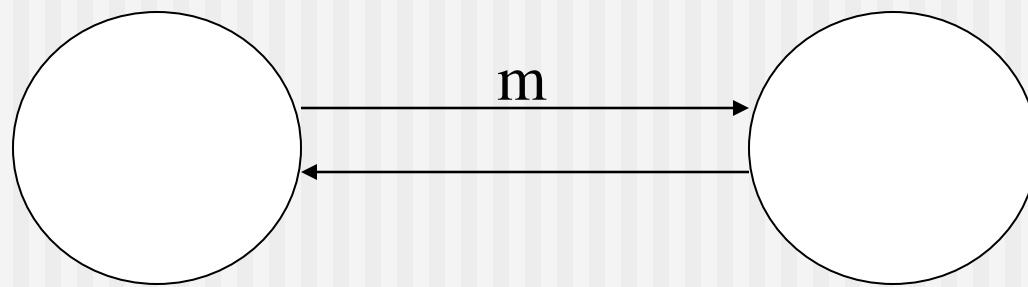
- Consumer code:

```
while (1)
{
    recv m;
    consume m;
    send ack;
}
```

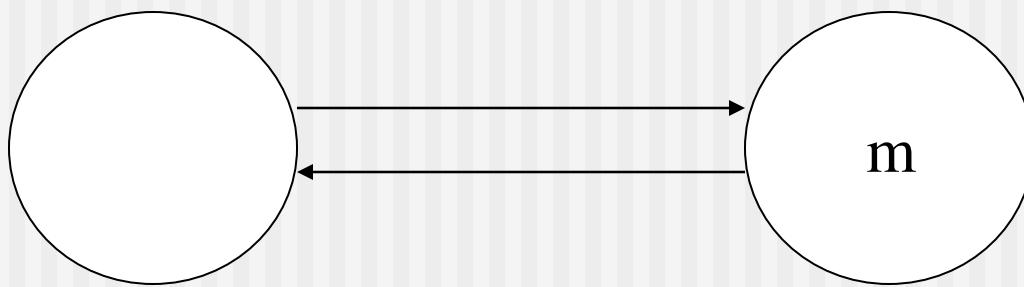
Example: Initial State



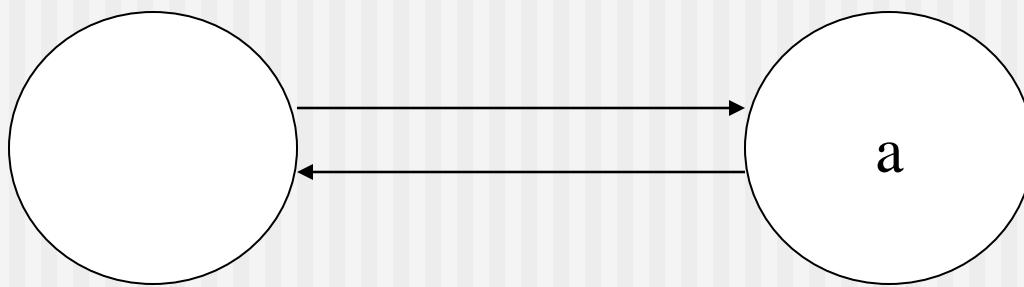
Example



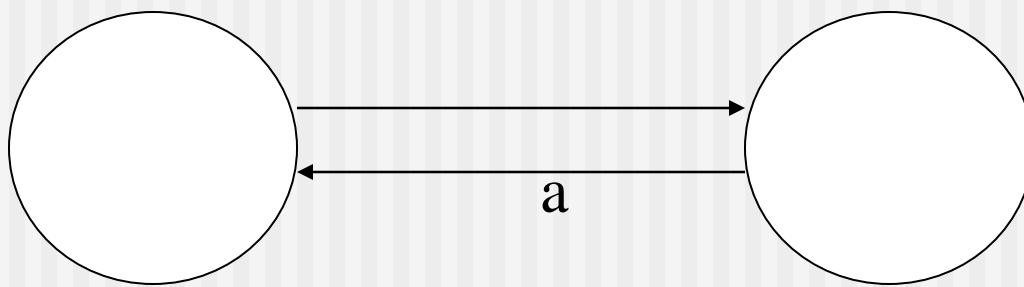
Example



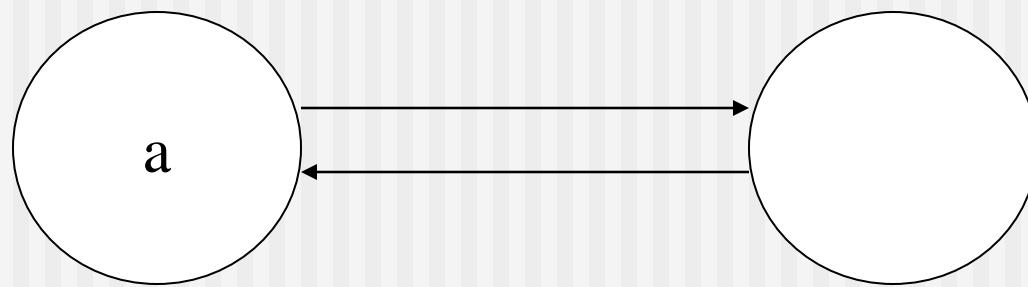
Example



Example



Example

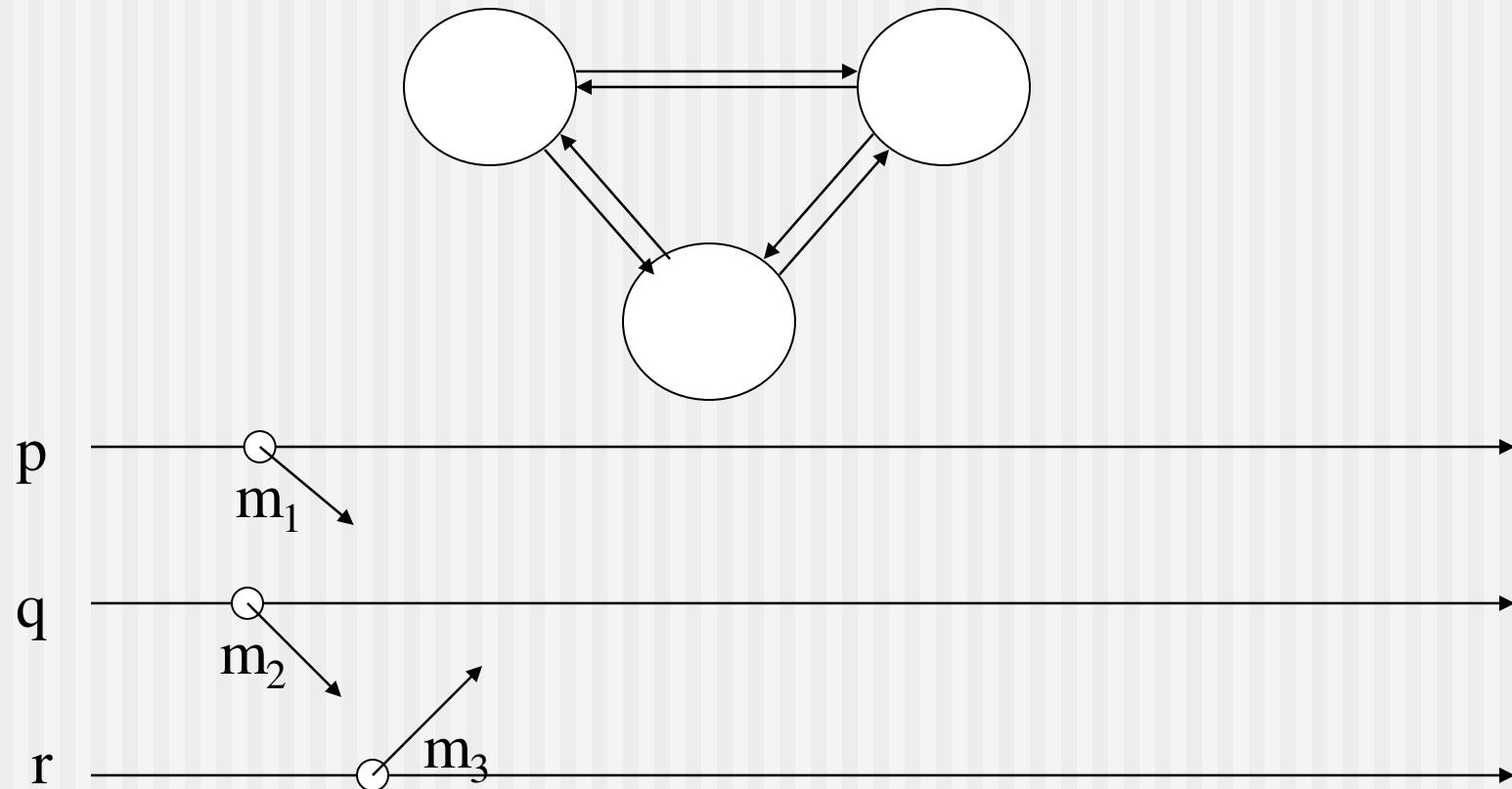


Deterministic state diagram



Non-deterministic computation

3 processes



Election Algorithms

- In distributed computing, **leader election** is the process of designating a single process as the organizer of some task distributed among several computers (nodes).
- Before the task is begun, all network nodes are unaware which node will serve as the "leader," or coordinator, of the task.
- After a leader election algorithm has been run, however, each node throughout the network recognizes a particular, unique node as the task leader.

Election Algorithms

- Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform special role.
- All processes in distributed systems may be equal characteristics, there is no way to select one of them to be special.
 - Assume have some “ID” that is a number, that every process knows the process number of every other process.
- “elect” process with the highest number as leader

Election Algorithms

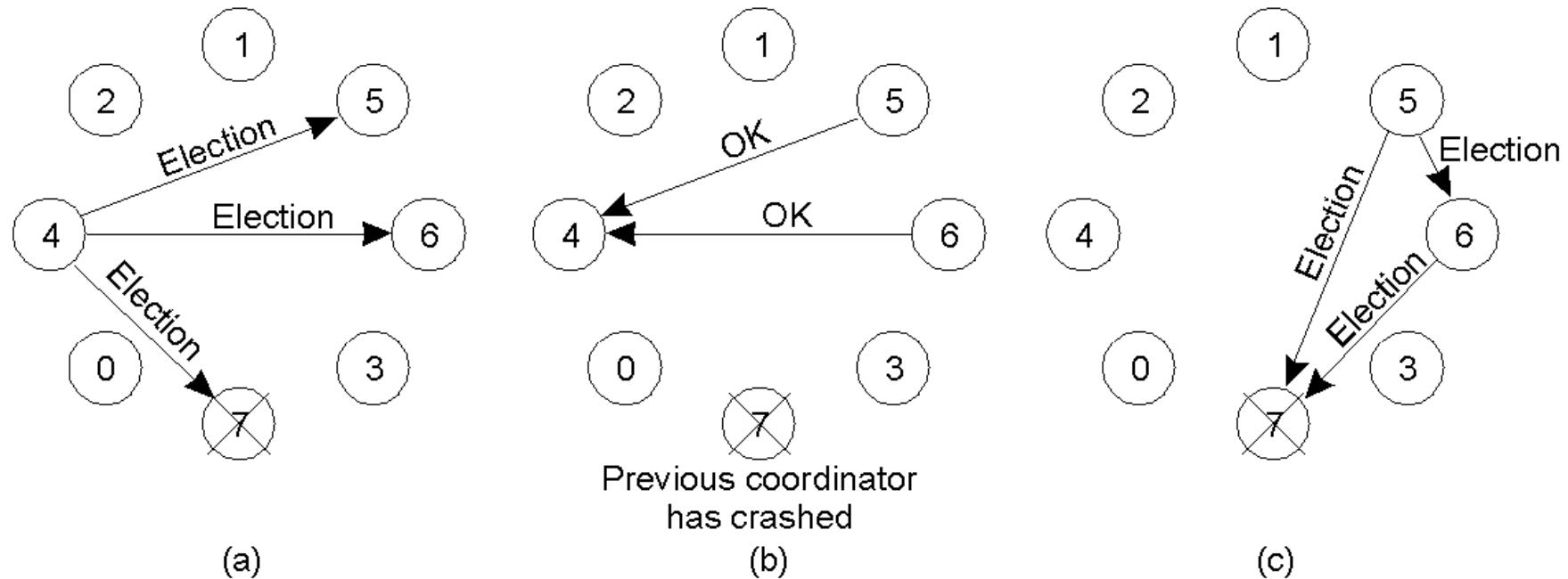
■ Election

- **Goal:** ensure that the election achieves an agreement among all the processes
- the P with the highest id is elected

The Bully Algorithm

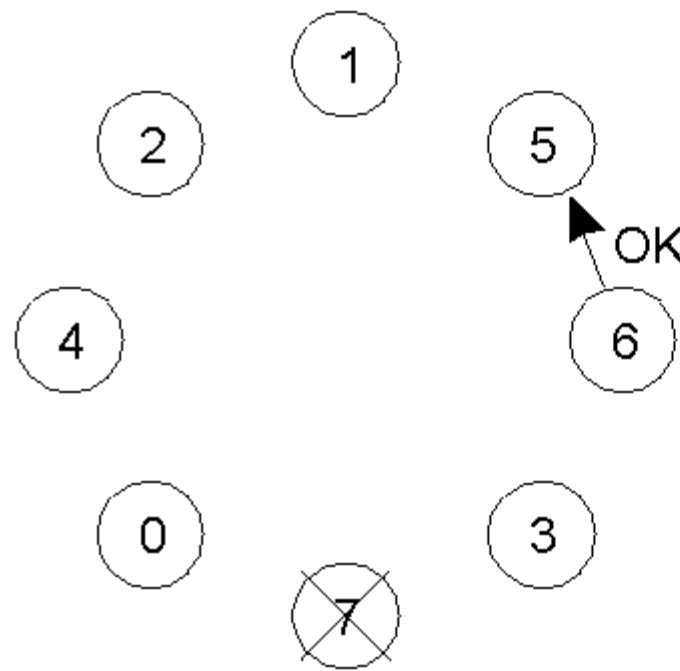
- When a process notices that the coordinator is no longer responding to requests, it **initiates an election**.
- A process ,P, holds an election as follows
 1. P sends an ELECTION message to all processes with higher numbers.
 2. If no one responds, P wins the election and becomes coordinator.
 3. If one of the higher-ups answers, it takes over. P's job is done
 4. Finally only a P (the new coordinator) will remain and it will inform the other by sending a msg
 5. If a process is restarted, the first action is to trigger an election

The Bully Algorithm (1)

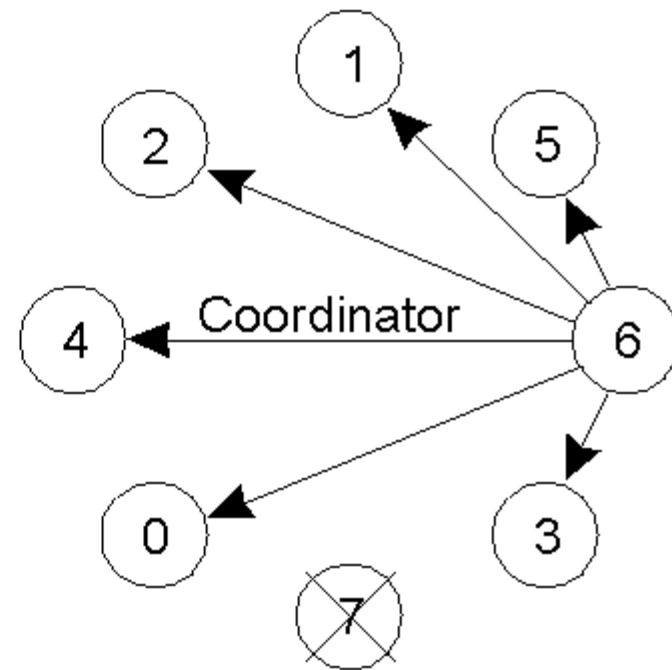


- Process 4 notices 7 down
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Algorithm (2)



(d)



(e)

- d) Process 6 tells process 5 to stop
- e) Process 6 wins and tells everyone
 - Eventually “biggest” (bully) wins
 - If processes 7 comes up, starts elections again

A Ring Algorithm

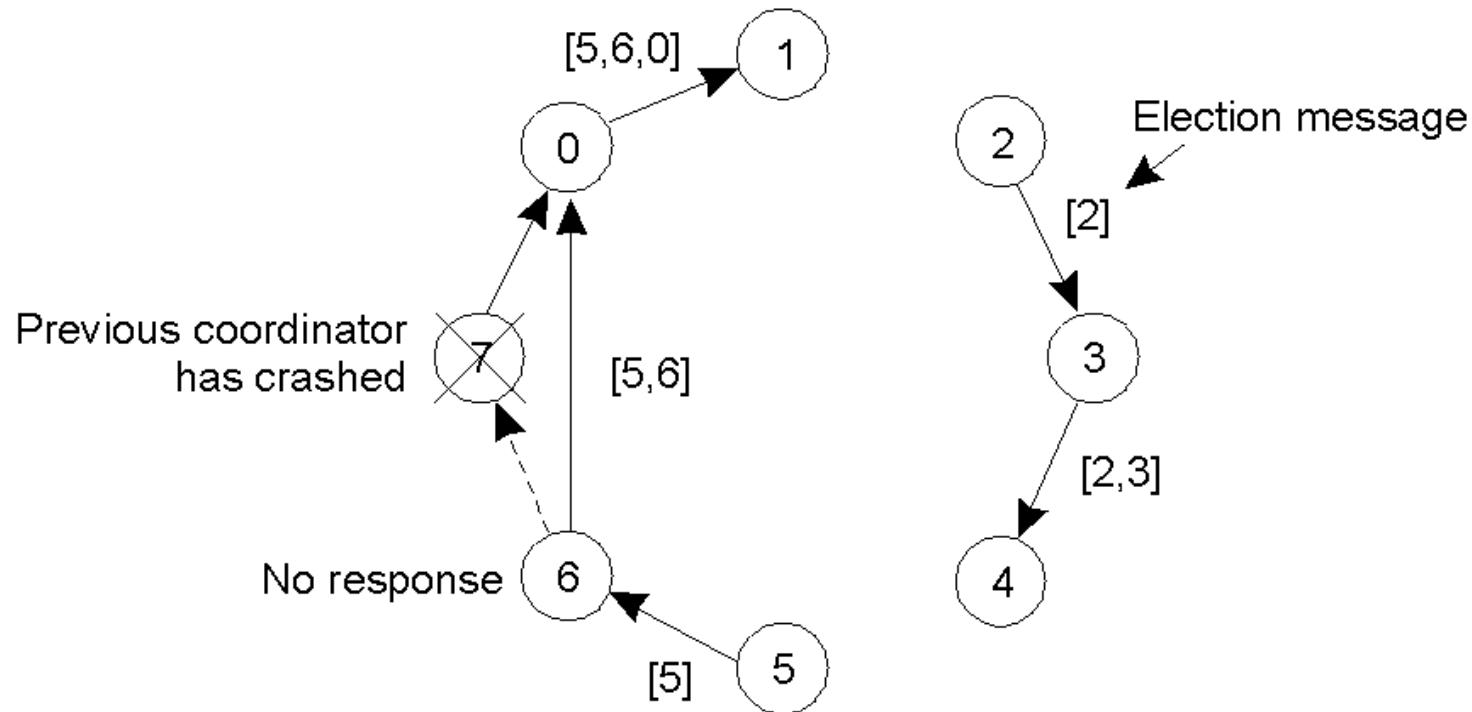
- processes are ordered in such way that each process knows who its successor is.
- When a P suspects a coordinator fault
 - Sends an ELECTION msg containing its Pid to the next (successor); if it is down, the sender skips over the successor & the msg is sent to the next of the ring
 - Each member of the network receives and propagates a msg to the next, **adding its id**

A Ring Algorithm

- When a msg return to a P who sent it (verify by inspecting the list), the msg is turned into COORDINATOR and circulated, to report:
 - New coordinator: P of the list with highest id
 - Multiple messages can circulate over the network

A Ring Algorithm

- Once around, change to COORDINATOR (biggest)



- Even if two ELECTIONS started at once, everyone will pick same leader

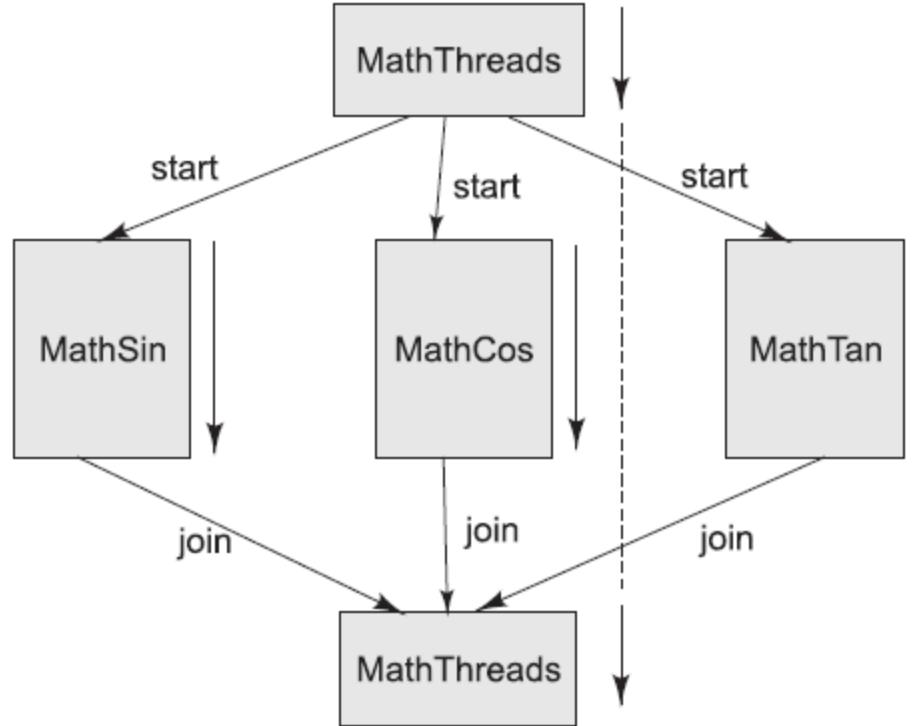
CONCURRENCY

Concurrency in a Non-Distributed System:

- Concurrency in distributed systems are familiar from the study of OS and multithreaded programming problems
- Prevent race conditions (race conditions that occur when concurrent processes access shared resources)
- Critical sections
- In non-distributed system above problems are solved by implementing mutual exclusion using local primitives such as
 - Locks
 - Semaphores

A PROGRAM WITH MULTIPLE THREADS

- To illustrate creation of multiple threads in a program performing concurrent operations, let us consider the processing of the following mathematical equation: $p = \sin(x) + \cos(y) + \tan(z)$



■ Concurrency in a Distributed System:

Distributed System introduces more challenges of concurrency due to

- No directly shared resources (e.g., memory)
- No global state
- No global clock
- No centralized algorithms
- presence of communication delays

DISTRIBUTED MUTUAL EXCLUSION

- Concurrent access to distributed resources
- Must prevent race conditions during critical regions

Requirements:

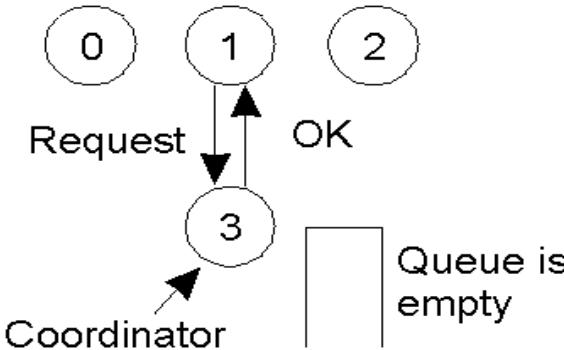
1. **Safety:** At most one process may execute the critical section at a time
2. **Liveness:** Requests to enter and exit the critical section ultimately succeed
3. **Ordering:** Requests are processed in happened-before ordering

METHOD 1: CENTRAL SERVER

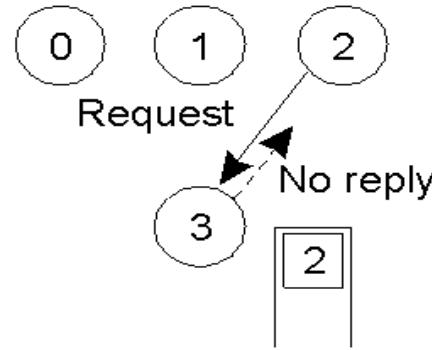
Simplest approach: use a central server that controls the entering and exiting of critical sections.

- Requests to enter and exit a critical section are sent to a lock server (or coordinator)
- Permission to enter is granted by receiving a token
- When critical section left, token is returned to the server

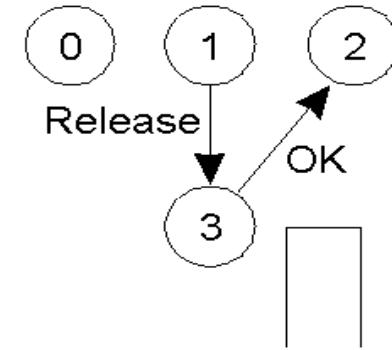
A Centralized Algorithm



(a)



(b)



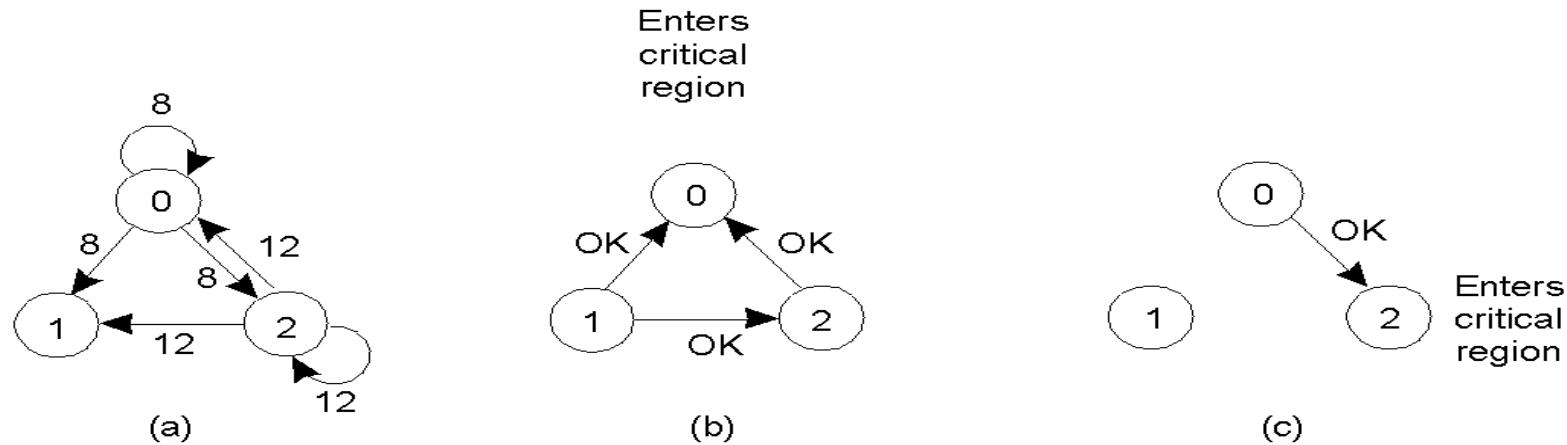
(c)

- a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
- b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (Or, can say “denied”)
- c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2.

Cont...

- This scheme is easy to implement, but it does not scale well due to the central authority.
- Moreover, it is vulnerable to failure of the central server.

A Distributed Algorithm

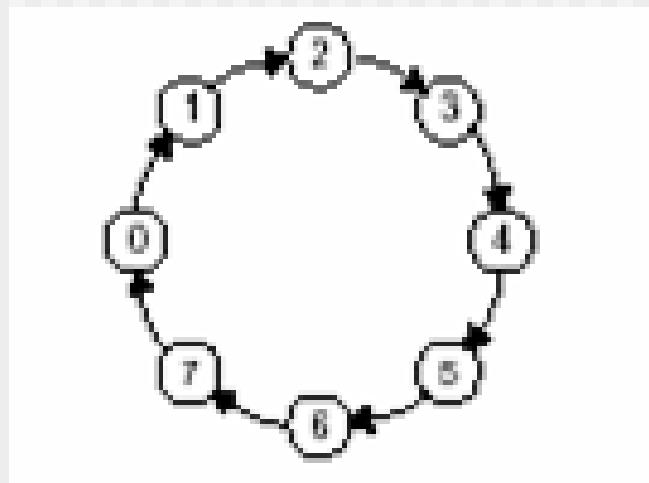


- a) Processes 0 and 2 want to enter the same critical region at the same moment.
- b) Process 1 doesn't want to enter into critical region, says “OK”. Process 0 has the lowest timestamp, so it wins. Queues up “OK” for 2.
- c) When process 0 is done, it sends an OK to 2 so can now enter the critical region.

METHOD 2: TOKEN RING

Implementation:

- All processes are organised in a logical ring structure
- A token message is forwarded along the ring
- Before entering the critical section, a process has to wait until the token comes
- Must retain the token until the critical section is left



Properties:

- Ring limits scalability
- Token messages consume bandwidth
- Failing nodes or channels can break the ring (token might be lost)

METHOD 3: USING MULTICASTS AND LOGICAL CLOCKS

- **Algorithm by Ricart & Agrawala:** proposed an algorithm for distributed mutual execution that makes use of logical clocks.
- Processes p_i maintain a Lamport clock and all process must be able to communicate pairwise
- At any moment, each process is in one of three states:
 1. **Released:** Outside of critical section
 2. **Wanted:** Waiting to enter critical section
 3. **Held:** Inside critical section

■ Process behaviour:

1. If a process wants to enter a critical section, it
 - multicasts a message(L_i , p_i) and
 - waits until it has received a reply from every process
2. If a process is in Released , it immediately replies to any request to enter the critical section
3. If a process is in Held , it delays replying until it is finished with the critical section
4. If a process is in Wanted , it replies to a request immediately only if the **requesting timestamp** is smaller than the one in its **own request**

Properties:

- Better scalability, but multicast leads to increasing overhead.
- Unfortunately, failure of any peer process can deny all other process entry to the critical section.

MUTUAL EXCLUSION: A COMPARISON

Messages Exchanged : Messages per entry/exit of critical section

- o Centralised: 3 (two to enter and one to leave)
- o Ring: $1 \rightarrow \infty$
- o Multicast: $2(n-1)$

Delay: Delay before entering critical section

- o Centralised: 2
- o Ring: $0 \rightarrow n-1$
- o Multicast: $2(n-1)$

Reliability:

- o Centralised: coordinator crashes
- o Ring: lost token, process crashes
- o Multicast: any process crashes

DISTRIBUTED TRANSACTIONS

TRANSACTIONS

- Transaction:
 - Comes from database world: the concept of a transaction originates from the database community as a mechanism to maintain the consistency of databases.
 - Defines a sequence of operations
 - Atomic in presence of multiple clients and failures
- Mutual Exclusion :
 - Protect shared data against simultaneous access
 - Allow multiple data items to be modified in single atomic action
- Transaction Model:
 - Operations:
 - Begin Transaction
 - End Transaction
 - Read
 - Write
 - End of Transaction:
 - Commit
 - Abort

ACID PROPERTIES

- **atomic:** must follow an "**all or nothing**" rule. Each transaction is said to be atomic if when one part of the transaction fails, the entire transaction fails and database state is left unchanged.
- **consistent:** concurrent transactions will not produce inconsistent results;
- **isolated:** transactions do not interfere with each other i.e. no intermediate state of a transaction is visible outside , it refers to the requirement that other operations cannot access data that has been modified during a transaction that has not yet completed.
- **durable:** after a commit, results are permanent (even if server or hardware fails)

CLASSIFICATION OF TRANSACTIONS

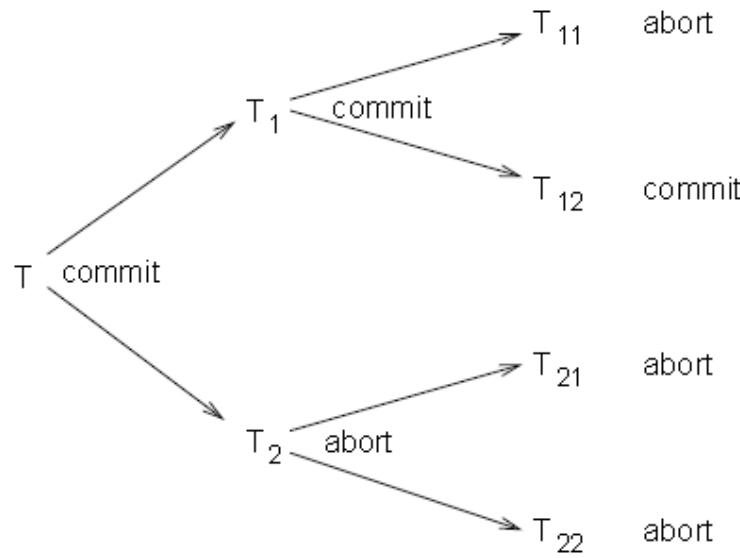
- **Flat:** sequence of operations that satisfies ACID
- **Nested:** *hierarchy* of transactions
- **Distributed:** (flat) transaction that is executed on distributed data
- Flat Transactions:
 - Simple
 - Failure → all changes undone

```
BeginTransaction
accountA -= 100;
accountB += 50;
accountC += 25;
accountD += 25;
EndTransaction
```

NESTED TRANSACTION

- Subdivide a complex transaction into many smaller *sub-transactions*
- As a result, if a single *sub-transaction* fails, the work that went into others is not necessarily wasted
- Example:
 - Booking a flight
 - Sydney → Manila
 - Manila → Amsterdam
 - Amsterdam → Toronto
- What to do?
 - Abort whole transaction
 - Partially commit transaction and try alternative for aborted part
 - Commit nonaborted parts of transaction

Cont...



- ❑ *Subtransactions* and parent transactions
- ❑ Parent transaction may commit even if some subtransactions aborted
- ❑ Parent transaction aborts → all subtransactions abort

Cont...

■ Subtransactions:

- Subtransaction can abort any time
- Subtransaction cannot commit until parent ready to commit
 - Subtransaction either aborts or commits provisionally
 - Provisionally committed subtransaction reports provisional commit list, containing all its provisionally committed subtransactions, to parent
 - On abort, all subtransactions in that list are aborted.

TRANSACTION IMPLEMENTATION

■ Private Workspace:

- Perform all *tentative* operations on a *shadow copy* of the server state
- Atomically swap with main copy on Commit
- Discard shadow on Abort.

Cont...

- Writeahead Log:

- In-place update with writeahead logging
 - Log is reverted when a transaction Aborts

CONCURRENCY CONTROL

- It is often necessary to allow transactions to occur simultaneously (for example, to allow multiple travel agents to simultaneously reserve seats on the same flight).
- Due to the consistency and isolation properties of transactions, concurrent transaction must not be allowed to interfere with each other.
- Concurrency control algorithms for transactions guarantee that multiple transactions can be executed simultaneously while providing a result that is the same as if they were executed one after another.

CONCURRENCY CONTROL

- **Simultaneous Transactions:**

- Clients accessing bank accounts
- Travel agents booking flights

- **Problems:**

- Simultaneous transactions may interfere
 - Lost update
 - Inconsistent retrieval
- **Consistency and Isolation** require that there is no interference

- **Concurrency Control Algorithms:**

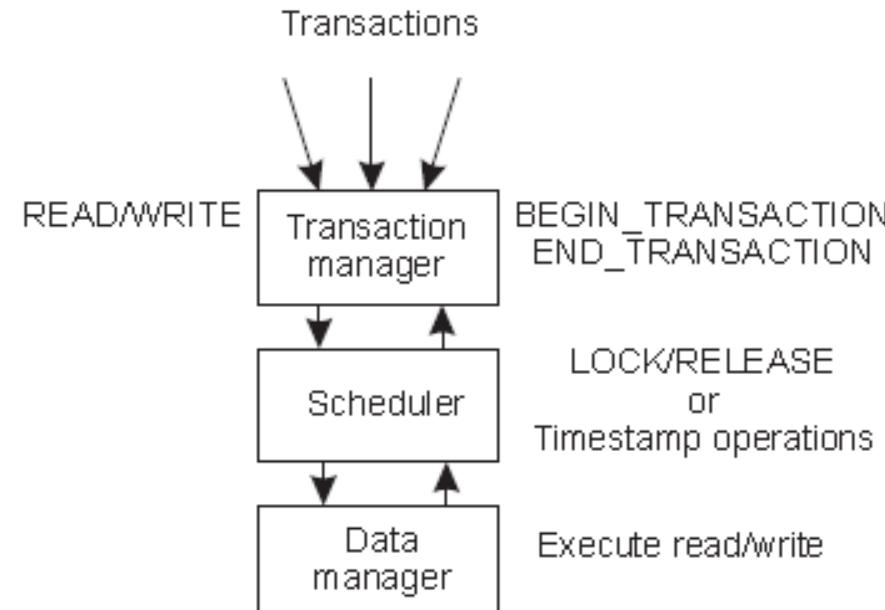
- Guarantee that multiple transactions can be executed simultaneously while still being isolated.
- As though transactions executed one after another

CONFLICTS AND SERIALISABILITY

- **conflict:** operations (from the same, or different transactions) that operate on same data
- **read-write conflict:** one of the operations is a write
- **write-write conflict:** more than one operation is a write
- **Define a *Schedule* of operations:**
 - Total ordering (interleaving) of operations
 - Legal schedules provide results as though the transactions were serialised (i.e., performed one after another) (*serial equivalence*)

MANAGING CONCURRENCY

■ Transaction Managers:



Dealing with Concurrency:

- Locking
- Timestamp Ordering
- Optimistic Control

LOCKING

- Pessimistic approach: prevent illegal schedules
 - The locking algorithms require that **each transaction obtains a lock** from a scheduler process before performing a read or a write operation.
 - The scheduler is responsible for granting and **releasing locks in such a way that legal schedules are produced**.
 - Ensures that only valid schedules result

TWO PHASE LOCKING (2PL)

- The most widely used locking approach is two-phase locking (2PL).
- In this approach a lock for a data item is granted to a process if no conflicting locks are held by other processes (otherwise the process requesting the lock blocks until the lock is available again).
- Lock is not released until operation executed by data manager
- No more locks granted after a release has taken place

All schedules formed using 2PL are serialisable.

PROBLEMS WITH LOCKING

- Deadlock
- Cascaded Aborts: If a transaction (T1) reads the results of a write of another transaction (T2) that is subsequently aborted, then the first transaction (T1) will also have to be aborted.

TIMESTAMP ORDERING

- Each transaction has unique timestamp ($ts(Ti)$)
- Each operation (TS(W), TS(R)) receives its transaction's timestamp
- Each data item has two timestamps:
 - read timestamp: $ts_{RD}(x)$ - the timestamp of the last read x
 - write timestamp: $ts_{WR}(x)$ - the timestamp of the last committed write x
- Timestamp ordering rule:
 - write request only valid if $TS(W) > ts_{WR}$ and $TS(W) \geq ts_{RD}$
 - read request only valid if $TS(R) > ts_{WR}$
- **Conflict resolution:**
 - Operation with lower timestamp executed first

OPTIMISTIC CONTROL

- Assume that no conflicts will occur.
 - Detect & resolve conflicts at commit time
 - A transaction is split into three phases:
 - Working (using shadow copies)
 - Validation
 - Update
 - In the **working phase** operations are carried out on shadow copies with no attempt to detect or order conflicting operations.

Cont....

- In the **validation phase** the scheduler attempts to detect conflicts with other transactions that were in progress during the working phase.
- If conflicts are detected then one of the conflicting transactions are aborted.

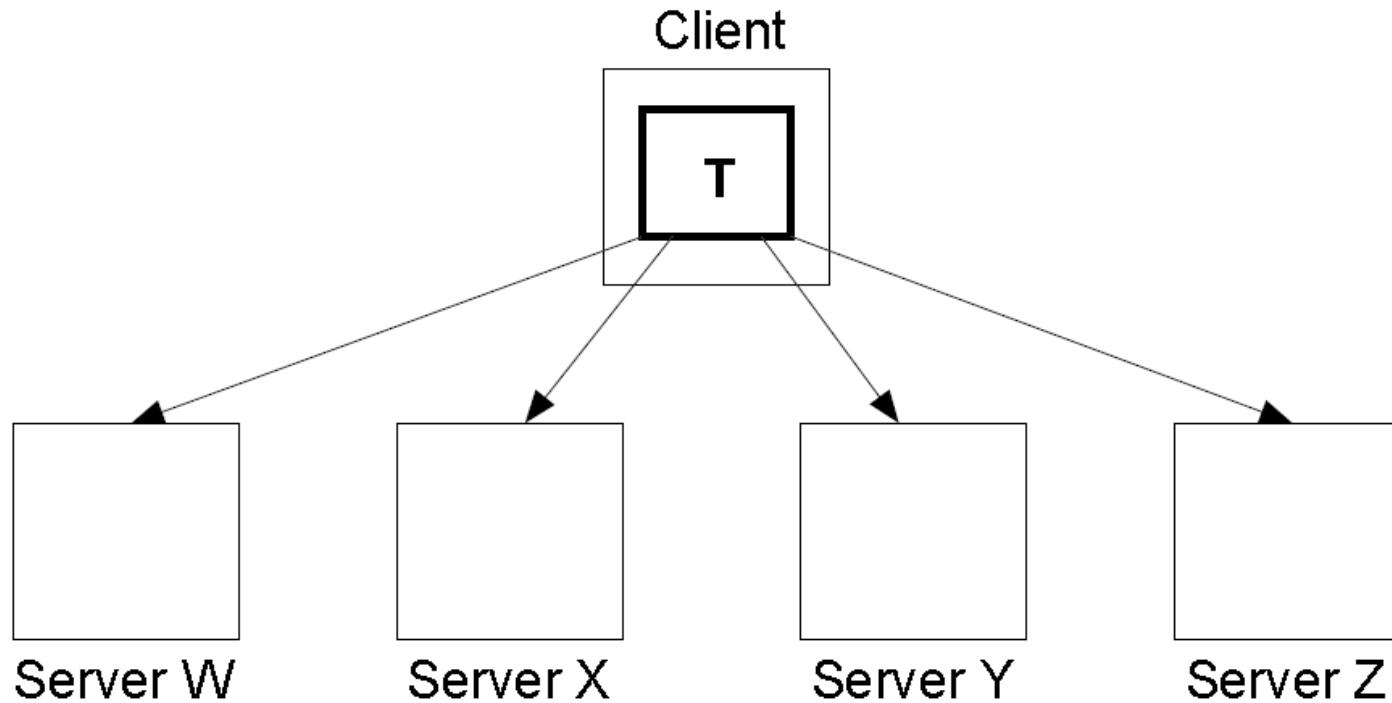
Cont...

- In the **update phase**, assuming that the transaction was not aborted, all the updates made on the shadow copy are made permanent.

