UNIT 2
ARCHITECTURE

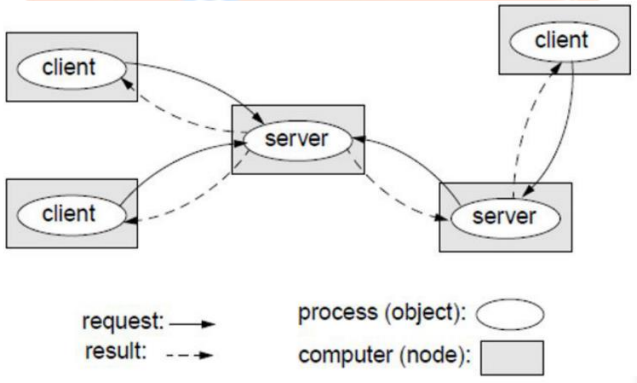
4 Hrs

ARCHITECTURAL MODEL

- The architectural model describes responsibilities distributed between system components and how are these components placed.
- **Client-server model**
- The system is structured as a set of processes, called servers that offer services to the users, called clients.
- The client-server model is usually based on a simple request/reply protocol, implemented with send/receive primitives or using remote procedure calls (RPC) or remote method invocation (RMI):
- The client sends a request (invocation) message to the server asking for some service;

CLIENT-SERVER MODEL

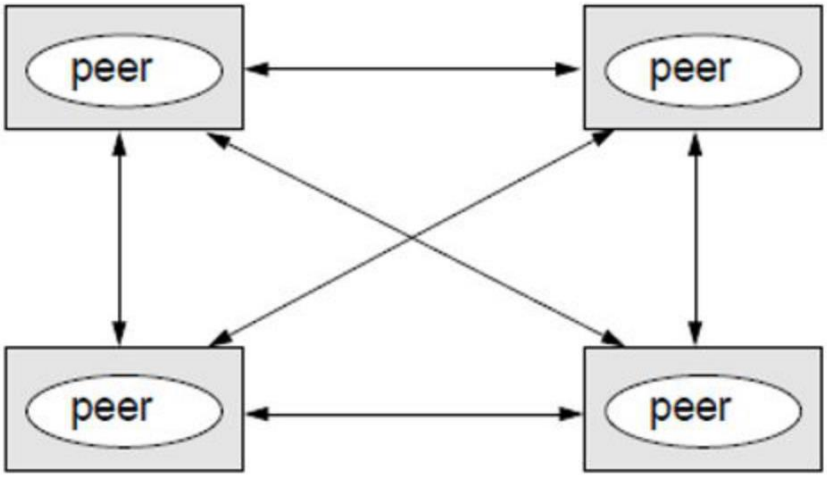
The server does the work and returns a result (e.g. the data requested) or an error code if the work could not be performed.



PEER TO PEER MODEL

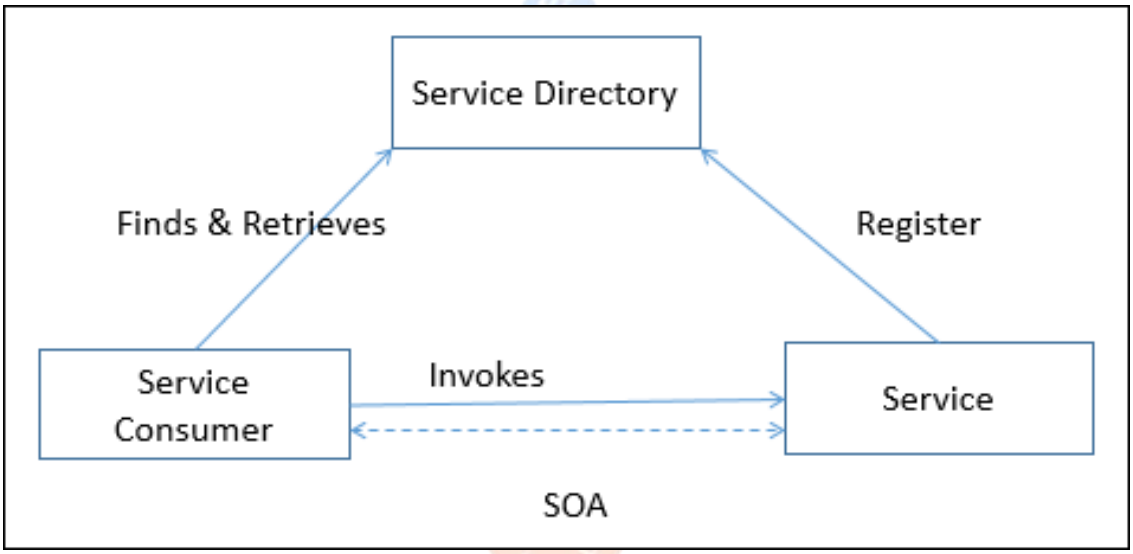
- All processes (objects) play a similar role.
- Processes (objects) interact without a particular distinction between clients and servers.
- The pattern of communication depends on the particular application.
- A large number of data objects are shared; any individual computer holds only a small part of the application database.
- Processing and communication loads for access to objects are distributed across many computers and access links.
- This is the most general and flexible model.

PEER TO PEER MODEL



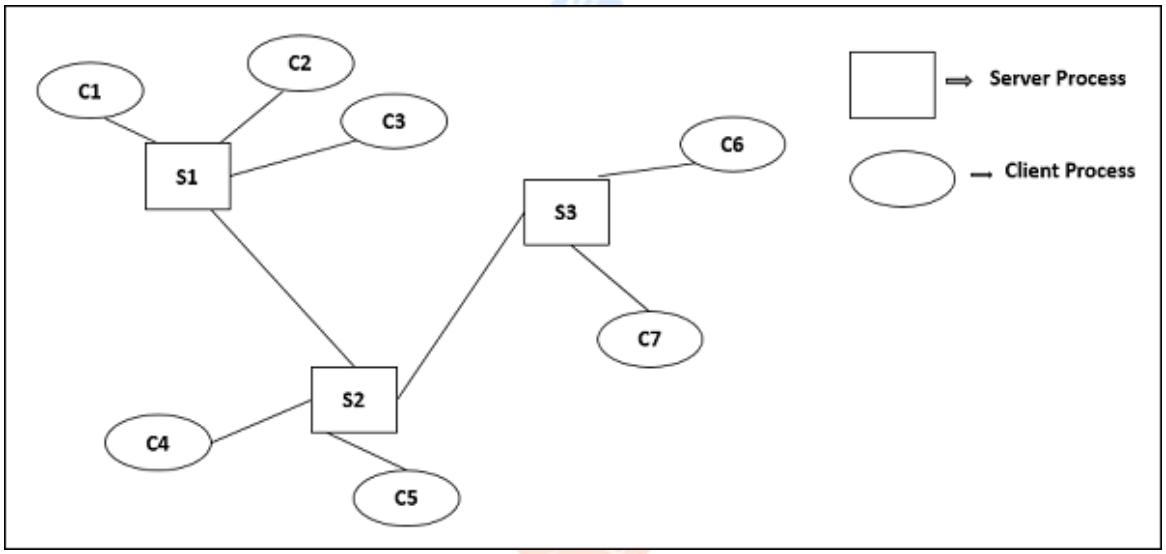
SERVICE-ORIENTED ARCHITECTURE (SOA)

- A service is a component of business functionality that is well-defined, self-contained, independent, published, and available to be used via a standard programming interface.
- The connections between services are conducted by common and universal message-oriented protocols such as the SOAP Web service protocol, which can deliver requests and responses between services loosely.
- Service-oriented architecture is a client/server design which support business-driven IT approach in which an application consists of software services and software service consumers (also known as clients or service requesters).