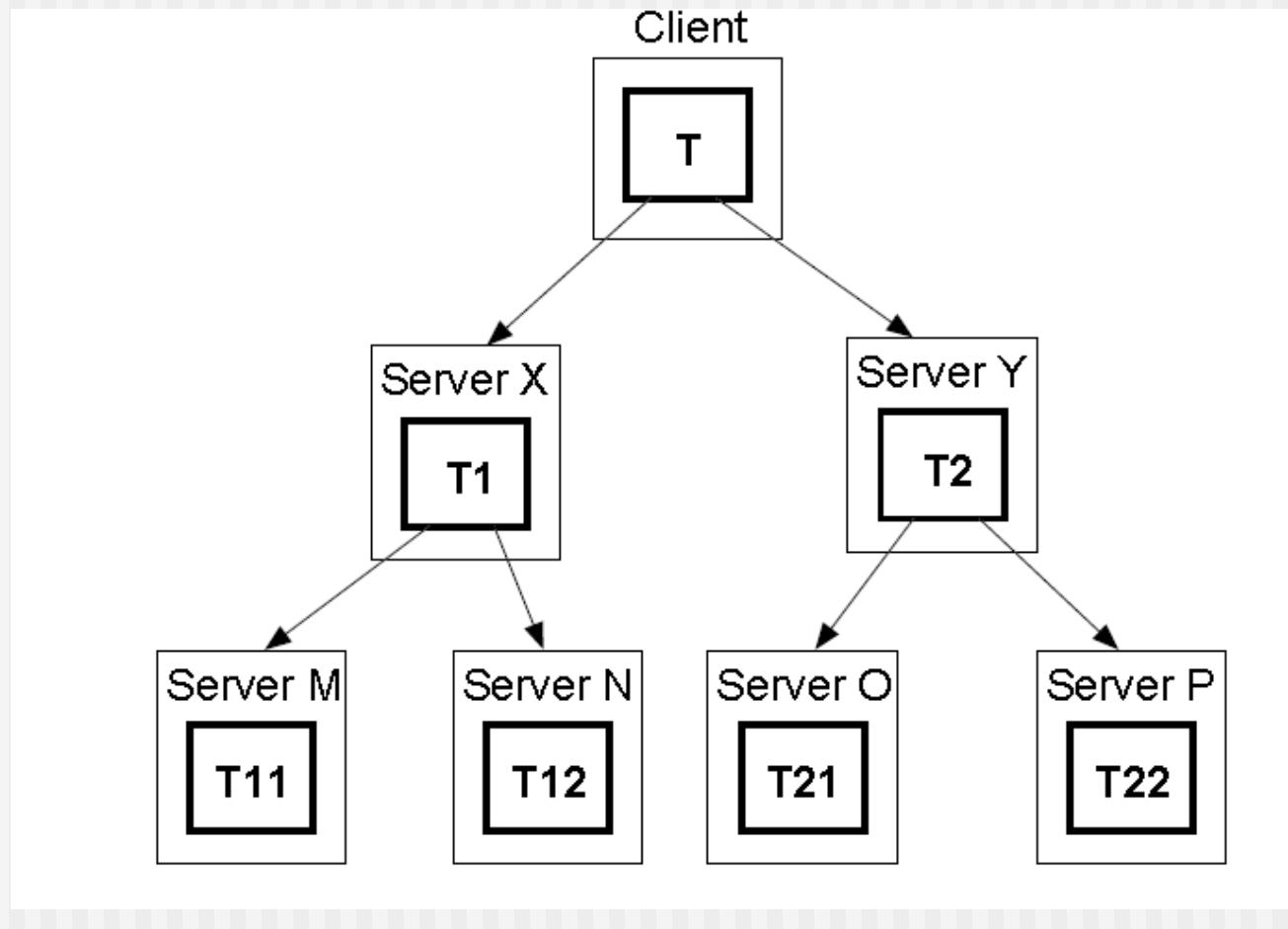
DISTRIBUTED TRANSACTIONS

- In distributed system, a single transaction will, in general, involve several servers:
 - transaction may require several services,
 - transaction involves files stored on different servers
- All servers must agree to *Commit* or *Abort*, and do this atomically.

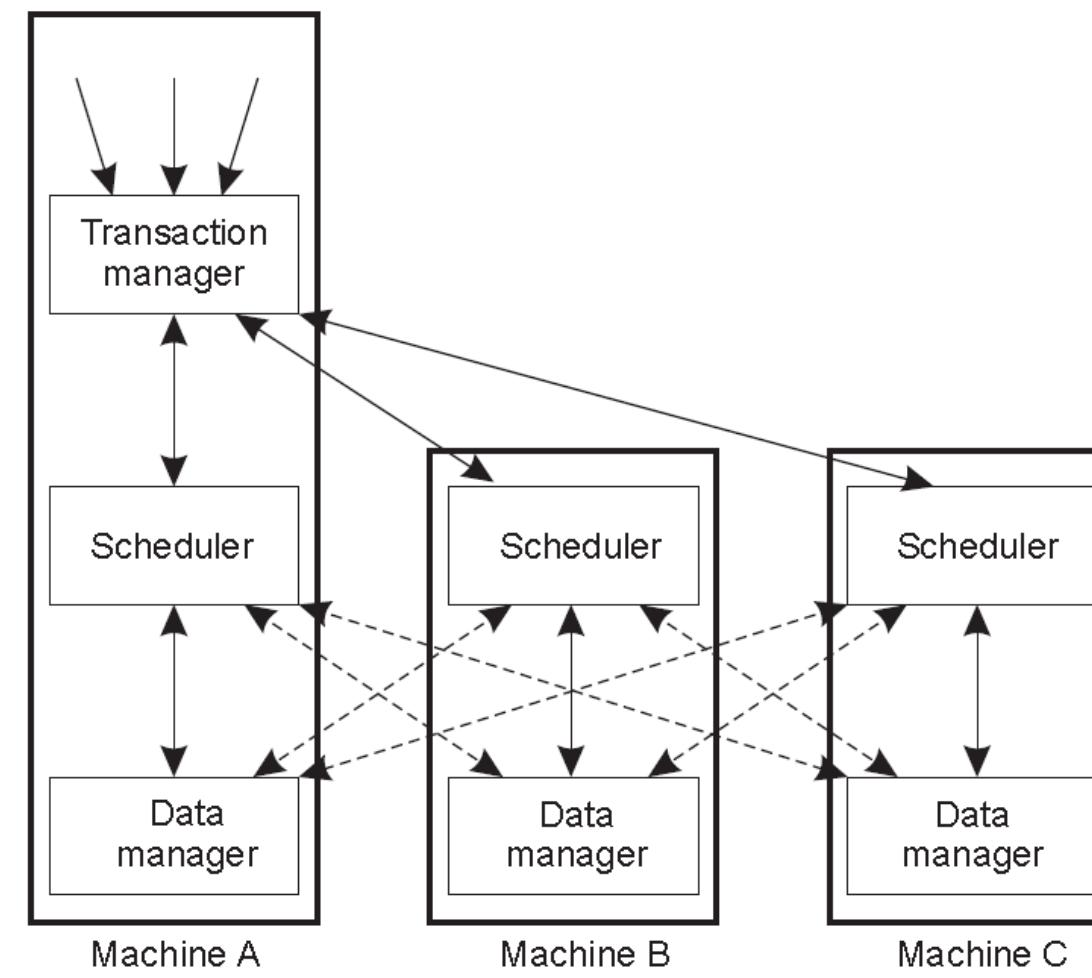
Distributed Flat Transaction:



Distributed Nested Transaction:



DISTRIBUTED CONCURRENCY CONTROL



DISTRIBUTED LOCKING

■ Distributed 2PL:

- Data can be replicated
- Scheduler on each machine responsible for locking own data
- Read lock: contact any replica
- Write lock: contact all replicas

ATOMICITY AND DISTRIBUTED TRANSACTIONS

■ **Distributed Transaction Organisation:**

- Each distributed transaction has a coordinator, the server handling the initial BeginTransaction call
- Coordinator maintains a list of workers, i.e. other servers involved in the transaction
- Each worker needs to know coordinator
- Coordinator is responsible for ensuring that whole transaction is atomically committed or aborted
 - Require a distributed commit protocol.

DISTRIBUTED ATOMIC COMMIT

- Transaction may only be able to commit when all workers are ready to commit (e.g. validation in optimistic concurrency)
- Distributed commit requires at least two phases:
 1. **Voting phase:** all workers vote on commit, coordinator then decides whether to commit or abort.
 2. **Completion phase:** all workers commit or abort according to decision.

Basic protocol is called two-phase commit (2PC)

Two-phase commit: Coordinator

1. Sends CanCommit, receives yes, abort;
2. Sends DoCommit, DoAbort

If yes messages are received from all workers, the coordinator sends a **DoCommit message** to all workers, which then implement the Commit. However, if any of the workers replies with **abort** instead of yes, the coordinator sends a DoAbort message to all workers to trigger a collective Abort.

Two-phase commit: Worker

1. Receives CanCommit, sends yes, abort;
2. Receives DoCommit, DoAbort

A worker on receiving a **CanCommit message** answers with either a **yes** or **abort** message, depending on the workers internal state. Afterward, it commits or aborts depending on the instructions from the coordinator.

The only extra case is where the worker is in an **abort cycle** and immediately answers a **CanCommit** with an **abort**.

■ Failures can be due to:

■ server failures/hosts:

- If a host fails in the 2PC protocol, then, after being restarted, it aborts all transactions.

restarting worker aborts all transactions

■ Failure of communication channels/network:

- coordinator aborts after timeout.

Two-phase commit with timeouts: Worker

- On timeout sends GetDecision
- Whenever a worker that is ready to commit when receiving a CanCommit message times out while waiting for the decision from the coordinator, it sends a special GetDecision message that triggers the coordinator to resend the decision on whether to Commit or Abort.
- These messages are handled by the coordinator once the decision between committing and aborting has been made. Moreover, the coordinator resends CanCommit messages if a worker does not issue a timely reply.

Two-phase commit with timeouts: Coordinator

- On timeout re-sends CanCommit, On GetDecision repeats decision

Limitations(2PC)

- Once node voted “yes”, cannot change its mind, even if crashes.
 - Atomic state update to ensure “yes” vote is stable.
- If coordinator crashes, all workers may be blocked.
 - Can use different protocols (e.g. three-phase commit),
 - in some circumstances workers can obtain result from other workers.

SUMMARY

- **Distributed Algorithms:**

- Timing models

- **Time:**

- Clock synchronisation
 - Logical clocks
 - Vector clocks

- **Concurrency Control:**

- Distributed mutual exclusion

- **Distributed transactions**

- **2PC**

Distributed Deadlock Detection

References:

- Mukesh Singhal, Niranjan Shivaratri. “Advanced concepts in operating systems”. Tata McGraw Hill publication.

Introduction

- **Deadlock:** a situation where a process or set of the process is blocked, waiting on an event that will never occur.
- Avoiding performance degradation due to deadlocks requires that a **system be deadlock free** or that **deadlocks be quickly detected and eliminated**.
- Deadlock in distributed systems are similar to deadlocks in single processor systems, only worse.
 - They are harder to avoid ,prevent or even detect.
 - They are hard to cure because all relevant information is scattered over many machines.
- In Distributed Systems, a process can request and release resources in any order
- If the sequence of allocation is not controlled, deadlocks can occur

Assumptions:

- System has only reusable resources
- Only exclusive access to resources
- Only one copy of each resource
- States of a process: running or blocked
 - Running state: process has all the resources
 - Blocked state: waiting on one or more resource

Types of Resources

- Reusable resource:
 - Description:
 - Used by one process at a time and not consumed by that use
 - Processes obtain resources that they later release for reuse by other processes
 - Examples:
 - Processor , I/O channels, main and secondary memory, files, databases
- Consumable resources
 - Description:
 - Created (produced) and destroyed (consumed) by a process
 - Examples:
 - Interrupts, signals, messages, and information in I/O buffers

Necessary Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource (One process holds a resource in a non-sharable mode & Other processes requesting resource must wait for resource to be released).
- **Hold and wait:** a process holding resource(s) is waiting to acquire additional resources held by other processes.
- **No preemption:** Resources not forcibly removed from a process holding it, a resource can be released only voluntarily by the process upon its task completion.

Necessary Conditions for Deadlock

- **Circular wait:** A closed chain of processes exists.

A set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that :

P_0 is waiting for a resource that is held by P_1 ,

P_1 is waiting for a resource that is held by $P_2, \dots,$

P_{n-1} is waiting for a resource that is held by P_n ,

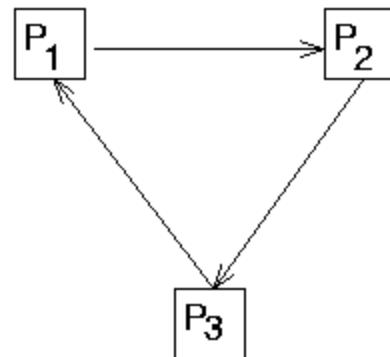
and P_n is waiting for a resource that is held by P_0 .

Types of Deadlocks

- Resource Deadlocks
 - Processes can simultaneously wait for several resources and can not proceed until they have acquired all those resources.
 - Deadlock occurs if each process in a set request resources held by another process in the same set, **and it must receive all the requested resources to move further.**
- Communication Deadlocks
 - Each process is waiting for communication (process's message) from another process, and will not communicate until it receives the communication for which it is waiting.

Wait-For Graphs (WFG)

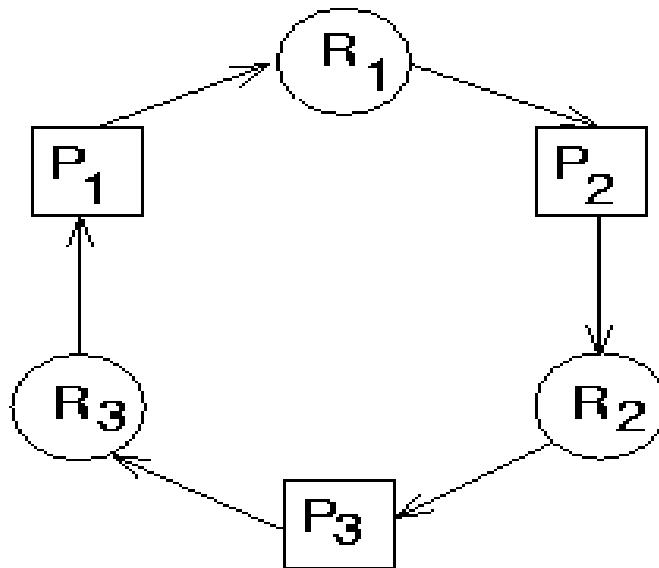
- In DS, the **system state** can be modeled by a directed graph called a *wait-for-graph*(WFG).



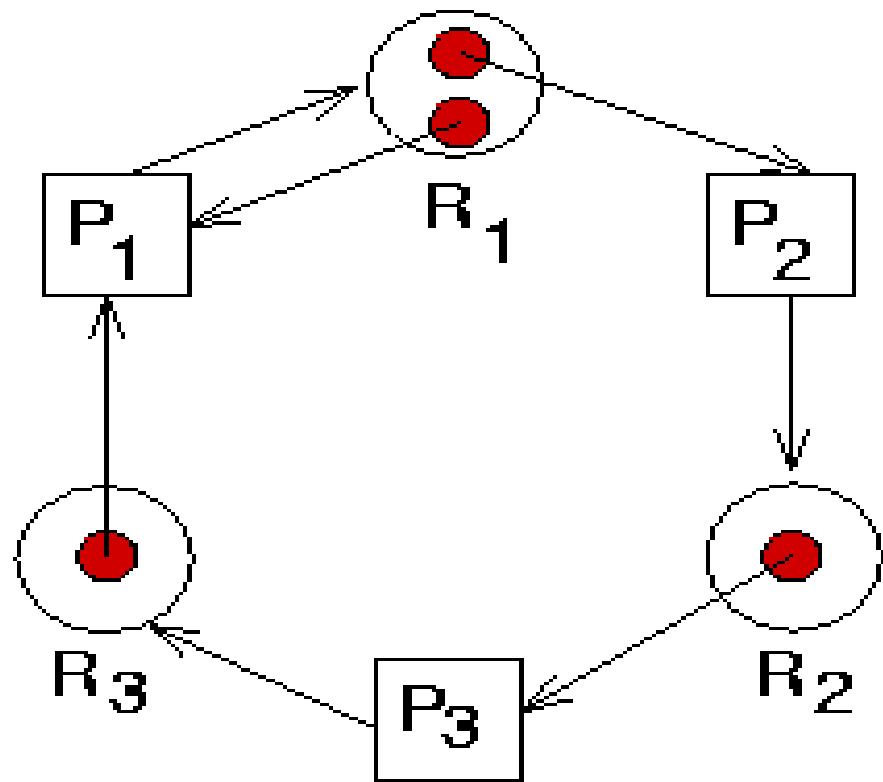
- Wait-for Graphs (WFG): $P_1 \rightarrow P_2$ implies P_1 is waiting for a resource from P_2 .
- WFG in databases
 - Transaction-wait-for Graphs (TWF)
 - Nodes are transactions and $T_1 \rightarrow T_2$ indicates that T_1 is blocked and waiting for T_2 to release some resource

Single-Unit Resource Allocation Graphs

- Nodes correspond to processes and resources.
- The **simplest request model** since a process is restricted to requesting only one unit of a resource at a time.
- The outdegree of nodes in the WFG is one.
- A **deadlock corresponds to a cycle** in the **wait-for-graph**, provided that there is only one unit of every resource in the system.



Multiunit Resource Allocation Graphs

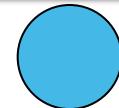


Resource allocation graph

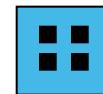
- To represent the state of process & resource interaction in DS
- Nodes of a graph → processes and resources.
- Edges of a graph → pending requests or assignment of resources.
- A pending request is represented by a *request edge* directed from the node of a requesting process to the node of the requested resource
- A resource assignment is represented by an *assignment edge* directed from the node of an assigned resource to the node of the assigned process.

Resource Allocation Graph

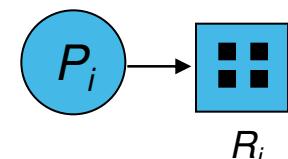
- Process



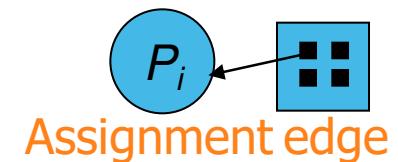
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



- P_i releases an instance of R_j



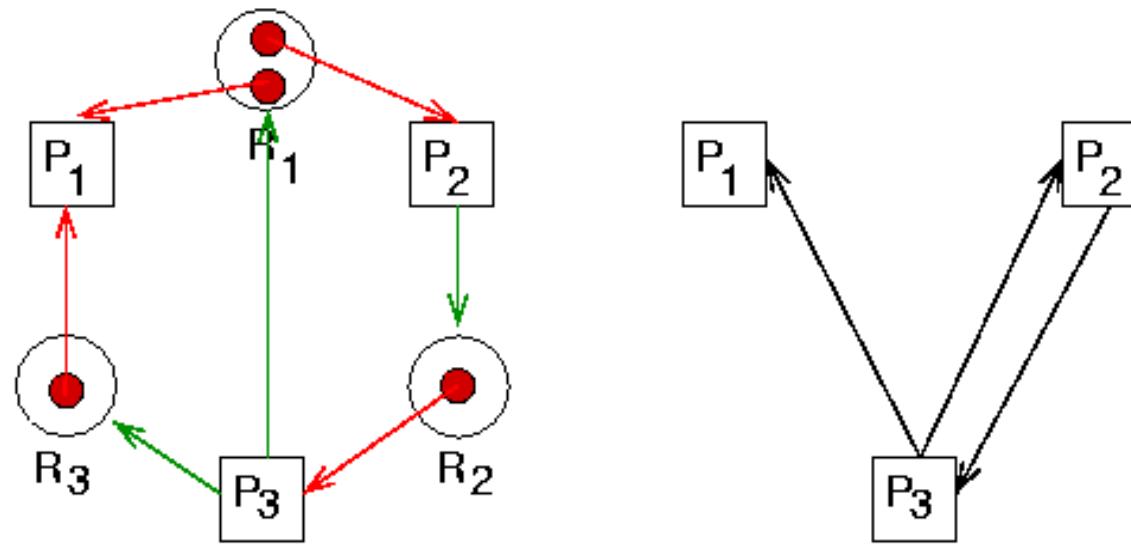
The sequence of
Process's resource utilization

Resource allocation graph

- System is deadlocked if its resource allocation graph contains a *directed cycle* or a *knot*
 - A **knot** in a directed graph is a collection of vertices and edges with the property that every vertex in the knot has outgoing edges, and **all outgoing edges from vertices in the knot terminate at other vertices in the knot.**

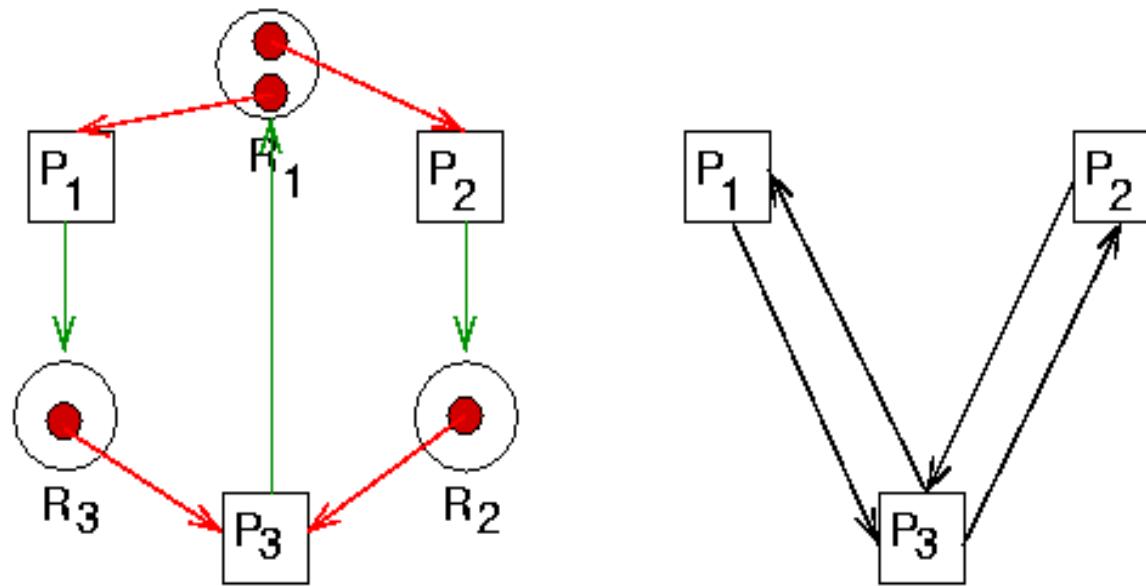
Resource allocation graph

- A cycle with no deadlock, and no knot.



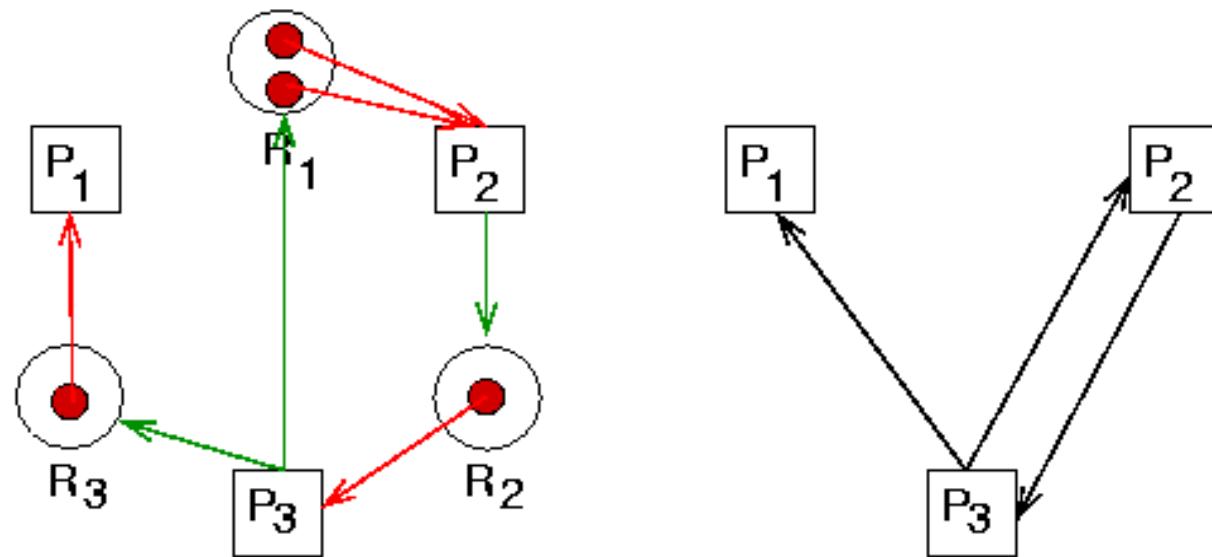
Resource allocation graph

- A knot is a sufficient condition for deadlock.

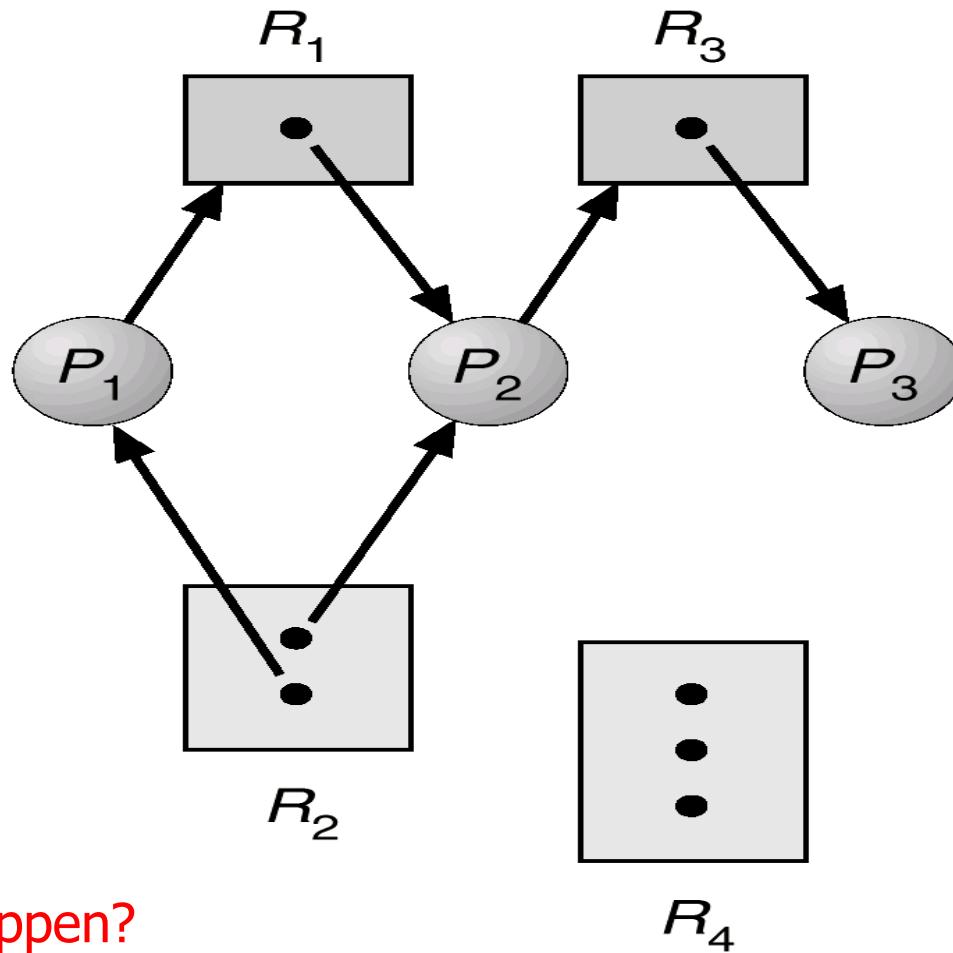


Resource allocation graph

- A knot is not a necessary condition for deadlock.

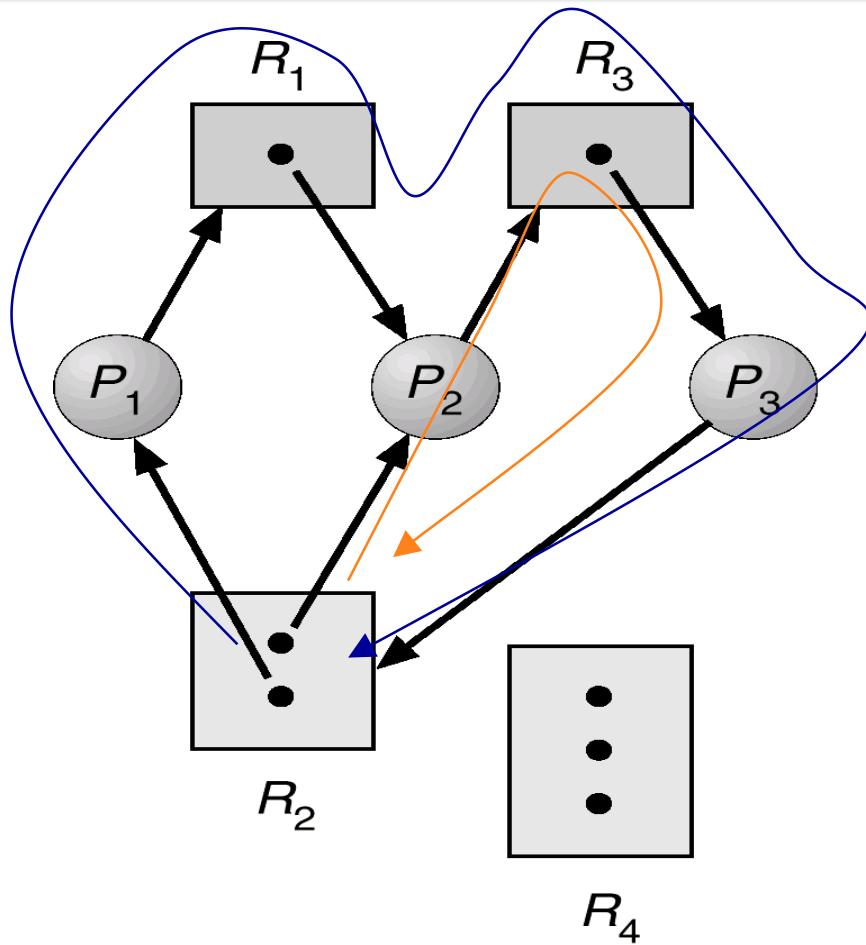


Resource-allocation graph



Can a deadlock happen?

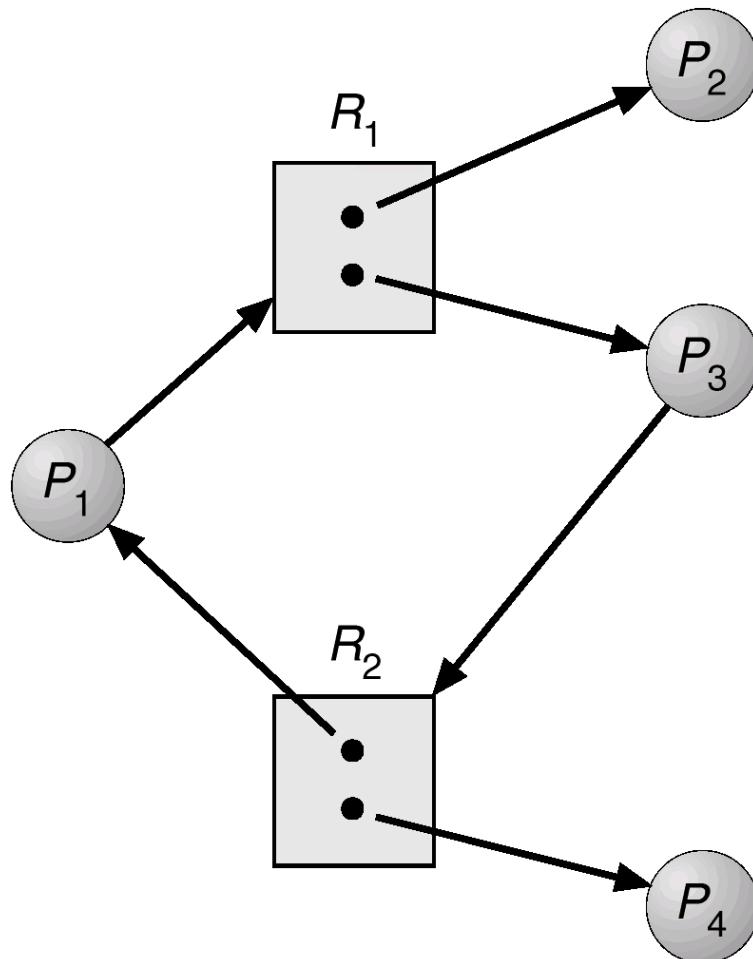
Resource-allocation graph



Can a deadlock happen?

There are two cycles found.

Resource Allocation Graph With A Cycle But No Deadlock



- If graph contains no cycles \Rightarrow no deadlock.
- If graph contains a cycle
 - if only one instance per resource type, then deadlock.
 - if several instances per resource type, possibility of deadlock.

AND, OR Models

- AND Model
 - A process/transaction can simultaneously request for multiple resources.
 - Remains blocked until it is granted *all* of the requested resources.
 - Resource deadlocked processes are modeled by AND
 - A cycle is sufficient to declare a deadlock with this model
- OR Model
 - A process/transaction can simultaneously request for multiple resources.
 - Remains blocked till *any one* of the requested resource is granted.
 - Communication deadlocks are modeled by OR
 - A cycle is a necessary condition
 - A knot is a sufficient condition

Deadlock Handling Strategies

- There are three strategies for handling deadlocks, viz., **deadlock prevention**, **deadlock avoidance**, and **deadlock detection**.

1. Deadlock Prevention:

- Resources are granted to requesting process in such a way that granting a request for a resource never leads a deadlock
- Can be achieved by
 - either having process acquire all the needed resources simultaneously before it begins execution,
 - or by preempting a process that holds the needed resource

Deadlock Handling Strategies

- Many drawbacks
 - Decreases the system concurrency
 - Set of processes can become deadlocked in resource acquiring phase
 - In many systems, future resource requirements are unpredictable

Deadlock Handling Strategies

2. Deadlock Avoidance:

- Before allocation, check for possible deadlocks.
- A resource is granted to a process if the resulting global system state is safe.
- A state is safe if **there exists at least one sequence of execution for all processes** such that all of them can run to completion
- Difficult in DS because,
 - It needs global state info in each site requiring huge storage and communication requirements
 - Due to the large number of processes and resources, it will be computationally expensive to check for a safe state

Deadlock Handling Strategies

- **Deadlock Detection:**
 - Resources are granted to requesting processes without any check
 - Periodically the status of resource allocation and pending requests is examined
 - Find cycles in WFG
- Two favorable conditions:
 - Once a cycle is formed in the WFG, it remains until it is detected and broken
 - Cycle detection can proceed concurrently with the normal activities of a system

Issues in Deadlock detection and resolution

- Two basic issues :
 - Detection of existing deadlocks
 - Resolution of detected deadlocks
- Detection of deadlocks involves:
 - Maintenance of the WFG
 - Search of the WFG for the presence of cycles(or knots)
- **Correct deadlock detection algorithm must satisfy the following two conditions:**
 1. Progress—No undetected deadlocks
 - Algorithm must detect all existing deadlocks in finite time
 2. Safety—No false deadlocks
 - Algorithm should not report deadlocks which are non-existent

Issues in Deadlock detection and resolution

- Resolution
 - Breaking existing dependencies in the system WFG
 - Roll back one or more processes and assign their resources to other deadlocked processes

Control organizations for Distributed Deadlocks

1. Centralized Control

- A *control site (designated site)* constructs global wait-for graphs (WFGs) and checks for directed cycles.
- WFG can be maintained continuously (or) built on-demand by requesting WFGs from individual sites.
- Simple and easy to implement
- Have a single point of failure
 - Communication links near the control sites are likely to be congested

Control organizations for Distributed Deadlocks

2. Distributed Control

- Responsibility for detecting a global deadlock is shared equally among all sites
- The global state graph is spread over many sites and several sites participate in the detection of a **global cycle**.
- Deadlock detection is initiated only when the waiting process is suspected to be a part of deadlock cycle
- Any site can initiate the deadlock detection process

- Not vulnerable to single point failure
- Difficult to design due to the lack of globally shared memory
- **Several sites may initiate detection for the same deadlock**

Centralized Algorithms

- **The completely centralized algorithm**
 - Control site maintains the WFG of the entire system and checks it for the existence of the deadlock
 - *Request resource* and *release resource* messages sent by each site to the control site
 - Simple and easy to implement
- Problem:
 - Highly inefficient
 - Large delays in user response, large communication overhead, congestion of communication links near the control site
 - Poor reliability

Centralized Algorithms

- Solution:
 - Can be mitigated partially by having each site maintain its resource status (WFG) locally and by having each site send its **resource status to a designated site periodically** for construction of the global WFG and the detection of deadlocks.
 - Due to inherent communication delays and the lack of synchronized clocks, **the designated site may get inconsistent view** of the system and detect false deadlocks
- Example:
 - R1 and R2 are stored at sites S1 and S2
 - Transactions T1 and T2 are started at sites S3 and S4 respectively:

Centralized Algorithms

T1

lock R1

unlock R1

lock R2

unlock R2

T2

lock R1

unlock R1

lock R2

unlock R2

Centralized Algorithms

- **Ho-Ramamoorthy 2-phase Algorithm**

- Each site maintains a status table of all local processes initiated at that site: includes all resources locked & all resources being waited on.
- Controller requests (periodically) the status table from each site.
- Controller then constructs WFG from these tables, searches for cycle(s).
- If no cycles, no deadlocks.

Centralized Algorithms

- Otherwise, (cycle exists): Request for state tables again.
- The designated site construct WFG *only* those transactions which are common to both reports.
- If the same cycle is detected again, system is in deadlock.
- Later proved: cycles in 2 consecutive reports *need not* result in a deadlock. Hence, this algorithm detects false deadlocks

Centralized Algorithms

- **Ho-Ramamoorthy 1-phase Algorithm**

- Each site maintains 2 status tables: *resource status* table (for all local resources) and *process status* table (for all local processes).
- Controller periodically collects these tables from each site.
- Constructs a WFG from transactions common to both the tables.
- No cycle, no deadlocks.
- A cycle means a deadlock.

Centralized Algorithms

- Does not detect false deadlocks:

E.g.

- Resource table at S1: R1 is waited upon by P2 ($R1 \leftarrow P2$)
- Process table at S2: P2 is waiting for R1 ($P2 \rightarrow R1$)
- then, the edge $P2 \rightarrow R1$ reflects current system state.

Distributed Algorithms

- All sites collectively cooperate **to detect a cycle in the state graph** that is likely to be distributed over several sites of the system
- Initiated when,
 - A process is forced to wait and, it can be initiated by the local site of the process.

Classification of Distributed Algorithms

- **Path-pushing:** resource dependency information disseminated through designated paths (in the graph).
- **Edge-chasing:** special messages or probes circulated along edges of WFG. Deadlock exists if the probe is received back by the initiator (single returned probe indicates a cycle).
- **Diffusion computation:** queries on status sent to process in WFG (all queries returned indicates a cycle).
- **Global state detection:** get a snapshot of the distributed system.

Edge-Chasing Algorithm

- Chandy-Misra-Haas's Algorithm for AND model (Resource Model)
 - If a process makes a request for a resource which fails or times out, the process generates a *probe message* and sends it to each of the processes **holding one or more** of its requested resources.
 - Each probe message contains the following information:
 - the *id* of the process that is blocked (*the one that initiates the probe message*);
 - the *id* of the process is sending this particular version of the probe message; and
 - the *id* of the process that should receive this probe message.
 - A probe(i, j, k) is used by a deadlock detection process P_i . This probe is sent by the home site of P_j to home site of P_k .
 - This probe message is circulated via the edges of the graph. Probe returning to P_i implies deadlock detection.

Edge-Chasing Algorithm

- Chandy-Misra-Haas's Algorithm for AND model (Resource Model)
 - Terms used:
 - P_j is *dependent* on P_k , if a sequence of $P_j, P_{i1}, \dots, P_{im}, P_k$ exists, such that each process except P_k in the sequence is blocked and each process , except the first one (P_j), holds a resource for which the previous process in the sequence is waiting.
 - P_j is *locally dependent* on P_k , if P_j is dependent upon P_k and both the processes are at same site.
 - Each process maintains a boolean array *dependent_i*: *dependent_i(j)* is true if P_i knows that P_j is dependent on it. (initially *dependent_i(j)* set to false for all i & j).

Chandy-Misra-Haas's Algorithm

Sending the probe:

if P_i is locally dependent on itself then deadlock.

else for all P_j and P_k such that

(a) P_i is locally dependent upon P_j , and

(b) P_j is waiting on P_k , and

(c) P_j and P_k are on different sites, send probe(i,j,k) to the home site of P_k .

Receiving the probe:

if (d) P_k is blocked, and

(e) $\text{dependent}_k(i)$ is false, and

(f) P_k has not replied to all requests of P_j ,
then begin

$\text{dependent}_k(i) := \text{true};$

 if $k = i$

 then declare that P_i is deadlocked

.....

Chandy-Misra-Haas's Algorithm

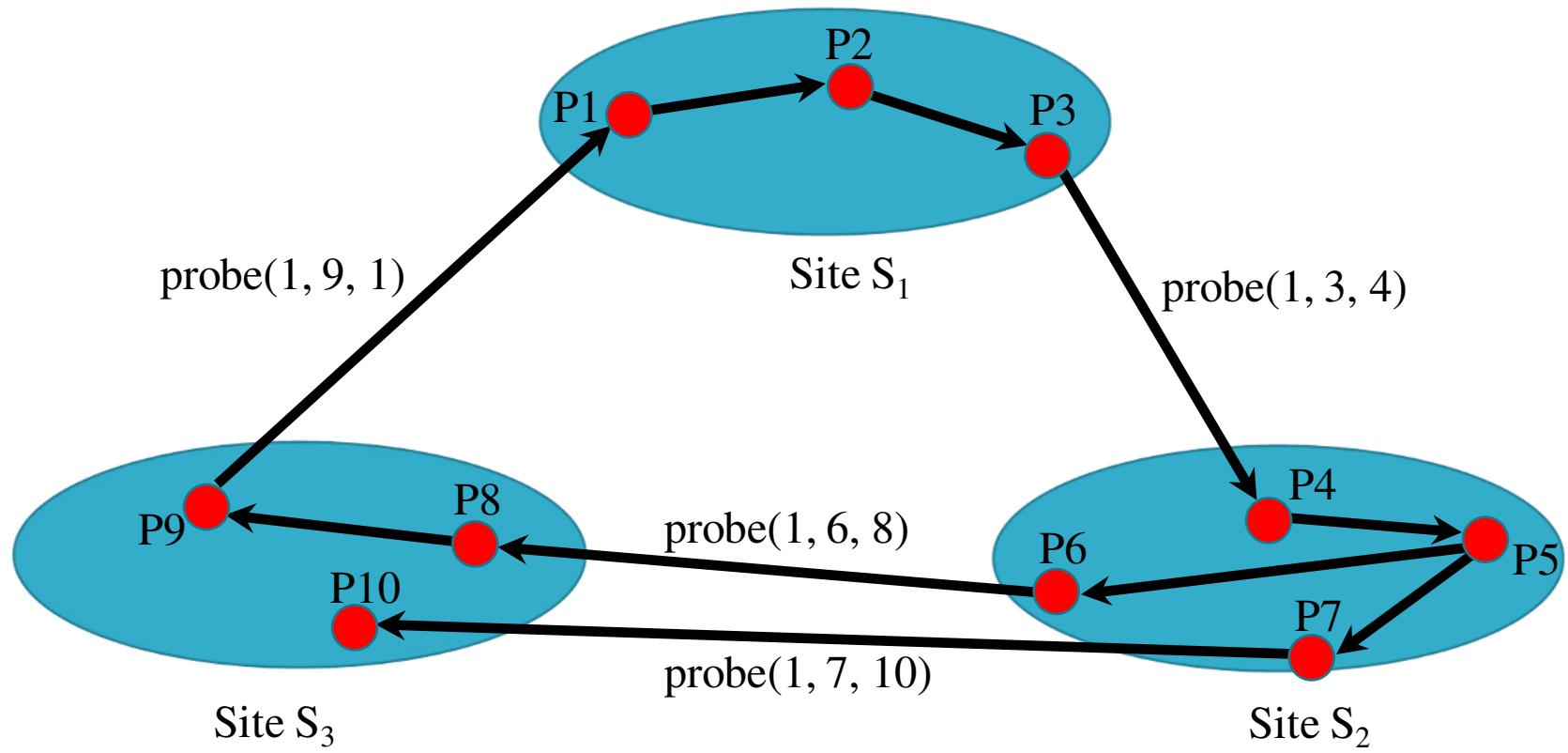
Receiving the probe (contd...):

else for all P_m and P_n such that

- (a') P_k is locally dependent upon P_m , and
- (b') P_m is waiting on P_n , and
- (c') P_m and P_n are on different sites, send probe(i, m, n)
to the home site of P_n .