MULTI-TIER ARCHITECTURE (N-TIER ARCHITECTURE)

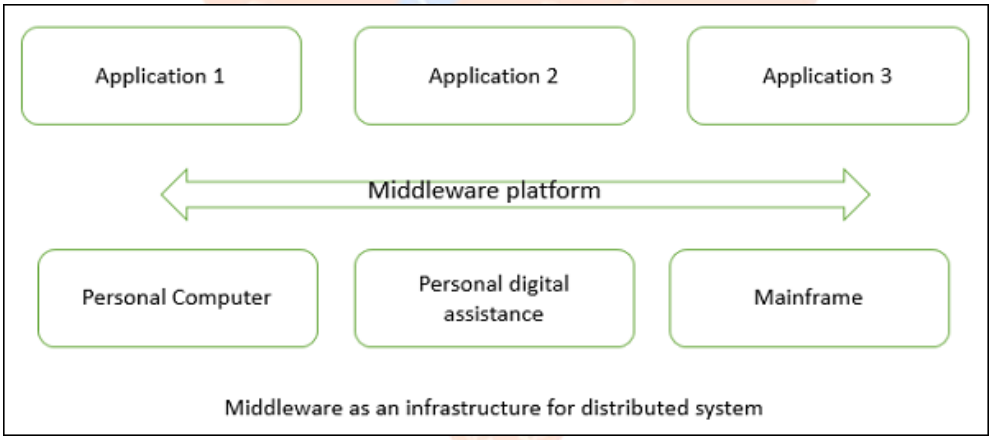
- Multi-tier architecture is a client–server architecture in which the functions such as presentation, application processing, and data management are physically separated.
- By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application.
- It provides a model by which developers can create flexible and reusable applications.



MIDDLEWARE

- Middleware is an infrastructure that appropriately supports the development and execution of distributed applications.
- It provides a buffer between the applications and the network.
- It sits in the middle of system and manages or supports the different components of a distributed system.
- Examples are transaction processing monitors, data convertors and communication controllers etc.

MIDDLEWARE



Middleware as an infrastructure for distributed system

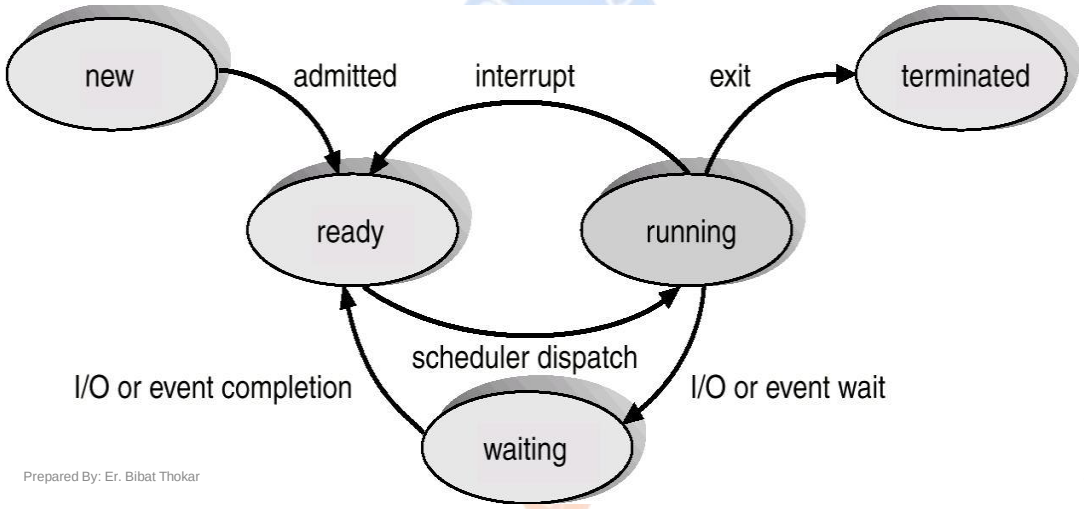
UNIT 3
PROCESSES

6 Hrs.

PROCESS

- A process is defined as an entity which represents the basic unit of work to be implemented in the system i.e. a process is a program in execution.
- The execution of a process must progress in a sequential fashion.
- In general, a process will need certain resources such as the CPU time, memory, files, I/O devices and etc. to accomplish its task.

PROCESS STATE (FIVE STATE MODEL)



PROCESS STATE (FIVE STATE MODEL)

- New: A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.
- Ready: Process that is prepared to execute when given the opportunity. i.e. they are not waiting on anything except the CPU availability.
- Running: The process that is currently being executed.

PROCESS STATE (FIVE STATE MODEL)

- Blocked: A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- Exit:
  - A process that has been released from the pool of executable processes by the OS, either because it is halted or because it is aborted for some reason
  - A process that has been released by OS either after normal termination or after abnormal termination (error).

THREADS

- A thread is a single sequence stream within in a process.
- Because threads have some of the properties of processes, they are sometimes called lightweight processes.
- In many respects, threads are popular way to improve application through parallelism.
- The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

THREADS

- Like a traditional process (i.e. process with one thread), a thread can be in any of several states (Running, Blocked, Ready or terminated).
- A thread consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like process.
- Threads shares address space, program code, global variables, and OS resources with other thread.

THREADS

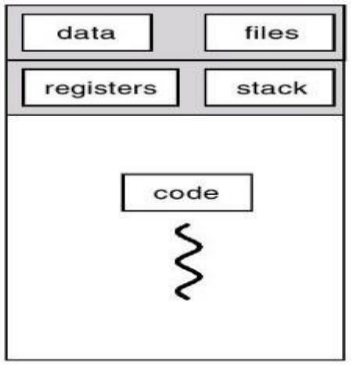


Figure: Process with Single Thread

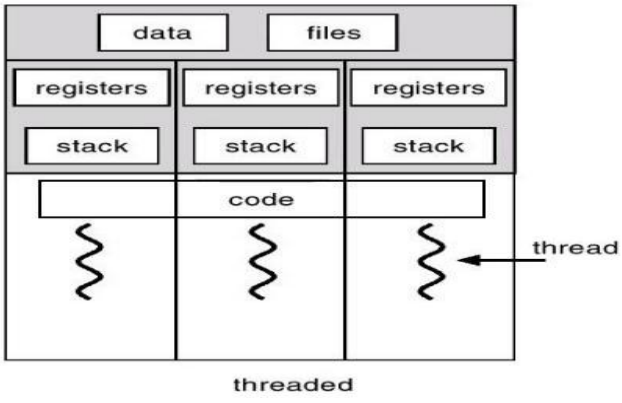


Figure: Process with Multiple Thread

WHY THREAD?

- Process with multiple threads makes a great server (e.g. print server).
- Increase responsiveness, i.e. with multiple threads in a process, if one threads blocks then other can still continue executing.
- Sharing of common data reduce requirement of inter-process communication.
- Proper utilization of multiprocessor by increasing concurrency.
- Threads are cheap to create and use very little resources.
- Context switching is fast (only have to save/reload PC, Stack, SP, Registers).

VIRTUALIZATION

- Virtualization is a process that allows for more efficient utilization of physical computer hardware and is the foundation of cloud computing.
- Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer-processors, memory, storage and more-to be divided into multiple virtual computers, commonly called virtual machines (VMs).

VIRTUALIZATION

- Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware.
- Virtualization enables cloud providers to serve users with their existing physical computer hardware; it enables cloud users to purchase only the computing resources they need when they need it, and to scale those resources cost-effectively as their workloads grow.

VIRTUALIZATION: ADVANTAGES

- Resource efficiency
  - Server virtualization lets you run several applications-each on its own VM with its own OS-on a single physical computer (typically an x86 server) without sacrificing reliability
- Easier management
  - Replacing physical computers with software-defined VMs makes it easier to use and manage policies written in software.
  - It allows you to create automated IT service management workflows.
  - For example, automated deployment and configuration tools enable administrators to define collections of virtual machines and applications as services, in software templates

VIRTUALIZATION: ADVANTAGES

- Minimal downtime
  - OS and application crashes can cause downtime and disrupt user productivity.
  - Admins can run multiple redundant virtual machines alongside each other and failover between them when problems arise.
- Faster provisioning
  - Buying, installing, and configuring hardware for each application is time-consuming.
  - Provided that the hardware is already in place, provisioning virtual machines to run all your applications is significantly faster.

VIRTUALIZATION: TYPES

- Desktop virtualization
  - Desktop virtualization lets you run multiple desktop operating systems, each in its own VM on the same computer.
- Network virtualization
  - Network virtualization uses software to create a “view” of the network that an administrator can use to manage the network from a single console.
  - It abstracts hardware elements and functions (e.g., connections, switches, routers, etc.) and abstracts them into software running on a hypervisor.

VIRTUALIZATION: TYPES

- Storage virtualization
  - Storage virtualization enables all the storage devices on the network-whether they're installed on individual servers or standalone storage units-to be accessed and managed as a single storage device.
- Data virtualization
  - Data virtualization tools create a software layer between the applications accessing the data and the systems storing it.
  - The layer translates an application's data request or query as needed and returns results that can span multiple systems

VIRTUALIZATION: TYPES

- Application virtualization
  - Application virtualization runs application software without installing it directly on the user's OS.
- Data center virtualization
  - Data center virtualization abstracts most of a data center's hardware into software, effectively enabling an administrator to divide a single physical data center into multiple virtual data centers for different clients.

VIRTUALIZATION: TYPES

- CPU virtualization
  - CPU (central processing unit) virtualization is the fundamental technology that makes hypervisors, virtual machines, and operating systems possible.
  - It allows a single CPU to be divided into multiple virtual CPUs for use by multiple VMs.
- GPU virtualization
  - A GPU (graphical processing unit) is a special multi-core processor that improves overall computing performance by taking over heavy-duty graphic or mathematical processing.

VIRTUALIZATION: TYPES

- Linux virtualization
  - Linux includes its own hypervisor, called the kernel-based virtual machine (KVM), which supports Intel and AMD's virtualization processor extensions so you can create x86-based VMs from within a Linux host OS.
- Cloud virtualization
  - As noted above, the cloud computing model depends on virtualization.
  - By virtualizing servers, storage, and other physical data center resources, cloud computing providers can offer a range of services to customers

CLIENT AND SERVER MODEL

- Client and server networking model is a model in which computers such as servers provide the network services to the other computers such as clients to perform a user based tasks.
- Strategies:
- An application program is known as a client program, running on the local machine that requests for a service from an application program known as a server program, running on the remote machine.
- A client program runs only when it requests for a service from the server while the server program runs all time as it does not know when its service is required.

CLIENT AND SERVER MODEL

- A server provides a service for many clients not just for a single client. Therefore, we can say that client-server follows the many-to-one relationship. Many clients can use the service of one server.
- Services are required frequently, and many users have a specific client-server application program. For example, the client-server application program allows the user to access the files, send e-mail, and so on.
- If the services are more customized, then we should have one generic application program that allows the user to access the services available on the remote computer.

CLIENT AND SERVER MODEL

- Client
  - A client is a program that runs on the local machine requesting service from the server.
  - A client program is a finite program means that the service started by the user and terminates when the service is completed.
- Server
  - A server is a program that runs on the remote machine providing services to the clients. When the client requests for a service, then the server opens the door for the incoming requests, but it never initiates the service.
  - A server program is an infinite program means that when it starts, it runs infinitely unless the problem arises. The server waits for the incoming requests from the clients. When the request arrives at the server, then it responds to the request.

ADVANTAGES OF CLIENT-SERVER NETWORKS

- Centralized:
  - Centralized back-up is possible in client-server networks, i.e., all the data is stored in a server.
- Security:
  - These networks are more secure as all the shared resources are centrally administered.

ADVANTAGES OF CLIENT-SERVER NETWORKS

- Performance:
  - The use of the dedicated server increases the speed of sharing resources. This increases the performance of the overall system.
- Scalability:
  - We can increase the number of clients and servers separately, i.e., the new element can be added, or we can add a new node in a network at any time.

DISADVANTAGES OF CLIENT-SERVER NETWORK

- Traffic Congestion is a big problem in Client/Server networks.
- When a large number of clients send requests to the same server may cause the problem of Traffic congestion.
- It does not have a robustness of a network, i.e., when the server is down, then the client requests cannot be met.

DISADVANTAGES OF CLIENT-SERVER NETWORK

- A client/server network is very decisive. Sometimes, regular computer hardware does not serve a certain number of clients. In such situations, specific hardware is required at the server side to complete the work.
- Sometimes the resources exist in the server but may not exist in the client. For example, If the application is web, then we cannot take the print out directly on printers without taking out the print view window on the web.

UNIT 4
COMMUNICATIONS

5 Hrs.

INTRODUCTION

- Communication between two processes in a distributed system is required to exchange various data, such as code or a file, between the processes.
- When one source process tries to communicate with multiple processes at once, it is called Group Communication
- Communication Paradigms:
  - Inter-process Communication
  - Remote Invocation
  - Indirect Communication

INTER-PROCESS COMMUNICATION

- Process that coexist on the memory at a given time are called concurrent process.
- The concurrent process may either be independent or cooperating.
- The independent process, as the name implies do not share any kind of information or data with each other. They just compete with each other for resources like CPU, I/O devices etc. that are required to accomplish their task.

INTER-PROCESS COMMUNICATION

- The cooperating processes on other hand share, need to exchange data or information with each other.
- The cooperating processes require some mechanism to exchange data or pass information to each other. One such mechanism is inter-process communication (IPC).
- Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.

ISSUES IN INTER-PROCESS COMMUNICATION

- How one process can pass information to another?
- The second has to do with making sure two or more processes do not get in each other's way, for example, two processes in an airline reservation system each trying to grab the last seat on a plane for a different customer.
- The third concerns proper sequencing when dependencies are present: if process A produces data and process B prints them, B has to wait until A has produced some data before starting to print.

CHARACTERISTICS OF IPC

- Synchronous and asynchronous communication
  - In synchronous form of communication, the sending and receiving processes synchronize at every message having blocking operations.
  - In asynchronous form of communication, the use of the send operation is non blocking.
    - The sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process.
  - The receive operation can have blocking and non-blocking variants.

CHARACTERISTICS OF IPC

- Message destinations
  - The messages are in the Internet protocols i.e. messages are sent to (Internet address, local port) pairs.
  - A local port is a message destination within a computer.
  - A port has exactly one receiver but can have many senders.
  - Processes may use multiple ports to receive messages.
  - Any process that knows the number of a port can send a message to it.

CHARACTERISTICS OF IPC

- Reliability
  - A point-to-point message service can be described as reliable if messages are guaranteed to be delivered.
  - A point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost.

CHARACTERISTICS OF IPC

- Ordering
  - Some applications require that messages be delivered in the order in which they were transmitted by the sender

INTER-PROCESS COMMUNICATION

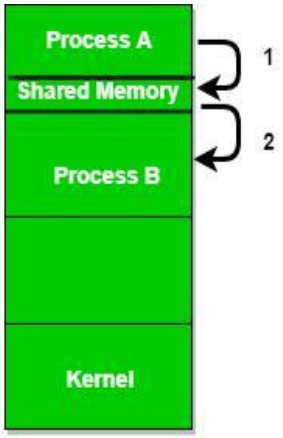
- IPC allows the processes running on the single system to communicate with other. Two basic communication model for providing IPC are:
  - Shared Memory
  - Message Passing

SHARED MEMORY

- Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it.
- One way of communication using shared memory can be imagined like this:
  - Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory.