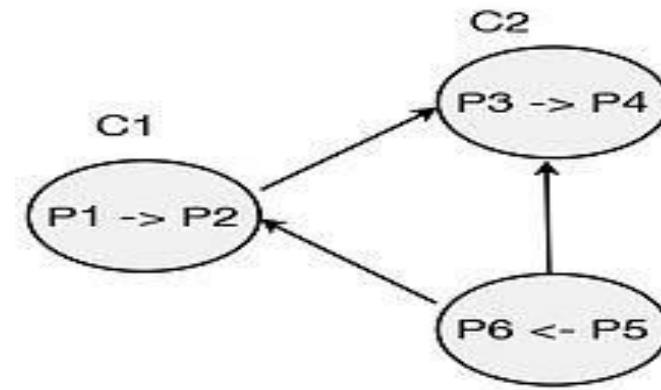
end.

Example



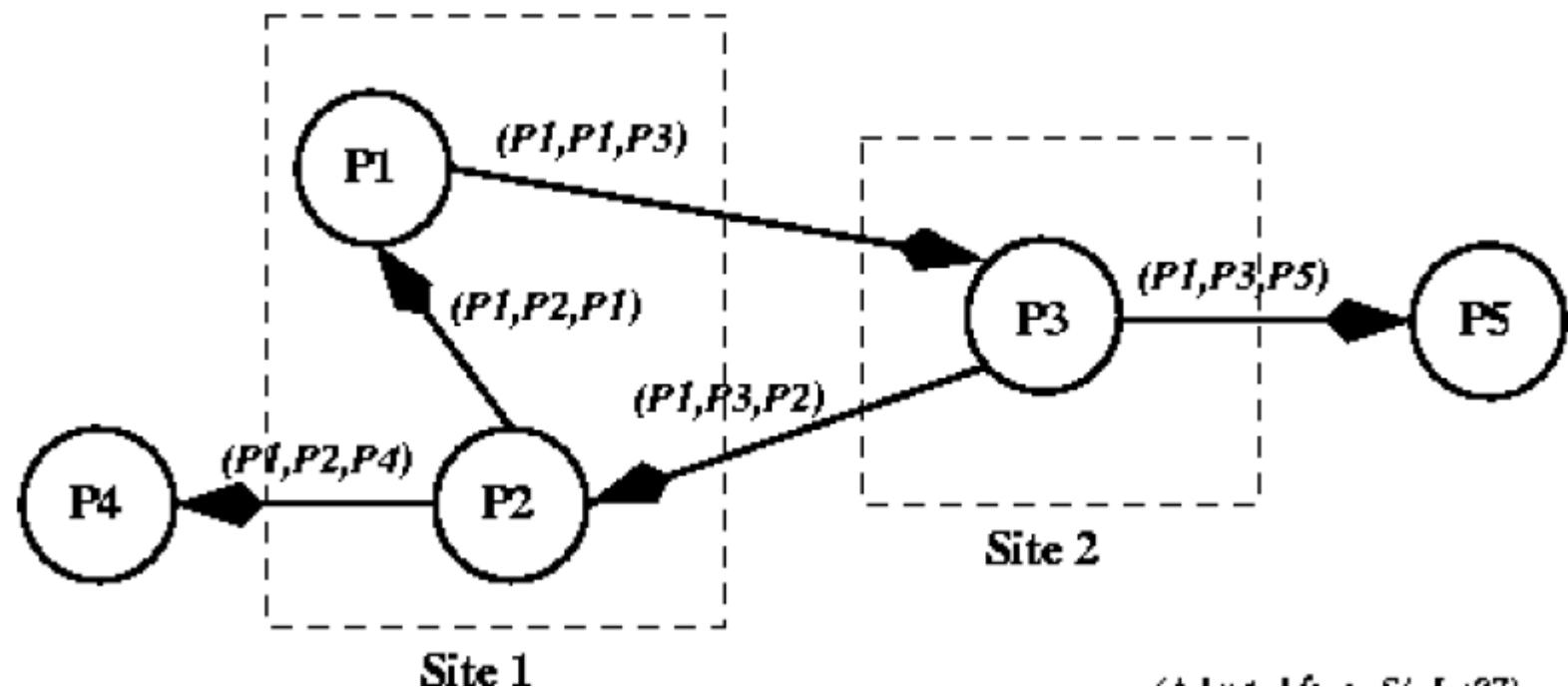
Example

- In below example, P_1 initiates the deadlock detection. C_1 sends the probe saying P_2 depends on P_3 . Once the message is received by C_2 , it checks whether P_3 is idle. P_3 is idle because it is locally dependent on P_4 and updates $dependent_3(2)$ to True.
- Same as above, C_2 sends probe to C_3 and C_3 sends probe to C_1 . At C_1 , P_1 is idle so it update $dependent_1(1)$ to true. Therefore it can be declared a deadlock has occurred.

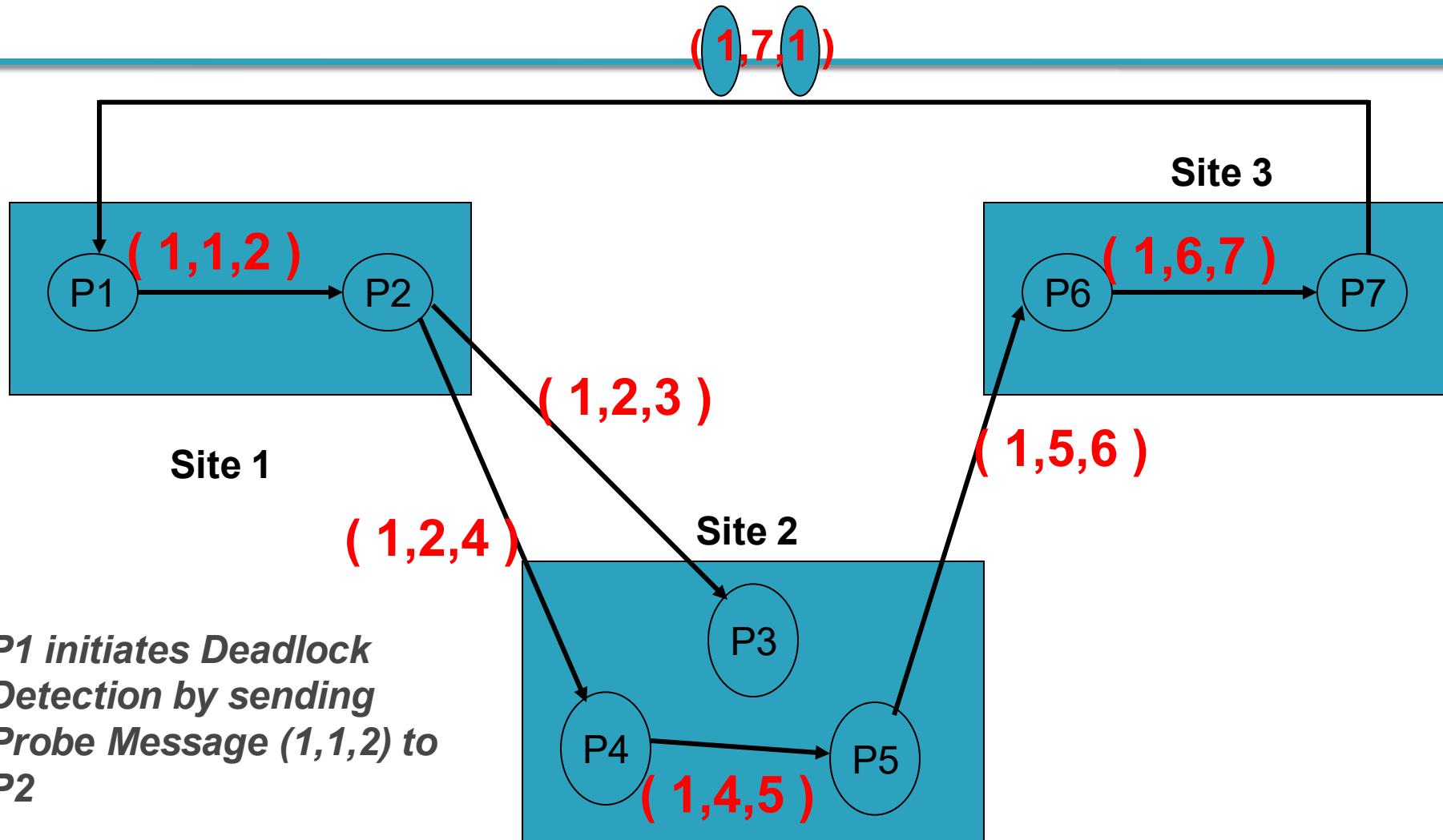


Example

- **P1** initiates the probe message, so that all the messages shown have **P1** as the initiator. When the probe message is received by process **P3**, it modifies it and sends it to two more processes. Eventually, the probe message returns to process **P1**. **Deadlock!**

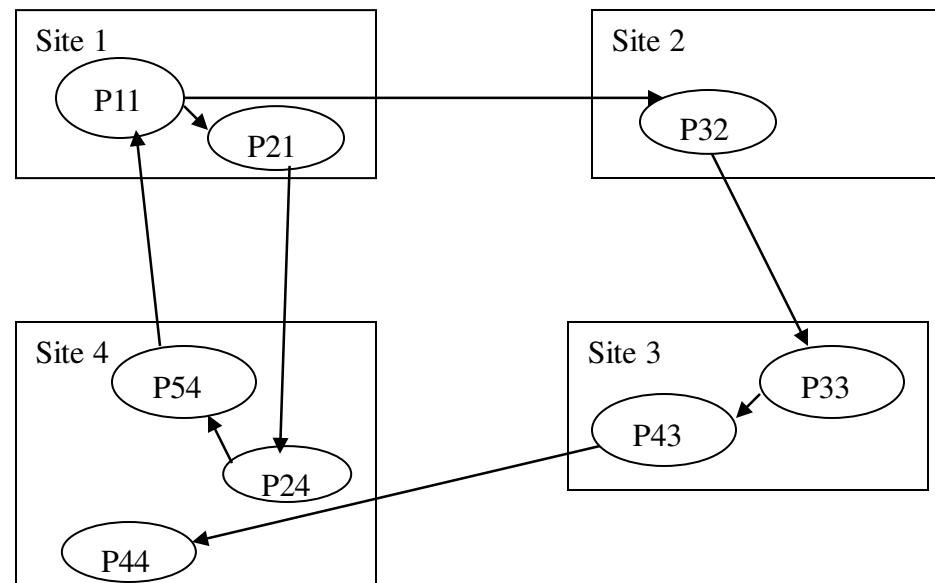


Example



Example

- If process P11 initiates Chandy-Misra edge chasing algorithm then find out whether deadlock is occurred in the system or not? Justify your answer.



Advantages of Chandy-Misra-Hass Deadlock Detection Algorithm

- The advantages of this algorithm include the following:
 1. It is easy to implement.
 2. Each probe message is of fixed length.
 3. There is very little computation.
 4. There is very little overhead.
 5. There is no need to construct a graph, nor to pass graph information to other sites.
 6. This algorithm does not find false deadlock.
 7. There is no need for special data structures.

A Diffusion Computation Based Algorithm

Algorithm for OR model (Communication Model)

- Deadlock Detection computation is diffused through WFG of the system
- Process can be in two states:
 - Active state- a process is executing.
 - Blocked state – a process is waiting to acquire a resource.
- A blocked process may start a diffusion- by sending query message to all the processes from whom it is waiting to receive a message.
- Messages takes two forms
 - Query message (i,j,k)
 - i = initiator of check
 - j = immediate sender
 - k = immediate recipient
 - Reply message (i,k,j)

A Diffusion Computation Based Algorithm

Terms used:

- Dependent set of process:
 - Set of processes from whom the process is waiting to receive message
- Engaging query:
 - The first query message received for the deadlock detection initiated by Pi.
- $\text{wait}_k(i)$:
 - It is a local boolean variable at process P_k indicates that it has been continuously blocked since it received the last engaging query from process Pi

A Diffusion Computation Based Algorithm

If an active process receives a query or reply message, it discards it. When a blocked process P_k receives a query (i,j,k) message, it takes the following actions:

- If this is the first query message received by P_k for deadlock detection initiated by P_i (*called the engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable $num_k(i)$ (*to the number of query messages sent*).
- If this is not an engaging query, then P_k returns a reply message to it immediately, provided P_k has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

An initiator detects a deadlock when it receives reply messages to all the query messages it had sent out.

Diffusion-based Algorithm

Initiation by a blocked process P_i :

send query(i,i,j) to all processes P_j in the dependent set DS_i of P_i ;
 $num(i) := |DS_i|$; $wait_i(i) := true$;

When a blocked process P_k receiving query(i,j,k):

if this is *engaging* query for process P_k /* first query from P_i */
then

 send query(i,k,m) to all P_m in DS_k ;

$num_k(i) := |DS_k|$; $wait_k(i) := true$;

else if

$wait_k(i)$

 then

 send a reply(i,k,j) to P_j .

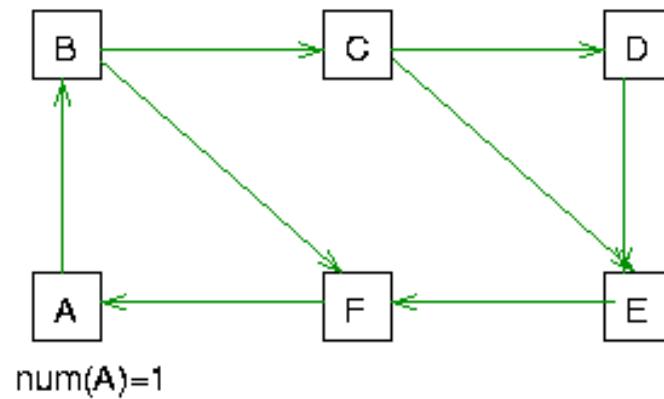
Diffusion-based Algorithm

Process P_k receiving reply(i, j, k)

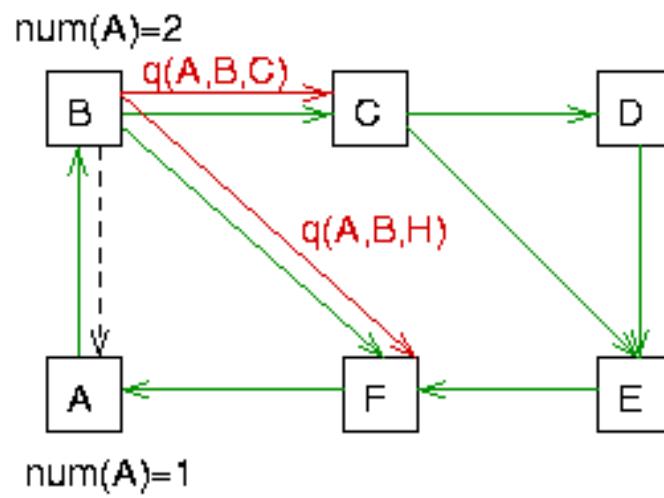
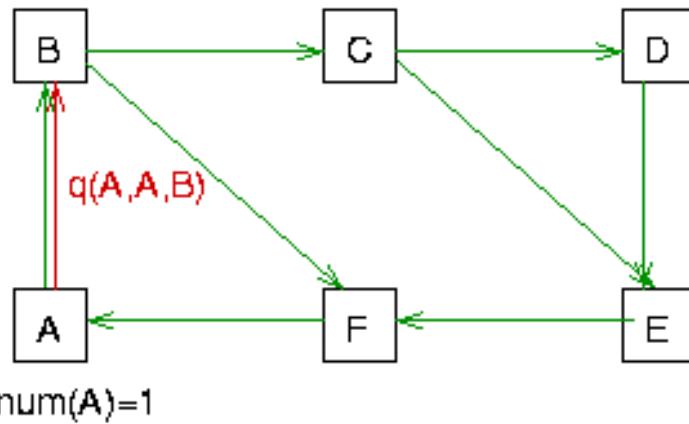
```
if
    waitk(i)
then begin
    numk(i) := numk(i) - 1;
    if
        numk(i) = 0 then
            if
                i = k then declare a deadlock.
            else
                send reply(i, k, m) to Pm, which sent the engaging query.
```

On receipt of $query(i,j,k)$ by m

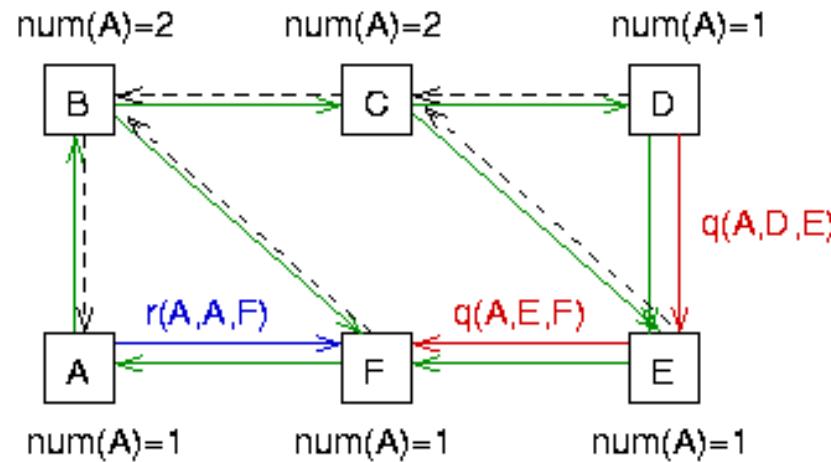
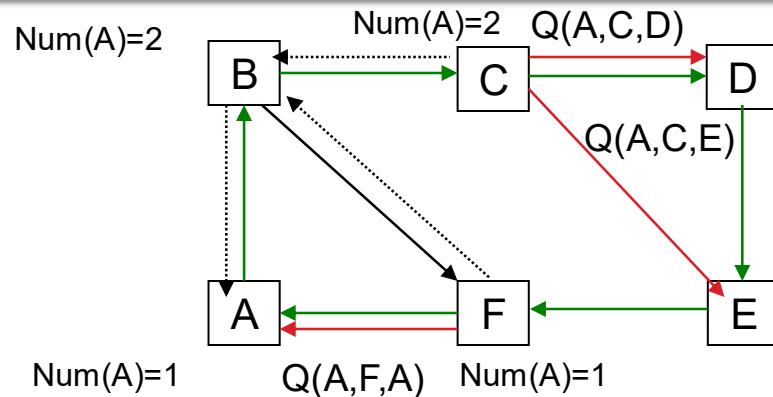
- if not blocked then discard the query
- if blocked
 - if this is an *engaging* query propagate $query(i,j,k)$
 - else
 - if not continuously blocked since engagement then discard the query
 - else send $reply(i,k,j)$ to j



On receipt of $query(i,j,k)$ by m

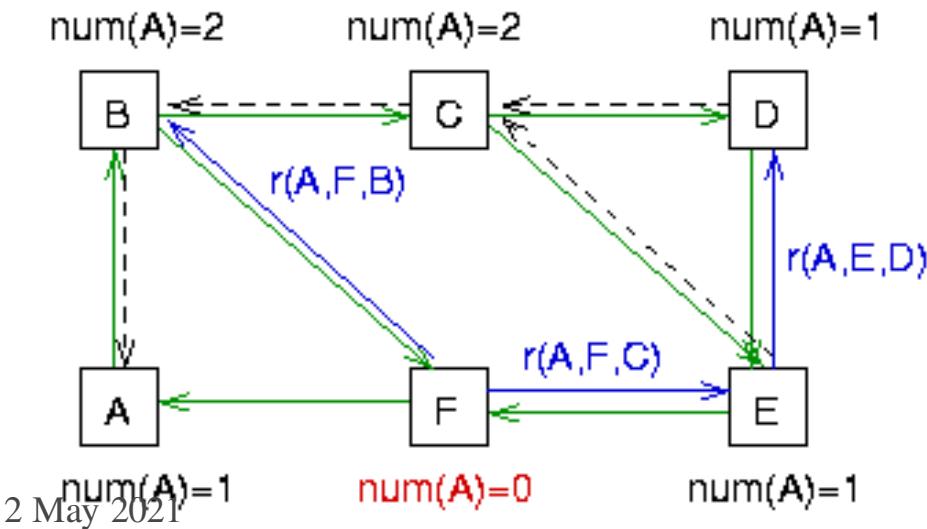


On receipt of $query(i,j,k)$ by m

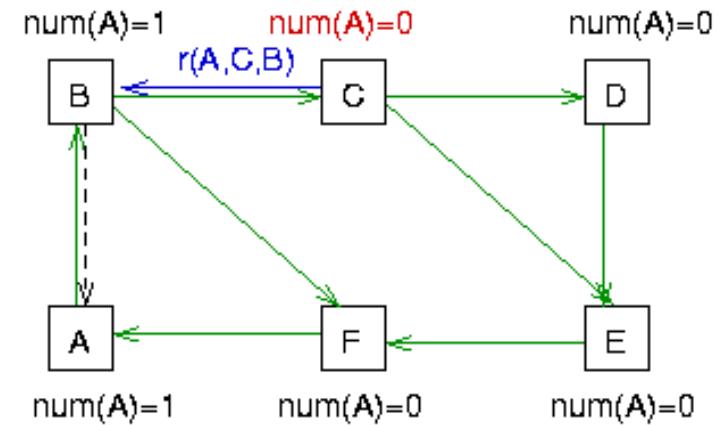
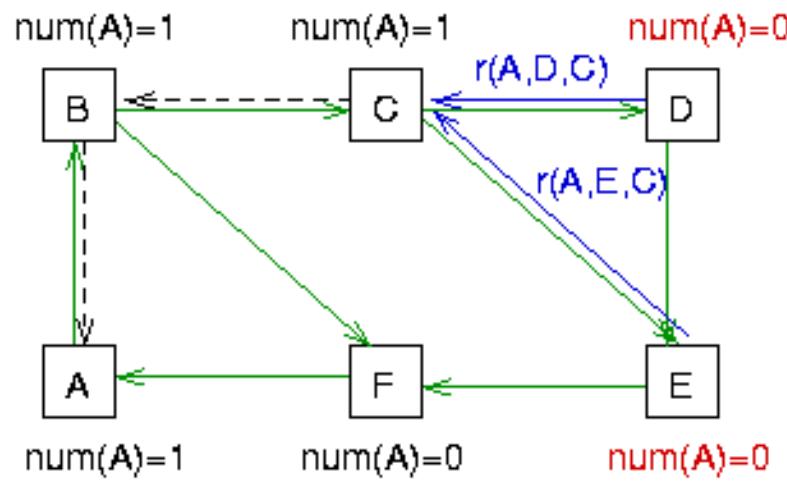


On receipt of $\text{reply}(i,j,k)$ by k

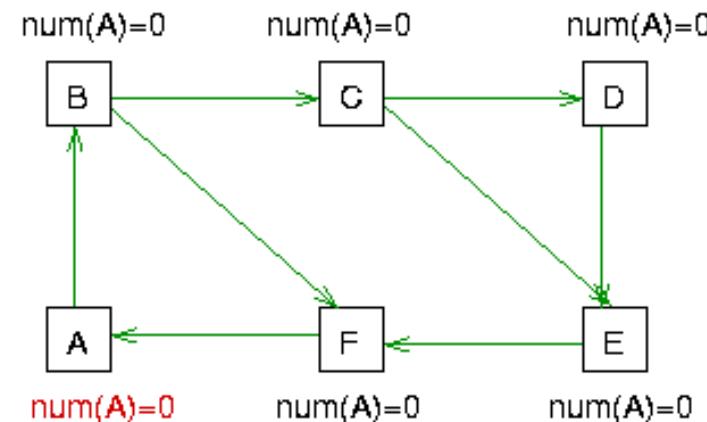
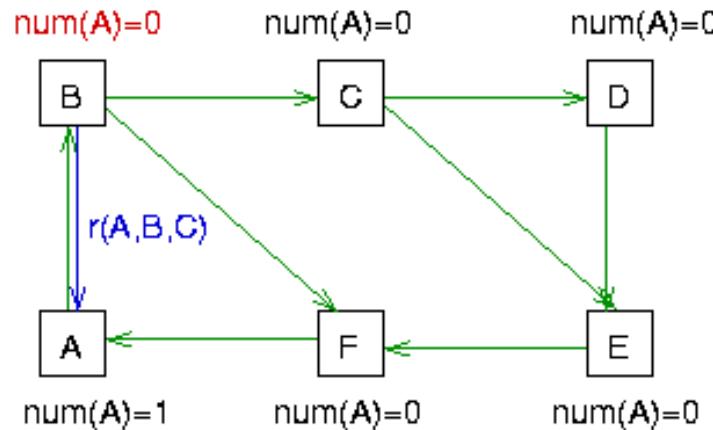
- if this is not the last reply
then just decrement the awaited reply count
- if this is the last reply then
 - if $i=k$ report a deadlock
 - else send $\text{reply}(i,k,m)$
to the engaging process m



On receipt of $\text{reply}(i,j,k)$ by k



On receipt of $\text{reply}(i,j,k)$ by k



Persistence & Resolution

- Deadlock persistence:
 - Average time a deadlock exists before it is resolved.
- Implication of persistence:
 - Resources unavailable for this period: **affects utilization**
 - Processes wait for this period unproductively: **affects response time.**
- Deadlock resolution:
 - Aborting at least one process/request involved in the deadlock.
 - Efficient resolution of deadlock requires knowledge of all processes and resources.
 - If every process detects a deadlock and tries to resolve it independently -> highly inefficient ! Several processes might be aborted.

Deadlock Resolution

- Priorities for processes/transactions can be useful for resolution.
 - Highest priority process initiates and detects deadlock .
 - When deadlock is detected, lowest priority process(s) can be aborted to resolve the deadlock.

Distributed Scheduling

References:

- Mukesh Singhal, Niranjan Shivaratri. “Advanced concepts in operating systems”. Tata McGraw Hill publication.

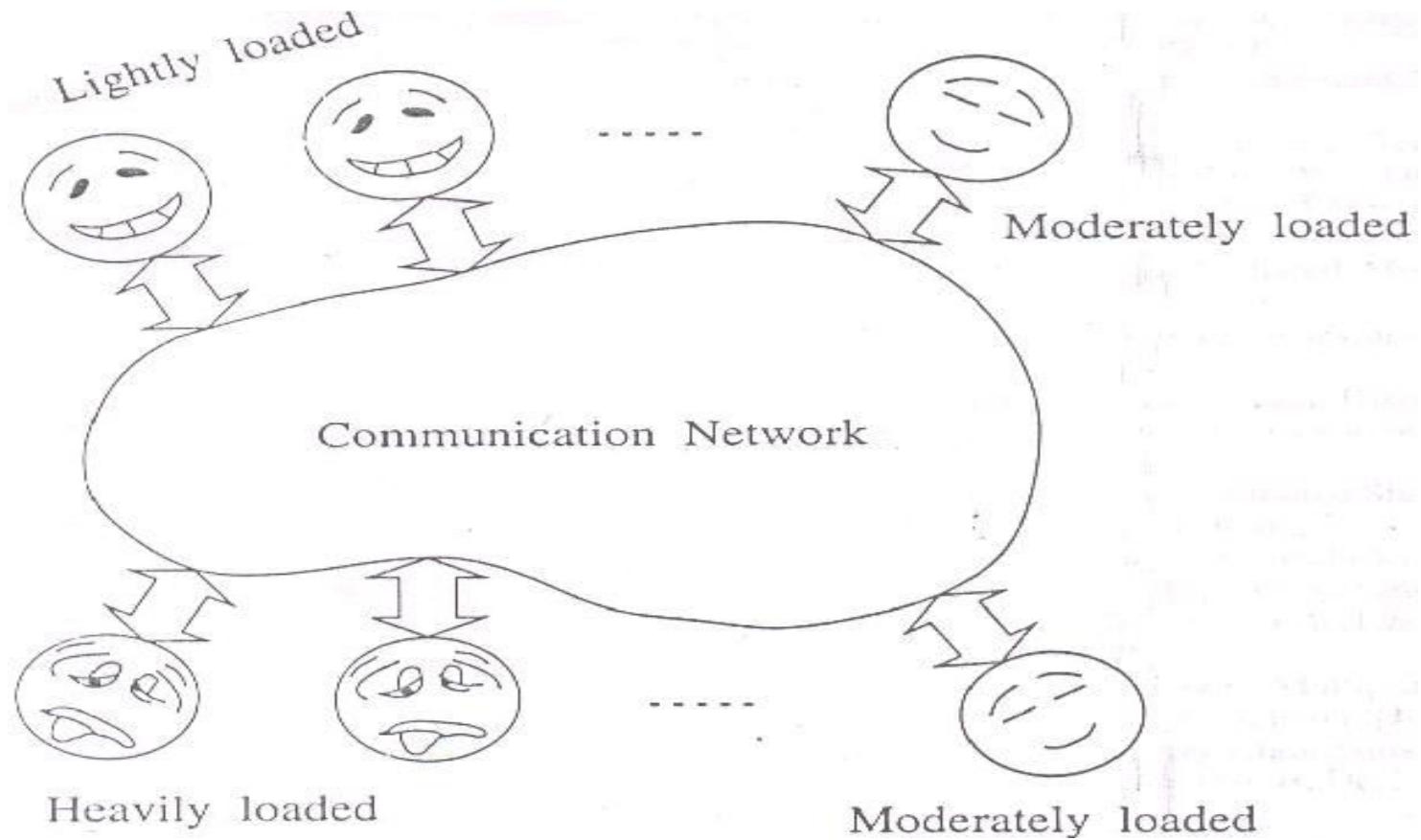
Introduction

- Distributed systems offer a tremendous processing capacity.
- however, in order to realize this tremendous computing capacity, and take full advantage of it, good resource allocation schemes are needed.
- A distributed scheduler is a resource management components of a distributed operating system that focuses on judiciously and transparently redistributing the load of the system among the computers such that overall performance of the system is maximized.

Introduction

- Due to random arrival of tasks and their random CPU service time requirements, there is a good possibility that several computers are **heavily loaded** (hence suffering from performance degradation), while others are **idle or lightly loaded**.

Introduction



Introduction

- **Goal:** To improve the performance of distributed system by appropriately transferring work from heavily loaded computer to idle or lightly loaded computer.
- Performance of a system
 - One of the metric is **average response time** of task which is the length of the time interval between its origination and completion.
 - Minimizing the **average response time** is often the goal of load distributing.
- Defining a proper characterization of a load at node is very important, as load distributing decisions are based on the load measured at one or more nodes.
- The mechanism used to measure load imposes minimal overhead.

Issues in Load Distributing

Several central issues in load distributing

1. Load:

- CPU queue length are good indicators of load .
- As a result , when all the tasks that the node has accepted have arrived , the node can become overloaded and require further task transfers to reduce its load.
- Prevention- Artificially increment CPU queue length at a node whenever the node accepts a remote task.
- Use timeout to avoid anomalies when task transfers fail.
- After the timeout, if the task has not yet arrived, the CPU queue length is decremented.

Issues in Load Distributing

2. Classification of load distributing algorithms:

Broadly characterized as:

- Static
 - Decision is hard wired in the algorithm, using a prior knowledge of the system.
 - They make no use of system state information (the loads at nodes).

Classification of load distributing algorithms

- Dynamic
 - Make use of system state information to make load distributing decisions
 - They collect, store and analyze the system state information.
 - Impose overhead to the system
- Adaptive
 - Special class of dynamic algorithm
 - they adapt their activities by **dynamically changing the parameters of the algorithm** to suit the changing system state

Issues in Load Distributing

3. Load distributed algorithms can further be classified as **Load balancing** vs. **Load sharing algorithms**

- Both types of algorithms attempt to reduce the likelihood of an *unshared state* by transferring tasks to lightly loaded nodes.
- Unshared state : A state in which one computer lies idle while at the same time tasks argue for service at another computer
- Load balancing algo:
 - Attempt to equalize the loads at all computers

Load balancing vs. Load sharing

- Task transfer are not **instantaneous** because of **communication delays**.
- Delays in transferring a task increase the duration of unshared state.
- To avoid lengthy unshared states, *blocking* task transfers from overloaded computers to computers that are **likely to become idle shortly** can be used.

Issues in Load Distributing

4. Preemptive vs. Non-preemptive transfers:

- Preemptive:
 - Transfer of a task that is partially executed
- Non preemptive:
 - Transfer of a task that has not yet started execution
- Information about the environment in which the task will execute must be transferred to the receiving node. Ex: user's current working directory, privileges inherited by the task

Requirements for Load Distributing

- Scalability: It should work in large distributed system
- Location transparency: It should hide the location of processes.
- Determinism: A transferred process must produce the same results it would produce if it was not transferred.
- Preemption: Remotely executed processes should be preempted and migrated elsewhere on demand.
- Heterogeneity: It should be able to distinguish among different processors.

Components of Load Distributing algorithm

- Four components
 - Transfer policy
 - Determines whether a node is in a suitable state to participate in a task transfer
 - Selection policy
 - determines which task should be transferred
 - Location policy
 - determines to which node a task selected for transfer should be sent
 - Information policy
 - responsible for motivating the collection of system state information

Transfer policy

- A large number of the transfer policies are threshold policies.
 - Better to decide the **sender and receiver**.
 - when a new task originates at a node, and the load at the node **exceeds** a threshold T , the transfer policy decides that the node is **sender**.
 - If the load at a node **falls** below, the transfer policy decides that the node can be a **receiver** for a remote task.

Selection Policy

- Selects a task for transfer, once the transfer policy decides that the node is a sender
- Simplest approach: to select the newly originated task
- Overhead incurred in task transfer should be compensated by the reduction in the response time realized by the task
- Bryant and Finkel propose another approach - a task is selected for transfer only if its response time will be improved upon transfer.
- Factors to consider:
 - Overhead incurred by transfer should be minimal. For example, a task of small size carries less overhead.
 - Number of location dependent system calls made by the selected task should be minimal

Location Policy

- To find suitable nodes to share load (sender or receiver)
- Widely used method : polling- a node polls another node to find out whether it is suitable node for load sharing
 - Either serially or in parallel (e.g., multicast)
 - Either randomly or on a nearest-neighbor basis
- Alternative to polling
 - Broadcast a query to find out if any node is available for load sharing

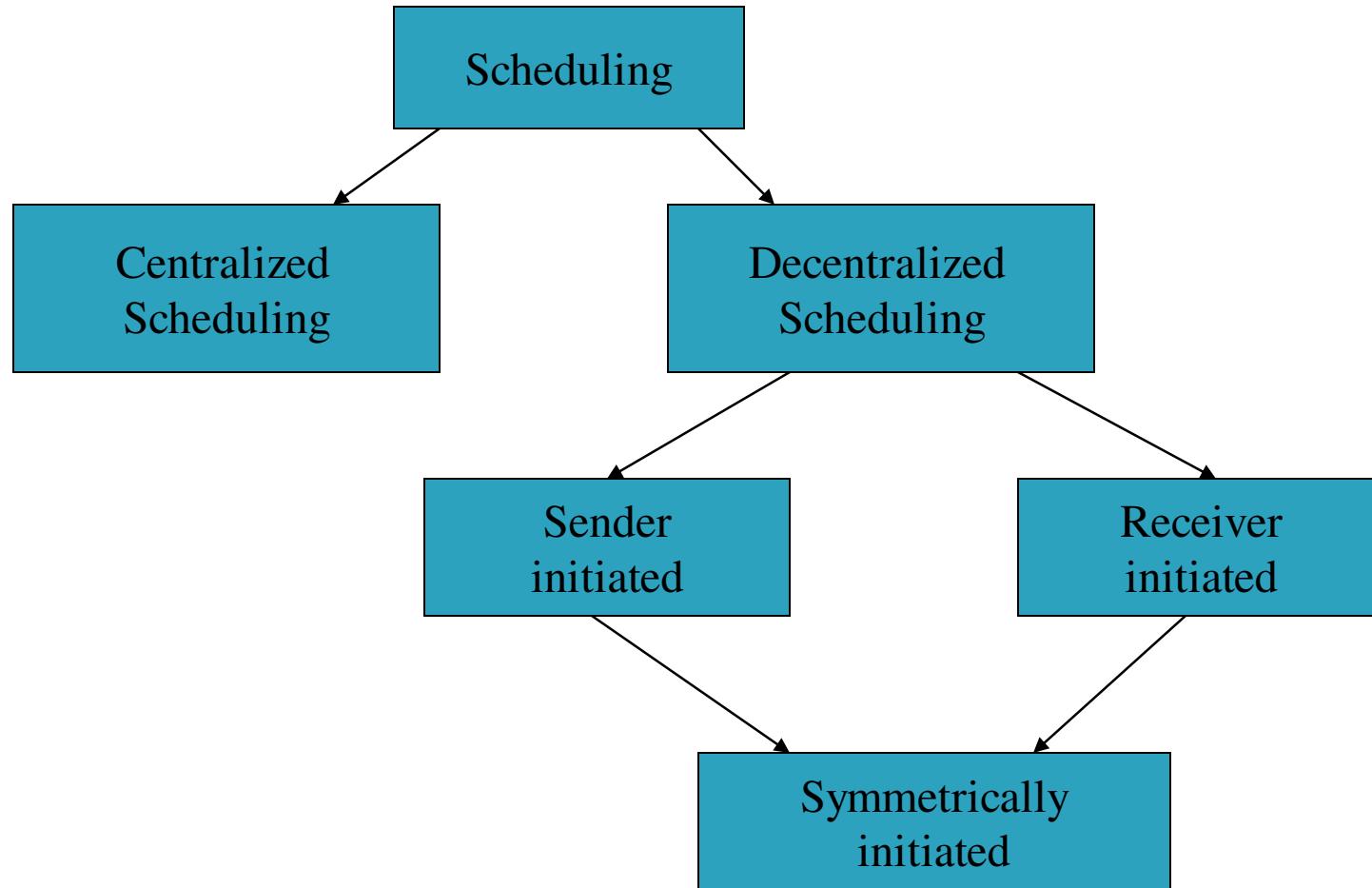
Information Policy

- To decide when, where and what information about the states of other nodes on the system should be collected
- One of three types:
 - Demand driven
 - Node collects the state of the other nodes only when it becomes either a sender or a receiver
 - dynamic policy
 - can be sender-initiated or receiver-initiated

Information Policy

- Periodic
 - Nodes exchange load information periodically
- State change driven
 - Nodes distribute state information whenever their state changes by a certain degree
 - Centralized policy- nodes send the state information to a centralized collection points and decentralized policy- nodes send the information to peers.

Types of Scheduler



Sender-Initiated algorithms

- Load distributing activity is initiated by an **overloaded node (sender)** that attempts to send a task to an underloaded node (receiver).
- **Transfer Policy:** A Threshold policy based on CPU queue length **used by the algorithms**
 - A node is identified as a **sender** if a new task originating at the node makes the queue length exceed a **threshold T**.
- **Selection Policy:** These sender-initiated algorithms consider only newly arrived tasks will be transferred.

Sender-Initiated algorithms

- **Location policy:**

1. **Random :**

- Node is randomly selected & task is simply transferred.
- No information exchange to assist in decision making
- **Problem** – useless task transfers may occur when a task is transferred to a node that is already heavily loaded.
- **Solution** – limit the number of times a task can be transferred
- It provides significant performance improvement over no load sharing

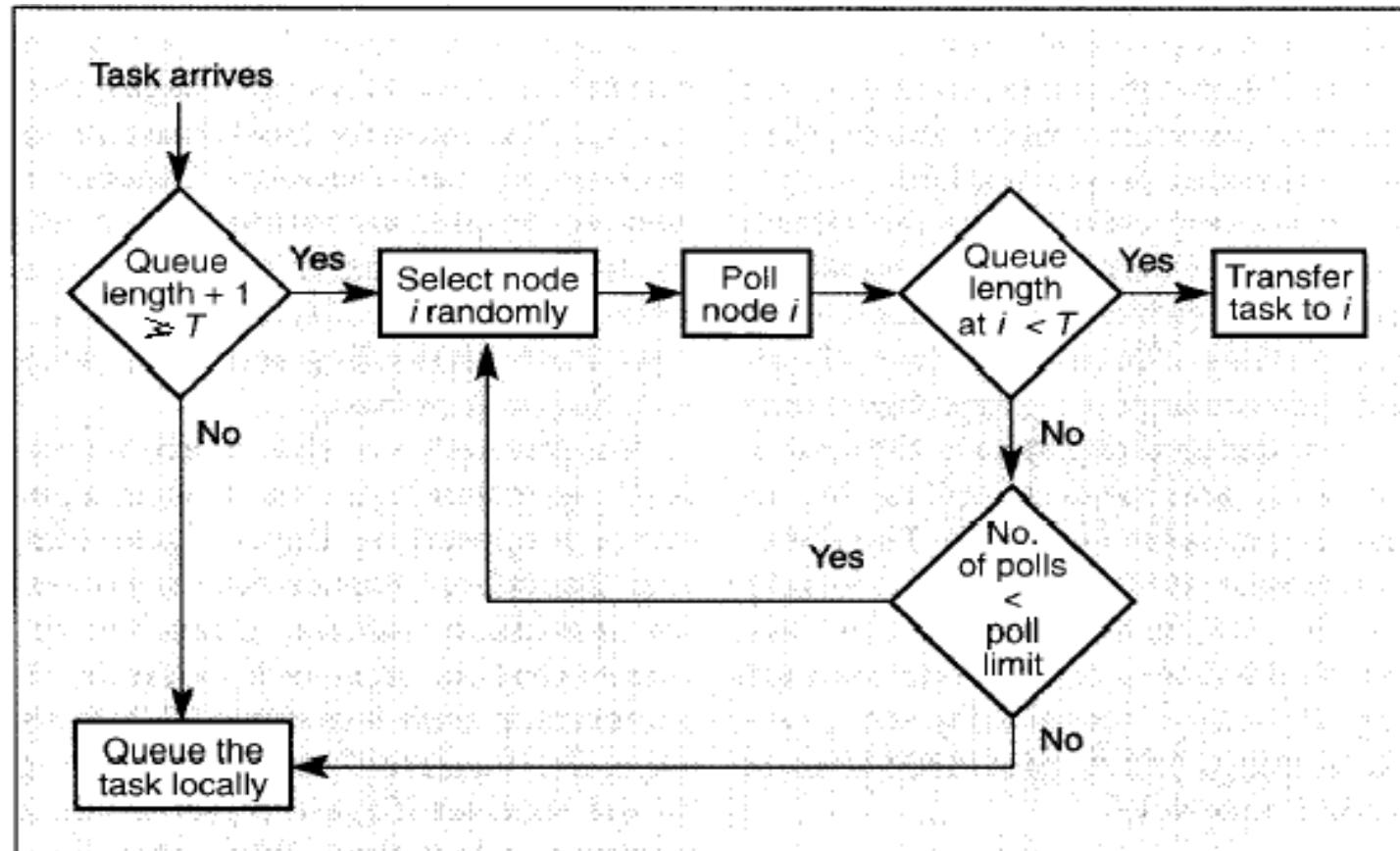
Sender-Initiated algorithms

- **Location policy:**

- 2. **Threshold :**

- The problem of useless task transfer can be avoided using polling (selected a random)
 - Node is polled to determine whether it is a receiver
 - The number of polls is limited by a parameter called ***PollLimit*** to keep the overhead low
 - A sender node will not poll any node more than once during one searching session of ***PollLimit*** polls
 - If no suitable node is found, then the sender node must execute the task
 - It provides significant performance improvement over the random location policy
 - Flowchart

Sender-Initiated algorithms



Sender-Initiated algorithms

- Location policy:

3. Shortest :

- It makes effort to choose the **best receiver** for a task
- Number of nodes (=PollLimit) are selected at random and are polled to determine their queue length
- Node with shortest queue length is selected
- The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task.
- Performance improvement is marginal indicating that more detailed state information does not necessarily result in significant improvement in system performance

Sender-Initiated algorithms

- Information Policy
 - Demand driven
- Stability
 - All three approaches for location policy cause system instability at high system loads.
 - In highly loaded system, probability of finding receiver is very low,
 - When the load **due to work arriving** and due to the **load sharing activity exceeds** the system's serving capacity, instability occurs.