SHARED MEMORY

- When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly.
- Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

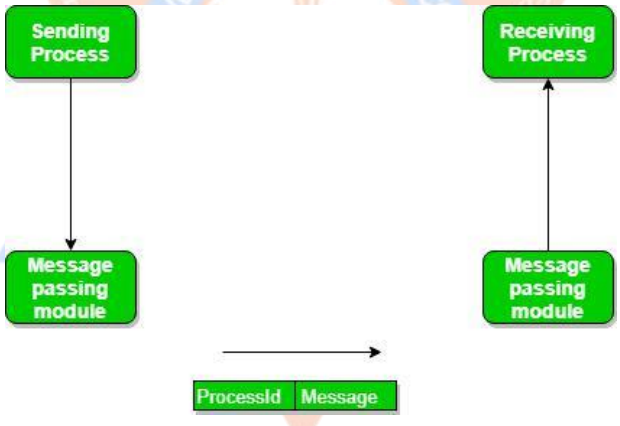


MESSAGE PASSING

- In this method, processes communicate with each other without using any kind of shared memory.
- If two processes p1 and p2 want to communicate with each other, they proceed as follow:
- Establish a communication link (if a link already exists, no need to establish it again).
- Start exchanging messages using basic primitives.

MESSAGE PASSING

- **We need at least two primitives:**
  - **send(message, destination) or send(message)**
  - **receive(message, host) or receive(message)**



SOCKETS

- A socket is one endpoint of a two-way communication link between two programs running on the network.
- Inter-process communication consists of transmitting a message between a socket in one process and a socket in another process.
- The java API for inter-process communication in the internet provides both datagram and stream communication.
- In both forms of communication, UDP and TCP, use the socket abstraction, which provides an endpoint for communication between processes.

REMOTE INVOCATION OR CLIENT SERVER COMMUNICATION

- It is the calling of a remote operation, procedure or method
- Remote Procedure Calls:
  - Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.

REMOTE INVOCATION OR CLIENT SERVER COMMUNICATION

- Remote Method Invocation:
- RMI (Remote Method Invocation) is a way that a programmer can write object-oriented programming in which objects on different computers can interact in a distributed network. This has the following two key features:
  - Space uncoupling: Senders do not need to know who they are sending to
  - Time uncoupling: Senders and receivers do not need to exist at the same time

REMOTE PROCEDURE CALL (RPC)

- RPC is a communication protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details.
- RPC is used to call other processes on the remote systems like a local system. A procedure call is also sometimes known as a function call or a subroutine call.

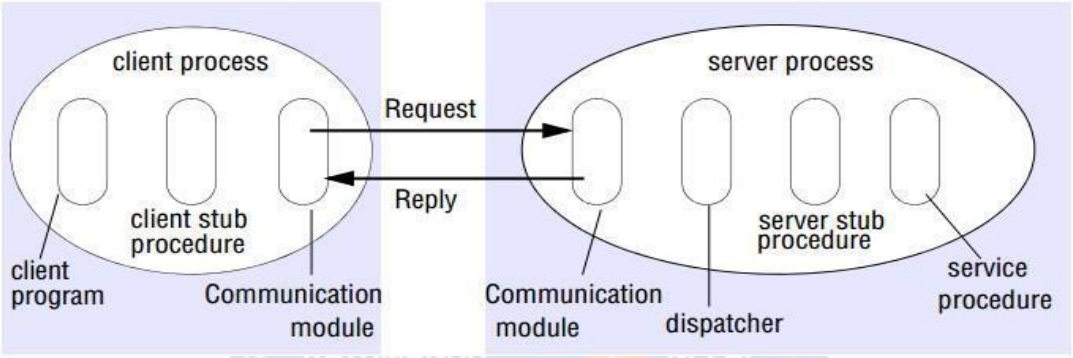
REMOTE PROCEDURE CALL (RPC)

- RPC uses the client-server model. The requesting program is a client, and the service-providing program is the server.
- Like a local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned.
- However, the use of lightweight processes or threads that share the same address space enables multiple RPCs to be performed concurrently.

INTERFACE DEFINITION LANGUAGE (IDL)

- IDL is the specification language used to describe a process component's application programming interface (API)
- IDL is commonly used in Remote Procedure Call. In this case, IDL provides a bridge between the machines at either end of the link that may be using different operating systems and computer languages.

REMOTE METHOD INVOCATION (RMI)



- A stub is a piece of code that is used to convert parameters during a remote procedure call (RPC).

STEPS

- The client calls the client stub. The call is a local procedure call with parameters pushed onto the stack in the normal way.
- The client stub packs the procedure parameters into a message and makes a system call to send the message. The packing of the procedure parameters is called marshalling.
- The client's local OS sends the message from the client machine to the remote server machine.
- The server OS passes the incoming packets to the server stub.

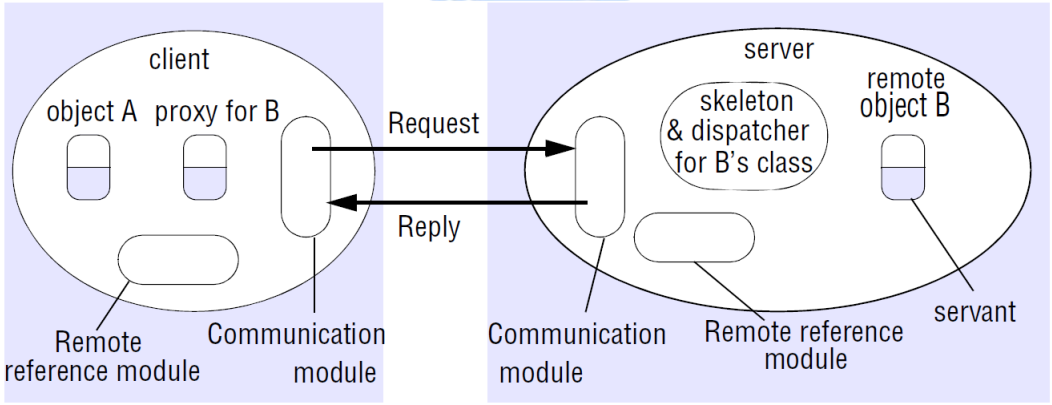
STEPS : CONTINUED

- The server stub unpacks the parameters (called unmarshalling) from the message.
- When the server procedure is finished, it returns to the server stub, which marshals the return values into a message. The server stub then hands the message to the transport layer.
- The transport layer sends the resulting message back to the client transport layer, which hands the message back to the client stub.
- The client stub unmarshalls the return parameters, and execution returns to the caller.

REMOTE METHOD INVOCATION (RMI)

- RMI uses an object oriented paradigm where the user needs to know the object and the method of the object he needs to invoke.
- RMI is a set of protocols being developed by Sun's JavaSoft division that enables Java objects to communicate remotely with other Java objects
- RMI handles the complexities of passing along the invocation from the local to the remote computer.
- But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called.

REMOTE METHOD INVOCATION (RMI)



STEPS

- Caller calls a local procedure implemented by the stub
- Stub marshals call type and the input arguments into a request message
- Client stub sends the message over the network to the server and blocks the current execution thread
- Server skeleton receives the request message from the network

STEPS: CONTINUED

- Skeleton unpacks call type from the request message and looks up the procedure on the called object
- Skeleton un-marshals procedure arguments
- Skeleton executes the procedure on the called object
- Called object performs a computation and returns the result
- Skeleton packs the output arguments into a response message

STEPS: CONTINUED

- Skeleton sends the message over the network back to the client
- Client stub receives the response message from the network
- Stub unpacks output arguments from the message
- Stub passes output arguments to the caller, releases execution thread and caller then continues in execution

RMI

- Remote Reference Module
  - Responsible for translating between local and remote object references and for creating remote object references.
  - When a remote object is to be passed as an argument or a result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.
  - A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object.

RMI

- Remote Reference Module
  - When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object.
  - In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

RMI

- Servants
  - Servant is an instance of a class that provides the body of a remote object.
  - It is the servant that eventually handles the remote requests passed on by the corresponding skeleton.
  - Servants live within a server process. They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted.

RMI

- Communication module
  - The two cooperating communication modules carry out the request-reply protocol, which transmits request and reply messages between the client and server
  - The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the request message

RMI

- Proxy
  - The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to a remote object.
  - It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client.
  - There is one proxy for each remote object for which a process holds a remote object reference.