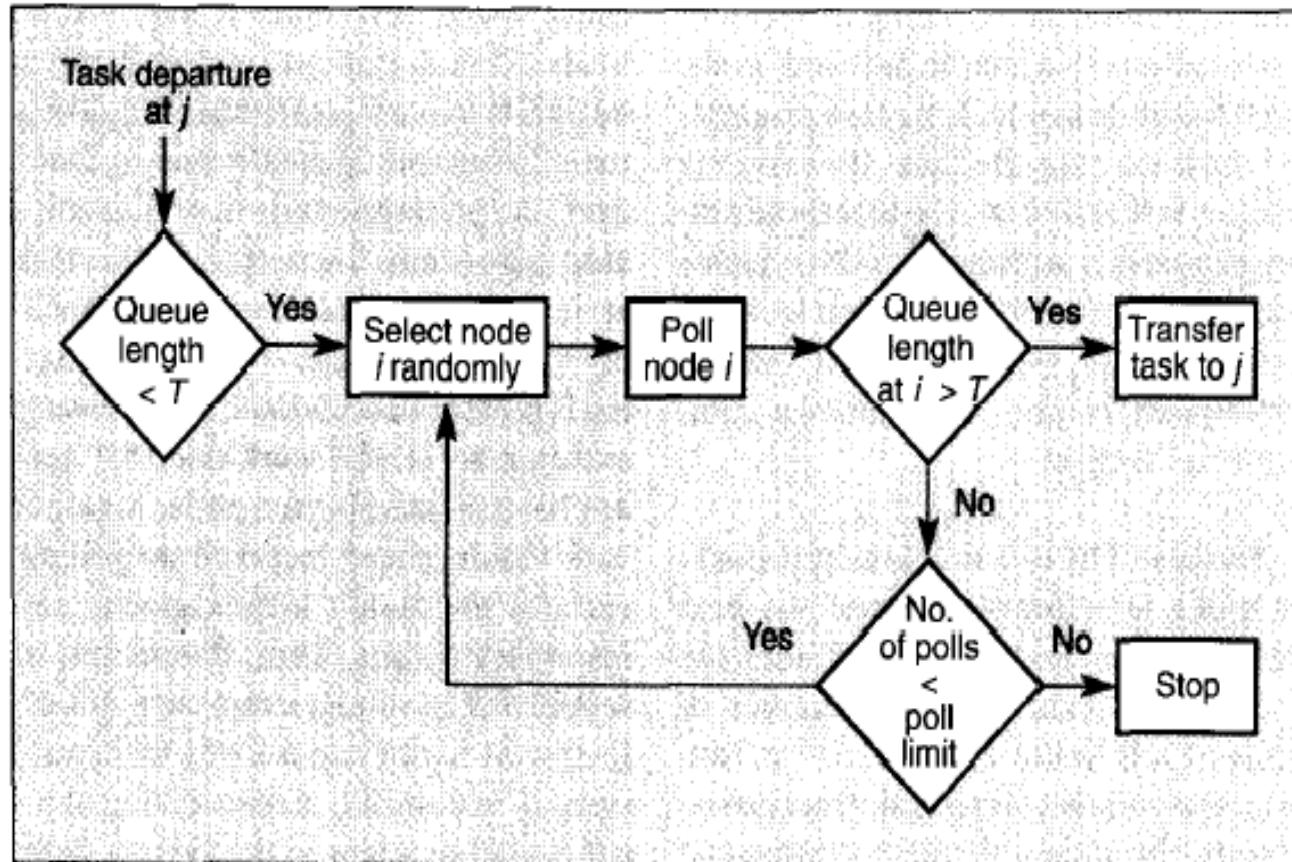
Receiver-Initiated Algorithms

- Load distributing activity is initiated from an under-loaded node (receiver) that is trying to obtain a task from an overloaded node (sender).
- **Transfer Policy:**
 - Threshold policy based on CPU queue length
 - Transfer policy is triggered when a task departs
- **Selection Policy:**
 - Any of the approaches as discussed earlier.

Receiver-Initiated Algorithms

- **Location Policy:**
 - Randomly selected node is polled
 - Above procedure is repeated until a **node that can transfer a task (i.e., a sender) is found** or a static PollLimit number of tries have failed to find a sender
 - If all polls fail to find a sender, the node waits until another task completes or until a predetermined period is over before initiating the search for a sender

Receiver-Initiated Algorithms



Receiver-Initiated Algorithms

- **Information Policy:**
 - Demand driven because the polling activity starts only after a node becomes a receiver
- **Stability:**
 - They do not cause system instability
 - Very little wastage of CPU cycles at high system loads
 - At low system loads, more receiver initiated polls do not cause system instability as spare CPU cycles are available
- **Drawback:**
 - Most transfers are preemptive and therefore expensive

Symmetrically Initiated Algorithms

- In symmetrically initiated algorithm, both senders and receivers search for receivers and senders, respectively, for task transfers.
- These algorithms have the advantages of both sender and receiver initiated algorithms.
- At low system loads, the sender-initiated component is more successful in finding under-loaded nodes. At high system loads, the receiver-initiated component is more successful in finding overloaded nodes.
- As in sender-initiated algorithms, cause system instability at high system loads.
- Receiver-initiated algorithms, a preemptive task transfer is necessary.

Symmetrically Initiated Algorithms

- **Transfer Policy:**
 - Threshold policy which uses two adaptive thresholds
 - Thresholds (two) are equidistant from the **node's estimate** of the **average load across all nodes**
- Example
 - Node's estimate of the average load = 2
 - Lower threshold = 1
 - Upper threshold = 3
- A node whose load is greater than upper threshold → a sender
- A node whose load is less than lower threshold → a receiver.
- Nodes that have loads between these thresholds lie within the acceptable range, so they are neither senders nor receivers.

Symmetrically Initiated Algorithms

- **Location Policy:** It has two components
 - Sender-Initiated Components
 - Receiver Initiated Components
- **Sender-initiated component:**
 - **Sender:**
 - Broadcast TooHigh message
 - Set TooHigh timeout alarm
 - Listen for an Accept message until timeout expires
 - **Receiver:**
 - Receive TooHigh message
 - Send Accept message to sender
 - Sets AwaitingTask timeout alarm
 - Increases its load value before accepting a task. Why????
 - If Awaiting Task timeout expires without the arrival of transferred task , the load value at the receiver is decreased.

Symmetrically Initiated Algorithms

- On receiving an *Accept* message, if the node is still a sender, it chooses the best process to transfer and transfer it to the node that responded.
- On expiration of TooHigh timeout, if no Accept message is received,
 - Sender guesses that its estimate of the average system load is too low (since no node has a load much lower)
 - Hence, it broadcasts a ChangeAverage message to increase the average load estimate at the other nodes.

Symmetrically Initiated Algorithms

- Receiver-initiated component
 - broadcasts a TooLow message
 - Sets a TooLow timeout alarm
 - listening for a TooHigh message
- If a TooHigh message is received, the receiver performs the same actions that it does under sender-initiated negotiation
- If the TooLow timeout expires before receiving any TooHigh message, the receiver broadcasts a ChangeAverage message to decrease the average load estimate at the other nodes

Symmetrically Initiated Algorithms

- **Selection policy:** This algorithm can make use of any of the approaches discussed earlier.
- **Information policy:** The information policy is demand-driven.
- **Key Points:**
 1. The average system load is determined individually at each node, imposing little overhead.
 2. Acceptable range determines the responsiveness of the algorithm.

Remote Procedure Call (RPC)

History

- 1984: Birrell & Nelson
 - Mechanisms to call procedures on other machines
 - Processes on machine A can call procedures on machine B
 - A is suspended
 - Execution continues on B
 - When B returns, control passed back to A
 - Goal: it appears to the programmer that a normal call is taking place

Introduction

- The Remote Procedure Calls (RPC) mechanism is a high-level communications paradigm for network applications.
- RPC is a powerful technique for constructing distributed, client/server based applications.
- It is based on extending the notion of conventional, or local procedure calling,
 - so that the **called procedure** need not exist in the same address space as the **calling procedure**.
 - The two **processes** may be on the same system, or they may be on different systems with a network connecting them.

Introduction

- By using RPC, programs on networked platforms can communicate with remote (and local) resources.
- RPC allows network applications to use specialized kinds of procedure calls designed to **hide** the details of underlying networking mechanisms.

Procedure

- Same as *routine*, *subroutine*, and *function*. A procedure is a section of a *program* that performs a specific task.
- An ordered set of tasks for performing some action.

Terms and Definitions

- **Client-** A process, such as a program or task, that **requests a service** provided by another program.
- **Server-** A process, such as a program or task, that **responds to requests** from a client.
- **Client Stub-** Module within a client application containing all of the functions necessary for the client to make remote procedure calls using the model of a traditional function call in a standalone application. The client stub is responsible for invoking the marshalling engine and some of the RPC application programming interfaces (APIs).
- **Server Stub -** Module within a server application or service that contains all of the functions necessary for the server to handle remote requests using local procedure calls.

How RPC Works

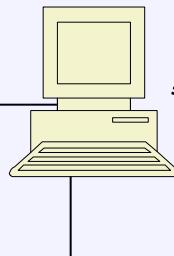
1. An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.
2. The client makes a procedure call that sends a request to the server and waits.
3. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a local procedure that performs the requested service, and sends the reply to the client.
4. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

How RPC Works

1. one procedure (**caller**) calls another procedure (**callee**)
2. the **caller waits** for the result from the callee
3. the **callee receives the request**, computes the results, and then **send them to the caller**
4. the **caller resumes** upon receiving the results from the callee

CALLER

invokes a procedure
on the callee



arguments

CALLEE

receives the request

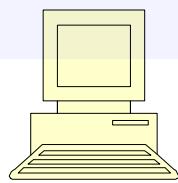
computes the results

sends the results

waits

resumes

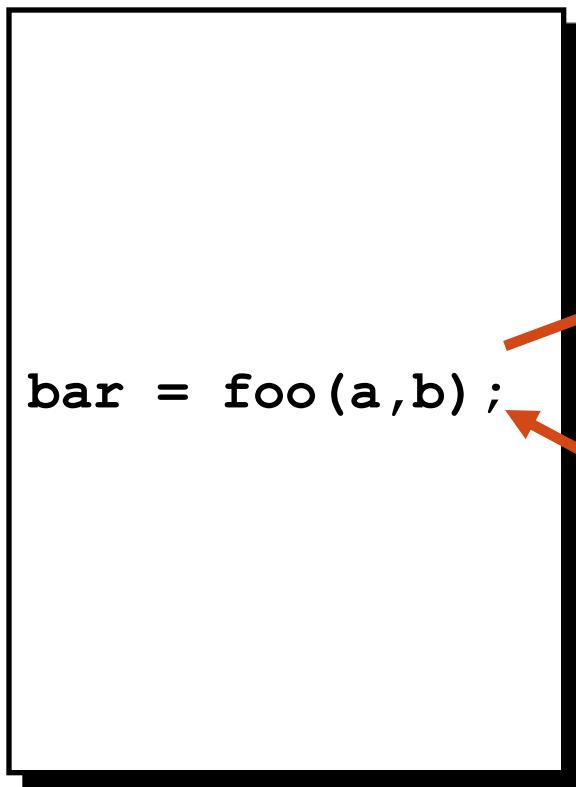
results



RPC call between two networked systems

Remote Subroutine

Client



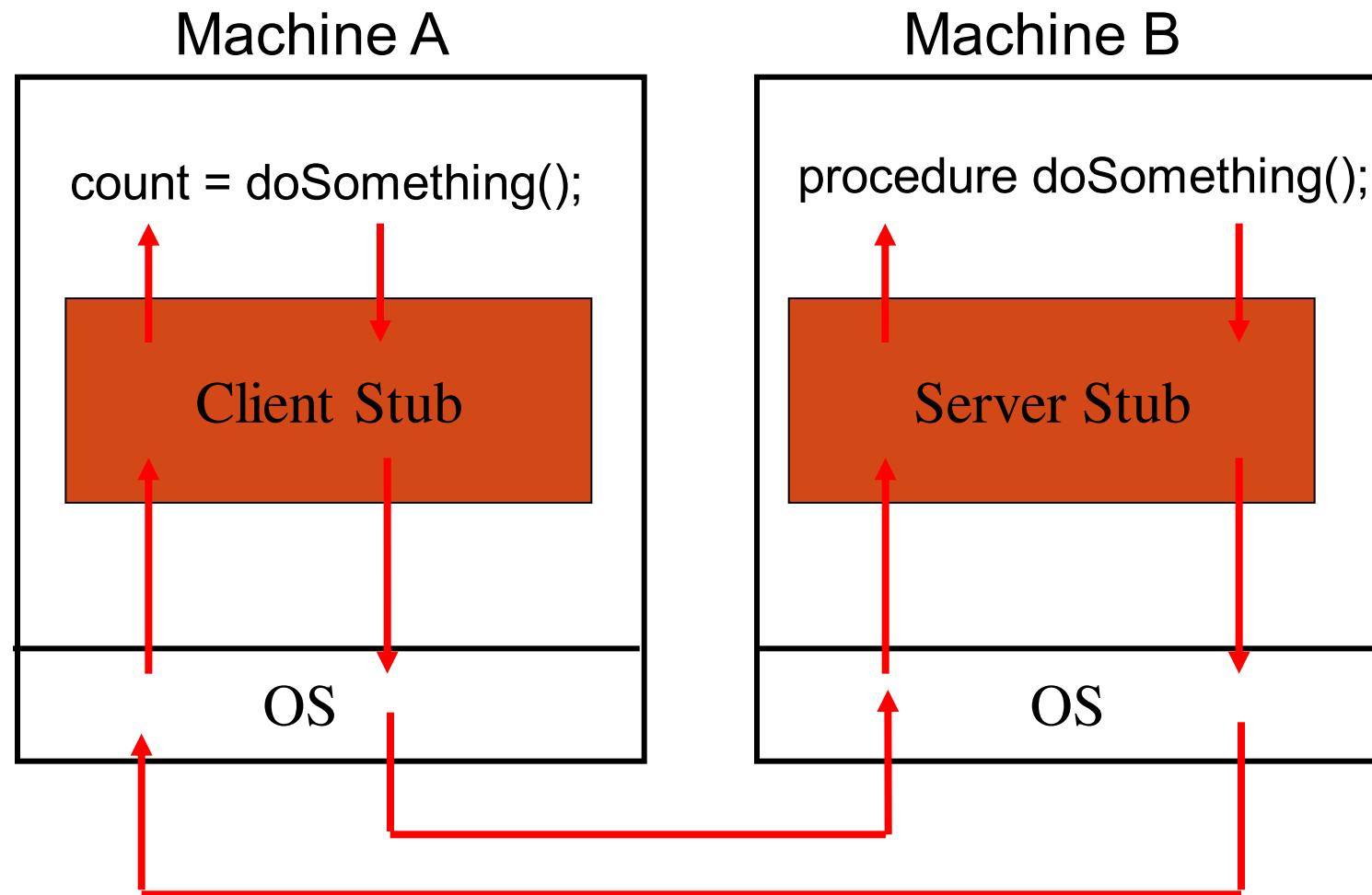
protocol

Server

An orange-bordered rectangle representing the Server. Inside, the C-like code for the subroutine `foo` is shown:

```
int foo(int x, int y) {  
    if (x>100)  
        return(y-2);  
    else if (x>10)  
        return(y-x);  
    else  
        return(x+y);  
}
```

Client and Server



Remote procedure call (RPC)

- A remote procedure call makes a call to a remote service look like a local call
 - RPC makes transparent whether server is local or remote
 - RPC allows applications to become distributed transparently
 - RPC makes architecture of remote machine transparent

Remote Procedure Call (RPC)

- The most common framework for newer protocols and for middleware
- Used both by operating systems and by applications
 - NFS is implemented as a set of RPCs
 - DCOM (**Distributed Component Object Model**) is a proprietary Microsoft technology for communication among software components distributed across networked computers.
 - CORBA (**Common Object Request Broker Architecture**) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms).
 - Java RMI (Java Remote Method Invocation) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked, possibly on different hosts. etc., are just RPC systems

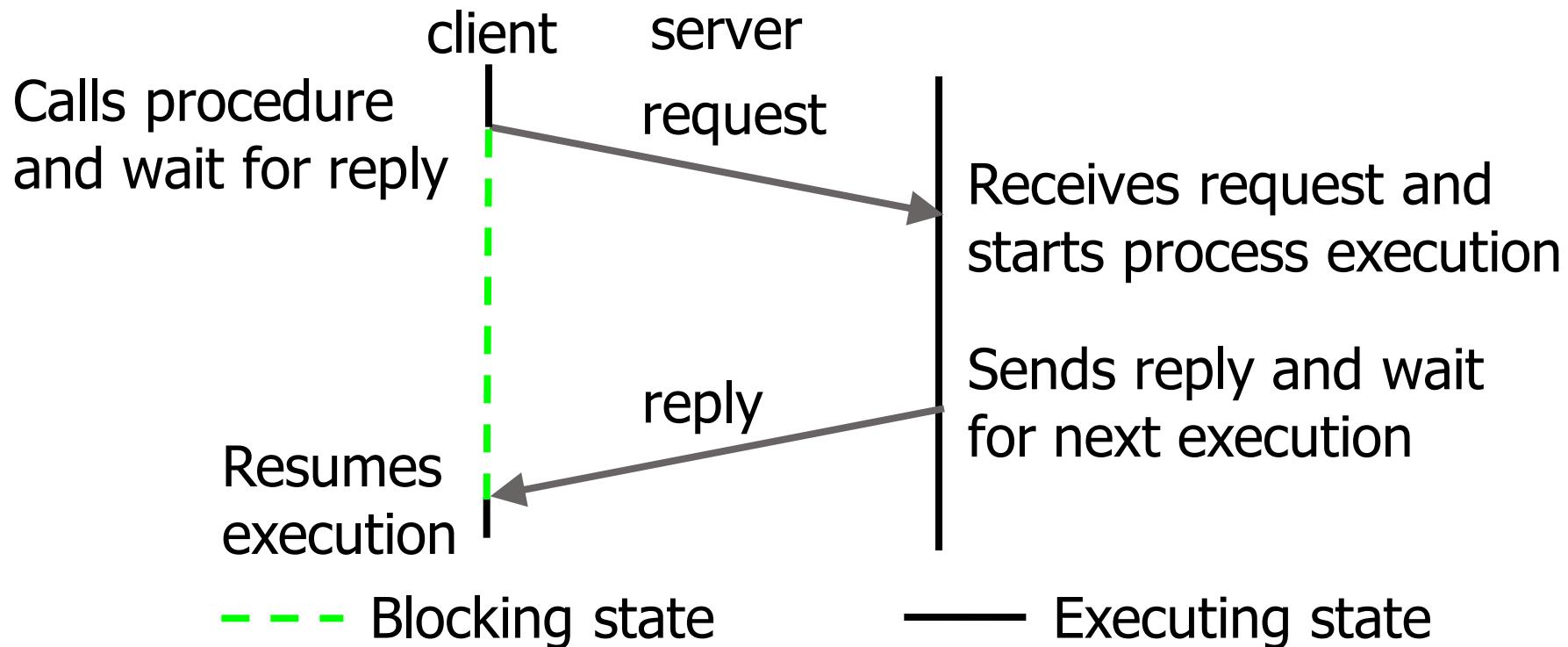
Remote Procedure Call (RPC)

- Fundamental idea: –
 - Server process exports an *interface* of procedures that can be called by client programs
 - Clients make local procedure/function calls
 - Under the covers, procedure/function call is converted into a message exchange with remote server process

Developing with RPC

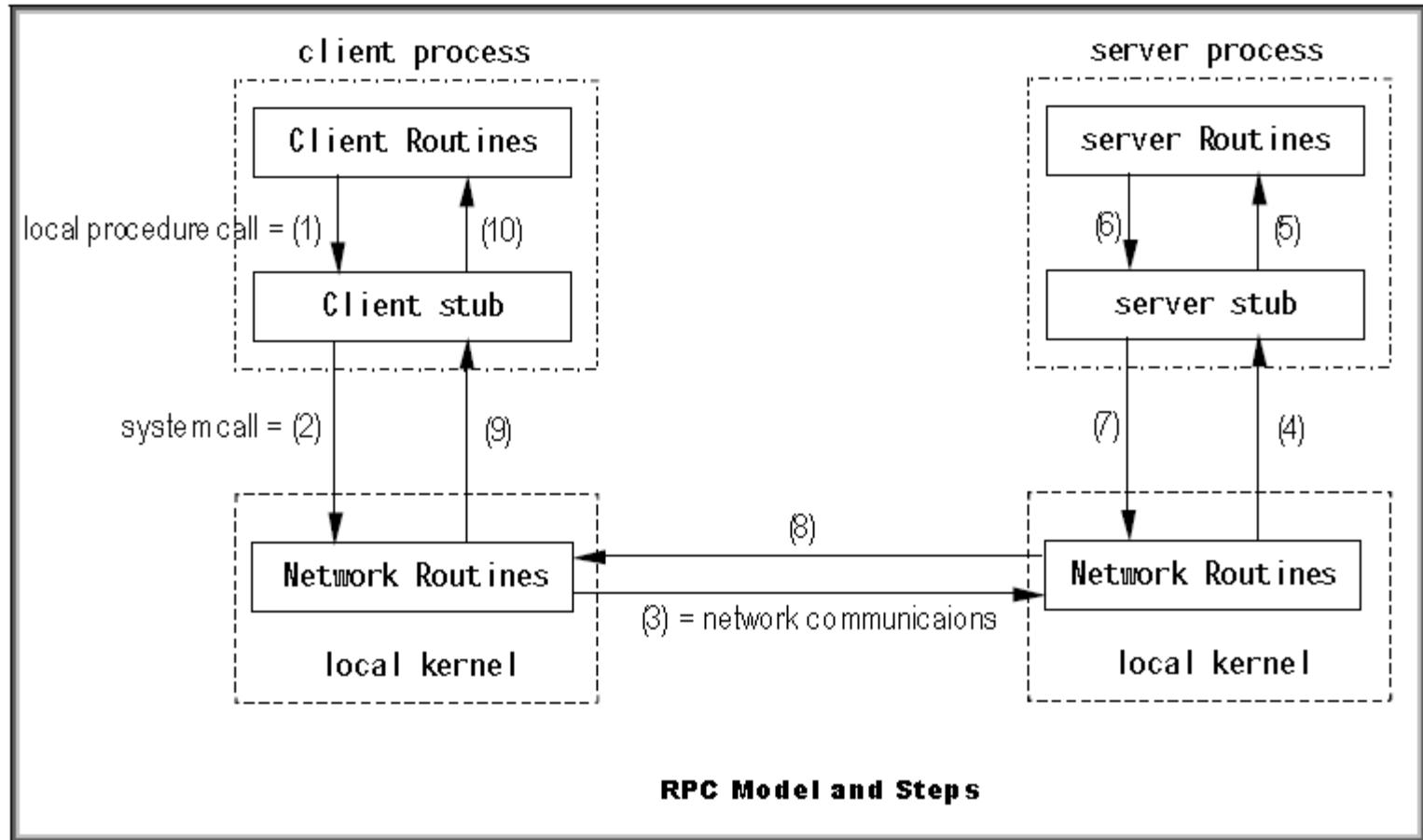
1. Define APIs between modules
 - Split application based on function, ease of development, and ease of maintenance
 - Don't worry whether modules run locally or remotely
2. Decide what runs locally and remotely
 - Decision may even be at run-time
3. Make APIs bullet proof
 - Deal with partial failures

The RPC model



RPC to be transparent- the calling procedure should not be aware that called procedure is executing on a different machine

RPC Execution Steps



Cont...

- (1) The **client** calls the procedure. It calls the client stub, which packages the arguments into network messages. Packaging is called **marshaling**.
- (2) The **client stub** executes a system call (usually write or sendto) into the local kernel to send the message.
- (3) The **kernel** uses the network routines (TCP or UDP) to send the network message to the remote host. Note that connection-oriented or connectionless protocols can be used.
- (4) A **server stub** receives the messages from the kernel and **unmarshals** the arguments.
- (5) The **server stub** executes a local procedure call to start the function and pass the parameters.
- (6) The **server process** executes the procedure and returns the result to the server stub.
- (7) The **server stub** marshals the results into network messages and passes them to the kernel.
- (8) The **messages** are sent across the network.
- (9) The **client stub** receives the network messages containing the results from the kernel using read or recv.
- (10) The **client stub** unmarshals the results and passes them to the client routine.

Stubs

- Client stub
 - Marshals arguments into machine-independent format
 - Sends request to server
 - Waits for response
 - Unmarshals result and returns to caller
- Server stub
 - Unmarshals arguments and builds stack frame
 - Calls procedure
 - Server stub marshalls results and sends reply

Marshalling Arguments

- *Marshalling* is the **packing of function parameters** into a message packet
 - The RPC stubs call functions to marshal or unmarshal the parameters
 - Client stub marshals the arguments into a message
 - Server stub unmarshals the arguments and uses them to invoke the service function (to execute the procedure locally)
 - on return:
 - the server stub marshals return values
 - the client stub unmarshals return values, and returns to the client program

Marshalling

- Argument encoding sometimes called marshalling
- Decoding -> unmarshalling
- issues

Data types

Conversion

Representation of data

- In a large distributed system, multiple machine types are present
- Each machine has its own representation for number, characters, and others data items.

3	2	1	0
0	0	0	5
7	6	5	4
L	L	I	J

(a)

0	1	2	3
5	0	0	0
4	5	6	7
J	I	L	L

(b)

0	1	2	3
0	0	0	5
4	5	6	7
L	L	I	J

(c)

- Original message on the Pentium (little-endian)
- The message after receipt on the Sun SPARC (big-endian)
- The message after being inverted. The little numbers in boxes indicate the address of each byte

Representation of Data (continued)

- The Sun RPC uses a standard format called **XDR**(eXternal Data Representation), a protocol for the machine-independent description and encoding of data.
- XDR is an data abstraction needed for machine independent communication. The client and server need not be machines of the same type.
- XDR is useful for transferring data between different computer architectures.
- RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to XDR representation before sending them over the wire.
- The process of **converting** from a particular machine representation to XDR format is called **serializing**, and the reverse process is called **deserializing**.

Semantics

- Call-by-reference not possible: the client and server don't share an address space. That is, addresses referenced by the server correspond to data residing in the client's address space.
- One approach is to simulate call-by-reference using *copy-restore*. In copy-restore, call-by-reference parameters are handled by sending a copy of the referenced data structure to the server, and on return replacing the client's copy with that modified by the server.
- However, copy-restore doesn't work in all cases. For instance, if the same argument is passed twice, two copies will be made, and references through one parameter only changes one of the copies.

RPC Binding

- How does the client know who to call, and where the service resides?
- The most flexible solution is to use *dynamic binding* and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

RPC Binding

- Binding is the process of **connecting the client to the server**
 - the server, when it starts up, exports its interface
 - identifies itself to a *network name server*
 - tells *RPC runtime* that it is alive and ready to accept calls
 - the client, before issuing any calls, imports the server
 - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Basic Concepts

interface

```
NAME      | type | type | type  
procedure_X | arg1 | arg2 | ret1  
procedure_Y | arg1 | ret1  
procedure_Z | arg1 | ret1 | ret2
```

local program

program module

```
...  
call procedure_X(arg1,arg2,ret1)  
...
```

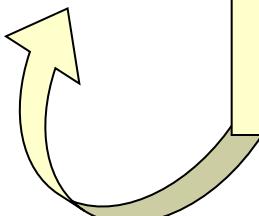
import
procedure_X

remote program

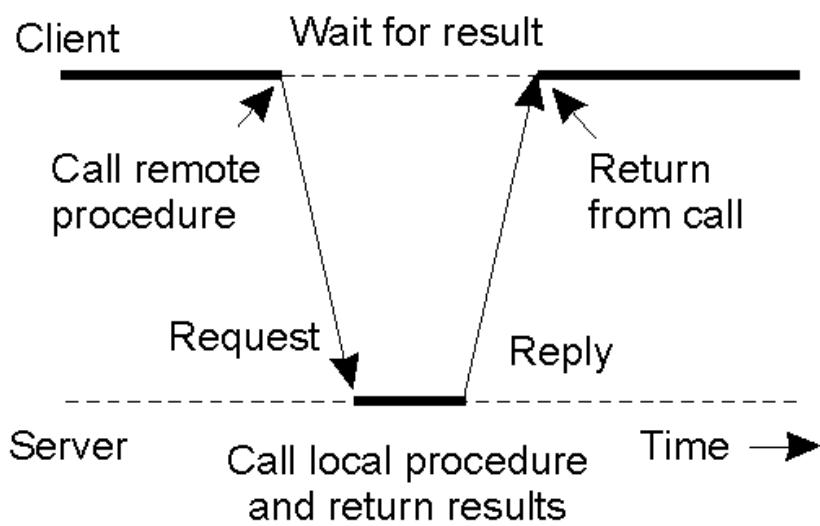
program module

```
implement  
procedure_X(arg1,arg2,ret1)  
{  
...  
}
```

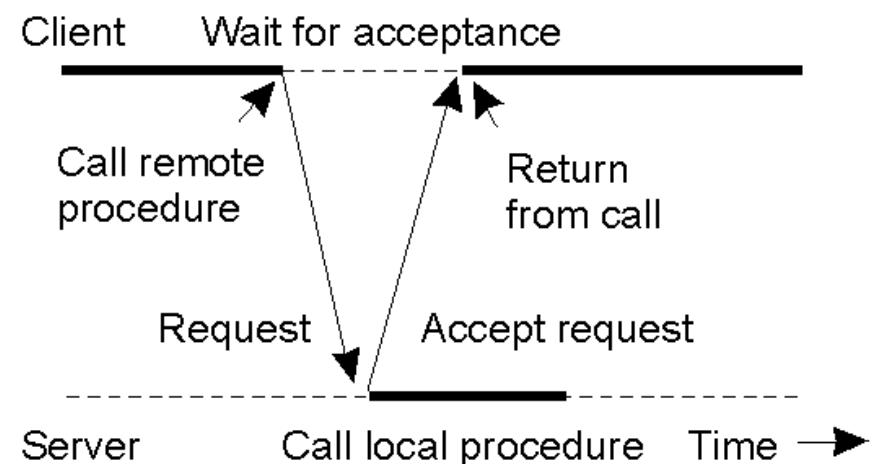
export
procedure_X



Asynchronous RPC



(a)



(b)

- a) The interconnection between client and server in a traditional RPC
- b) The interaction using asynchronous RPC

Cont ...

- Asynchronous RPC resolves the following limitations of traditional, synchronous RPC
- Slow or delayed clients
- Slow or delayed servers
- Transfer of large amounts of data

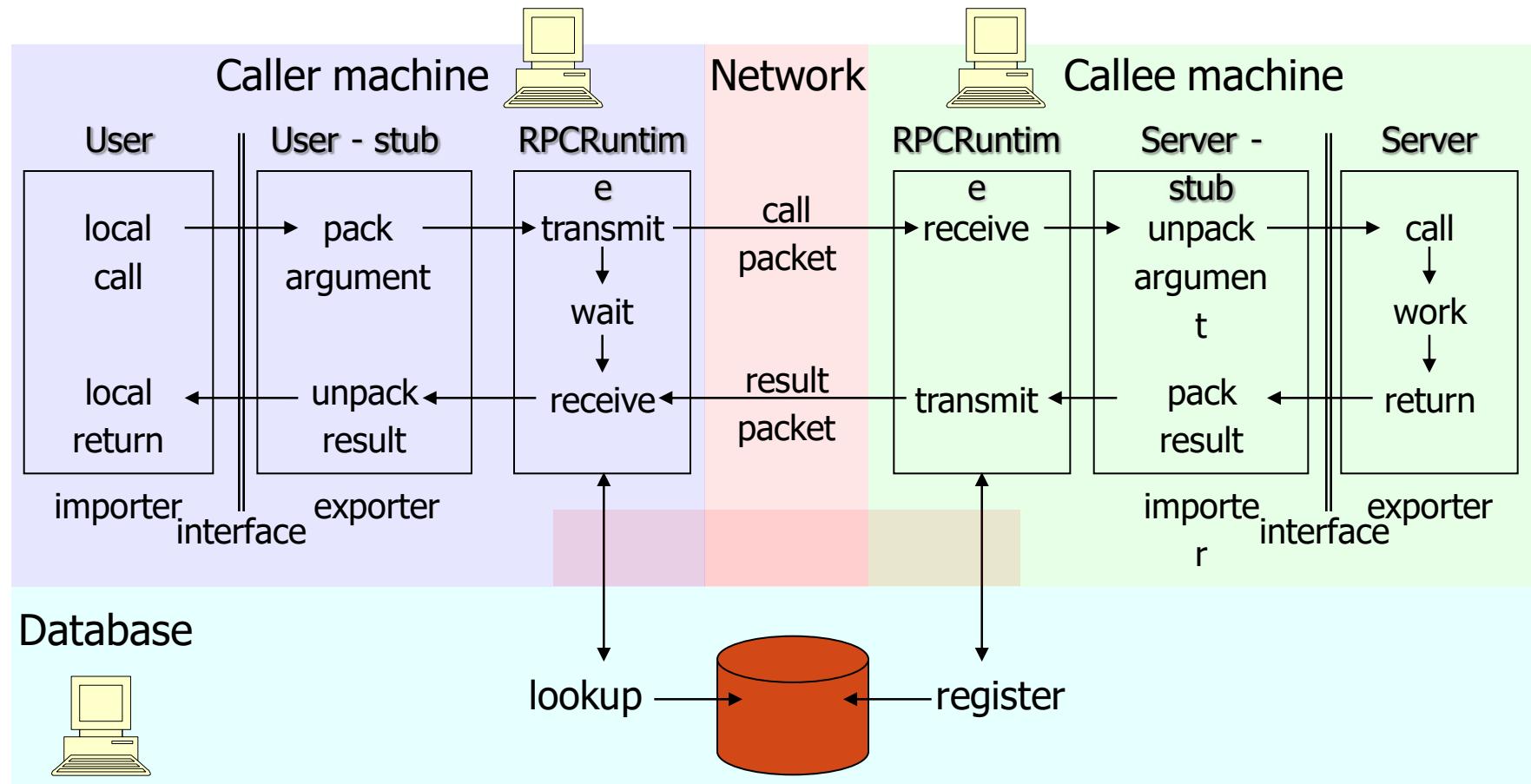
RPC Issues

- transparency-Applications should be prepared to deal with RPC failure
- **RPC is more vulnerable to failure**
 - ❖ machine failure (Server crash/Client crash while server is still running code for it)
 - ❖ communication failure
- address based arguments
 - ❖ the address space is not shared
- programming integration
 - ❖ integration into programming environment
- data integrity

RPC Issues

- data transfer protocols
 - ❖ network protocols
- binding
 - ❖ caller determines
 - the location
 - the identity
 - ❖ ... of the callee
- security
 - ❖ open communication network
 - ❖ Authenticate client
 - ❖ Authenticate server

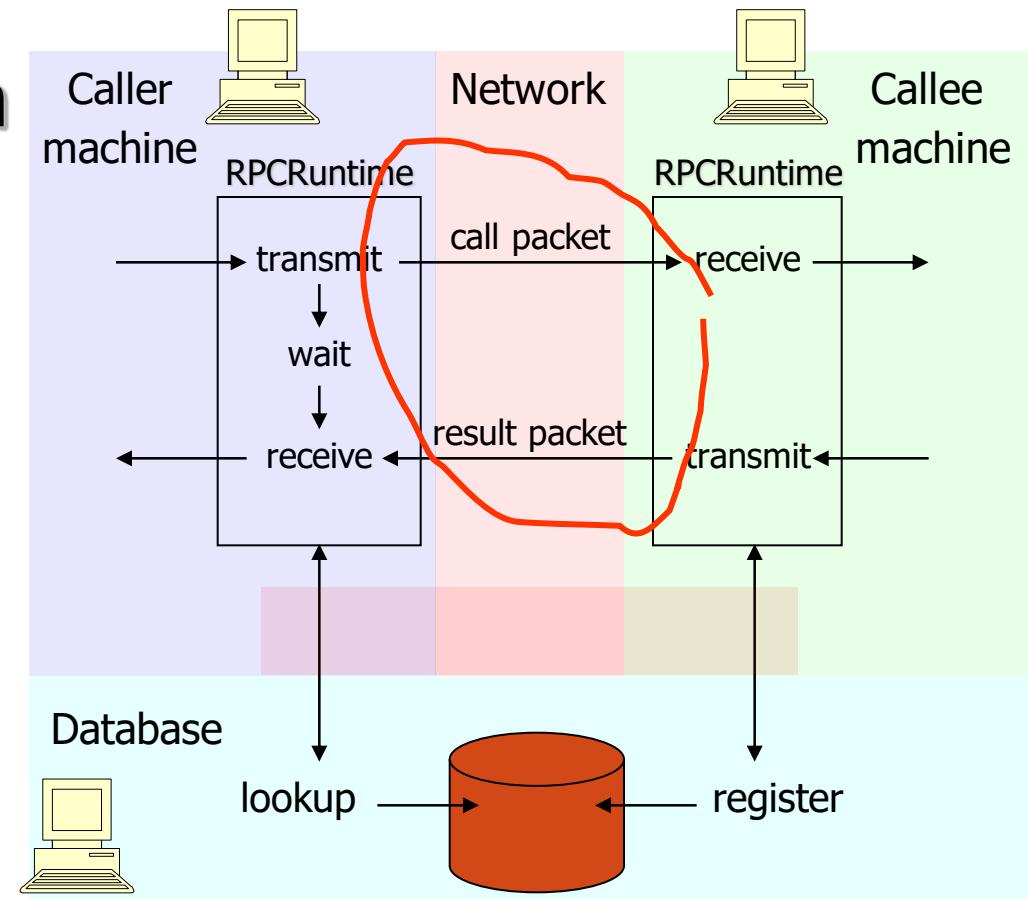
RPC Implementation – Overview



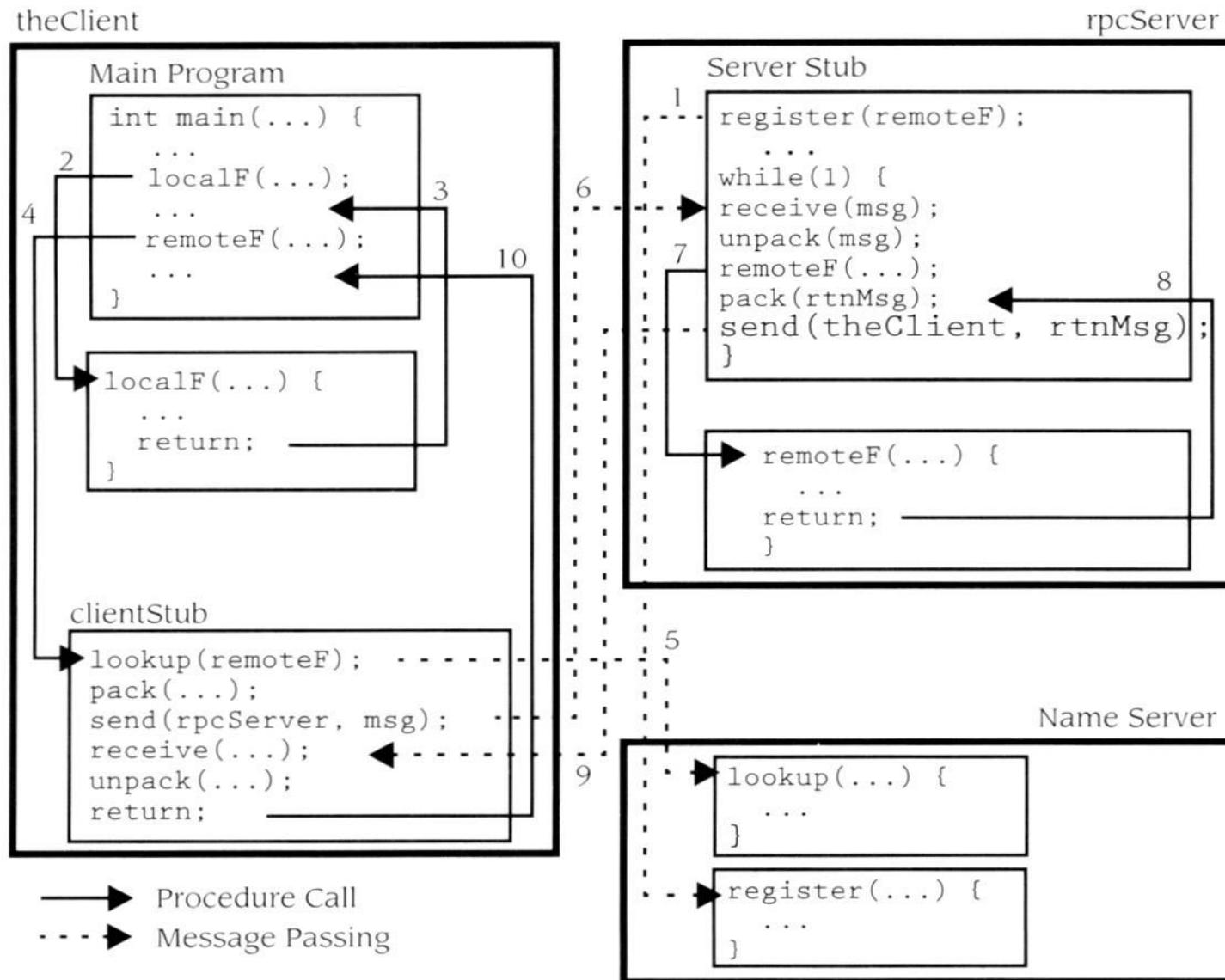
RPC Implementation – RPCRuntime

RPCRuntime deals with

- data (re)transmission
- data acknowledge
- packet routing
- encryption
- binding



RPC Implementation – RPC flow



Failures with RPC

- Failure types may be:
 1. The client is unable to locate the server
 - The server is not available
 - The stubs versions are not adequate
 2. The client request towards the server is lost:
 - After a timeout, the client retransmits the same request
 3. The server response from the server to the client is lost:
 - The client will retransmit the same request.

Failures with RPC (contd..)

4. The server is down after receiving the client request (3 cases):

- At least once:
 - When the machine restarts, retransmits the requests many times until the response arrives.
- At most once:
 - The operation is done either once or never.
- Exactly once:
 - The most desirable kind of semantics, where every call is carried out exactly once, no more and no less.
 - Unfortunately, such semantics cannot be achieved at low cost; if the client transmits a request, and the server crashes, the client has no way of knowing whether the server had received and processed the request before crashing.

RPC Application Development

To develop an RPC application the following steps are needed:

- Specify the protocol for client server communication
- Develop the client program
- Develop the server program

The programs will be compiled separately. The communication protocol is achieved by generated stubs and these stubs and rpc (and other libraries) will need to be linked in.

The *rpcgen* Protocol Compiler

- The easiest way to define and generate the protocol is to use a **protocol compiler** such as **rpcgen**.
- **rpcgen** provides programmers a simple and direct way to write distributed applications.
- **rpcgen** is a compiler. It accepts a remote program interface definition written in a language, **RPC Language**, which is similar to C.

Contd...

- It produces a C language output which includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions.
- The client stubs interface with the RPC library and effectively hide the network from their callers.
- The server stub similarly hides the network from the server procedures that are to be invoked by remote clients.

Contd...

The output of rpcgen is:

- A header file of definitions common to the server and the client
- A set of XDR routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

RPC specification

- A file with a ``.x" suffix acts as a remote procedure specification file. It defines **functions** that will be remotely executed.
- Multiple functions may be defined at once. They are numbered from one upward, and any of these may be remotely executed.
- The specification defines a program that will run remotely.
- The program has a name, a version number and a unique identifying number (chosen by you).

RPC versions and numbers

- Each RPC procedure is uniquely identified by a *program number, version number, and procedure number*.
- The program number identifies a group of related remote procedures, each of which has a different procedure number.
- Program numbers are given out in groups of hexadecimal 20000000.
 - 0 - 1fffffff defined by Sun,
 - 20000000 - 3fffffff defined by user
- Version numbers are incremented when functionality is changed in the remote program.
- More than one version of a remote program can be defined and a version can have more than one procedure defined.

Converting Local Procedures into Remote Procedures (an example)

- Assume an application that runs on a single machine. Suppose we want to convert it to run over the network. [a simple example program that calculates pi]
- To declare a remote procedure, first, determine the data types of all procedure-calling arguments and the result argument.

Contd...

- Writing protocol specification in RPC language to describe the remote version of calcu_pi.

- /* pi.x: Remote pi calculation protocol */

```
program PIPROG {  
    version CALCULATORS {  
        double CALCULATORS() = 1;  
    } = 1;  
} = 0x39876543;
```

The program PIPROG is 0x39876543, Version number referred to symbolically as CALCULATORS is 1.

Contd...

In this example,

- CALCU_PI procedure is declared to be:
 - the procedure 1,
 - in version 1 of the remote program
- PIPROG, with the program number 0x39876543.
- Notice that the **program and procedure names** are declared with all capital letters. This is not required, but is a good convention to follow.

Contd...

- To compile a **.x** file using

rpcgen -a -C pi.x

where:

option **-a** tells rpcgen to generate all of the supporting files

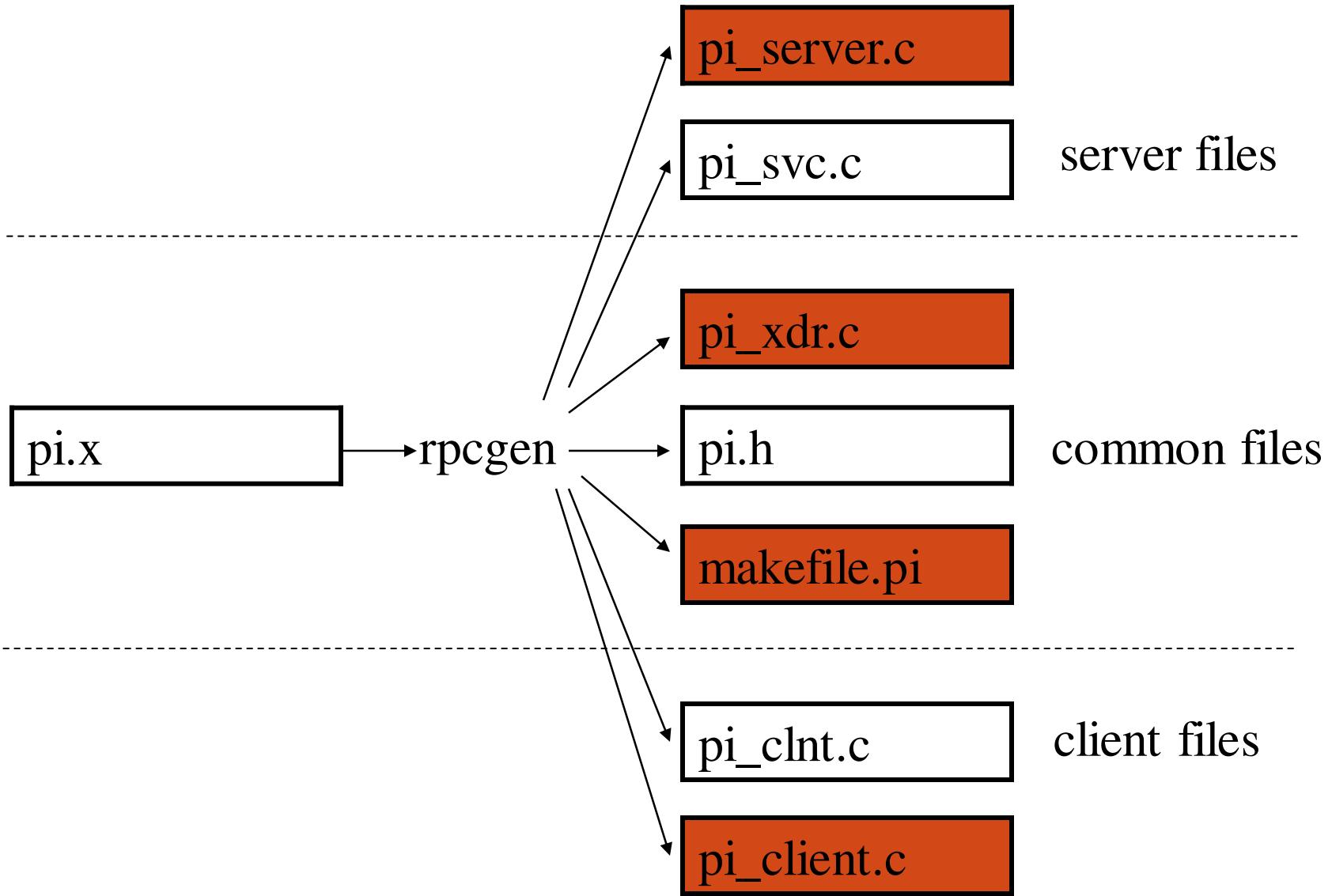
option **-C** indicates ANSI C is used

Contd....

This will generate the files:

- pi_clnt.c - the client stub - usually is not modified
- pi_svc.c - the server stub - usually is not modified
- pi.h - the header file that contains all of the XDR types generated from the specification
- makefile.pi - makefile for compiling all of the client and server code
- pi_client.c - client skeleton, need to be modified
- pi_server.c – server skeleton, need to be modified
- pi_xdr.c- Contains XDR filters needed by the client and server stubs – usually is not modified

rpcgen Files



Contd...

- Make changes to pi_client.c file to give actual arguments to call(if required) and to display the output. then save the changes and run the following command.

Compile the client side program and link it with client stub:

```
$ gcc pi_client.c pi_clnt.c -o pi_client -lnsl
```

- Write the logic at the location indicated and save the changes. Now, run the last command.

Compile the server side program and link it with server stub:

```
$ gcc pi_server.c pi_svc.c -o pi_server -lnsl
```

Contd....

How to run?

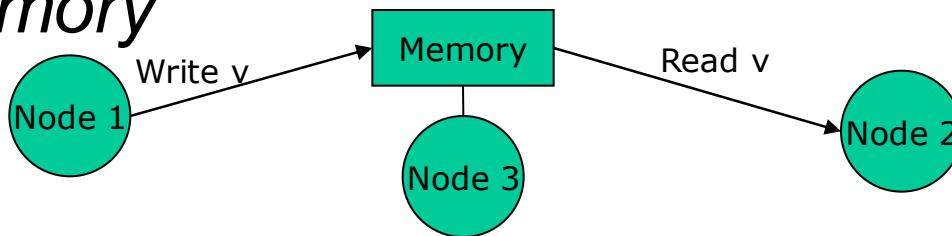
- Copy the server executable program pi_server to the remote machine and run.
- Run the client executable program pi_client on local host with the server host name.
- After running the server we can get the desired output.

- rpc-application development.doc

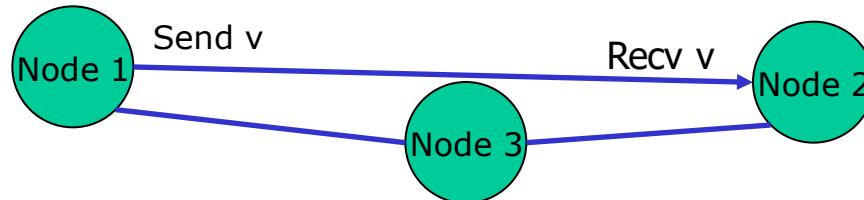
Distributed Shared Memory

There are 2 Ways of Information Sharing

- Using a *Shared Memory*
 - A node **writes** information in the *memory*
 - Another node **reads** information from the *memory*



- Using *Message Passing*
 - A node **sends** a *message* to another node
 - The second node **receives** the *message* from the other

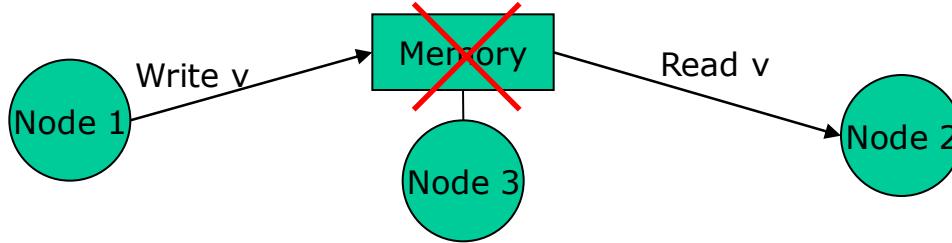


Shared Memory is Easier to Use

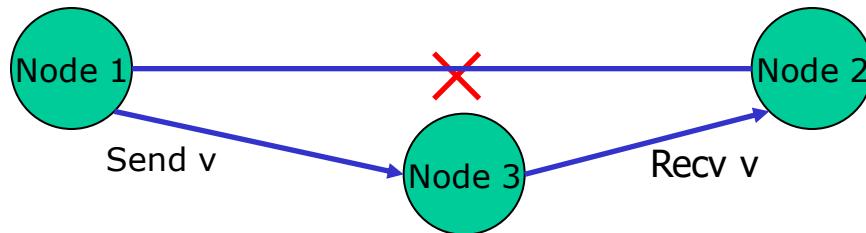
- *Shared Memory* is **easy** to use
 - If information is **written**, collaboration progresses!
- *Message Passing* is **difficult** to use
 - To which node the information should be **sent**?

Message Passing Tolerates Failures

- *Shared Memory* is **failure-prone**
 - Communication trusts on memory availability



- *Message-Passing* is **fault-tolerant**
 - As long as there is a way to route a message



DSM Introduction

- DSM provides a virtual address space shared among processes on loosely coupled processors.

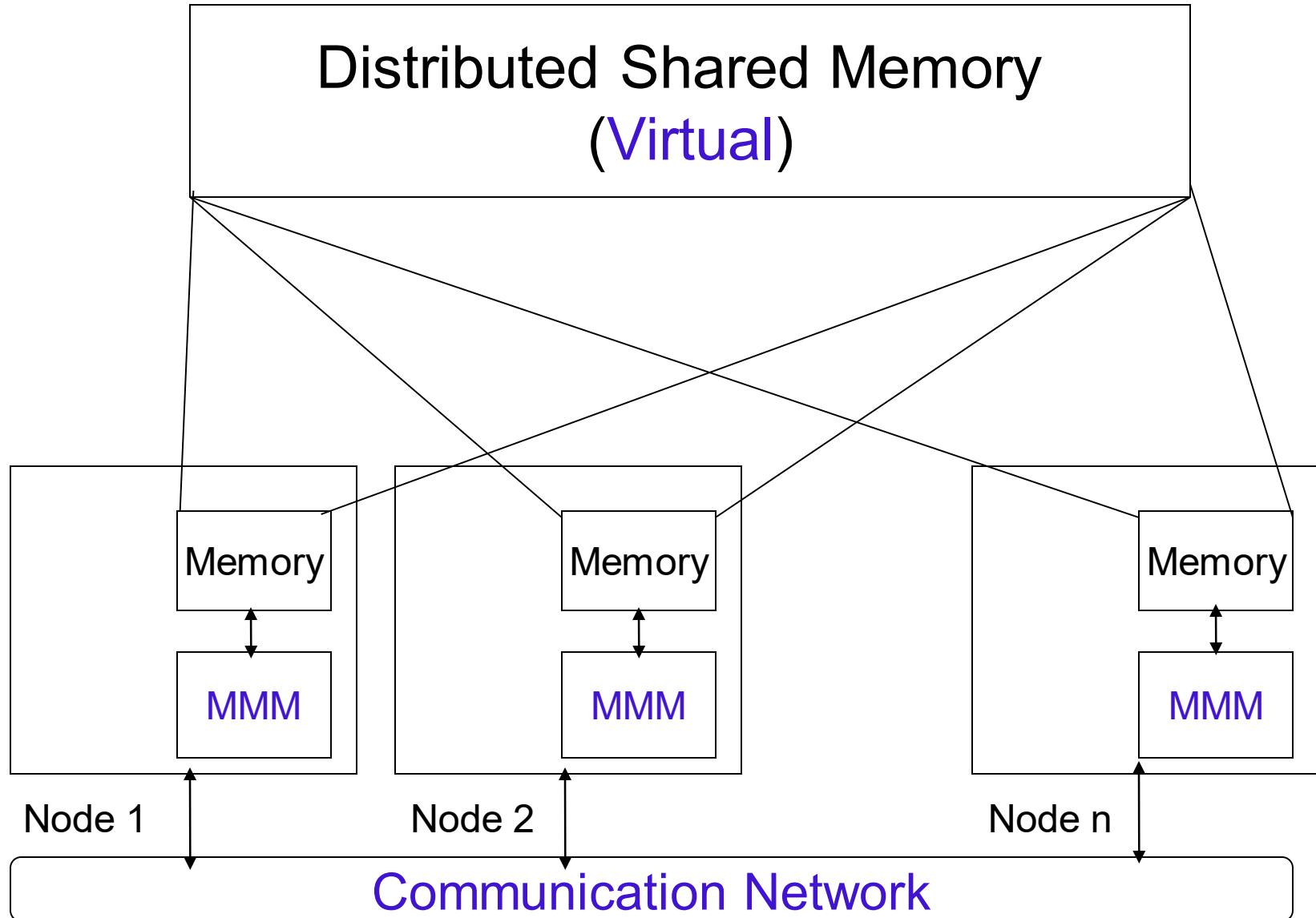
Formal Definition:

A Distributed Shared Memory System is a pair (P, M) where P is a set of N processors $\{ P_1, P_2, P_3, \dots, P_n \}$ and M is a shared memory.

Each process P_i sequentially executes read and write operations on data items in M in the order defined by the program running on it.

- It is an abstraction that integrates the local memory of different machines in a network environment into a single logical entity shared by cooperating processes executing on multiple sites.
- Due to the virtual existence of the share memory, DSM is sometimes also referred to as Distributed Shared virtual Memory (DSVM).

DSM Introduction



DSM Introduction

- To facilitate memory operation, the shared memory space is partitioned into blocks.
- When a process on a node accesses some data from a memory block of the shared memory space, the local memory manager handles the request.
- Variation of this general approach are used in different implementations depending on whether the DSM system allows replication and/or migration of shared-memory data blocks.

Design and Implementation Issues of DSM

Granularity

- **Granularity** refers to the block size of a DSM system , the unit of sharing and the unit of data transfer across the network.
- Possible units are a few words, a page, or a few pages.

Factors influencing block size selection

Other factors that influence the choice of block size are described below.

- **Paging overhead:** a process is likely to access a large region of its shared address space in a small amount of time. So paging overhead is less for large block sizes as compared to the paging overhead for small block sizes.

Granularity

- **Directory Size:** Information regarding blocks is maintained in directory. Obviously, the larger the block size, the smaller the directory which results in reduced directory management overhead.
- **Thrashing:** When data items in the same data block are being updated by multiple nodes at the same time, no real work can get done. A DSM must use a policy to avoid this situation (usually known as thrashing).
- **False Sharing:** It occurs when two different processes access two unrelated variables that reside in the same data block. The larger is the block size, the higher is the probability of false sharing.

Structure of shared memory space

- Structure defines the abstract view of the shared memory space to be presented to the application programmers of a DSM system.
- It may appear to its programmers as storage for words or a storage for data objects.

Approaches to structure shared memory space:

1. No Structuring

- Most DSM systems do not structure their shared-memory space.
- In these systems Shared memory is simply a linear array of words.
- Advantage is that it is convenient to choose any suitable page size as the unit of sharing and may be used for all applications.

Structure of shared memory space

- So it is simple and easy to design such a DSM system.
2. **Structuring by data type**
- Shared memory space is structured either as **a collection of objects** or **as a collection of variables** in the source language.
 - Granularity in such a DSM system is an **object** or a **variable**.
 - Since the sizes of the objects and variables vary greatly, these DSM systems use **variable grain size** to match the size of the object/variable being accessed by the application.
 - Variable grain size complicates design and implementation of these DSM systems.

Structure of shared memory space

3. Structuring as a database

- Shared memory space is ordered as an **associative memory** called a **tuple space**, which is a collection of immutable tuples with typed data items in their fields.
- To perform updates, old data items in the DSM are replaced by new data items.

Consistency Models

- A consistency model basically refers to the degree of consistency that has to be maintained for the shared-memory data for the memory to work correctly for a certain set of applications.
- A consistency model is essentially a contract between the software and the memory. If the software agrees to obey certain rules, the memory promises to work correctly.
- Consistency requirements vary from application to application.
- Applications that depend on a stronger consistency model may not perform correctly if executed in a system that supports only weaker consistency model.
- If a system supports the stronger consistency model, then the weaker consistency model is automatically supported but the converse is not true.

Strict Consistency Model

- This is the strongest form of memory coherence.
- Value returned by a **read operation** on a memory address is always the same as the **value written** by the **most recent write operation** to that address. All writes become visible to all processes.
- Implementation of the strict consistency model requires the existence of an **absolute global time** so that memory read/write operations can be **correctly ordered** to make the meaning of “most recent” clear.
- Absolute synchronization of clocks of all the nodes of a distributed system is not possible.

Sequential Consistency Model

- All processes see the **same order** of all **memory access operations** on the shared memory.
- The exact order in which the memory access operations are interleaved does not matter.
- Example: If the three operations read (r1), write (w1), read (r2) are performed on a memory address in that order, any of the orderings (r1, w1, r2), (r1, r2, w1), (w1, r1, r2), (w1, r2, r1), (r2, r1, w1), (r2, w1, r1) **of three operations is acceptable provided all processes see the same ordering.**
- This model is weaker than that of the strict consistency model.

Sequential Consistency Model

- Disadvantage is that it does not guarantee that a read operation on a particular memory address always **returns the same value as written by the most recent write operation** to that address.
- It provides **one-copy/single-copy** semantics because all the processes sharing a memory location always see exactly the same contents stored in it. So it is acceptable by most applications.

Causal Consistency Model

- A shared memory system is said to support the causal consistency model if all **write operations** that are potentially related are seen by all processors in the same (correct) order.
- Write operations that are **not potentially causally** related may be seen by different processes in different orders.
- **Example:** if a write operation (w2) is causally related to another write opeation (w1), the acceptable order is (w1, w2) because the value written by w2 might have been influenced in some way by the value written by w1. (w2, w1) is not an acceptable order.

PRAM Consistency Model

- This model proves a weaker consistency semantics than the consistency models described so far.
- It only ensures that all **write operations** performed by a **single process** are seen by **all other processes in the order in which they were performed** as if all the write operations performed by a **single process are in a pipeline**.
- But Write operations performed by **different processes** may be seen by different processes in different order.
- Example: If **w11** and **w12** are two write operations performed by a process **p1** in that order, and **w21** and **w22** are two write operations performed by a process **p2** in that order, process **p3** may see them in the order **[(w11, w12), (w21,w22)]** and another process **p4** may see them in the order **[(w21, w22), (w11,w12)]**.

Processor Consistency Model

- It is similar to PRAM model with an additional restriction of **memory coherence**
- For any memory location all processes agree on the same order of all write operations to that location. In effect, **processor consistency** ensures that all write operations performed on the same memory location are seen by all processes in the same order.
- This means that for the earlier example, if w12 and w22 are write operations for writing to the same memory locations x, all processes must see them in the same order – w12 before w22 or w22 before w12.

Weak Consistency Model

- It is designed to take advantage of the following two characteristics **common to many applications**.
- **It is not necessary to show the change in memory done by every write operation to other processes.** The result of several write operations can be combined and send to other processes when they need it.

Weak Consistency Model

- For this, a DSM system that supports the weak consistency model uses a special variable called a synchronization variable.
- When a synchronization variable is accessed by a process, (1) all changes made to the memory by the process are propagated to other nodes,(when process exits the critical section) (2) all changes made to the memory by other processes are propagated from other nodes to the process's node (when process enters a critical section)

Release Consistency Model

- To overcome the disadvantage of weak consistency model, this is designed.
- It provides a mechanism to clearly tell the system whether a process is entering a critical section or exiting from a critical section so that the system can decide and perform only either the first or the second operation when a synchronization variable is accessed by a process.
- Two synchronization variables, acquire and release, are used.

Discussion of Consistency Models

- It is difficult to grade the consistency models based on **performance** because quite different results are usually obtained for **different applications**.
- Therefore, in the design of a DSM system, the choice of consistency model usually depends on several other factors, such as **how easy is it to implement**, **how much concurrency does it allow**, and **how easy is it to use**.
- *Strict consistency model is never used in the design of a DSM system because its implementation is practically impossible.*
- Sequential consistency model is commonly used.
- Others are the main choices in the weaker category.
- Last two provides better concurrency.

Advantages of DSM

- Simpler abstraction
- Better portability of distributed application programs
- Better performance of some applications
- Flexible communication environment
- Ease of process migration.

Distributed File Systems

Organization

- General concepts
 - Introduction
- NFS
- AFS
- CODA

Introduction

- The main purposes of using file in operating systems are:
 - *Permanent storage of information*
 - *Sharing the information:* A file can be created by one application and the shared with different applications.
- A file system is a **subsystem of an operating system** that perform file Management activities such as organization, storing, retrieval, naming, Sharing and protection of files.

What Distributed File Systems Provide

- Access to data stored at servers using *file system interfaces*
- What are the file system interfaces?
 - Open a file, check status of a file, close a file
 - Read data from a file
 - Write data to a file
 - Lock a file or part of a file
 - List files in a directory, create/delete a directory
 - Delete a file, rename a file, add a symlink (*is a special type of file that contains a reference to another file or directory in the form of an absolute or relative path*) to a file
 - etc

Why DFSs are Useful

- Data sharing among multiple users
- User mobility
- Location transparency
- Backups and centralized management

Cont...

■ Clients and Servers:

- Clients access files and directories that are provided by one or more file servers.
- File Servers provide a client with a file service interface and a **view** of the file system
- Servers allow clients to perform operations from the file service interface on the files and directories
- Operations: add/remove, read/write
- Servers may provide different views to different clients

Cont...

- A DFS provide three types of services, that can be considered as a Component of the system.
 - *Storage service*
 - *True file service*
 - *Name service. Also called directory service*

Desirable features

- A good distributed file system should have the following features.
- ***Transparency***:- Between the different types of transparency as previously detailed :
 - **Location**: a client cannot tell where a file is located, where the name of a file does not reveal any hint of the file's physical storage location.
 - Location transparent: location of the file doesn't appear in the name of the file
 - ex: /server1/dir1/file specifies the server but not where the server is located -> server can move the file in the network without changing the path
 - **Migration**: a file can transparently move to another server
 - **Replication**: multiple copies of a file may exist
 - **Concurrency**: multiple clients access the same file

Cont...

- **Flexibility:** In a flexible DFS, it must be possible to add or replace file servers.
- Also, a DFS should support multiple underlying file system types (e.g., various Unix file systems, various Windows file systems, etc.)

Cont...

- Reliability:
- Consistency: employing replication and allowing concurrent access to files may introduce consistency problems.
- Security: clients must authenticate themselves and servers must determine whether clients are authorized to perform requested operation.
Furthermore communication between clients and the file server must be secured.
- Fault tolerance: clients should be able to continue working if a file server crashes. Likewise, data must not be lost and a restarted file server must be able to recover to a valid state.

Cont...

- **Performance:** In order for a DFS to offer good performance it may be necessary to distribute requests across **multiple servers**.
- Multiple servers may also be required if the amount of data stored by a file system is very large.

Cont...

- **Scalability:** A scalable DFS will avoid centralized components such as a centralized naming service, a centralized locking facility, and a centralized file store.
- A scalable DFS must be able to handle an increasing number of files and users.
- It must also be able to handle growth over a geographic area (e.g., clients that are widely spread over the world), as well as clients from different administrative domains.

Cont...

- Another features are:

- *User mobility*
- *Simplicity and ease of use*
- *High availability*
- *High reliability*
- *Security*
- *Heterogeneity*

Accessing Remote Files

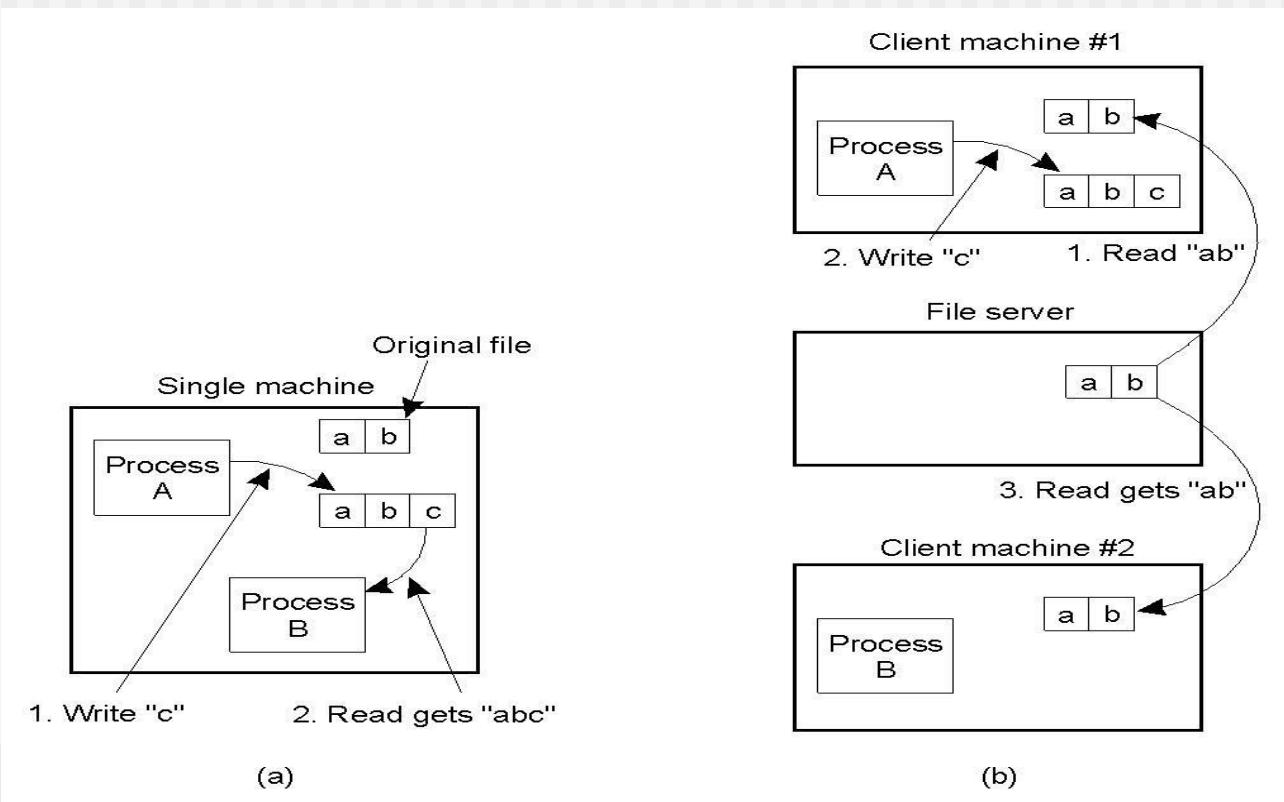
- There are several models to service a client's file access request when the accessed file is a remote file
 - *Remote service model*:- The client's request is performed at the server's node.
 - *Data-caching model* :-It attempts to reduce the amount of network traffic by taking advantage of the locality feature found in file access. The client's request is performed at the client's node itself by using the cached data. Compared with remote service model, this model greatly reduces network traffic.
- Almost all existing distributed file systems implement some form of caching.
- One problem, referred to as the *cache consistency problem* the cached data consistent with the original file content.

Units of Data Transfer

- In the systems that use the data-caching model, an important issue is to decide the unit of data transfer. The other models are:
 - *File-level transfer model*. In this model, when an operation requires file data to be transferred across the network in either direction between a client and a server, **the whole file is moved**.
 - *Block-level transfer model*. In this model, file data transfers across the network between a client and a server take place in **units of file blocks**.
 - *Byte-level transfer model*. In this model, file data transfers across the network between a client and a server take place in **units of bytes**.
 - *Record-level transfer model*. In this model, the transference unit is the **record**.

Semantics of File Sharing

- (a) On a single processor, when a *read* follows a *write* , the value returned by the *read* is the value just written.
- (b) In a distributed system with caching, absolute values may be returned.



Cont....

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

Four ways of dealing with the shared files in a distributed system.

Semantics of File Sharing . Unix

- Every operation on a file is instantly visible to all parties
- A Read following a Write will return the value just written
 - For all users of the file
- Enforces (requires) a total global order on all file operations to return most recent value
 - On a single physical machine this results from using a shared I-Node to control all file operations
 - File data is thus shared data structure among all users
- Distributed filesystem must replicate this behavior

Semantics of File Sharing . Unix

- Distributed UNIX Semantics
 - Could use a single centralized server which would thus serialize all file operations
 - Provides poor performance
- In the case of a DFS, it is possible to achieve such semantics if there is only a single file server and no client-side caching is used.
- In practice, such a system is unrealistic because caches are needed for performance and write-through caches are expensive.
- Furthermore deploying only a single file server is bad for scalability.
- Because of this it is impossible to achieve Unix semantics with distributed file systems.

Semantics of File Sharing . Session

- In the case of session semantics, changes to an open file are only locally visible.
- Only after a file is closed, are changes propagated to the server (and other clients).
- This raises the issue of what happens if two clients modify the same file simultaneously.
- It is generally up to the server to resolve conflicts and merge the changes.
- Another problem with session semantics is that parent and child processes cannot share file pointers if they are running on different machines.

Semantics of File Sharing . Inmutable files

- Immutable files cannot be altered after they have been closed.
- In order to change a file, instead of overwriting the contents of the existing file a new file must be created.
- This file may then replace the old one as a whole.
- This approach to modifying files does require that directories (unlike files) be updatable.
- Problems with this approach include a **race condition** when two clients try to replace the same file as well as the question of what to do with processes that are reading a file at the same time as it is being replaced by another process.

Semantics of File Sharing . Atomic Transactions

- In the transaction model, a sequence of file manipulations can be executed indivisibly, which implies that two transactions can never interfere.
- Changes are all or nothing
 - Begin-Transaction
 - End-Transaction
- System responsible for enforcing serialization
 - Ensuring that concurrent transactions produce results consistent with some serial execution
 - Transaction systems commonly track the read/write component operations
- Familiar aid of atomicity provided by transaction model to implementers of distributed systems
 - Commit and rollback both very useful in simplifying implementation
- This is the standard model for databases, but it is expensive to implement.

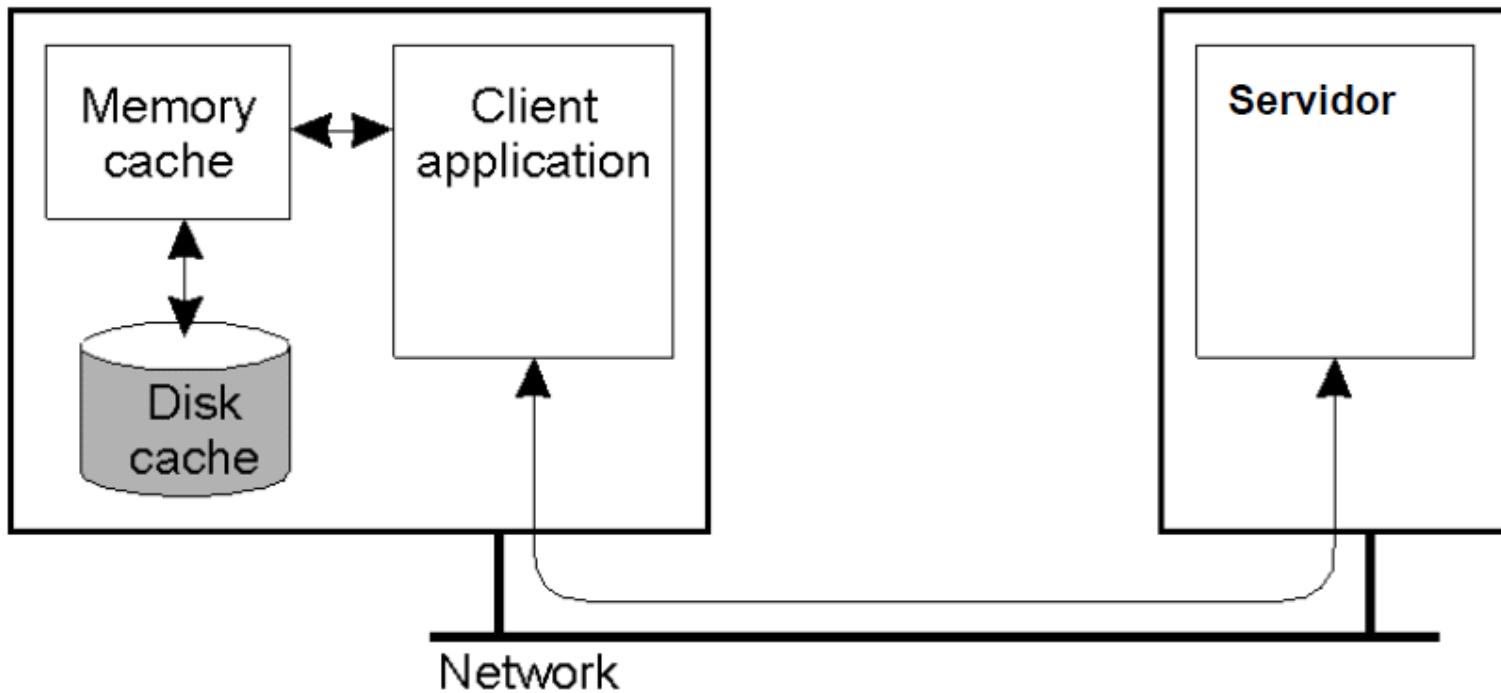
Caching

- Caching is often used to improve the performance of a DFS.
- In a DFS caching involves storing either a whole file, or the results of file service operations.
- Caching can be performed at two locations: **at the server and at the client**.
- Server-side caching makes use of file caching provided by the host operating system. This is transparent to the server and helps to improve the server's performance by **reducing costly disk accesses**.

Cont...

- Client-side caching comes in two flavours: on-disk caching, and in-memory caching.
- On-disk caching involves the creation of (temporary) files on the client's disk. These can either be complete files (as in the upload/download model) or they can contain partial file state, attributes, etc.
- In-memory caching stores the results of requests in the client-machine's memory. This can be process-local (in the client process), in the kernel, or in a separate dedicated caching process.

Caching...



cache hit-

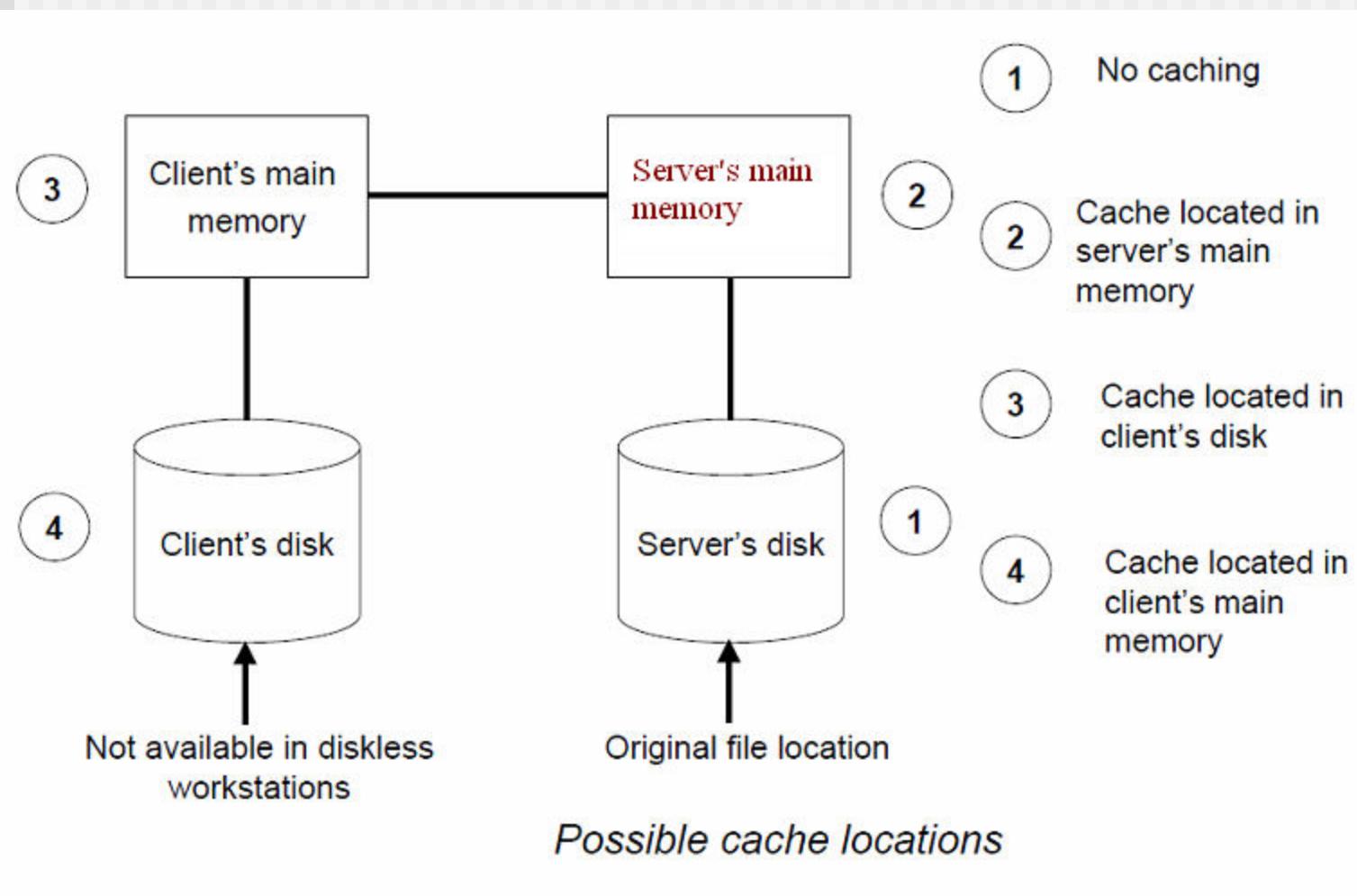
for example, a web browser program might check its local cache on disk to see if **it has a local copy of the contents of a web page at a particular URL**.

The percentage of accesses that result in cache hits is known as the **hit rate** or **hit ratio** of the cache.

Caching...

- regarding caching there are several key decisions as in distributed systems:
 - *Granularity of cached data* (large versus small)
 - *Cache size* (large versus small, fixed versus dynamically changing)
 - *Replacement policy*
- In distributed systems a file-caching scheme should also address the following key decisions:
 - *Cache location*
 - *Modification propagation*
 - *Cache validation*

Caching...



Caching...

Cache location	Access cost on cache hit
Server's main memory	One network access
Client's disk	One disk access
Client's main mermory	-----

Caching...

■ Modification propagation-

- The aim is keeping file data cached at multiple client nodes consistent.
- There are several approaches related with:
 - When to propagate modifications made to a cached data to the corresponding file server
 - How to verify the validity of cached data
- The modification propagation used has a critical effect on the system's performance and reliability.
- The file semantics supported depends greatly on the modification propagation scheme used.

Caching...

■ Write through Scheme.-

When a cache entry is modified, the new value is immediately sent to the server for updating the original copy of the file.

- **Advantages:** reliability and suitability for UNIX-like semantics.
- **Drawback:** write-through caches are too expensive to be useful, the consistency of caches will be damaged.

■ Delayed-Write Scheme.-

- The aim is to reduce network traffic for writes.
- where writes are not propagated to the server immediately, but in the background later on, and write-on-close where the server receives updates only after the file is closed.

Caching...

- Delayed-Write Scheme helps in performance improvement for write accesses due the following reasons:
 - Write accesses complete more quickly because the new value is written only in the cache , the client performing the write.
 - Adding a delay to write-on-close has the benefit of avoiding redundant writes if a file is deleted shortly after it has been closed.
 - Gathering of all file updates and sending them together to the server is more efficient than sending each update separately.
- However can be some reliability problems.
Modifications not yet send to the server from a client's cache will be lost if the client crashes.

Caching...