RMI

- Dispatcher
  - A server has one dispatcher and one skeleton for each class representing a remote object.
  - The dispatcher receives request messages from the communication module. It selects the appropriate method in the skeleton, passing on the request message

RMI

- Skeleton
  - The class of a remote object has a skeleton, which implements the methods in the remote interface.
  - A skeleton method unmarshals the arguments in the request message and invokes the corresponding method in the servant.
  - It waits for the invocation to complete and then marshals the result, together with any exceptions, in a reply message to the sending proxy's method.

DIFFERENCE BETWEEN RPC AND RMI

Remote Procedure Call (RPC)	Remote Method Invocation (RMI)
RPC is a library and OS dependent platform	Whereas it is JAVA platform
RPC supports procedural programming	RMI supports object oriented programming
RPC is less efficient in comparison of RMI	While RMI is more efficient than RPC
RPC creates more overhead	While it creates less overhead than RPC
The parameters which are passed in RPC are ordinary of normal data	While in RMI, object are passed as parameter
RPC is the older version of RMI	While it is successor version of RPC
There is high provision of ease of programming in RPC	While there is low provision of ease of programming in RMI
RPC does not provide any security	While it provides client level security
It's development cost is huge	While it's development cost id fair of reasonable
There is a huge problem of versioning RPC	While there is possible versioning using RDMI
There is multiple codes are simple application in RPC	While there is multiple codes are not needed for simple application in RMI

INDIRECT COMMUNICATION

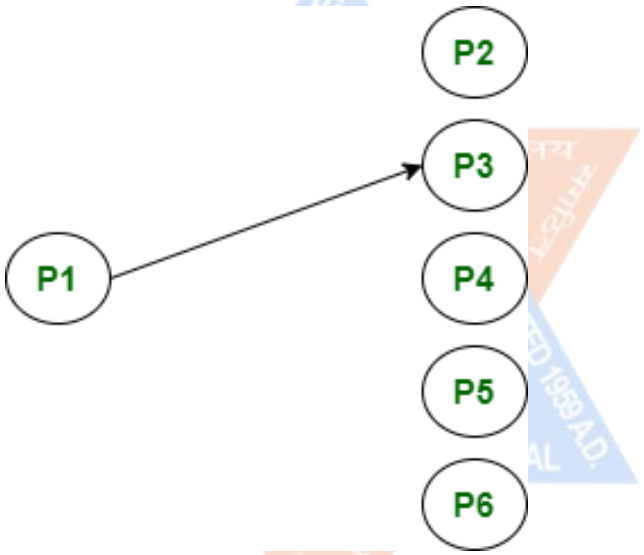
- Group communication
  - Group communication is concerned with the delivery of messages to a set of recipients and hence is a multiparty communication paradigm supporting one-to-many communication.
  - A group identifier uniquely identifies the group. The recipients join the group and receive the messages.

GROUP COMMUNICATION

- Senders send messages to the group based on the group identifier and hence do not need to know the recipients of the message.
- Types:
  - Unicast Communication
  - Multicast Communication
  - Broadcast Communication

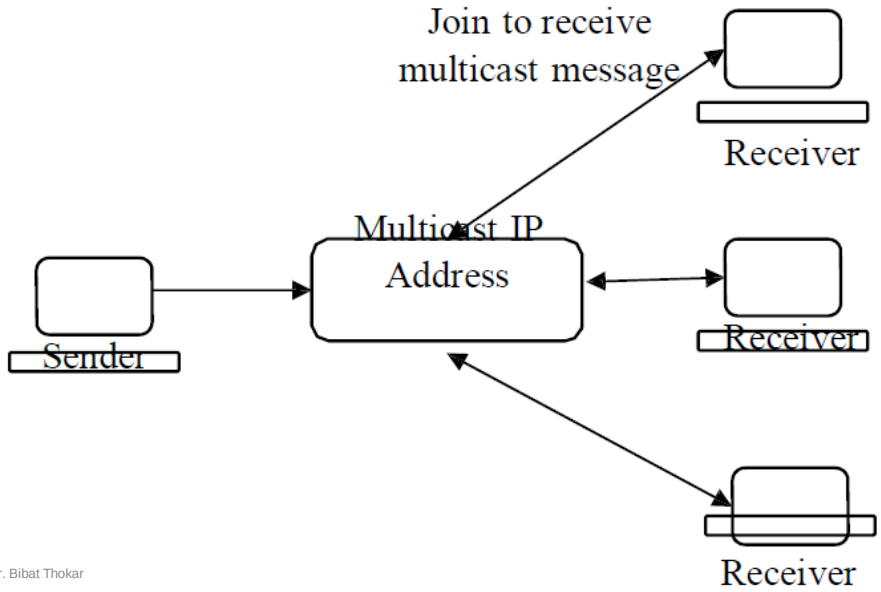
UNICAST COMMUNICATION

- When the host process tries to communicate with a single process in a distributed system at the same time.
- Although, same information may be passed to multiple processes. This works best for two processes communicating as only it has to treat a specific process only.
- However, it leads to overheads as it has to find exact process and then exchange information/data.



MULTICAST COMMUNICATION

- Multicast (one-to-many or many-to-many distribution) is group communication where information is addressed to a group of destination computers simultaneously.
- Multicast operation is an operation that sends a single message from one process to each of the members of a group of processes.
- The simplest way of multicasting, provides no guarantees about message delivery or ordering.



MULTICAST COMMUNICATION: CHARACTERISTICS

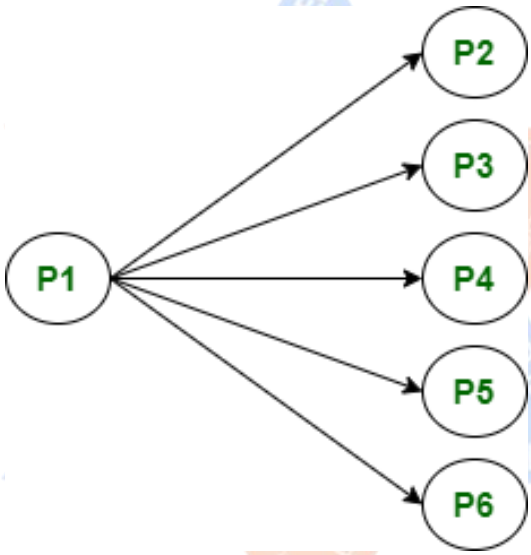
- Fault tolerance based on replicated services:
  - A replicated service consists of a group of servers.
  - Client requests are multicast to all the members of the group, each of which performs an identical operation.
- Better performance through replicated data
  - Data are replicated to increase the performance of a service.

MULTICAST COMMUNICATION: CHARACTERISTICS

- Finding the discovery servers in spontaneous networking
  - Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
- Propagation of event notifications
  - Multicast to a group may be used to notify processes when something happens.

BROADCAST COMMUNICATION

- When the host process tries to communicate with every process in a distributed system at same time.
- Broadcast communication comes in handy when a common stream of information is to be delivered to each and every process in most efficient manner possible.
- Since it does not require any processing whatsoever, communication is very fast in comparison to other modes of communication.
- However, it does not support a large number of processes and cannot treat a specific process individually.



MESSAGE PASSING INTERFACE (MPI)

- Message Passing Interface (MPI) is a standardized and portable message-passing system to function on a wide variety of parallel computers.
- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- The primary goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.
- The interface attempts to be: practical, portable, efficient and flexible.

MESSAGE PASSING INTERFACE (MPI)

- The basic principles of exchanging messages between two processes using send and receive operations is covered in MPI
- There are two types of message passing:
  - Synchronous message passing: This is implemented by blocking send and receive calls.
  - Asynchronous message passing: This is implemented by a non-blocking form of send.
- MPI runs on virtually any hardware platform like distributed memory, shared memory and in hybrid memory.

UNIT 5
NAMING

5 Hrs.

OVERVIEW

- Name, Identifies and Addresses
- Structured Naming
- Attribute-based Naming
- Case Study: The Global Name Service

NAMING: INTRODUCTION

- Names are used to share resources, to uniquely identify entities, to refer to locations, and more.
- An important issue with naming is that a name can be resolved to the entity it refers to. Name resolution thus allows a process to access the named entity.