■ Cache Validation Schemes.-

- A file data may simultaneously reside in the cache of multiple nodes.
- The modification propagation policy only specifies when the master copy of a file at the server node is updated upon modification of a cache entry.

■ Client initiated approach.-

- Checking before every access. This approach defeats the main purpose of caching, but it is suitable for supporting **UNIX-like semantics**.
- Periodic checking. A check is initiated very fixed interval of time.
- Check on file open. With this option, a client's cache entry is validated only when the client opens the corresponding file for use. It is **suitable for supporting session semantics**.

■ Server initiated approach.-

- maintaining a record of which files are cached by which clients.

File Replication

- The main approach to **improving the performance and fault tolerance** of a DFS is to replicate its content.
- A replicating DFS maintains multiple copies of files on different servers.
- This can prevent **data loss, protect a system against down time of a single server, and distribute the overall workload.**

File Replication

- There are three approaches to replication in a DFS:
 - Explicit replication : The client explicitly writes files to multiple servers. This approach requires explicit support from the client and **does not provide transparency**.
 - Lazy file replication : The server automatically copies files to other servers after the files are written. Remote files are only brought up to date when the files are sent to the server. How often this happens is up to the implementation and **affects the consistency of the file state**.
 - Group file replication : write requests are simultaneously sent to a group of servers. This keeps all the replicas up to date, and allows clients to read consistent file state from any replica.

File Replication

■ Differences between Replication and Caching

- A replica is associated with a server, whereas a cached copy is normally associated with a client.
- The existence of a cached copy is primarily dependent on the locality in file access patterns, whereas the existence of a replica normally depends on availability and performance requirements.
- As compared to a cached copy, a replica is widely known, secure, available, complete and accurate.
- A cached copy is dependent upon a replica. Only by periodic revalidation with respect to a replica can a cached copy be useful

File Replication ...

- The possible benefits that offer the replication of data are:
 - Increased availability
 - Increased reliability
 - Improved response time
 - Reduced network traffic
 - Improved system throughput
 - Better scalability
 - Autonomous operation

Fault tolerance

- The fault tolerance is an important issue in the design of a distributed file system. The characteristics of that kind of systems make possible several fault situations.
- The primary file properties that directly influence the availability of a distributed system to tolerate faults are:
 - **Availability.** It is a reference about the fraction of the time for which the file is available for use. **Replication** is a mechanism for improving the availability of a file.
 - **Robustness.** It refers to its power to survive crashes of the storage device.
 - **Recoverability.** Ability to be rolled back to an earlier, consistent state when an operation on the file fails or is aborted by the client.

Distinctions between Stateful vs Stateless service

The file servers that implement a distributed file service can be stateless or stateful.

- Failure Recovery-
- stateful server
 - If a stateful server loses all its volatile state in a crash.
 - Restore state by recovery protocol based on a dialog with clients ,or abort operations that were underway when the crash occurred.
 - Server needs to be aware of client failures in order to reclaim space allocated to record the state of crashed client processes (orphan detection and elimination).
- stateless server
 - With stateless server ,the effects of server failures and recovery are almost unnoticeable.
 - The main advantage of stateless servers is that they can easily recover from failure. Because there is no state that must be restored, **a failed server can simply restart after a crash and immediately provide services to clients as though nothing happened.**
 - A newly restored server can respond to a self-contained request without any difficulty.

Distinctions between Stateful vs Stateless service

- Penalties for using the robust stateless service:
 - Longer request messages.
 - Slower request processing.
 - Difficulty in providing UNIX file semantics.
- Some environments require stateful service.
 - A server utilizing server-initiated cache validation cannot provide stateless service , since it maintains a record of **which files are cached by which clients**.
 - UNIX use of file descriptors and implicit offsets is inherently stateful ;**servers must maintain tables to map the file descriptors to inodes ,and store the current offset within a file.**

NFS Introduction

- NFS was originally developed by Sun for use on its UNIX-based workstations, but has been implemented for many other systems as well.
- The basic idea behind NFS is that each file server provides a standardized view of its local file system (each NFS server supports the same model).
- This approach allows a **heterogeneous collection of processes**, possibly running on different operating systems and machines, to share a common file system.

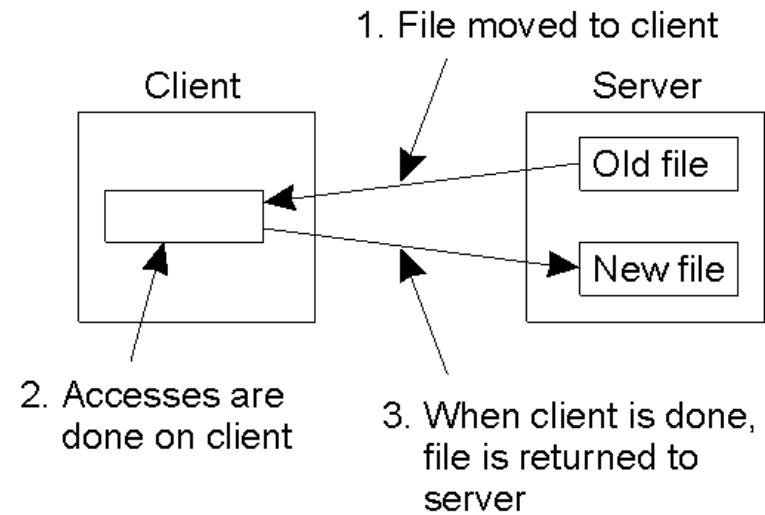
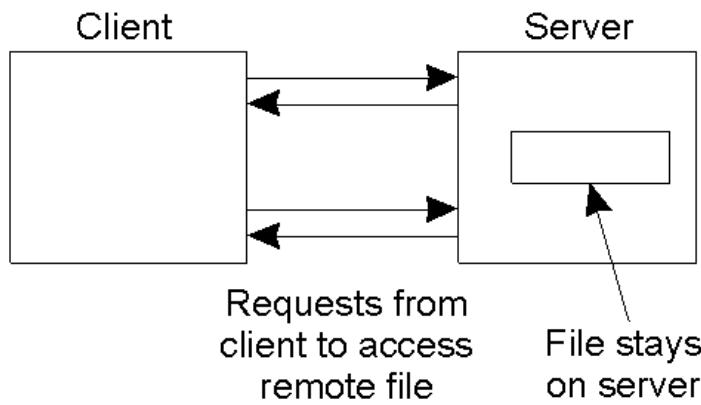
Contd...

- NFS version 1, 2, 3 &4.
- NFS is not so much a true file system, as a collection of protocols that together provide clients with a model of a distributed file system.
- The NFS protocols have been designed in such a way that different implementations should easily interoperate. In this way, NFS can run on heterogeneous collection of computers.
- An open standard with clear and simple interfaces.

Cont...

- Supports many of the design requirements already mentioned:
 - transparency
 - heterogeneity
 - efficiency
 - fault tolerance

NFS Architecture



- (a) The remote access model
- (b) The upload/download model

Cont...

There are benefits and drawbacks to both models.

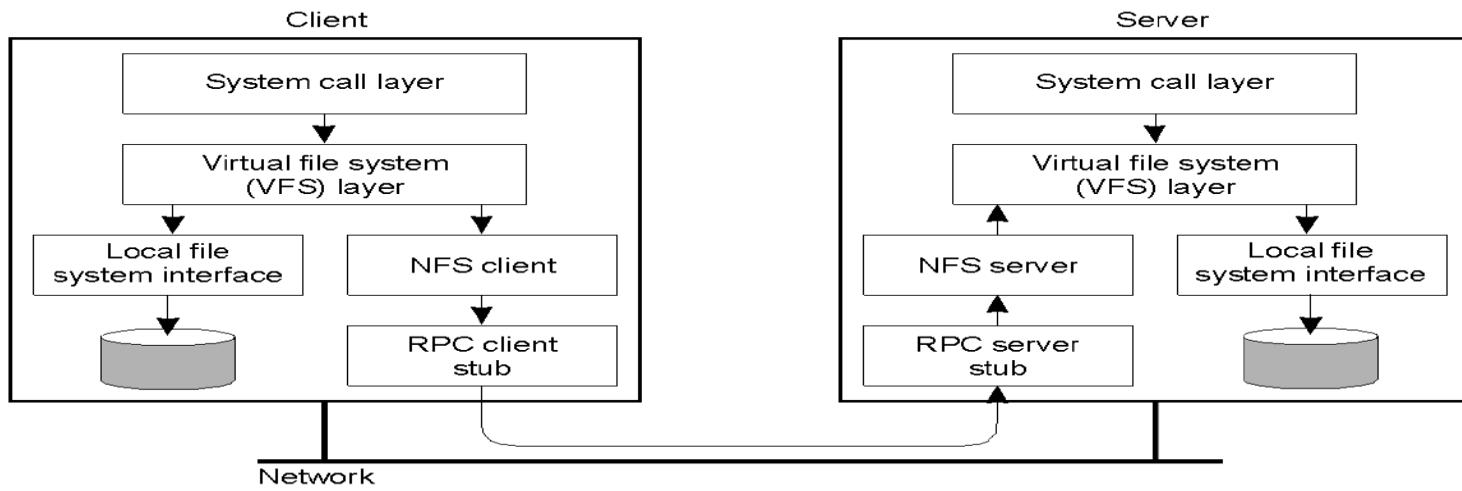
The remote access model -

- all operations are performed at the server itself, with clients simply sending commands to the server.
- It makes it possible for the file server to order all operations and therefore allow concurrent modifications to the files.
 - A drawback is that the client can only use files if it has contact with the file server. If the file server goes down, or the network connection is broken, then the client loses access to the files.

The upload/download model -

- files are downloaded from the server to the client. Modifications are performed directly at the client after which the file is uploaded back to the server.
- It can avoid generating traffic every time. Also, a client can potentially use a file even if it cannot access the file server.
 - A drawback of performing operations locally and then sending an updated file back to the server is that concurrent modification of a file by different clients can cause problems.

NFS Architecture...



- A client accesses the file system using the system calls provided by its local operating system.
- The local OS file system interface is replaced by an interface to the **Virtual File System (VFS)**, which is a standard for interfacing to different (distributed) file systems. The whole idea of the **VFS** is to hide the differences between various file systems.
- Operations on the **VFS** interface are either passed to a local file system, or passed to a separate component known as the **NFS Client**, which takes care of handling access to files stored at a remote server.
- In **NFS** all, client -server communication is done through **RPCs**.
- On the server side, the **NFS server** is responsible for handling incoming client request. The RPC stub unmarshals requests and the **NFS server** converts them to regular **VFS file** operations that are subsequently passed to the **VFS** layer.

NFS Architecture...

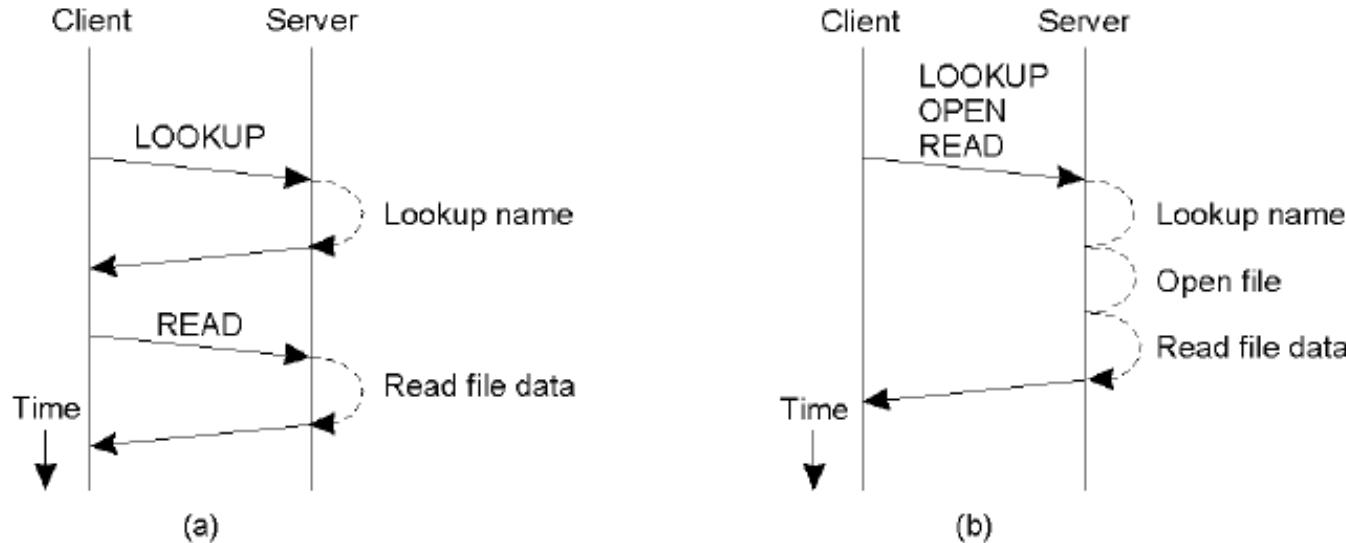
- An important advantage of this scheme is that **NFS** is largely independent of local file systems.
- It really does not matter whether the operating systems at the client or server implements a UNIX file system, a Windows 2000 file system, or even an old MS-DOS file system.
- The only important issue is that these file systems are accommodating with the file system model offered by **NFS**.

File System Model

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
getattr	Yes	Yes	Read the attribute values for a file
setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

An incomplete list of file system operations supported by NFS.

Communication



- a) Reading data from a file in NFS version 3.
- b) Reading data using a ***compound procedure*** in version 4. By this procedure several RPCs can be grouped into a single request.

NFS. Characteristics

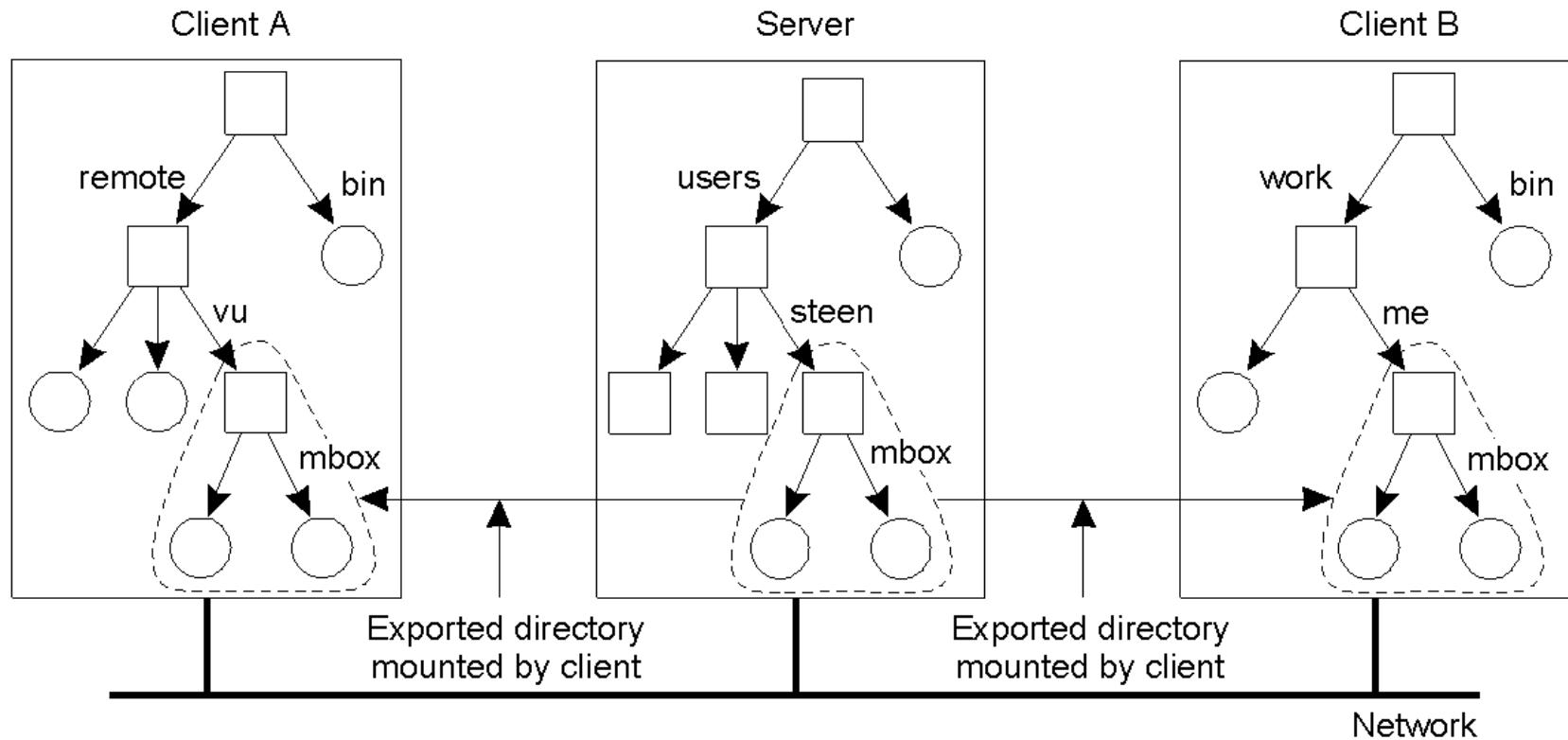
- Stateless server, so the user's identity and access rights must be checked by the server on each request.
 - In the local file system they are checked only on open()
- Every client request is accompanied by the user ID and group ID
- Server is exposed to imposter attacks unless the **user ID and group ID are protected by encryption**
- Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution
 - Mount operation:
 - Mount (remotehost, remotedirectory, localdirectory)
 - Server maintains a table of clients who have mounted file systems at that server
 - Each client maintains a table of mounted file systems holding:
 - < IP address, port number, file handle>

“File handle a number that the operating system assigns temporarily to a file when it is opened. The handle is used throughout the session to access the file. A special area of main memory is reserved for file handles, and the size of this area determines how many files can be open at once.”

NFS Naming

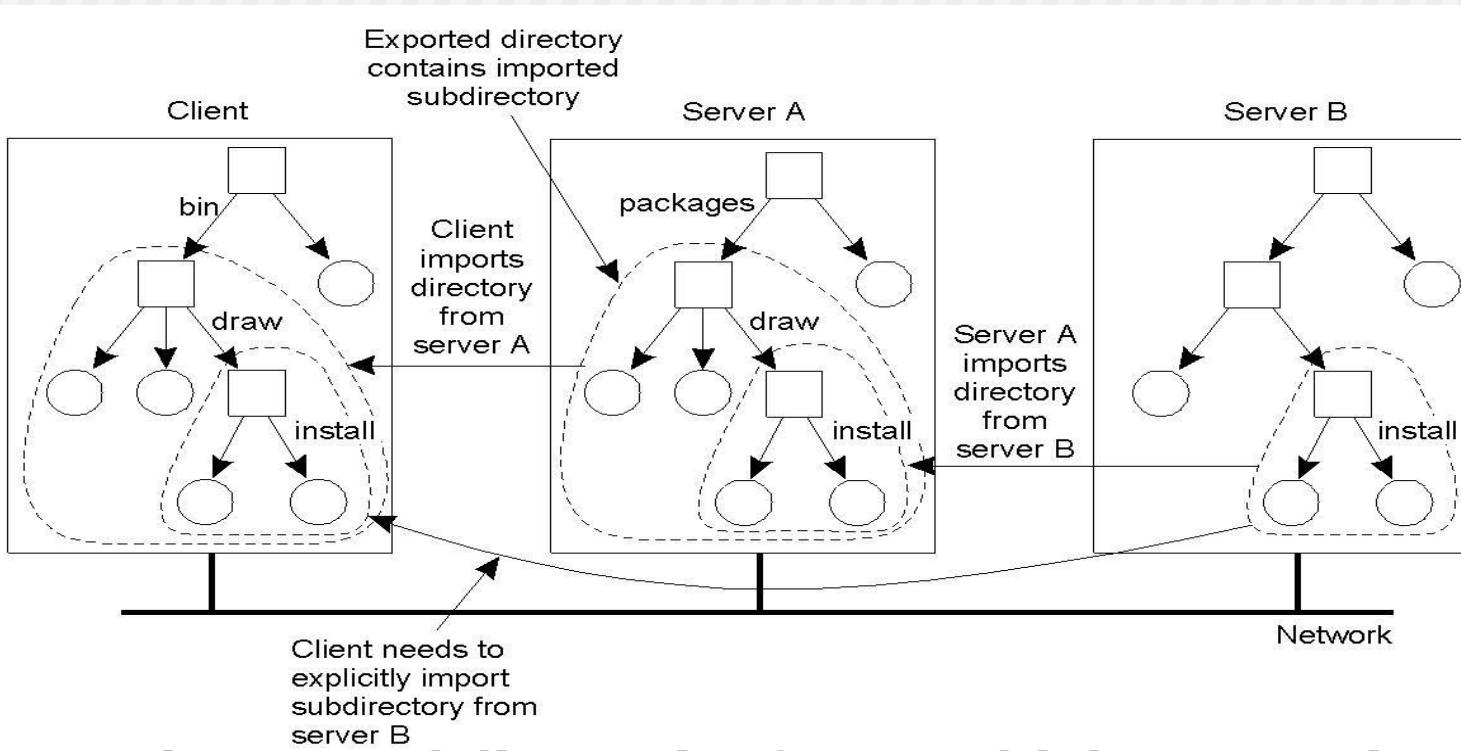
- The fundamental idea underlying the NFS naming model is to provide clients complete transparent access to a remote file system as maintained by a server.
- This transparency is achieved by letting a client be able to mount a remote file system into its own local file system.
- Instead of mounting an entire file system, NFS allows client to mount only part of a file systems.
- This design approach has a serious implication: in principle, users do not share **name spaces**. The drawback of this approach in a distributed file system, is that sharing files much harder.

NFS Naming



Mounting (part of) a remote file system in NFS

NFS Naming



Mounting nested directories from multiple servers in NFS

- An NFS server can not itself mount directories that are exported by other servers. However, it is not allowed to export those directories to its own clients.
- Instead, a client will have to explicitly mount such a directory from the server that maintains it.
- This restriction comes from simplicity. If a server could export a directory that is mounted from another server, it would have to return special file handles that include an identifier for a server. NFS does not support such file handles.

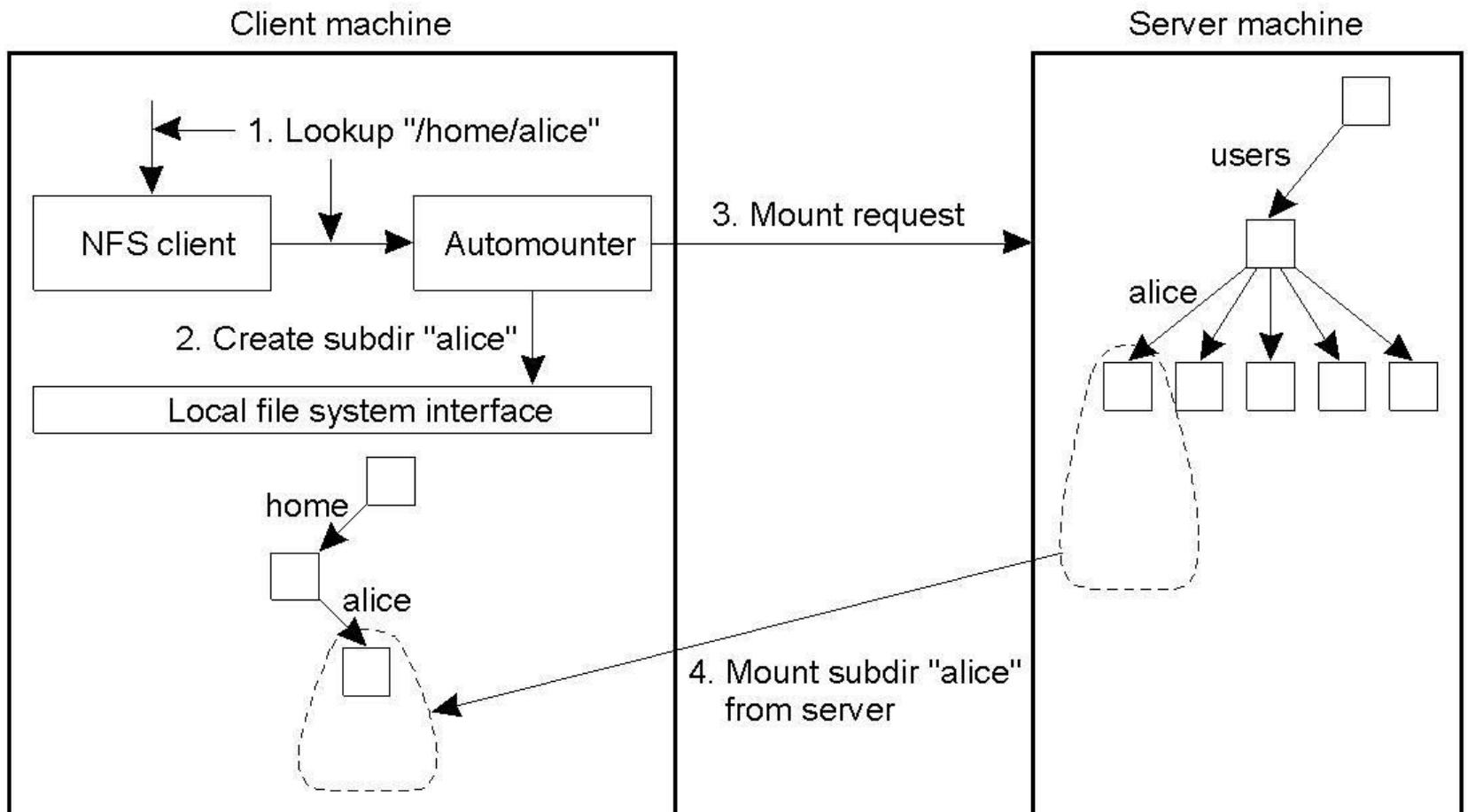
NFS. Automounting

- Problem with the NFS naming model has to do with deciding *when a remote file system should be mounted*.
- Consider a large system with thousands of users. Assume that each user has a local directory */home* that is used to mount the home directories of other users.
 - For Example: - Alice's home directory may be locally available to her as */home/Alice*, although the actual files are stored on a remote server. This directory can be automatically mounted when Alice logs into her workstation.
- for every user, logging in could incur a lot of communication and administrative overhead. In addition, it would require that all users are known in advance.
- On demand mounting of a remote file system is handled in NFS by an **automounter**, which runs as a separate process on the client machine. The principle underlying an automounter is relatively simple.

Cont...

- By considering a simple automounter, assume that for each user, the home directories of all users are available through the local directory */home*.
 - When a client machine boots, the automounter starts with mounting this directory.
- The effect of this local mount is that whenever program attempts to access */home*, the OS kernel will forward a *lookup* operation to the NFS client, which in this case, will forward the request to the automounter in its role as NFS server, as shown in below figure.

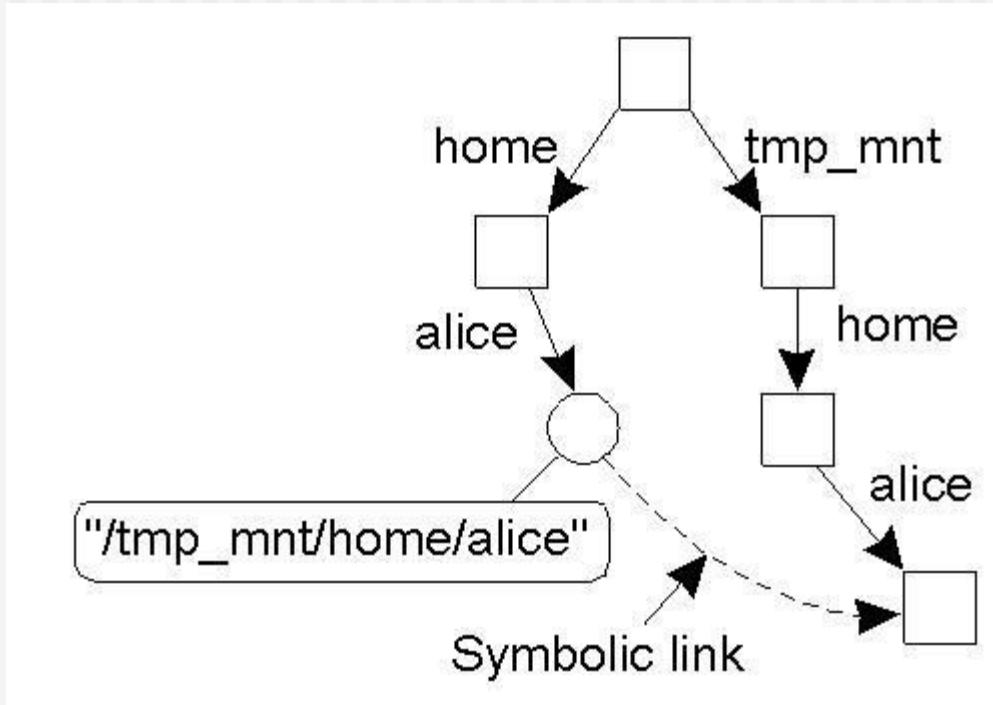
Cont...



Cont...

- Automounter first creates a subdirectory */alice in /home*. it then looks up the NFS server that exports Alice's home directory to subsequently and mount that directory in */home/alice*. At that point, the login program can proceed.
- A simple solution is to let the automounter mount directories in a special subdirectory, **and install a symbolic link to each mount directory**. This approach is shown in below figure.

Cont...



Cont...

- In the figure (last slide), the user home directories are mounted as subdirectories of */tmp_mnt*. When Alice logs in , the automounter mounts her home directory in */tmp_mnt/home/alice* and creates a symbolic link */home/alice* that refers to that subdirectory.
- The NFS server that exports Alic's home directory in contacted directory without further involment of automounter.

NFS. File Attributes

- An NFS file has a number of associated attributes. In version 3, the set of attributes was fixed and every implementation was expected to support those attributes. With version 4, the set of file attributes has been split into a set of ***mandatory attributes*** that every implementation must support, a set of ***recommended*** attributes that should be preferably supported, and an additional set of named attributes.

Attribute	Description
TYPE	The type of the file (regular, directory, symbolic link)
SIZE	The length of the file in bytes
CHANGE	Indicator for a client to see if and/or when the file has changed
FSID	Server-unique identifier of the file's file system

Some general mandatory file attributes in NFS.

Cont...

Attribute	Description
ACL	an access control list associated with the file
FILEHANDLE	The server-provided file handle of this file
FILEID	A file-system unique identifier for this file
FS_LOCATIONS	Locations in the network where this file system may be found
OWNER	The character-string name of the file's owner
TIME_ACCESS	Time when the file data were last accessed
TIME MODIFY	Time when the file data were last modified
TIME_CREATE	Time when the file was created

Some general recommended file attributes.

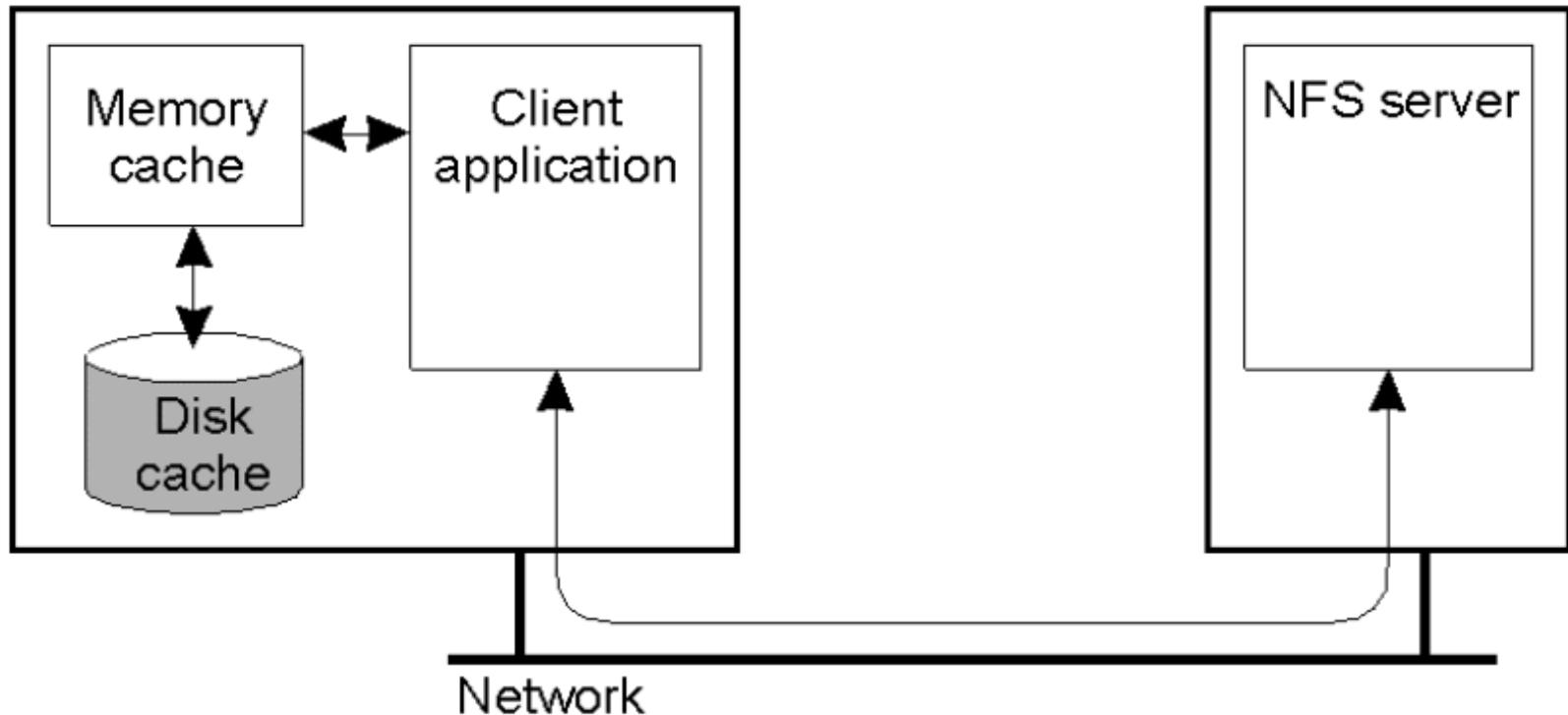
Synchronization: File Locking in NFS

Operation	Description
Lock	Creates a lock for a range of bytes
Lockt	Test whether a conflicting lock has been granted
Locku	Remove a lock from a range of bytes
Renew	Renew the leas on a specified lock

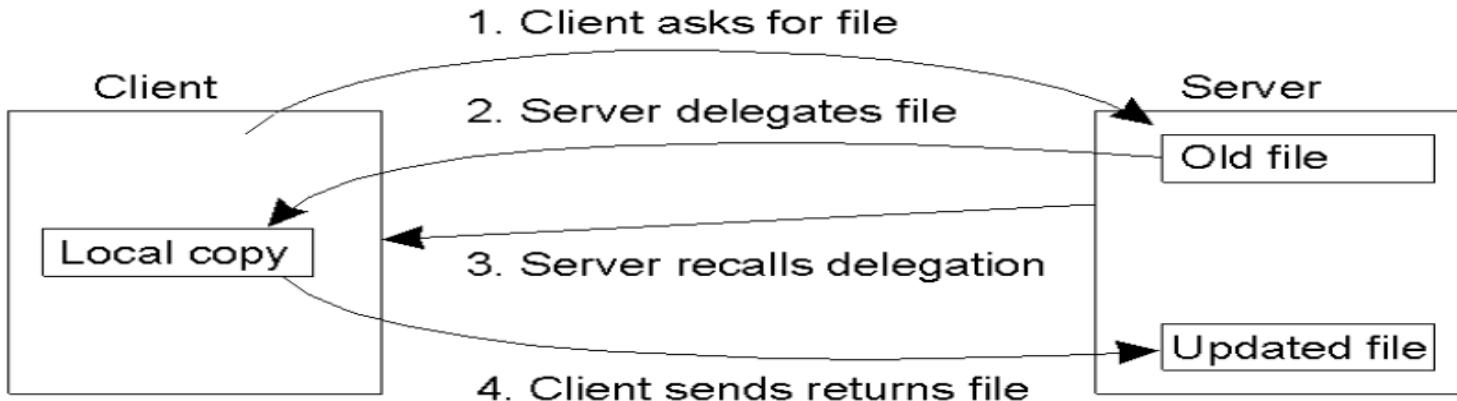
NFS version 4 operations related to file locking.

- NFS distinguishes read locks from write locks. Multiple clients can simultaneously access the same part of a file provided they only **read data**. A **write lock** is needed to obtain limited access to modify part of a file.
- **Lock** operation is used to request a read or write lock on a consecutive range of bytes in a file. If the lock cannot be granted due to another conflicting lock, the client gets back an error message and has to ask the server at a latter time. The client request to be put on a FIFO-ordered list maintained by the server. As soon as conflicting lock has been removed, the server will grant the next lock to the client at the top of the list.
- Removing a lock from a file is done by means of **locku** operation.
- Using the **renew** operation, a client requests the server to renew the lease on its lock.

NFS Client Caching

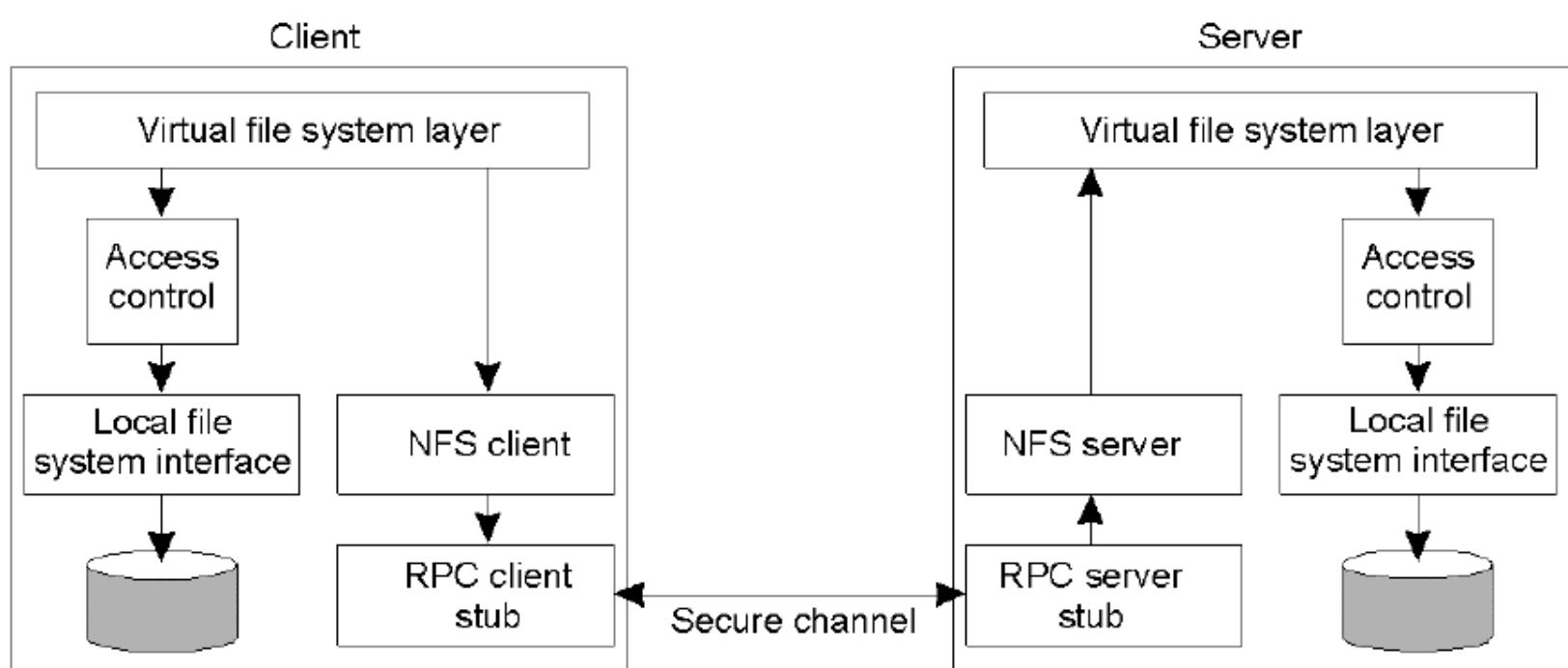


NFS Client Caching



- A server may delegate some of its rights to a client when a file is opened.
- An important consequence of delegating a file to a client is that the server needs to be able to recall the delegation, *for example*,
 - when another client on a different machine needs to obtain access rights to the file. Recalling a delegation requires that the server can do a callback to the client.

NFS The security architecture



Cont...

Security:

- Traditionally (NFS v3), clients were trusted. Three ways of authentication:
 - System authentication
Client passes user ID and group ID to server.
(i.e., almost no security)
 - Secure NFS using Diffie-Hellman (public key crypto)
 - More complex (implementation and key management)
 - Kerberos
 - Secure
 - Entry costs
- Enhanced security with NFS v4:
 - RPCSEC_GSS as general security framework
 - Message integrity and confidentiality

NFS. Access Control

Operation	Description
Read_data	Permission to read the data contained in a file
Write_data	Permission to modify a file's data
Append_data	Permission to append data to a file
Execute	Permission to execute a file
List_directory	Permission to list the contents of a directory
Add_file	Permission to add a new file to a directory
Add_subdirectory	Permission to create a subdirectory to a directory
Delete	Permission to delete a file
Delete_child	Permission to delete a file or directory within a directory
Read_acl	Permission to read the ACL
Write_acl	Permission to write the ACL
Read_attributes	The ability to read the other basic attributes of a file
Write_attributes	Permission to change the other basic attributes of a file
Read_namedAttrs	Permission to read the named attributes of a file
Write_namedAttrs	Permission to write the named attributes of a file
Write_owner	Permission to change the owner
Synchronize	Permission to access a file locally at the server with synchronous reads and writes

The classification of operations recognized by NFS with respect to access control.

NFS. Scalability

- The performance of a single server can be increased by the addition of processors, disks and controllers.
- When the limits of that process are reached, additional servers must be installed and the file systems must be reallocated between them.
- When loads go beyond the maximum performance, a distributed file system that supports replication of updatable files,
 - or one that reduces the protocol traffic by the caching of whole files, may offer a better solution.

Distributed Algorithms

COURSE OBJECTIVES

- To expose students to both the abstraction and details of file systems.
- To provide students with contemporary knowledge in parallel and distributed computing.
- To focus on performance and flexibility issues related to systems design decisions.
- Introduce a variety of methodologies and approaches for reasoning about concurrent and distributed programs

COURSE OUTCOMES

- After successful completion of this course, student will be able to Understand the concepts and issues related to distributed systems.
- Design and develop the programs for distributed environment.
- Manage performance, reliability and other issues while designing in distributed environment.

COURSE CONTENT**INTRODUCTION TO DISTRIBUTED SYSTEMS**

(04 Hours)

Review of Networks, Operating Systems – Concurrent Programming - Characteristics & Properties of DSs – Taxonomy - Design goals – Transparency Issues

DISTRIBUTED COMPUTING PARADIGMS

(02 Hours)

Basic Message Passing Model – The Client Server, Message Passing, Remote Procedure Call Model – RPC in conventional languages and in Java - The Distributed Objects – The Collaborative Application

INTER-PROCESS COMMUNICATION MECHANISMS

(08 Hours)

Communication in Distributed Systems, Socket Programming -Client Server examples, I/O Multiplexing, Inetd Super Server – Secure Sockets – The SSL & the Java Secure Socket Extension

PROCESS MODELS IN DISTRIBUTED SYSTEMS

(08 Hours)

Processes, Threads - Code Migration; Software Agents – CSP Distributed Processes - Naming with Mobile Entities - Unreferenced Objects

SYNCHRONIZATION

(04 Hours)

Clock Synchronization – Logical clocks – Election Algorithms – Distributed Mutual Exclusion

CONSISTENCY AND REPLICATION

(04 Hours)

Motivation, Object Replication, Consistency Models, Distribution Protocols – Consistency Protocols

FAULT TOLERANCE

(04 Hours)

Failure Models – Process Resilience – Reliable Client Server and Group Communications – Distributed Commit Protocols – Check-pointing and Recovery - Distributed Databases - Distributed Transactions

DISTRIBUTED FILE SYSTEMS

(04 Hours)

SUN NFS – Naming issues in DFS – Examples of contemporary DFSes – Comparisons

MISCELLANEOUS TOPICS

(04 Hours)

Distributed Object-based Systems – COM – CORBA – Architecture and Programming – Distributed Coordination based systems – TIB/RENDEZVOUS - JINI

Tutorials will be based on the coverage of the above topics separately and relevant research papers prescribed in the class

(14 Hours)

(Total Contact Time: 42 Hours + 14 Hours = 56 Hours)**PRACTICALS**

- 1.Implementation of the distributed applications/miniprojects using the TCP/IP Socket API as specified in the class e.g. a typical WWW server, a distributed Readers-Writers problem etc.
- 2.Implementation of the distributed applications/ miniprojects using the SUN ONC RPC Model, JAVA RMI, any Java-CORBA ORB as specified in the class

BOOKS RECOMMENDED

- 1.A S Tanenbaum, Martin Steen,"Distributed Systems: Principles and Paradigms", 2/E,PHI, 2006
- 2.Nancy A. Lynch, "Distributed Algorithms", Morgan Kaufmann, 1996
- 3.W Richard Stevens, "Unix Network Programming: Vol 1, Networking APIS: Sockets & XTI",2/E, Pearson Education,1998
- 4.Courlouis,Dollimore,Kindberg, "Distributed Systems Concepts & Design", 4/E,Pearson Ed. 2005
- 5.MukeshSinghal, Niranjan G. Shivaratri, "Advanced concepts in operating systems: distributed, database, and multiprocessor operating systems", MGH, 1/E, 1994.

Textbook & Readings

Textbook

1. Andrew S. Tanenbaum & Van Steen, “Distributed Systems Principles and Paradigms”, Publisher: PHI, Year: 2007.
2. P.K.Sinha, “Distributed Operating Systems”, Publisher: PHI, Year: 2007.
3. Mukesh Singhal, Niranjan G. Shivaratri (Contributor) “Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems”, MGH, 1994.

Reference Book

1. Hagit Attiya, Jennifer Welch “Distributed Computing: Fundamentals, Simulations, and Advanced Topics”, 2/E, Jon Wiley & sons, March 2004.
2. M.L.Liu, “Distributed Computing -- Concepts and Application”, Publisher Addison Wesley.

Readings will also include papers.

Assignments:

This course will involve listening to lectures, reading papers, writing reviews of papers, participating in class discussions, presenting papers and leading class discussions.

Students will be required to write reviews for paper they read.

Definitions:

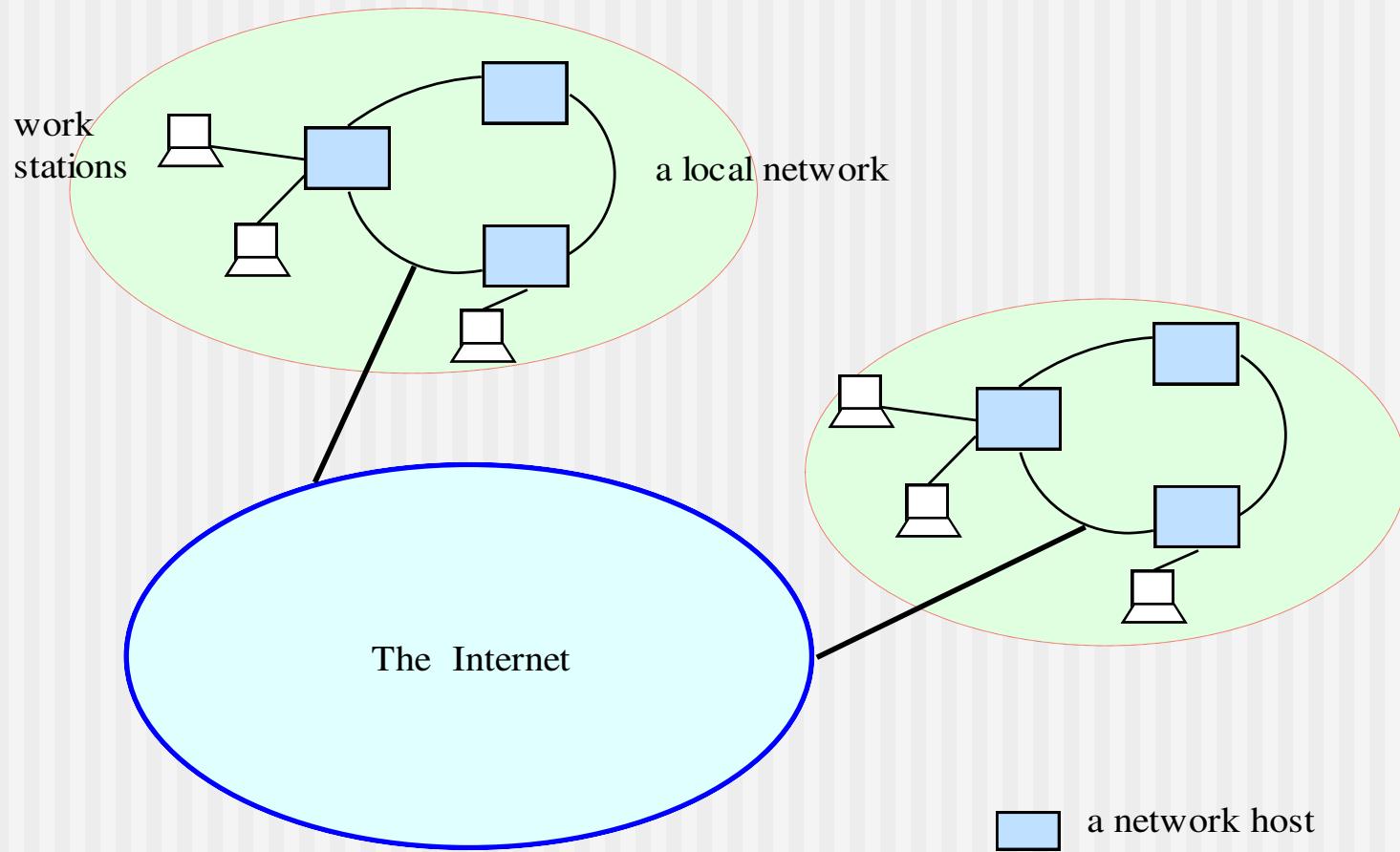
Distributed System

- Distributed Systems - what and why?
- Architecture
- Challenges
- Overview - principles and paradigms

Distributed system, distributed computing

- Early computing was performed on a single processor. Uni-processor computing can be called *centralized computing*.
- A *distributed system* is a collection of independent computers, interconnected via a network, capable of collaborating on a task.
- *Distributed computing* is computing performed in a distributed system.

Distributed Systems



DISTRIBUTED SYSTEMS

■ What is a distributed system?

Andrew Tannenbaum defines it as follows:

- “A distributed system is a collection of independent computers that appear to its users as a single coherent system.”

- Is there any such system?
- We will learn about the challenges in building “true” distributed systems

“A distributed system is a collection of independent computers that are used jointly to perform a single task or to provide a single service.”

Why Distributed?

- **Resource and Data Sharing**
 - Printers, databases, multimedia servers etc.
- **Availability, Reliability**
 - The loss of some instances can be hidden
- **Scalability, Extensibility**
 - System grows with demands (e.g. extra servers)
- **Performance**
 - Huge power (CPU, memory etc.) available
 - *Horizontal distribution (same logical level is distr.)*
- **Inherent distribution, communication**
 - Organizational distribution, e-mail, video conference
 - *Vertical distribution (corresponding to org. struct.)*

Parallel Computing vs Distributed Computing

- *Parallel* computing is more **tightly coupled to multi-threading**, or how to make full use of a single CPU.
- *Distributed* computing refers to the notion of **divide and conquer**, executing **sub-tasks** on different machines and then **merging** the results.
- However, since we stepped into the *Big Data* era, it seems the distinction is indeed, and most systems today **use a combination of parallel and distributed computing**.
- An example - Hadoop with the **Map/Reduce** paradigm, a clearly distributed system with **workers** executing tasks on different machines, but also taking full advantage of each machine with some parallel computing.

Problems of Distribution

- **Concurrency, Security**
 - Clients must not disturb each other
- **Partial failure**
 - We often do not know, where is the error (e.g. RPC)
- **Location, Migration, Replication**
 - Clients must be able to find their servers
- **Heterogeneity**
 - Hardware, platforms, languages, management
- **Convergence**
 - Between distributed systems and telecommunication

Examples of distributed systems

- **Network of workstations (NOW):** a group of networked personal workstations connected to one or more server machines.
- **The Internet**
- **An intranet:** a network of computers and workstations within an organization, separated from the Internet via a protective device (a firewall).
- A large-scale distributed system – **eBay**
- **Collection of Web servers:** distributed database of hypertext and multimedia documents.
- Distributed file system on a LAN
- Domain Name Service (DNS)
- small-scale distributed system
-Home Smart

Computers in a Distributed System

- Workstations: computers used by end-users to perform computing
- Server machines: computers which provide resources and services
- Personal Assistance Devices: handheld computers connected to the system via a wireless communication link.

THE ADVANTAGES OF DISTRIBUTED SYSTEMS

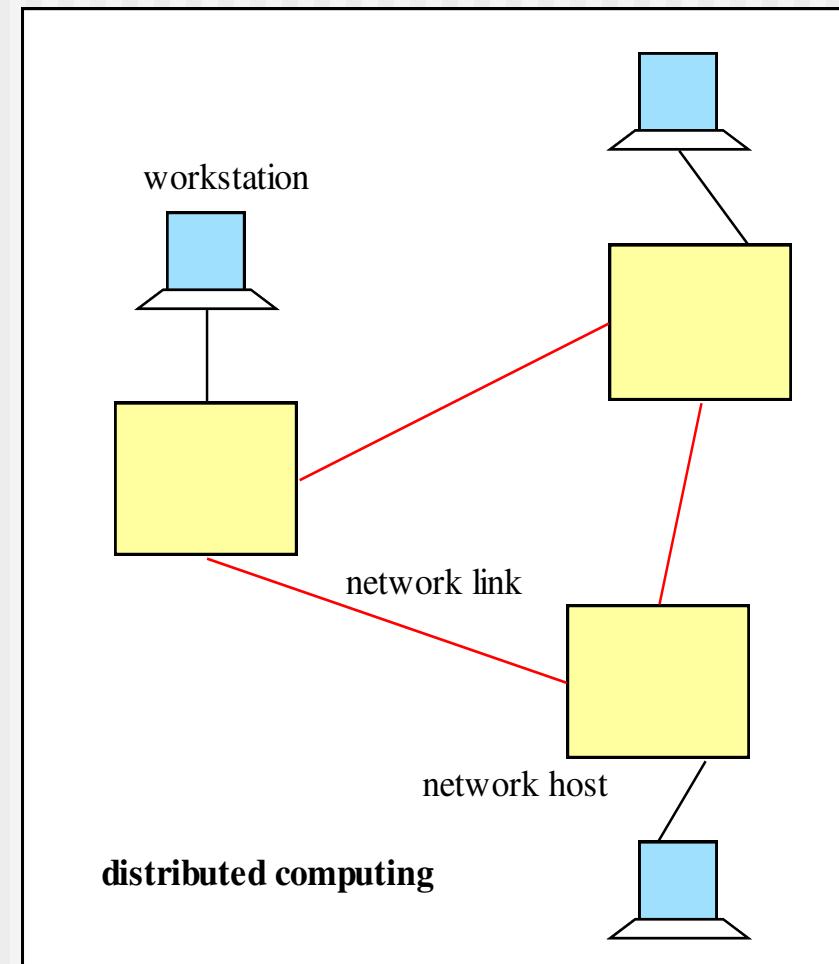
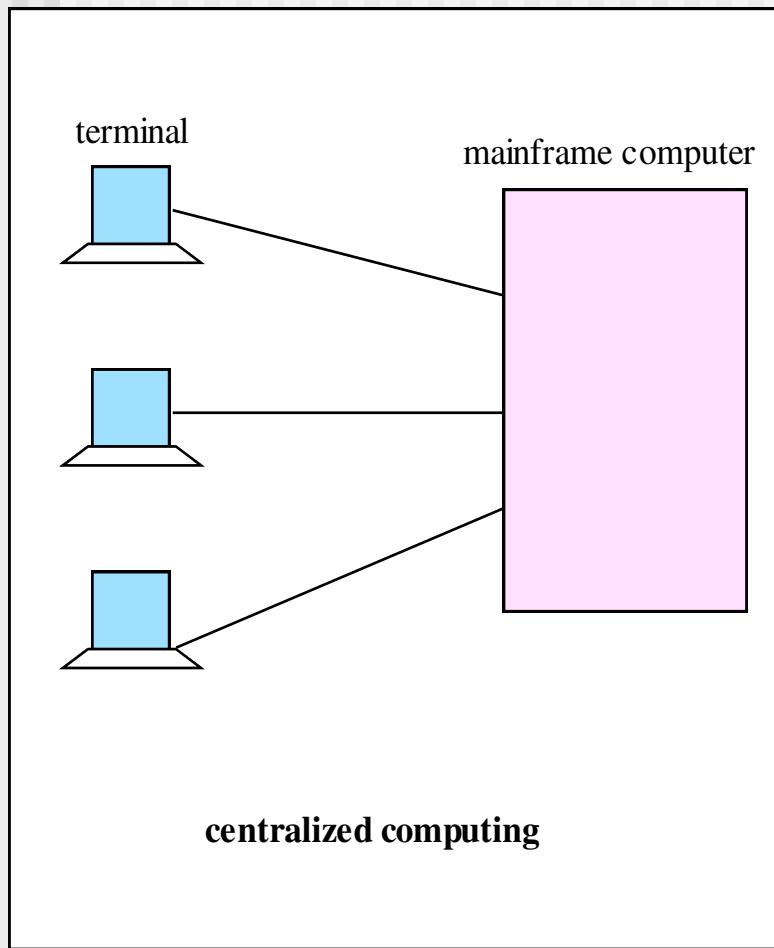
- What are economic and technical reasons for having distributed systems?
 - **Economics:**
 - **Performance:** By using the combined processing and storage capacity of many nodes, performance levels can be reached that is out of the scope of centralized machines.
 - **Scalability:** Resources such as processing and storage capacity can be increased incrementally.
 - **Inherent distribution:** Some applications, such as email and the Web (where users are spread out over the whole world), are naturally distributed. This includes cases where users are geographically dispersed as well as when single resources (e.g., printers, data) need to be shared.
 - **Reliability:** By having redundant components, the impact of hardware and software faults on users can be reduced.
 - Better **flexibility** in meeting users' needs.

THE DISADVANTAGES OF DISTRIBUTED SYSTEMS

- **Multiple Points of Failures:** the **failure** of one or more **participating computers**, or one or more **network links**, can bring trouble.
- **Security Concerns:** In a distributed system, there are more opportunities for unauthorized attack.
- **Software complexity:** Distributed software is more complex and harder to develop than conventional software.

Distributed systems are hard to build and understand.

Centralized vs. Distributed Computing



DISTRIBUTED SYSTEM ARCHITECTURE

Hardware Architecture:

- § Uniprocessor
- § Multiprocessor
- § Multicomputer

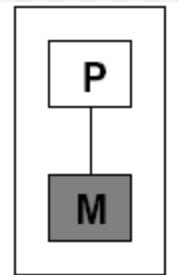
Software Architecture:

- § Uniprocessor OS
- § Multiprocessor OS
- § Network OS (NOS)
- § Distributed OS (DOS)
- § Middleware

- A key characteristic of distributed systems that includes
 - a hardware aspect (independent computers)
 - and a software aspect (performing a task and providing a service)

HARDWARE ARCHITECTURE

■ Uniprocessor:

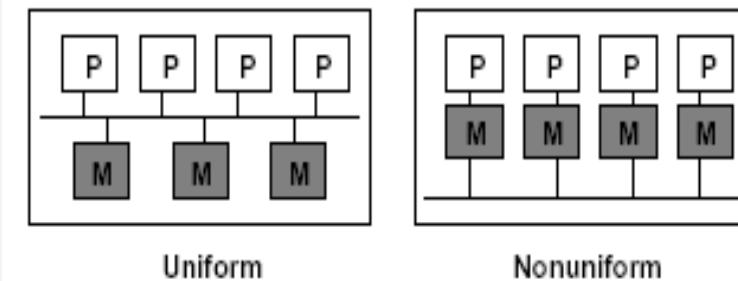


Properties:

1. Single processor
2. Direct memory access

Continued.....

- **Multiprocessor:** those that have shared memory , usually called multiprocessors, and those do not , sometimes called Multicomputers.
- Typically, there are 2 major types of Parallel Architectures that are common in the industry: Shared Memory Architecture and Distributed Memory Architecture. Shared Memory Architecture, again, is of 2 types: Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA).

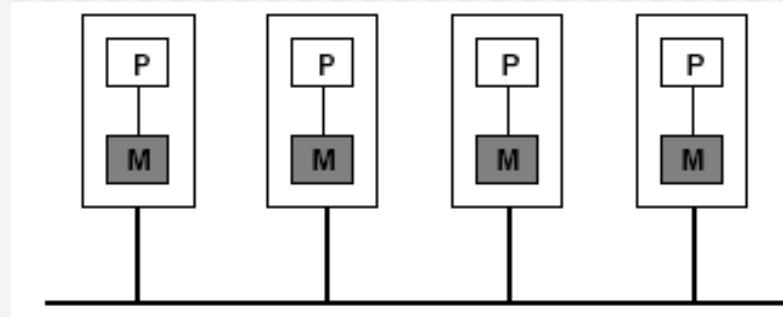


Properties:

1. Multiple processors
2. Direct memory access
3. Uniform memory access -access time to a memory location is independent of which processor makes the request
4. Non-uniform memory access- the memory access time depends on the memory location relative to a processor.

Continued.....

■ Multicomputer:

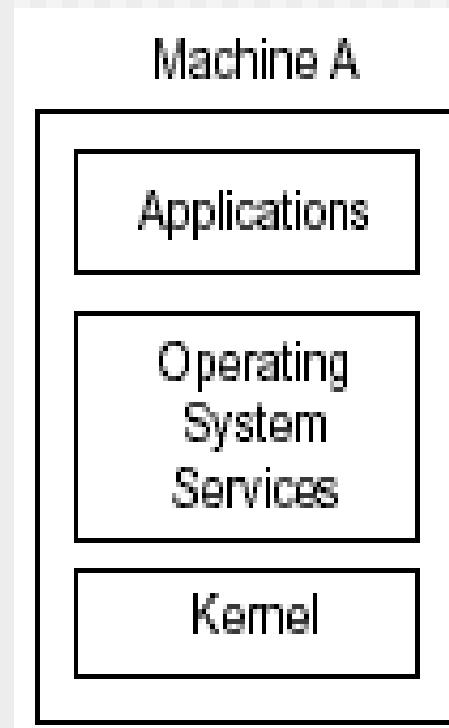


Properties:

1. Multiple computers
2. Network Connection (to support higher bandwidth)
3. Homogeneous (all nodes support same physical architecture) vs. Heterogeneous (does not support same physical architecture)

SOFTWARE ARCHITECTURE

- **Uniprocessor OS:**



Continued....

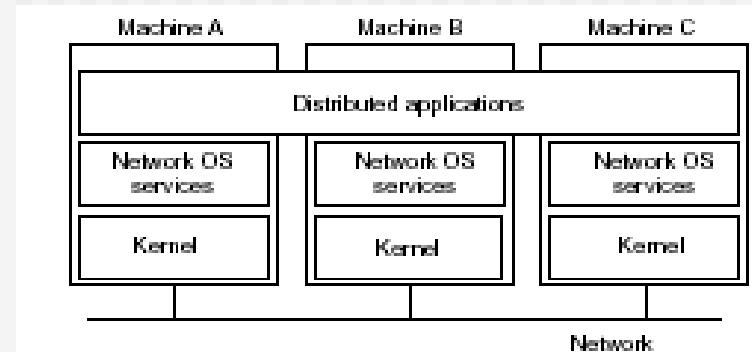
- **Multiprocessor OS:**

Similar to a uniprocessor OS but:

- Kernel designed to handle multiple CPUs
- Communication uses same primitives as uniprocessor OS
- Tightly-coupled software on tightly-coupled hardware
- Examples: high-performance servers
- shared memory
- single run queue

Continued....

■ Network OS:

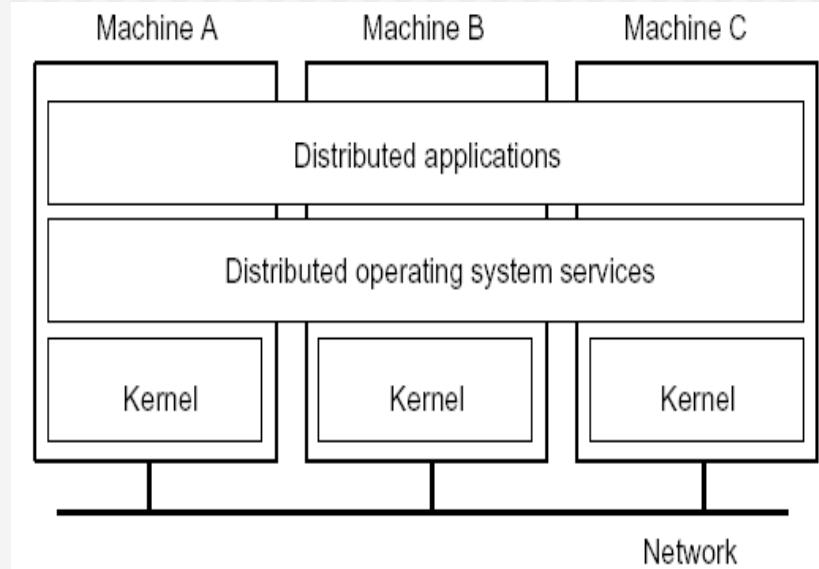


Properties:

1. No single system image (Users are aware of multiplicity of machines)
2. Individual nodes are highly autonomous
3. Access to resources of various machines is done explicitly by:
 - Remote logging into the appropriate remote machine (telnet)
 - Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism
4. Examples: Linux, Windows

Continued....

- **Distributed OS:** A tightlycoupled operating system is generally referred to as a **distributed operating system (DOS)**, and is used for managing multiprocessors and homogeneous multicompilers.



Properties:

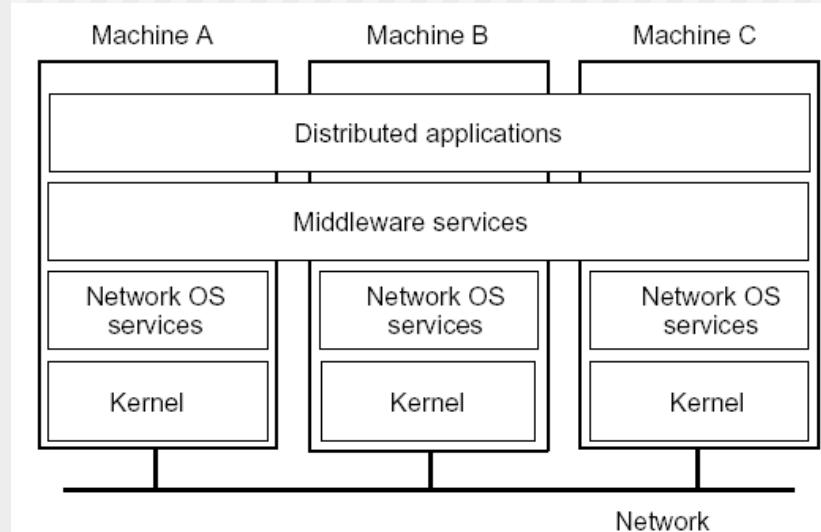
1. **Users not aware of multiplicity of machines**
 - Access to remote resources similar to access to local resources
2. **High degree of transparency** (single system image)
3. **Support distributed services** (services may include distributed shared memory, assignment of tasks to processors, distributed storage, interprocess communication, transparent sharing of resources, distributed resource management, etc.)
4. **Homogeneous hardware**

Middleware

- Neither a distributed operating system or a network operating system really qualifies as a distributed system according to the definition.
- **Distributed operating system is not intended to handle a collection of *independent* computers, while a network operating system does not provide a view of a *single coherent system*.**
- *The question comes to mind whether it is possible to develop a distributed system that has the best of both worlds: the scalability and openness of network operating systems and the transparency and related ease of use of distributed operating systems.*
- The **solution** is to be found in an additional layer of software that is used in network operating systems to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency.
- Many modern distributed systems are constructed by means of such an additional layer of what is called **middleware**.

Continued....

- **Middleware:** Middleware enables the components to coordinate their activities.



Properties:

1. System independent interface for distributed programming
2. Improves transparency (e.g., hides heterogeneity)
3. Provides services (e.g., naming service, transactions, etc.)
4. Provides programming model (e.g., distributed objects)

Continued....

■ Why is Middleware ‘Winning’?:

1. Builds on commonly available abstractions of network OS
2. Examples: RPC, NFS, CORBA
3. Usually runs in user space
5. less error
6. Independence from OS, network protocol, programming language, etc.

Comparison between Systems

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Software Concepts

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer at top of NOS implementing general-purpose services	Provide distribution transparency

Openness

- An open distributed system is a system that **offers services according to standard rules** that describe syntax and semantics of the services.
 - **Systems should conform to well-defined Interfaces**
 - **System should support Interoperability**
 - Components of different origin can communicate
 - **System should support Portability**
 - Components work on different platforms
 - **Standards – a necessity**

DISTRIBUTED SYSTEMS IN CONTEXT

Networking:

- Network Protocols, routing protocols, etc.
- Distributed Systems: **make use of networks**

Operating Systems:

- Resource management for single systems
- Distributed Systems: **management of distributed resources**

BASIC PROBLEMS AND CHALLENGES IN DISTRIBUTED SYSTEMS

The distributed nature of a systems gives rise of the following challenges:

- Transparency
- Scalability
- Dependability
- Performance
- Flexibility

TRANSPARENCY

**Concealment of the separation of the components of a distributed system
(single image view).**

■ **There are a number of forms of transparency:**

- **Access:** Local and remote resources accessed in same way
 - the users should not need or able to recognize whether a resource (hardware or software) is remote or local.
 - This is facilitated by distributed operating system.
- **Location:** Users unaware of location of resources
- **Migration:** Resources can migrate without name change
- **Replication:** Users unaware of existence of multiple copies
- **Failure:** Users unaware of the failure of individual components
- **Concurrency:** Users unaware of sharing resources with others

SCALABILITY

A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity

- **Scale has three dimensions:**

- **Size:** number of users and resources (problem: overloading)
- **Geography:** distance between users and resources (problem: communication)
- **Administration:** number of organizations that apply administrative control over parts of the system (problem: administrative mess)

Note:

- Scalability often conflicts with (small system) performance

DEPENDABILITY

Dependability of distributed systems is a double-edged sword:

- Distributed systems promise higher availability:
 - Replication
 - But availability may degrade:
 - More potential points of failure
-
- Dependability requires **consistency**, security, and fault tolerance

PERFORMANCE

- Any system should strive for maximum performance
- In distributed systems, performance directly conflicts with some other desirable properties....
 - Transparency
 - Security
 - Dependability
 - Scalability

FLEXIBILITY

“A flexible distributed system can be configured to provide exactly the services that a user or programmer needs”

Flexibility generally provides a number of key properties:

- Extensibility: Components/services can be changed or added
- Openness of interfaces and specification
- Interoperability

PRINCIPLES

There are several key principles underlying all distributed systems. As such, any distributed system can be described based on how those principles apply to that system.

- System Architecture
- Communication
- Synchronization
- Replication and Consistency
- Fault Tolerance
- Security
- Naming

During the rest of the course we will examine each of these principles in detail.

PARADIGMS

Most distributed systems are built based on a particular paradigm (or model)

- Shared memory
- Distributed objects
- Distributed file system
- Shared documents
- Distributed coordination
- Agents

This course will cover the first three in detail.

MISCELLANEOUS “RULES OF THUMB”

Here we present some rules of thumb that are relevant to the study and design of distributed systems

- **Trade-offs** - Many of the challenges provide conflicting requirements. For example better scalability can cause worse overall performance. Have to make trade-offs - what is more important?
- **Separation of Concerns**- Split a problem into individual concerns and address each separately
- **End-to-End Argument**- Some communication functions can only be reliably implemented at the application level
- **Mechanism** -A system should build mechanisms that allow flexible application of policies.
- **Keep It Simple**- make things as simple as possible, but no simpler.

OVERVIEW OF COURSE

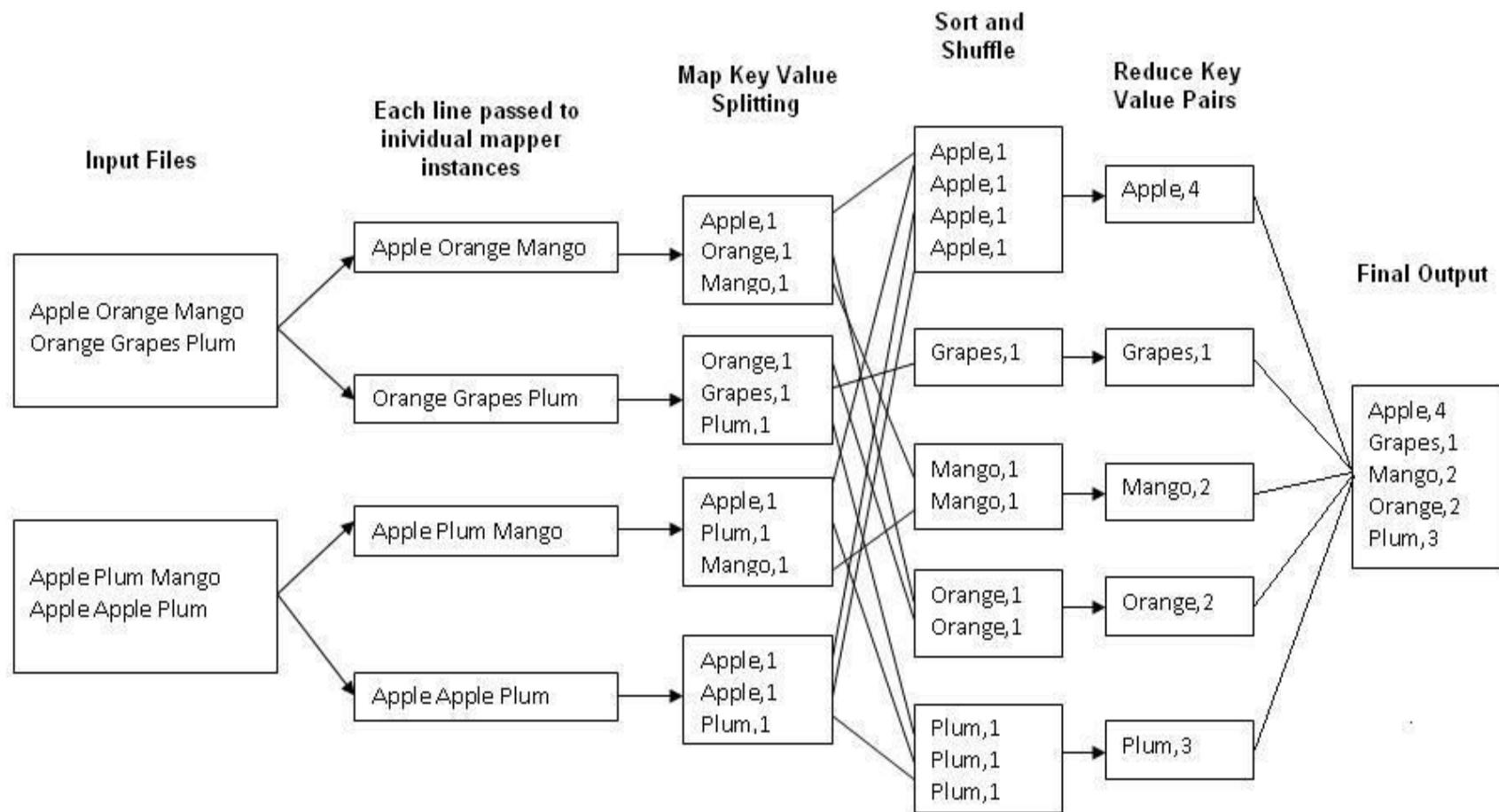
- Introduction
- System Architecture and Communication
- Replication and Consistency and Distributed Shared Memory
- Middleware and Distributed Objects
- Synchronization and Coordination
- Fault Tolerance
- Security
- Naming
- Distributed File Systems
- Extras:
 - Some reading materials
 - Research papers

-
- **Location Transparency:** The name of a file does not give any hint of file's physical storage location.
 - **Location Independence:** The name of a file does not need to be changed when file's physical storage location changes.

MapReduce

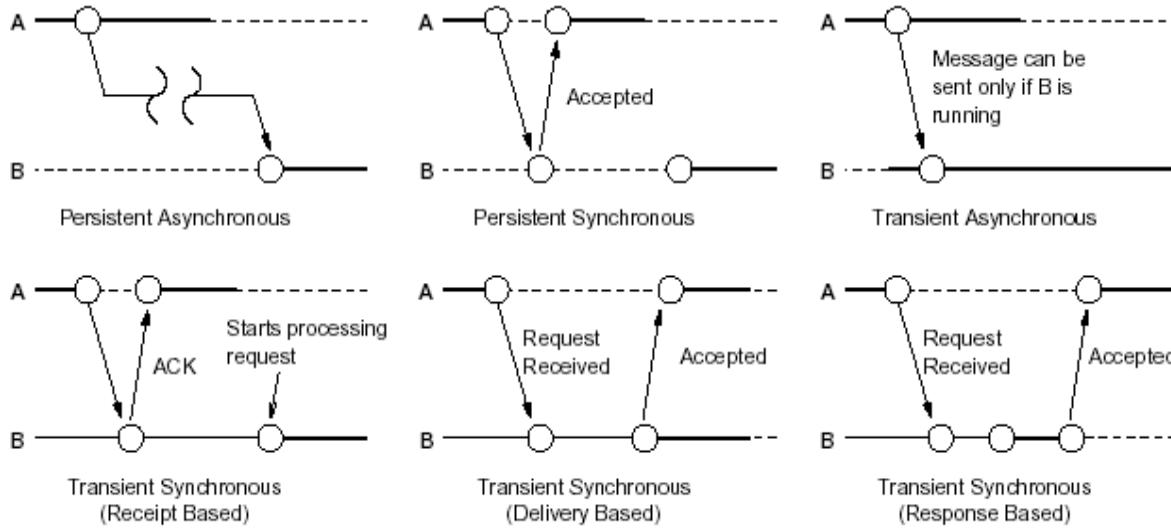
- A **MapReduce** job usually **splits** the input data-set into **independent chunks** which are processed by the map tasks in a completely parallel manner. The framework **sorts the outputs of the maps**, which are then **input to the reduce tasks**. Typically both the input and the output of the job are stored in a file-system.
- **Word Count - Hadoop Map Reduce Example**
 - Word count is a typical example where Hadoop map reduce developers start their hands on with.
 - This sample map reduce is intended to **count the no of occurrences of each word** in the provided input files.
- **What are the minimum requirements?**
 - Input text files – any text file
 - [The modern platform for data management and analytics](#)
 - The mapper, reducer and driver classes to process the input files

Word Count - Hadoop Map Reduce Example



-
- **Integration**
Combining software or hardware components or both into an overall system.
 - **Portability**
Relating to or being software that can run on two or more kinds of computers or with two or more kinds of operating systems.
 - **Interoperability**
The ability of software and hardware on multiple machines from multiple vendors to communicate.

Explanation for the different types of messages figure:



A. Persistent Asynchronous

- a. Sender A is not waiting for Receiver B's response, so it is Asynchronous.
- b. Sender A sends message at the time Recipient B is not active; B gets the message when it becomes active and the message doesn't get lost. So it is Persistent.

B. Persistent Synchronous

- a. Sender A is waiting for response from Recipient B, so it is Synchronous.
- b. Recipient B is not active, still the message is sent and not lost, so it is Persistent.

C. Transient Asynchronous

- a. Sender A continues its work without waiting for response from Receiver, so it is Asynchronous
- b. Sender A is sending the message to Recipient B when B is active, so it is Transient.

D. Transient Synchronous (Receipt Based)

- a. Sender A is waiting for response from Recipient B, so it is Synchronous.
- b. Recipient B acknowledges A's message immediately upon receiving it, without doing any kind of processing and so it is Receipt Based Synchronous.
- c. Sender A sends the message when Recipient B is active, so it is Transient.

E. Transient Synchronous (Delivery Based)

- a. Sender A is waiting for response from Recipient B, so it is Synchronous.
- b. Recipient B acknowledges A's message only when it starts processing it and so it is Delivery Based Synchronous.
- c. Sender A sends the message when Recipient B is active, so it is Transient.

F. Transient Synchronous (Response Based)

- a. Sender A is waiting for response from Recipient B, so it is Synchronous.
- b. Recipient B sends message to sender A only when it completes entire processing of the request and so it is Response based Synchronous.
- c. Sender A sends the message when Recipient B is active, so it is Transient.

Assignment-I Solution

1 Solution

Correct Option: If events a and b are causally related, then event a happened before event b.

Explanation:

Scalar clocks satisfy the monotonicity and hence the consistency property: for two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$.

2 Solution

Correct Option: a: 6, b: 7, c: 6, d: 7, e: 8, f: 9

Explanation: Solved by the property of Scalar Time

3 Solution

Correct Option: (P): (iii), Q: (iv), R: (i), (S): (ii)

Explanation:

- (P) Access transparency hides differences in data representation on different systems and provides uniform operations to access system resources.
- (Q) Migration transparency allows relocating resources without changing names.
- (R) Failure transparency refers to the system being reliable and fault-tolerant.
- (S) Relocation transparency is the ability to relocate the resources as they are being accessed is relocation transparency.

4 Solution

Correct Option: True

Explanation: There is no anonymous leader election algorithm for synchronous ring systems that is uniform

5 Solution

Correct Option: n-1, d

Explanation: There is an asynchronous convergecast algorithm with message complexity $(n-1)$ and time complexity d , when a rooted spanning tree of n nodes with depth d is known in advance.

6 Solution

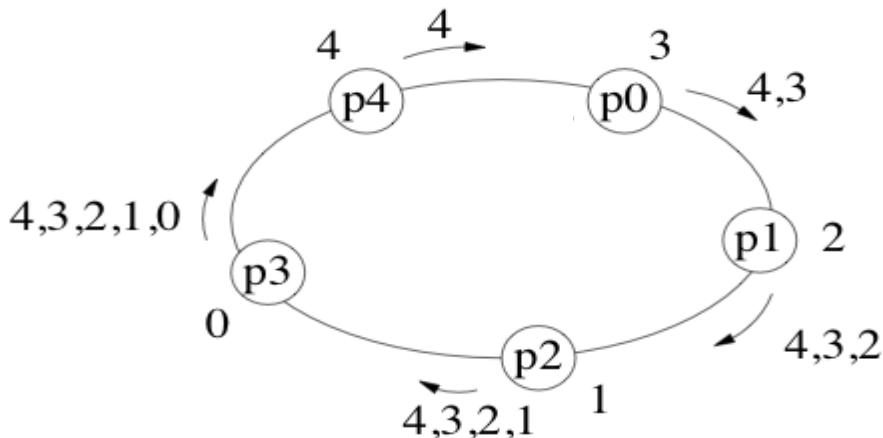
Correct Option: O(m), O(m)

Explanation: There is an asynchronous algorithm to find a depth-first search spanning tree of a network with m edges and n nodes, given a distinguished node, with message complexity O(m) and time complexity O(m)

7 Solution

Correct Option: 15

Explanation:



Total number of messages is $n + (n-1) + (n-2) + \dots + 1$

largest id travels all around the ring (n messages)

2nd largest id travels until reaching largest

3rd largest id travels until reaching largest and so on

$$5 + (5-1) + (5-2) + (5-3) + (5-4) = 15 \text{ messages}$$

8 Solution

Correct Option: $2k+1, 4*2^k$

Explanation:

k-neighbourhood of a processor p_i in the ring to be the set of processors that are at distance at most k from p_i in the ring (either to the left or to the right). Note that the k-neighbourhood of a processor includes exactly $2k+1$ processors.

Probe distance in phase k is 2^k and Number of messages initiated by a processor in phase k is at most $4*2^k$ (probes and replies in both directions)

9 Solution

Correct Option: (1, 7, 4)

Explanation: If we do an element-by element comparison:

(1, 7, 4) is neither \geq nor \leq to (2, 7, 3)

$1 < 2, 7 = 7, 4 > 3.$

10 Solution

Correct Option: A5: (4, 3, 3), B3: (1, 3, 0), C2: (1, 2, 2)

Explanation: Solved by the property of vector clock

11 Solution

Correct Option: Layered, Middleware

Explanation: The distributed system uses a layered architecture to break down the complexity of system design. The Middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level.

12 Solution

Correct Option: Asynchrony

Explanation: Asynchrony: Absolute & relative timings of events cannot always be known precisely.

13 Solution

Correct Option: Broadcast, Convergecast

Explanation:

Broadcast is used to send the information to all.

Convergecast is used to collect the information.

14 Solution

Correct Option: Both are true

Explanation:

A uniform algorithm does not use the ring size (same algorithm for each size ring)

A non-uniform algorithm uses the ring size (different algorithm for each size ring)

15 Solution

Correct Option: Both are false

Explanation:

Correct statements are:

By the property of scalar clocks and vector clocks:

The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j , $C(e_i) < C(e_j) \Rightarrow e_i \not\rightarrow e_j$

The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.