NAMES, ADDRESSES AND IDENTIFIES

- A name in a distributed system is a string of bits or characters that is used to refer to an entity.
- Entity: An entity in a distributed system can be practically anything. Typical examples include resources such as hosts, printers, disks, and files.

NAMES, ADDRESSES AND IDENTIFIES

- Other well-known examples of entities that are often explicitly named are processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on.
- Entities can be operated on. For example, a resource such as a printer offers an interface containing operations for printing a document, requesting the status of a print job.

NAMES, ADDRESSES AND IDENTIFIES

- To operate on an entity, it is necessary to access it, for which we need an access point. An access point is yet another, but special, kind of entity in a distributed system.
- The name of an access point is called an address. The address of an access point of an entity is also simply called an address of that entity.

NAMES, ADDRESSES AND IDENTIFIES

- An entity can offer more than one access point. As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address
- An entity may change its access points in the course of time. For example, when a mobile computer moves to another location, it is often assigned a different IP address than the one it had before.

NAMES, ADDRESSES AND IDENTIFIES

- Identifier is a name that uniquely identifies an entity. The identifier is unique and refers to only one entity.
- A true identifier is a name that has the following properties:
  - An identifier refers to at most one entity.
  - Each entity is referred to by at most one identifier.
  - An identifier always refers to the same entity (i.e., it is never reused).

NAMES, ADDRESSES AND IDENTIFIES

- In many computer systems, addresses and identifiers are represented in machine-readable form only, that is, in the form of bit strings.
- For example, an Ethernet address is essentially a random string of 48 bits. Likewise, memory addresses are typically represented as 32-bit or 64-bit strings.

NAMES, ADDRESSES AND IDENTIFIES

- Name space:
  - Contains all valid names recognized and managed by a service
- Name resolution:
  - A process to look up information/attributes from a name

NAMING SYSTEMS

- Structured Naming
- Attribute-based Naming

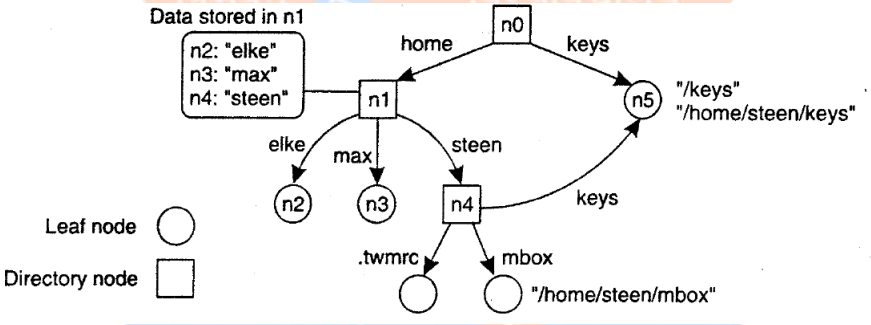
STRUCTURED NAMING

- Names are commonly organized into what is called a name space.
- Name space for structured names can be represented as a labeled, directed graph with two types of nodes:
- Leaf Nodes and
- Directory Nodes.

STRUCTURED NAMING

- A leaf node represents a named entity and has no outgoing edges.
  - It stores information on the entity it is representing.
  - For example: its address so that a client can access it.
- A directory node has a number of outgoing edges each labeled with a name.
  - It stores a table in which an outgoing edge is represented as a pair (edge label, node identifier).
  - Such a table is called a directory table.

GENERAL NAMING GRAPH WITH A SINGLE ROOT NODE



STRUCTURED NAMING

- The graph has one node, namely no, which has only outgoing and no incoming edges. Such a node is called the root (node) of the naming graph.
- Although it is possible for a naming graph to have several root nodes, for simplicity, many naming systems have only one.
- It is an example of directed acyclic graph. In such an organization, a node can have more than one incoming edge, but the graph is not permitted to have a cycle.

NAME RESOLUTION

- Name Space offer a convenient mechanism for storing and retrieving information about entities by means of names.
- With a given path name, name space can look up information stored in the node referred to by that name. This process of looking up a name is called name resolution.
  - Parse the path name and extract the components and then use the hierarchical structure for resolving each component.
- If name resolution is done within a machine, it is a centralized approach.

RESOLVING FILE NAMES ACROSS MACHINES

- Name Resolution in case of distributed systems is similar to that of a centralized approach but different machines may be involved in resolving different parts of the name.
- Directory node in the foreign name space (i.e. a remote or different machine's name space) is called mounting point.
- Normally, the mounting point is the root of a name space. During the name resolution, the mounting point is looked up and resolution proceeds by accessing its directory table.

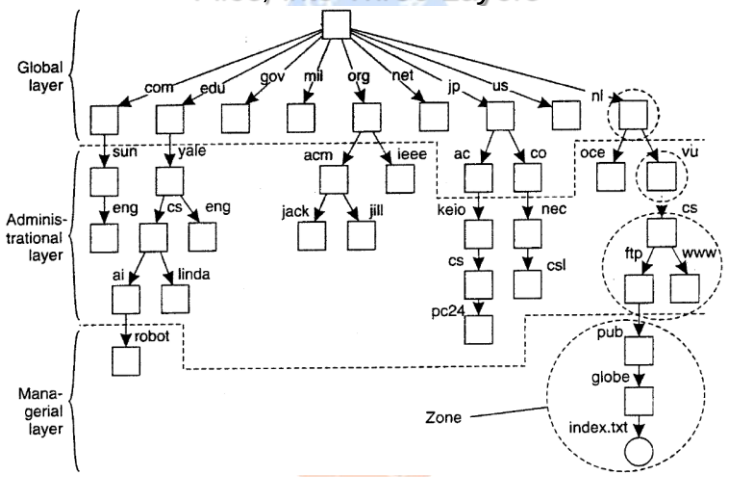
RESOLVING FILE NAMES ACROSS MACHINES

- If the client machine and the file server are both configured with Network File System (NFS) and once the directory of file server is mounted on the client machine, client machine can access the files of that directory like local files.
- In NFS, remote files are accessed using local names(location independence). OS maintains a mount table with the mappings.
- Example : nfs://its.cs.vu.nl//home/steem, this NFS URL names a file called /home/steem on an NFS file server its.cs.vu.nl, which can be accessed by means of the NFS protocol.

NAME SPACE DISTRIBUTION

- Naming in large distributed systems: System may be global in scope (e.g., Internet, WWW). Name space is organized hierarchically.
- Name space is distributed and has three logical layers - global layer, administration layer and managerial layer.
- The global layer is formed by highest-level nodes(root node and other directory nodes close to it). Represents groups of organization and are characterized by their stability.

Partitioning of the DNS Name Space, including Internet-accessible Files, into Three Layers



NAME SPACE DISTRIBUTION

- The administration layer is managed by directory nodes that are together within a single organization. Represent group of entities that belong to the same organization.
- The nodes in these layer are relatively stable. The managerial nodes consists of the nodes that change frequently.
- The more stable a layer, the longer are the lookups valid (and can be cached longer)
- Example: In the partitioning of the DNS name space, domain names like .com,.edu,.au etc come under global layers, then organization name like sun, yale etc come under administration layer.

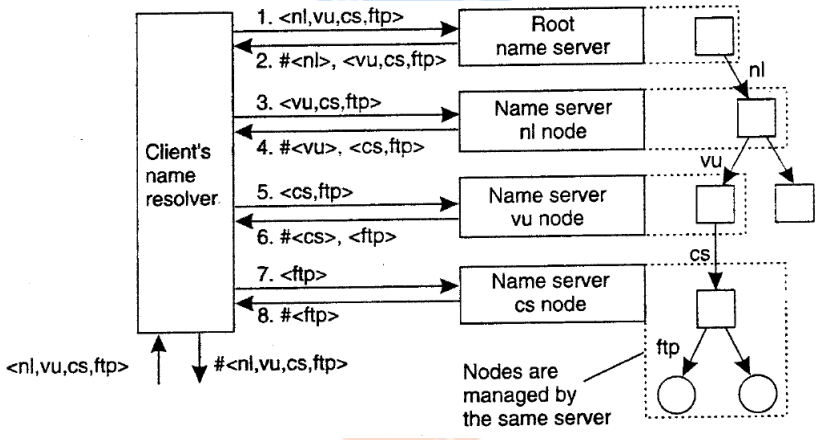
IMPLEMENTATION OF NAME SPACE

- Iterative Resolution
- Recursive Resolution

ITERATIVE RESOLUTION

- In iterative naming system, each component/name is resolved one at a time, going from top level domain and descend down the hierarchy and resolve one at a time.
- Example: ftp://ftp.cs.vu.nl/pub/globe/index.html. In this resolution root server can resolve only the label nl, for which it will return the address of the associated name server.
- Client the passes the remaining path name to a name server, which resolves the label vu and returns the remaining name and this is passed to a name server and so on until all the components are resolved.

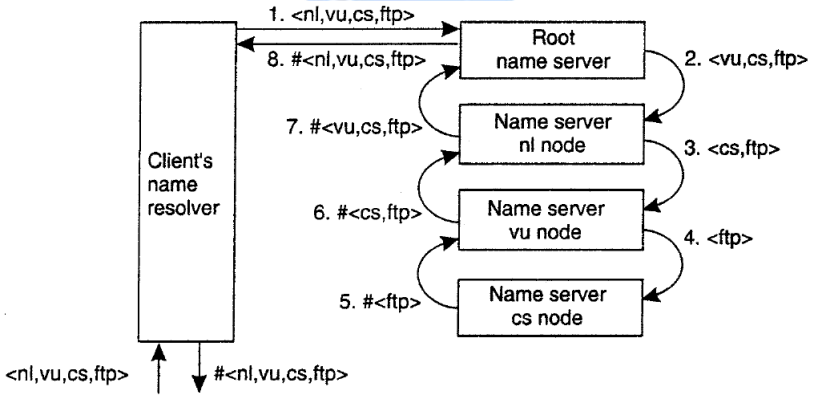
PRINCIPLE OF ITERATIVE NAME RESOLUTION



RECURSIVE RESOLUTION

- Starting from the root, root server will resolve and passes the result to the next name server it finds and so on until all the components are resolved and results will be trickled in opposite direction like in recursion and root name server passes the result to client machine.
- Example: ftp://ftp.cs.vu.nl/pub/globe/index.html. In recursive resolution, root name server resolves only the label nl and passes the remaining path name to a name server, which resolves the label vu and so on until all the components are resolved and results trickle back up to the root name server which pass that to the client.

PRINCIPLE OF RECURSIVE NAME RESOLUTION



PROS AND CONS

- In iterative resolution, caching can reduce overheads, as it eliminates few look-ups and can reduce burden on root name server. On the other hand, communication costs are quite high in case of iterative resolution compared to the recursive resolution
- In recursive resolution, caching results is more effective compared to the iterative name resolution. But, in recursive resolution burden on root name server is quite high, as every request goes through root server. Communication costs are often cheaper in case of recursive resolution comparatively.

STRUCTURED NAMING: EXAMPLE DOMAIN NAME SYSTEM (DNS)

- One of the largest distributed naming services in use today is the Internet
- Domain Name System (DNS). DNS is primarily used for looking up IP addresses of hosts and mail servers.

DNS NAME SPACE

- The DNS name space is hierarchically organized as a rooted tree. A label is a case-insensitive string made up of alphanumeric characters.
- A label has a maximum length of 63 characters; the length of a complete path name is restricted to 255 characters.
- The string representation of a path name consists of listing its labels, starting with the rightmost one, and separating the labels by a dot (H. ").

DNS NAME SPACE

- The root is represented by a dot. So, for example, the path name root: <nl, VU, cs, flits>, is represented by the string flits.cs. vu.nl., which includes the rightmost dot to indicate the root node. We generally omit this dot for readability.
- Because each node in the DNS name space has exactly one incoming edge (with the exception of the root node, which has no incoming edges), the label attached to a node's incoming edge is also used as the name for that node.
- A subtree is called a domain; a path name to its root node is called a domain name. Note that, just like a path name, a domain name can be either absolute or relative.

DNS IMPLEMENTATION

- DNS name space can be divided into a global layer and an administrational layer. The managerial layer, which is generally formed by local file systems, is formally not part of DNS and is therefore also not managed by it.
- Each zone is implemented by a name server, which is virtually always replicated for availability.
- Updates for a zone are normally handled by the primary name server. Updates take place by modifying the DNS database local to the primary server.

DNS IMPLEMENTATION

- Secondary name servers do not access the database directly, but, instead, request the primary server to transfer its content. The latter is called a zone transfer in DNS terminology.
- A DNS database is implemented as a (small) collection of files, of which the most important one contains all the resource records for all the nodes in a particular zone.
- This approach allows nodes to be simply identified by means of their domain name, by which the notion of a node identifier reduces to an (implicit) index into a file.

ATTRIBUTE-BASED NAMING

- Structured names generally provide a unique and location-independent way of referring to entities.
- Moreover, structured names have been partly designed to provide a human-friendly way to name entities so that they can be conveniently accessed.
- However, location independence and human friendliness are not the only criterion for naming entities

ATTRIBUTE-BASED NAMING

- A popular one in distributed systems is to describe an entity in terms of (attribute, value) pairs, generally referred to as attribute-based naming.
- In this approach, an entity is assumed to have an associated collection of attributes, each attribute says something about that entity.
- By specifying which values a specific attribute should have, a user essentially constrains the set of entities that he/she is interested in.
- It is up to the naming system to return one or more entities that meet the user's description.

DIRECTORY SERVICES

- Attribute-based naming systems are also known as directory services, whereas systems that support structured naming are generally called naming systems.
- With directory services, entities have a set of associated attributes that can be used for searching. In some cases, the choice of attributes can be relatively simple.

DIRECTORY SERVICES

- For example, in an e-mail system, messages can be tagged with attributes for the sender, recipient, subject, and so on.
- However, even in the case of e-mail, matters become difficult when other types of descriptors are needed, as is illustrated by the difficulty of developing filters that will allow only certain messages (based on their descriptors) to be passed through.

DIRECTORY SERVICES

- Unlike structured naming systems, looking up values in an attribute-based naming system essentially requires an exhaustive search through all descriptors.
- When considering performance, such a search is less of problem within a single data store, but separate techniques need to be applied when the data is distributed across multiple, potentially dispersed computers.
- Different approaches to solving this problem in distributed systems:

HIERARCHICAL IMPLEMENTATIONS: LDAP

- A common approach to tackling distributed directory services is to combine structured naming with attribute-based naming.
- Many of these systems use, or rely on the lightweight directory access protocol commonly referred simply as LDAP
- Conceptually, an LDAP directory service consists of a number of records, usually referred to as directory entries.
- A directory entry is comparable to a resource record in DNS. Each record is made up of a collection of (attribute. value) pairs, where each attribute has an associated type.