Assignment-II Solutions: Distributed Systems (Week-2)

Q. 1 Consider the following statement:

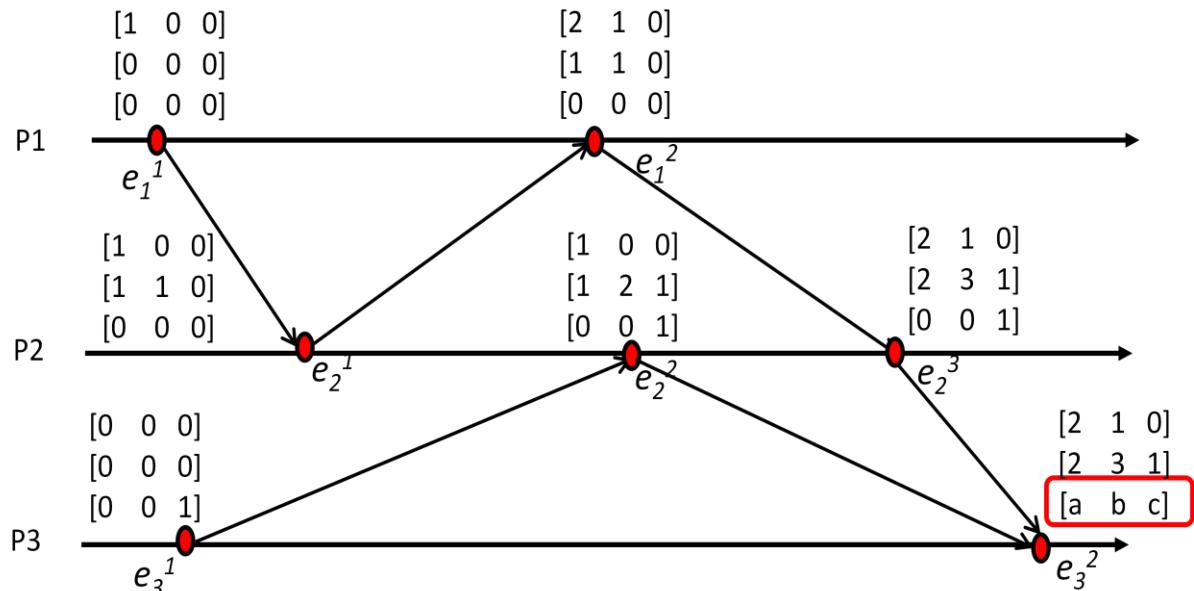
The Chandy-Lamport global snapshot algorithm works correctly for non-FIFO channels.

- A. True
- B. False

Ans: B. False

Explanation: Chandy-Lamport global snapshot algorithm requires FIFO channels

Q. 2 Find out the correct matrix clock values of a, b, c at P3 process as highlighted in the given figure:

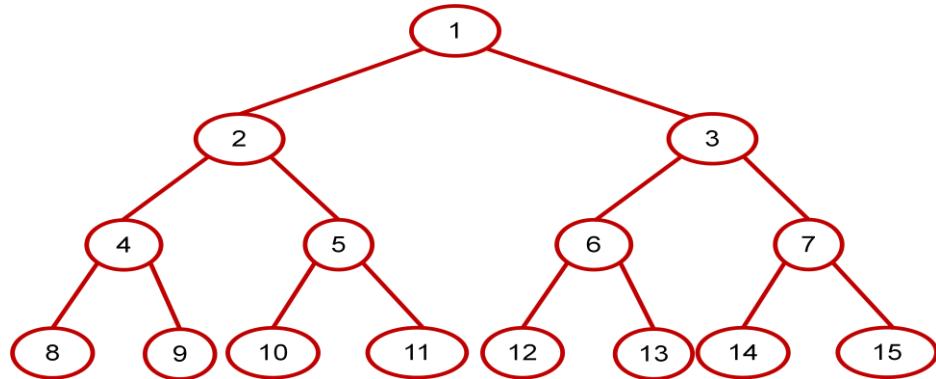


- A. $[0 \ 0 \ 2]$
- B. $[2 \ 3 \ 1]$
- C. $[2 \ 3 \ 2]$
- D. $[0 \ 0 \ 1]$

Ans: C. $[2 \ 3 \ 2]$

Explanation: Solved by the property of Matrix Clocks

Q. 3 Consider a tree structured quorum with 15 sites as shown in below figure:



when node 2 fails, the possible quorums can be formed as:

- A. {1-2-4-8}, {1-2-4-9}, {1-2-5-10}, {1-2-5-11}, {1-3-6-12}, {1-3-6-13}, {1-3-7-14} and {1-3-7-15}
- B. {1-4-8-5-10}, {1-4-8-5-11}, {1-4-9-5-10}, {1-4-9-5-11}, {1-3-6-12}, {1-3-6-13}, {1-3-7-14} and {1-3-7-15}
- C. {1-3-6-12}, {1-3-6-13}, {1-3-7-14} and {1-3-7-15}
- D. {1-4-8-5-10}, {1-4-8-5-11}, {1-4-9-5-10}, {1-4-9-5-11}

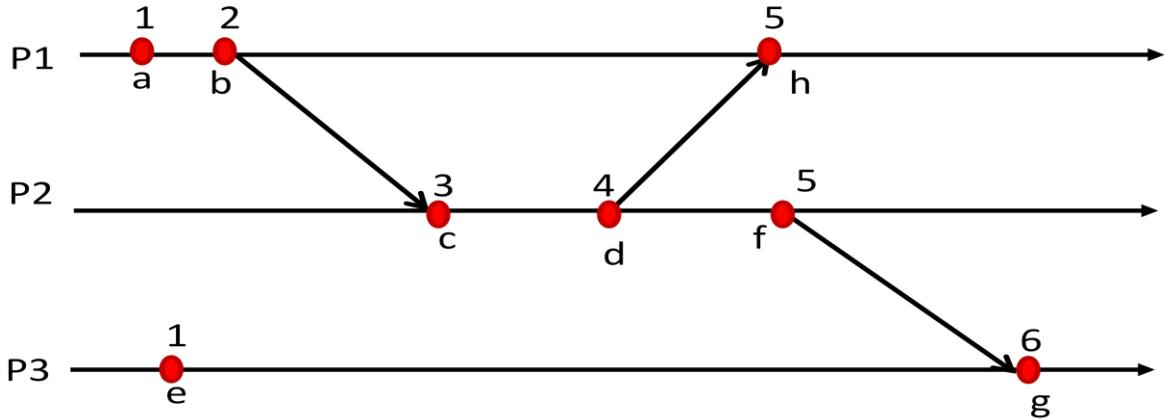
Ans: B. {1-4-8-5-10}, {1-4-8-5-11}, {1-4-9-5-10}, {1-4-9-5-11}, {1-3-6-12}, {1-3-6-13}, {1-3-7-14} and {1-3-7-15}

Explanation: If any site fails, the algorithm substitutes for that site two possible paths starting from the site's two children and ending in leaf nodes.

For example, when node 2 fails, we consider possible paths starting from children 4 and 5 and ending at leaf nodes. The possible paths starting from child 4 are 4-8 and 4-9, and from child 5 are 5-10 and 5-11. So, when node 2 fails, the following eight quorums can be formed:

{1-4-8-5-10}, {1-4-8-5-11}, {1-4-9-5-10}, {1-4-9-5-11}, {1-3-6-12}, {1-3-6-13}, {1-3-7-14} and {1-3-7-15}

Q. 4 Consider the following figure:



Specify how many possible consistent cuts are there that contain the event h.

- A. 2
- B. 3
- C. 4
- D. 5

Ans: D. 5

Explanation: There are total five possible consistent cuts that contain the event h.

Q. 5. For each critical section (CS) execution, Ricart-Agrawala algorithm requires _____ messages per CS execution and the Synchronization delay in the algorithm is _____.

- A. $3(N - 1), T$
- B. $2(N - 1), T$
- C. $(N - 1), 2T$
- D. $(N - 1), T$

Ans: B. $2(N - 1), T$

Explanation: For each CS execution, Ricart-Agrawala algorithm requires $(N - 1)$ REQUEST messages and $(N - 1)$ REPLY messages. Thus, it requires $2(N - 1)$ messages per CS execution. Synchronization delay in the algorithm is T .

Q. 6 The properties hold for quorums in a coterie are:

- A. Total ordered set and Minimality property
- B. Partial ordered set and Intersection property
- C. Minimality property and Intersection property
- D. Graceful degradation and Relinquish property

Ans: C. Minimality property and Intersection property

Explanation:

A **coterie C** is defined as a set of sets, where each set $g \in C$ is called a **quorum**. The following properties hold for quorums in a coterie:

Intersection property: For every quorum $g, h \in C$, $g \cap h \neq \emptyset$.

For example, sets $\{1,2,3\}$, $\{2,5,7\}$ and $\{5,7,9\}$ cannot be quorums in a coterie because the first and third sets **do not have a common element**.

Minimality property: There should be no quorums g, h in **coterie C** such that $g \supseteq h$. For example, sets $\{1,2,3\}$ and $\{1,3\}$ cannot be quorums in a coterie because the first set is a superset of the second.

Q. 7 The Network Time Protocol (NTP) which is widely used for clock synchronization on the Internet uses the _____ method. The design of NTP involves a _____ of time servers

- A. Differential delay, Binary tree
- B. Offset Delay Estimation, Hierarchical tree
- C. NTP timestamps, Quorum
- D. Roundtrip delay, Hierarchical tree

Ans: B. Offset Delay Estimation, Hierarchical tree

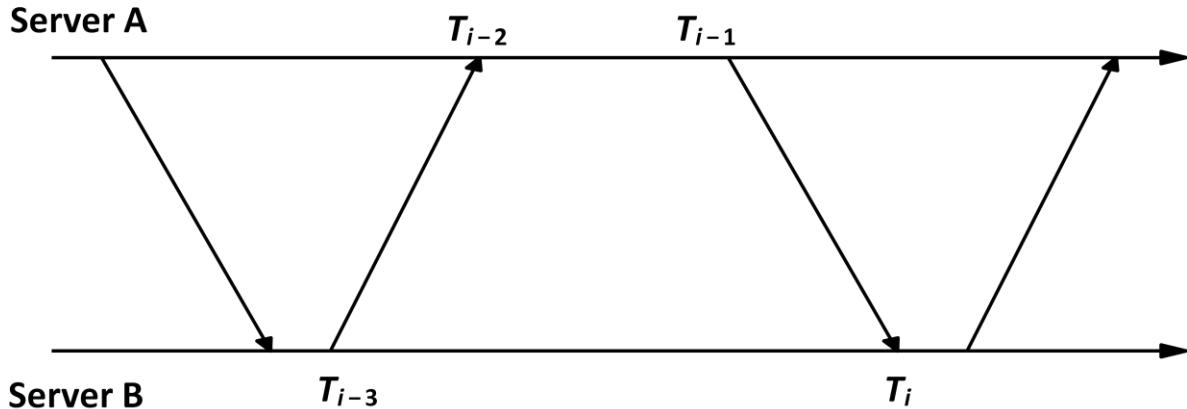
Explanation:

The Network Time Protocol (NTP) which is widely used for clock synchronization on the Internet uses the Offset Delay Estimation method.

The design of NTP involves a hierarchical tree of time servers:

- (i) The primary server at the root synchronizes with the UTC.
- (ii) The next level contains secondary servers, which act as a backup to the primary server.
- (iii) At the lowest level is the synchronization subnet which has the clients.

Q. 8 Consider the following timing diagram for the two servers:



The **offset O_i** and **round-trip delay D_i** can be estimated as:

- A. $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$ and $D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- B. $O_i = (T_{i-2} - T_i)/2$ and $D_i = (T_i - T_{i-2})$
- C. $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)$ and $D_i = (T_i - T_{i-2})$
- D. $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)$ and $D_i = ((T_i - T_{i-3}) - (T_{i-1} - T_{i-2}))/2$

Ans: $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$ and $D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$

Explanation: Derived from the derivation of Clock offset and delay estimation in the Network Time Protocol (NTP)

Q. 9 At 8:27:340 (hr, min, 1/100 sec.), server B requests time from the time-server A. At 8:27:510, server B receives a reply from timeserver A with the timestamp of 8:27:275.

Find out the drift of B's clock with respect to the time-server A's clock (assume there is no processing time at the time-server for time service).

- A. 2 sec
- B. 3 sec
- C. -1.5 sec
- D. -2 sec

Ans: C) -1.5 sec

Explanation: RTT: Reply – Request = 510-340 = 170 1/100sec.

Adjusted local time: Server + RTT/2 = 275 + 85 = 360 1/100sec.

Drift: Adjusted local time – local time = 360 – 510 = -150 1/100sec. = -1.5 sec

Q. 10 The global state recording part of a single instance of the Chandy-Lamport algorithm requires _____ messages and _____ time, where e is the number of edges in the network and d is the diameter of the network.

- A. $O(\log e)$, $O(d)$
- B. $O(e)$, $O(\log d)$
- C. $O(e)$, $O(2d)$
- D. $O(e)$, $O(d)$

Ans: D. $O(e)$, $O(d)$

Explanation: The recording part of a single instance of the algorithm requires $O(e)$ messages and $O(d)$ time, where e is the number of edges in the network and d is the diameter of the network.

Q. 11 For each critical section (CS) execution, maekawa's algorithm requires _____ messages per CS execution and the Synchronization delay in the algorithm is _____.

- A. \sqrt{N} , T
- B. $2\sqrt{N}$, T
- C. $3\sqrt{N}$, T
- D. $3\sqrt{N}$, $2T$

Ans: D. $3\sqrt{N}$, $2T$

Explanation: Since the size of a request set is \sqrt{N} , an execution of the CS requires \sqrt{N} REQUEST, \sqrt{N} REPLY, and \sqrt{N} RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution.

Synchronization delay in this algorithm is $2T$. This is because after a site S_i exits the CS, it first releases all the sites in R_i and then one of those sites sends a REPLY message to the next site that executes the CS.

Q. 12 Agarwal-EI Abbadi quorum-based algorithm uses _____ where All the sites in the system are logically organized into a

-
- A. Coteries, Binary Tree
 - B. Tree-structured quorums, Complete binary tree
 - C. Grid, List
 - D. Unrooted Tree Structure, AVL Tree

Ans: B) Tree-structured quorums, Complete binary tree

Explanation: Agarwal-EI Abbadi quorum-based algorithm uses '**tree-structured quorums**'. All the sites in the system are logically organized into a **complete binary tree**.

- For a complete binary tree with **level 'k'**, we have **$2k + 1 - 1$ sites** with its root at level k and leaves at level 0.

Quiz Assignment-III Solutions: Distributed Systems (Week-3)

Q. 1 Under heavy load, the Raymond's Tree Algorithm requires exchange of only _____ messages per critical section execution.

- A) N, where N is the sites
- B) 4
- C) 3
- D) 0

Ans: B) 4

Explanation: In heavy load, the algorithm requires exchange of only four messages per CS execution.

Q. 2 Consider the following statement:

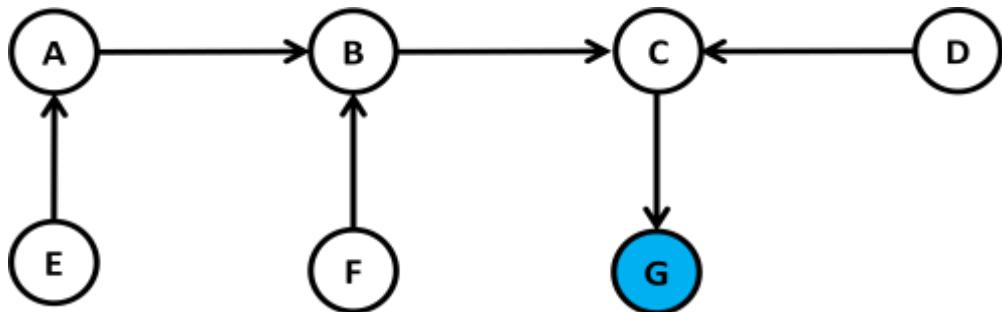
"In Suzuki-Kasami's Broadcast Algorithm, if a site does not hold the token when it makes a request, the algorithm requires $5N-1$ messages to obtain the token."

- A) True
- B) False

Ans: B) False

Explanation: If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token.

Q. 3 Consider the given tree of Raymond's Tree Algorithm as shown in figure, where node 'G' is the privileged node. Calculate the content of holder variables for various nodes



- A. $\text{HOLDER}_A = E$, $\text{HOLDER}_B = A$, $\text{HOLDER}_C = B$, $\text{HOLDER}_D = C$, $\text{HOLDER}_E = F$,
 $\text{HOLDER}_F = E$, $\text{HOLDER}_G = F$
- B. $\text{HOLDER}_A = B$, $\text{HOLDER}_B = C$, $\text{HOLDER}_C = G$, $\text{HOLDER}_D = C$, $\text{HOLDER}_E = A$,
 $\text{HOLDER}_F = B$, $\text{HOLDER}_G = \text{self}$

- C. $\text{HOLDER}_A = B$, $\text{HOLDER}_B = F$, $\text{HOLDER}_C = B$, $\text{HOLDER}_D = C$, $\text{HOLDER}_E = A$,
 $\text{HOLDER}_F = \text{self}$, $\text{HOLDER}_G = C$
- D. $\text{HOLDER}_A = B$, $\text{HOLDER}_B = C$, $\text{HOLDER}_C = G$, $\text{HOLDER}_D = C$, $\text{HOLDER}_E = A$,
 $\text{HOLDER}_F = B$, $\text{HOLDER}_G = G$

Ans: B. $\text{HOLDER}_A = B$, $\text{HOLDER}_B = C$, $\text{HOLDER}_C = G$, $\text{HOLDER}_D = C$, $\text{HOLDER}_E = A$,
 $\text{HOLDER}_F = B$, $\text{HOLDER}_G = \text{self}$

Explanation:

Each node maintains a **HOLDER** variable that provides information about the placement of the privilege in relation to the node itself.

- A node stores in its **HOLDER** variable the identity of a node that it thinks has the privilege or leads to the node having the privilege.
- For two nodes X and Y, if $\text{HOLDER}_X = Y$, we could redraw the undirected edge between the nodes X and Y as a directed edge from X to Y. In same way we can find:

$\text{HOLDER}_A = B$, $\text{HOLDER}_B = C$, $\text{HOLDER}_C = G$, $\text{HOLDER}_D = C$, $\text{HOLDER}_E = A$, $\text{HOLDER}_F = B$, $\text{HOLDER}_G = \text{self}$

Q. 4 Consider the following regarding the Log-based Rollback Recovery Schemes

Scheme	Paradigm
(P) Pessimistic Logging	(i) Low failure free overhead and simpler recovery
(Q) Optimistic Logging	(ii) It reduces failure free overhead, but complicates recovery.
(R) Casual Logging	(iii) It simplifies recovery but hurts the failure-free performance.

Match the scheme to the paradigms they are based on.

- A. (P): (i), Q: (ii), R: (iii)
B. (P): (iii), Q: (ii), R: (i)
C. (P): (ii), Q: (iii), R: (i)
D. (P): (iii), Q: (i), R: (ii)

Ans: B. (P): (iii), Q: (ii), R: (i)

Explanation: Refer the definitions of Log-based Rollback Recovery Schemes

Q.5 Consider the following statement:

“Pease showed that in a fully connected network, it is impossible to reach an agreement if number of faulty processors ‘f’ exceeds $\lfloor (n - 1)/3 \rfloor$ where n is the number of processors”

- A. True
B. False

Ans: A. True

Explanation: Please statement is correct

Q. 6 "Koo-Toueg algorithm is a uncoordinated checkpointing and recovery technique that takes a consistent set of checkpointing and avoids domino effect and livelock problems during the recovery"

- A. True
- B. False

Ans: B. False

Explanation: Koo and Toueg (1987) proposed a coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and avoids 'domino effect' and 'livelock problems' during the recovery

Q 7. Messages whose 'send' is done but 'receive' is undone due to rollback are called

- A. In-transit message
- B. Lost messages
- C. Orphan messages
- D. Duplicate messages

Ans: B. Lost messages

Explanation:

Lost messages –messages whose 'send' is done but 'receive' is undone due to rollback

Q. 8 Consider the following statements related to Process Failure Models

- i. Fail-stop: In this model, a properly functioning process may fail by stopping execution from some instant thenceforth. Additionally, other processes can learn that the process has failed.
 - ii. Crash: A properly functioning process may fail by intermittently receiving only some of the messages sent to it, or by crashing.
 - iii. Receive omission: In this model, a properly functioning process may fail by stopping to function from any instance thenceforth. Unlike the fail-stop model, other processes do not learn of this crash.
 - iv. Send omission: A properly functioning process may fail by intermittently sending only some of the messages it is supposed to send, or by crashing.
- A. Only (i) & (ii) are true
 - B. Only (i), (ii) and (iii) are true
 - C. Only (i) & (iv) are true
 - D. All are true

Ans: Only (i) & (iv) are true

Explanation: The correct definitions are:

- i. Fail-stop: In this model, a properly functioning process may fail by stopping execution from some instant thenceforth. Additionally, other processes can learn that the process has failed.
- ii. Crash: In this model, a properly functioning process may fail by stopping to function from any instance thenceforth. Unlike the fail-stop model, other processes do not learn of this crash.
- iii. Receive omission: A properly functioning process may fail by intermittently receiving only some of the messages sent to it, or by crashing.
- iv. Send omission: A properly functioning process may fail by intermittently sending only some of the messages it is supposed to send, or by crashing.

Q. 9 Cascaded rollback which causes the system to roll back to too far in the computation (even to the beginning), in spite of all the checkpoints is known as:

- A. Rollback
- B. Phantom Effect
- C. Livelock
- D. Domino Effect

Ans: D. Domino Effect

Explanation: Refer the definition of Domino Effect

Q. 10 Consider the given table of agreement problems and match the correct pair:

Agreement Problem	Agreement Condition
(P) Byzantine Agreement Problem	(i) All processes have an initial value and All non-faulty processes must agree on the same (single) value.
(Q) Consensus Problem	(ii) All processes have an initial value and All non-faulty processes must agree on the same array of values $A[v_1 \dots v_n]$.
(R) Interactive Consistency Problem	(iii) Single source has an initial value and All non-faulty processes must agree on the same value.

- A. (P): (i), (Q): (ii), (R): (iii)
- B. (P): (iii), (Q): (ii), (R): (i)
- C. (P): (iii), (Q): (i), (R): (ii)
- D. (P): (ii), (Q): (iii), (R): (i)

Ans: C. (P): (iii), (Q): (i), (R): (ii)

Explanation: Refer the definitions of Agreement Problem

Quiz Assignment-IV Solutions: Distributed Systems (Week-4)

Q. 1 In wait for graph (WFG), a directed edge from node P1 to node P2 indicates that:

- A. P1 is blocked and is waiting for P2 to release some resource
- B. P2 is blocked and is waiting for P1 to release some resource
- C. P1 is blocked and is waiting for P2 to leave the system
- D. P2 is blocked and is waiting for P1 to leave the system

Ans: A) P1 is blocked and is waiting for P2 to release some resource

Explanation: The state of the system can be modeled by directed graph, called a wait for graph (WFG). In a WFG , nodes are processes and there is a directed edge from node P1 to node P2 if P1 is blocked and is waiting for P2 to release some resource.

Q. 2 In the wait for graph, if there exists a directed cycle or knot:

- A. then a deadlock does not exist
- B. then a deadlock exists
- C. then the system is in a safe state
- D. either b or c

Ans: B) then a deadlock exists

Explanation: A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Q. 3 Consider the following statements:

A deadlock detection algorithm must satisfy the following two conditions:

Condition 1: Progress (No false deadlocks): The algorithm should not report deadlocks which do not exist.

Condition 2: Safety (No undetected deadlocks): The algorithm must detect all existing deadlocks in finite time.

- A. Both conditions are true
- B. Both conditions are false
- C. Only condition 1 is true
- D. Only condition 2 is true

Ans: B) Both conditions are false

Explanation: A deadlock detection algorithm must satisfy the following two conditions:

(i) Progress (No undetected deadlocks): The algorithm must detect all existing deadlocks in finite time. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

(ii) Safety (No false deadlocks): The algorithm should not report deadlocks which do not exist (called phantom or false deadlocks).

Q. 4 Consider the following regarding the models of deadlock:

Model of Deadlock	Design Paradigm
(P) Single Resource Model	(i) No assumptions are made regarding the underlying structure of resource requests.
(Q) AND Model	(ii) Presence of a knot indicates a deadlock.
(R) OR Model	(iii) The out degree of a node in the WFG can be more than 1.
(S) Unrestricted model,	(iv) Maximum out-degree of a node in a WFG can be 1

Match the deadlock models to the design paradigms they are based on.

- A. (P): (i), Q: (ii), R: (iii), (S): (iv)
- B. (P): (iv), Q: (iii), R: (ii), (S): (i)
- C. (P): (iii), Q: (iv), R: (i), (S): (ii)
- D. (P): (ii), Q: (iv), R: (i), (S): (iii)

Ans: B) (P): (iv), Q: (iii), R: (ii), (S): (i)

Explanation:

The Single Resource Model: In the single resource model, a process can have at most one outstanding request for only one unit of a resource. Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

The AND Model: In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process. The out degree of a node in the WFG for AND model can be more than 1.

OR Model: In the OR model, the presence of a knot indicates a deadlock.

Unrestricted Model: In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests. Only one assumption that the deadlock is stable is made and hence it is the most general model.

Q. 5 Consider the following statements related to distributed deadlock detection algorithms:

Statement 1: In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG. The basic idea is to build a global WFG for each site of the distributed system.

Statement 2: In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is verified by propagating special messages called probes, along the edges of the graph. These probe messages are different than the request and reply messages.

- A. Both statements are true
- B. Both statements are false
- C. Only statement 1 is true
- D. Only statement 2 is true

Ans: A) Both statements are true

Q. 6 Consider the following statements related to consistency models:

(i) Strict consistency (SC): Only Write operations issued by the same processor and to the same memory location must be seen by others in that order.

(ii) PRAM memory: Only Write operations issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

(iii) Slow Memory: Any Read to a location (variable) is required to return the value written by the most recent Write to that location (variable) as per a global time reference.

- A. All are True
- B. All are False
- C. Only (i) and (iii) are true
- D. Only (ii) is true

Ans: D) Only (ii) is true

Explanations: (i) Strict consistency (SC): Any Read to a location (variable) is required to return the value written by the most recent Write to that location (variable) as per a global time reference.

(ii) PRAM memory: Only Write operations issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

(iii) Slow Memory: Only Write operations issued by the same processor and to the same memory location must be seen by others in that order.

Q. 7 Choose the correct consistency model that defines the following conditions:

- I. All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.
 - II. Accesses to sync variables are sequentially consistent
 - III. Access to sync variable is not permitted unless all Writes elsewhere have completed
 - IV. No data access is allowed until all previous synchronization variable accesses have been performed
-
- A. Weak consistency
 - B. Causal consistency
 - C. Processor consistency
 - D. Program consistency

Ans: A) Weak consistency

Q. 8 The space complexity is the _____ registers and time complexity is _____ time for n-process bakery algorithm of shared memory mutual exclusion

- A. Upper bound of n , $O(n^2)$
- B. Upper bound of n , $O(n)$
- C. Lower bound of n , $O(n)$
- D. Lower bound of n , $O(n^2)$

Ans: C) Lower bound of n , $O(n)$

Explanation: For Bakery algorithm

Space complexity: lower bound of n registers

Time complexity: $O(n)$ time

Q. 9 In GHS algorithm for Minimum Spanning Tree (MST), an edge adjacent to the fragment with smallest weight and that does not create a cycle is called as:

- A. Weight outgoing edge (WOE)
- B. Least weight outgoing edge (LWOE)
- C. Link outgoing weight edge (LOWE)
- D. Minimum weight outgoing edge (MWOE)

Ans: D) Minimum weight outgoing edge (MWOE)

Explanation: Minimum weight outgoing edge (MWOE) An edge adjacent to the fragment with smallest weight and that does not create a cycle.

Q. 10 The GHS algorithm computes the minimum spanning tree using _____ messages for a graph of N nodes and E edges

- A. Atleast $2|E|$
- B. Atleast $5N \log N$
- C. Atmost $3N\log N + 2|E|$
- D. Atmost $5N \log N + 2|E|$

Ans: D) At most $5N \log N + 2|E|$

Explanation: The upper bound of GHS algorithm is $5N \log N + 2|E|$

Q. 11 Consider the following properties of minimum spanning tree (MST):

MST Property 1: Given a fragment of an MST, let e be a minimum-weight outgoing edge of the fragment. Then joining e and its adjacent non-fragment node to the fragment yields another fragment of an MST.

MST Property 2: If all the edges of a connected graph have same weights, then the MST is unique.

- A. Both are true
- B. Both are false
- C. Only property 1 is true
- D. Only property 2 is true

Ans: C) Only property 1 is true

Explanation: MST Property 1: Given a fragment of an MST, let e be a minimum-weight outgoing edge of the fragment. Then joining e and its adjacent non-fragment node to the fragment yields another fragment of an MST.

MST Property 2: If all the edges of a connected graph have different weights, then the MST is unique.

Quiz Assignment-V Solutions: Distributed Systems (Week-5)

Q. 1 A state in which a process has finished its computation and will not restart any action unless it receives a message is called as

- A. Partially terminated state
- B. Locally terminated state
- C. Globally terminated state
- D. Terminating state

Ans: Locally terminated state

Explanation: A distributed computation is globally terminated if every process is locally terminated and there is no message in transit between any processes.

“Locally terminated” state is a state in which a process has finished its computation and will not restart any action unless it receives a message.

Q. 2 In spanning-tree-based termination detection algorithm of Topor, the best case message complexity is _____ and worst case complexity of the algorithm is _____, where N is the number of processes and M is the number of computation messages exchanged

- A. $O(N)$, $O(M)$
- B. $O(N)$, $O(N*M)$
- C. $O(N^2)$, $O(N^2)$
- D. $O(M)$, $O(N)$

Ans: $O(N)$, $O(N*M)$

Explanation: The best case message complexity of the algorithm is $O(N)$, where N is the number of processes in the computation, which occurs when all nodes send all computation messages in the first round.

The worst case complexity of the algorithm is $O(N*M)$, where M is the number of computation messages exchanged, which occurs when only computation message is exchanged every time the algorithm is executed.

Q. 3 Consider the following statements about termination detection (TD) algorithm

Statement 1: Execution of a termination detection algorithm cannot indefinitely delay the underlying computation.

Statement 2: The termination detection algorithm required addition of new communication channels between processes.

- A. Statement 1 is true and statement 2 is false
- B. Statement 1 is false and statement 2 is true
- C. Both statements are false
- D. Both statements are true

Ans: Statement 1 is true and statement 2 is false

Explanation: A termination detection (TD) algorithm must ensure the following:

1. Execution of a TD algorithm cannot indefinitely delay the underlying computation.
2. The termination detection algorithm must not require addition of new communication channels between processes.

Q. 4 Find out the correct relation between the different message ordering paradigms, where SYNC, FIFO, A and CO denote the set of all possible executions ordered by synchronous order, FIFO order, non- FIFO order and causal order respectively.

- A. $\text{SYNC} \subset \text{FIFO} \subset \text{A} \subset \text{CO}$
- B. $\text{SYNC} \subset \text{FIFO} \subset \text{CO} \subset \text{A}$
- C. $\text{SYNC} \subset \text{CO} \subset \text{FIFO} \subset \text{A}$
- D. $\text{A} \subset \text{FIFO} \subset \text{CO} \subset \text{SYNCH}$

Ans: $\text{SYNC} \subset \text{CO} \subset \text{FIFO} \subset \text{A}$

Q. 5 Consider the dijkstra's self-stabilizing token ring system.

A legitimate state must satisfy the following constraints:

- 1) There must be at least one privilege in the system (liveness or no deadlock).
- 2) Every move from a legal state must again put the system into a legal state (closure).
- 3) During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation).
- 4) Given any two legal states, there is a series of moves that change one legal state to the other (reachability).

- A. All constraints are false
- B. All constraints are true
- C. Constraint 1&2 are true and Constraint 3&4 are false
- D. Constraint 1&2 are false and Constraint 3&4 are true

Ans: All constraints are true

Explanation: A legitimate state must satisfy the following constraints:

1. There must be at least one privilege in the system (liveness or no deadlock).
2. Every move from a legal state must again put the system into a legal state (closure).
3. During an infinite execution, each machine should enjoy a privilege an infinite number of times (no starvation).
4. Given any two legal states, there is a series of moves that change one legal state to the other (reachability).

Q. 6 As proven by Ghosh, the minimum number of states required in a self-stabilizing ring is:

- A. One state
- B. Two states
- C. Three states
- D. Four states

Ans: Three states

Explanation: Dijkstra offered three solutions for a directed ring with n machines, 0, 1,, $n-1$, each having K states, (i) $K \geq n$, (ii) $K = 4$, (iii) $K = 3$.

It was later proven by Ghosh that a minimum of three states is required in a self-stabilizing ring.

Q. 7 Consider the following statements about three phase distributed algorithm

Statement 1: The three phase distributed algorithm is closely structured along the lines of Lamport's algorithm for mutual exclusion.

Statement 2: This algorithm uses $3(n - 1)$ messages for $n - 1$ destinations.

- A. Statement 1 is true and statement 2 is false
- B. Statement 1 is false and statement 2 is true
- C. Both statements are false
- D. Both statements are true

Ans: Both statements are true

Explanation: The 3 phase algorithm is closely structured along the lines of Lamport's algorithm for mutual exclusion. This algorithm uses three phases and $3(n - 1)$ messages for $n - 1$ destinations

Q. 8 Consider the following table of Application-Level Multicast Algorithms:

Algorithm	Paradigm
(P) Communication history-based algorithms	(i) Pinwheel
(Q) Privilege-based	(ii) Propagation tree
(R) Moving sequencer	(iii) RST algorithm
(S) Fixed sequencer	(iv) Totem

Match the algorithm to the correct example

- A. (P): (iii), (Q): (iv), (R): (i), (S): (ii)
- B. (P): (iv), (Q): (iii), (R): (ii), (S): (i)
- C. (P): (i), (Q): (ii), (R): (iv), (S): (iii)
- D. (P): (i), (Q): (iv), (R): (iii), (S): (ii)

Ans: (P): (iii), (Q): (iv), (R): (i), (S): (ii)

Q. 9 Consider the following properties of different multicast algorithms:

Statement 1: Privilege-based multicast algorithms provide (i) causal ordering if closed groups are assumed, and (ii) total ordering.

Statement 2: Moving sequencer algorithms, which work with open groups, provide total ordering.

Statement 3: Fixed sequencer algorithms provide total ordering.

- A. Only statement 1&2 are true
- B. Only statement 1&3 are true
- C. All statements are false
- D. All statements are true

Ans: All statements are true

Quiz Assignment-VI Solutions: Distributed Systems (Week-6)

Q. 1 Consider the following statement:

“A randomized algorithm can ensure that a leader is elected with some probability”. Thus we can solve a variant of the leader election problem that relaxes the condition that eventually a leader must be elected in every admissible execution. The relaxed version of the leader election problem requires:

Safety: In every configuration of every admissible execution, at most one processor is in an elected state

Liveness: At least one processor is elected with some non-zero probability

By considering the above statement, identify the valid situation:

- A. Safety property can be relaxed that is, the algorithm can elect two leaders and the liveness condition has to hold, and the algorithm will always terminate with a leader.
- B. Safety property has to hold with certainty: that is, the algorithm should never elect two leaders and the liveness condition is relaxed, and the algorithm need not always terminate with a leader, rather, it is required to do so with nonzero probability.
- C. Both Safety and liveness property can be relaxed
- D. Both Safety and liveness property has to hold

Answer: B) Safety property has to hold with certainty: that is, the algorithm should never elect two leaders and the liveness condition is relaxed, and the algorithm need not always terminate with a leader, rather, it is required to do so with nonzero probability.

Q. 2 True (or) False

Consider the following statement:

“The approach that is used to devising a randomized leader election algorithm is to use randomization to create asymmetry by having processors choose random pseudo-identifiers, drawn from some range, and then execute a deterministic leader election algorithm.”

- A. True
- B. False

Ans: A) True

Q. 3 Peer-to-peer (P2P) network systems use _____ organization of the network overlay for flexibly sharing resources (e.g., files and multimedia documents) stored across network-wide computers.

- A. Physical-level
- B. Network-level
- C. Transport-level
- D. Application-level

Ans: D) Application-level

Q. 4 In P2P networks, the ongoing entry and exit of various nodes, as well as dynamic insertion and deletion of objects is termed as _____

- A. Peer
- B. Churn
- C. Chunk
- D. Objects

Ans: B) Churn

Q. 5 Which of these is an example of Unstructured P2P overlay network?

- A. CAN
- B. CHORD
- C. Kademila
- D. Gnutella

Ans: D) Gnutella

Q. 6 Consider the following statements:

Statement 1: Exact keyword queries, range queries, attribute-based queries, and other complex queries can be supported in unstructured overlays.

Statement 2: Range queries, keyword queries, attribute queries are difficult to support in structured overlays.

- A. Both statements are false
- B. Both statements are true
- C. Statement 1 is true and statement 2 is false
- D. Statement 1 is false and statement 2 is true

Ans: B) Both statements are true

Q. 7 What is the characteristic of P2P network?

- A. Fault Tolerance
- B. Self-Adaptation
- C. Dealing with instability
- D. All of these

Ans: D) All of these

Q. 8 To keep the GFS (Google File System), highly available there are two strategies used namely _____ and _____.

- A. Fast Recovery , Garbage Collection
- B. Fast Recovery , Chunk Replication
- C. Master Replication , Data Integrity
- D. None of these

Ans: B) Fast Recovery, Chunk Replication

Q. 9 In GFS (Google File System) files are divided in _____ chunks.

- A. Variable Size.
- B. Fixed Size
- C. Both Fixed Size and Variable Size chunks
- D. None of these

Ans: B) Fixed Size

Q. 10 GFS (Google File System) supports following operations:

- A. read, write , make, delete
- B. update, add , read, write
- C. snapshot, remove, del
- D. read, write , open , close

Ans: D) read, write , open , close

Quiz Assignment-VII Solutions: Distributed Systems (Week-7)

Q. 1 The number of maps is usually driven by the total size of _____

- A. tasks
- B. inputs
- C. outputs
- D. None of these

Ans: B) inputs

Explanation: Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs.

Q. 2 _____ function is responsible for consolidating the results produced by each of the Map() functions/tasks.

- A. Reducer
- B. Map
- C. Reduce
- D. All of the mentioned

Ans: C) Reduce

Explanation: The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function.

Q. 3 The Mapreduce framework groups Reducer inputs by key in _____ stage.

- A. sort
- B. shuffle
- C. reduce
- D. None of these

Ans: A) Sort

Explanation: When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

Q. 4 In the local disk of the HDFS namenode the files which are stored persistently are:

- A. Namespace image and edit log
- B. Block locations and namespace image
- C. Edit log and block locations
- D. Namespace image, edit log and block locations

Ans: A) Namespace image and edit log

Q. 5 In HDFS, application data are stored on other servers called_____. All servers are _____ and communicate with each other using _____ protocols.

- A. NameNodes, fully connected, TCP-based
- B. DataNodes, fully connected, TCP-based
- C. NameNodes, loosely connected, UDP-based
- D. DataNodes, loosely connected, UDP-based

Ans: B) DataNodes, fully connected, TCP-based

Q. 6 In Spark, A _____ is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

- A. Spark Streaming
- B. FlatMap
- C. Driver
- D. Resilient Distributed Dataset (RDD)

Ans: D) Resilient Distributed Dataset (RDD)

Explanation: Resilient Distributed Dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

Q. 7 In Spark, a dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called_____, which passes each element through a user-provided function of type $A \rightarrow List[B]$.

- A. Map
- B. Filter
- C. flatMap
- D. Reduce

Ans: C) flatMap

Explanation: A dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called flatMap

Q. 8 Spark also allows programmers to create two restricted types of shared variables to support two simple but common usage patterns such as _____ and_____.

- A. Map, Reduce
- B. Flatmap, Filter
- C. Sort, Shuffle
- D. Broadcast, Accumulators

Ans: D) Broadcast, Accumulators

Explanation: Spark also lets programmers create two restricted types of shared variables to support two simple but common usage patterns such as Broadcast variables and Accumulators.

Spark lets the programmer create a “broadcast variable” object that wraps the value and ensures that it is only copied to each worker once.

Accumulators: These are variables that workers can only “add” to using an associative operation, and that only the driver can read.

Q. 9 Consider the following statements:

Statement-1: Reactive routing protocols ask each host (or many hosts) to maintain global topology information, thus a route can be provided immediately when requested.

Statement-2: Proactive routing protocols have the feature on-demand. Each host computes route for a specific destination only when necessary.

- A. Both statements are false
- B. Both statements are true
- C. Statement 1 is true and statement 2 is false
- D. Statement 1 is false and statement 2 is true

Ans: A) Both statements are false

Explanation: (i) Proactive routing protocols ask each host (or many hosts) to maintain global topology information, thus a route can be provided immediately when requested. But large amount of control messages are required to keep each host updated for the newest topology changes.

(ii) Reactive routing protocols have the feature on-demand. Each host computes route for a specific destination only when necessary. Topology changes which do not influence active routes do not trigger any route maintenance function, thus communication overhead is lower compared to proactive routing protocol.

Q. 10 Most of the on-demand routing protocols use _____ for route discovery. Flooding suffers from _____ problem.

- A. Broadcast, Flooding
- B. Shortest-path, Convergecast
- C. Flooding, Broadcast storm
- D. Broadcast, Convergecast

Ans: C) Flooding, Broadcast storm

Explanation: On-demand routing protocols attract much attention due to their better scalability and lower protocol overhead. But most of them use flooding for route discovery. Flooding suffers from broadcast storm problem.

Broadcast storm problem refers to the fact that flooding may result in excessive redundancy, contention, and collision. This causes high protocol overhead and interference to other ongoing communication sessions.

Quiz Assignment-VIII Solutions: Distributed Systems (Week-8)

Q. 1 Kerberos primarily addresses _____ using a _____ cryptosystem. Kerberos is an authentication system designed for MIT's Project Athena.

- A. Client-server authentication, asymmetric
- B. Client-server authentication, symmetric
- C. Trusted authentication, asymmetric
- D. Privacy issue, symmetric

Ans: B) Client-server authentication, symmetric

Q. 2 The _____ allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.

- A. SSL record protocol
- B. SSL TCP segment
- C. SSL handshake protocol
- D. None of these

Ans: C) SSL handshake protocol

Q. 3 The computers that process transactions for the bitcoin network are commonly called:

- A. Truckers
- B. Linesman
- C. GPU
- D. Miners

Ans: D) Miners

Q. 4 In bitcoin proof-of-work is essentially _____

- A. One-IP-address-one-vote
- B. One-CPU-one-vote
- C. One-Disk-one-vote
- D. One-Memory-one-vote

Ans: B) One-CPU-one-vote

Explanation: Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains.

Q. 5 What is the name of the general ledger that tracks all bitcoin transactions?

- A. The Gox-Chain
- B. The Block-link
- C. The Block-chain
- D. Ledger-Link

Ans: C) The Block-chain

Q. 6 In blockchain technology, _____ are basically computer programs that can automatically execute the terms of a contract.

- A. Smart contracts
- B. Smart Property
- C. Public ledger
- D. Smart Consensus

Ans: A) Smart contracts

Explanation: Smart contracts are basically computer programs that can automatically execute the terms of a contract. When a pre-configured condition in a smart contract among participating entities is met then the parties involved in a contractual agreement can be automatically made payments as per the contract in a transparent manner.

Q. 7 The _____ and _____ are two important characteristics of blockchain technology.

- A. Checkpoints, Non-anonymity
- B. Consensus, Agreement
- C. Distributed consensus, Anonymity
- D. None of these

Ans: C) distributed consensus, anonymity

Q. 8 Consider the following statements:

Statement 1: In one-way authentication, only one principal verifies the identity of the other principal.

Statement 2: In mutual authentication, both communicating principals verify each other's identity.

- A. Both statements are false
- B. Both statements are true
- C. Statement 1 is true and statement 2 is false
- D. Statement 1 is false and statement 2 is true

Ans: B) Both statements are true