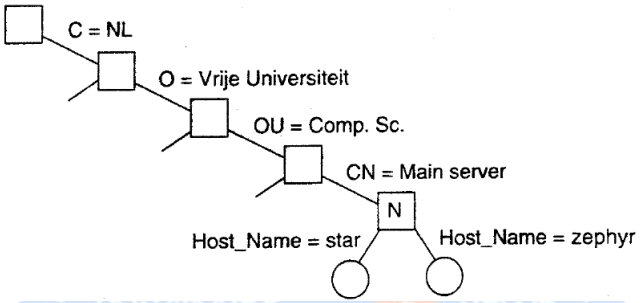
HIERARCHICAL IMPLEMENTATIONS: LDAP

- A distinction is made between single-valued attributes and multiple-valued attributes. The latter typically represent arrays and lists
- The collection of all directory entries in an LDAP directory service is called a directory information base (DIB).
- An important aspect of a DIB is that each record is uniquely named so that it can be looked up.
- Such a globally unique name appears as a sequence of naming attributes in each record. Each naming attribute is called a relative distinguished name, or RDN

HIERARCHICAL IMPLEMENTATIONS: LDAP

- As in DNS, the use of globally unique names by listing RDNs in sequence, leads to a hierarchy of the collection of directory entries, which is referred to as a directory information tree (DIT).
- A DIT essentially forms the naming graph of an LDAP directory service in which each node represents a directory entry.
- In addition. a node may also act as a directory in the traditional sense, in that there may be several children for which the node acts as parent as shown in the picture:

HIERARCHICAL IMPLEMENTATIONS: LDAP



DECENTRALIZED IMPLEMENTATIONS

- The key issue here is that (attribute, value) pairs need to be efficiently mapped so that searching can be done efficiently, i.e., by avoiding an exhaustive search through the entire attribute space.
- To establish such a mapping, Mapping to Distributed Hash Tables can be implemented.

MAPPING TO DISTRIBUTED HASH TABLES (DHT)

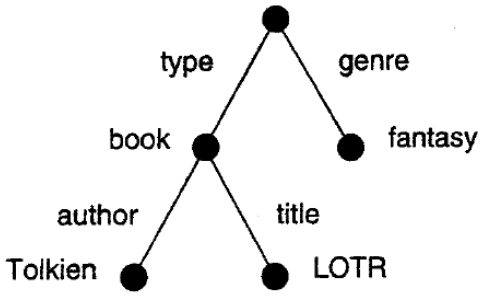
- Consider the case where (attribute, value) pairs need to be supported by a DHT-based system.
- Assume that queries consist of a conjunction of pairs as with LDAP, that is. a user specifies a list of attributes, along with the unique value he wants to see for every respective attribute.
- Advantage of this type of query is that no ranges need to be supported. Range queries may significantly increase the complexity of mapping pairs to a DHT.
- Each entity (referred to as a resource) is assumed to be described by means of possibly hierarchically organized attributes

MAPPING TO DISTRIBUTED HASH TABLES (DHT)

- Each such description is translated into an attribute-value tree (AVTree) which is then used as the basis for an encoding that maps well onto a DHT-based system.
- The main issue is to transform the AVTrees into a collection of keys that can be looked up in a DHT system.
- In this case, every path originating in the root is assigned a unique hash value, where a path description starts with a link (representing an attribute), and ends either in a node (value), or another link.

MAPPING TO DISTRIBUTED HASH TABLES (DHT)

```
description {
    type = book
    description {
        author = Tolkien
        title = LOTR
    }
    genre = fantasy
}
```



(a)General description of a resource (b)Its representation as an AVTree.

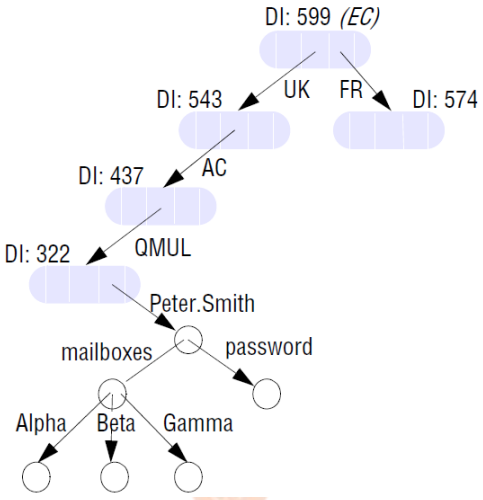
CASE STUDY: GLOBAL NAME SERVICES

- A Global Name Service (GNS) was designed and implemented by Lampson and colleagues at the DEC Systems Research Center to provide facilities for resource location, mail addressing and authentication.
- The designers of the GNS also recognized that the naming database is likely to have a long lifetime and that it must continue to operate effectively while it grows from small to large scale and while the network on which it is based evolves.
- The structure of the name space may change during that time to reflect changes in organizational structures.
- The service should accommodate changes in the names of the individuals, organizations and groups that it holds, and changes in the naming structure such as those that occur when one company is taken over by another

CASE STUDY: GLOBAL NAME SERVICES

- The GNS manages a naming database that is composed of a tree of directories holding names and values.
- Directories are named by multi-part pathnames referred to a root, or relative to a working directory, much like file names in a UNIX file system. Each directory is also assigned an integer, which serves as a unique directory identifier (DI).
- Names in the GNS have two parts: <directory name, value name>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree

GNS Directory Tree and Value Tree for user Peter.Smith



CASE STUDY: GLOBAL NAME SERVICES

- As an example in the picture, the DIs are illustrated as small integers (although they are actually chosen from a range of integers to ensure uniqueness).
- The attributes of a user Peter.Smith in the directory QMUL would be stored in the value tree named <EC/UK/AC/QMUL, Peter.Smith>.

CASE STUDY: GLOBAL NAME SERVICES

- The value tree includes a password, which can be referenced as <EC/UK/AC/QMUL, Peter.Smith/password>, and several mail addresses, each of which would be listed in the value tree as a single node with the name <EC/UK/AC/QMUL, Peter.Smith/mailboxes>.
- The directory tree is partitioned and stored in many servers, with each partition replicated in several servers.

UNIT-6
CO-ORDINATION

7 Hrs.

OVERVIEW

- Clock Synchronization
- Logical Clocks
- Mutual Exclusion
- Election Algorithm
- Location System
- Distributed Event Matching
- Gossip-based Co-ordination

CLOCK SYNCHRONIZATION

- Multiple processes in distributed system do not simultaneously access a shared resource, such as printer, but instead cooperate in granting each other temporary exclusive access.
- Multiple processes may sometimes need to agree on the ordering of events, such as whether message ml from process P was sent before or after message m2 from process Q.
- As it turns out, synchronization in distributed systems is often much more difficult compared to synchronization in uniprocessor or multiprocessor systems.

CLOCK SYNCHRONIZATION

- In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it.
- Clock synchronization is a method of synchronizing clock values of any two nodes in a distributed system with the use of external reference clock or internal clock value of the node.

PHYSICAL CLOCK

- Each computer contains an electronic device that counts oscillations in a crystal at a definite frequency and store division of count to frequency in a register to provide time. Such device is called physical clock and the time shown is physical time.
- Since, different computers in a distributed system have different crystals that run at different rates, the physical clock gradually get out of synchronization and provide different time values.
- Due to this, it is very difficult to handle and maintain time in critical real time systems. Consistency of distributed data during any modification is based on time factor.

LOGICAL CLOCK

- Logical clock is a mechanism for capturing causal and chronological relationships in a distributed system.
- A physically synchronous global clock may not be present in a distributed system. In such systems a logical clock allows global ordering on events from different processes.
- Logical clock is a virtual clock that records the relative ordering of events in a process.
- A logical clock is a monotonically increasing software counter.
- It is realized whenever relative ordering of events is more important than the physical time.
- The value of logical clock is used to assign time stamps to the events.

SYNCHRONIZATION OF LOGICAL CLOCKS

- Algorithms for synchronization of Logical clocks are:
- Lamport Clock Method
- Vector Clock Method

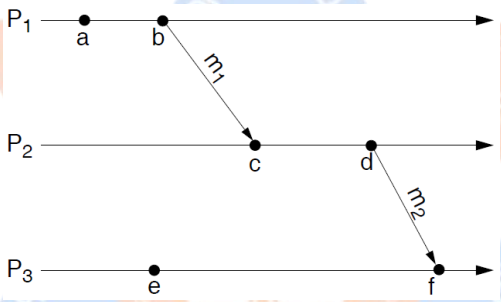
LAMPORT CLOCK METHOD

- The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed computer system.
- As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method.

LAMPORT CLOCK METHOD

- If two events occurred in the same process then they occurred in the order observed following the respective process:
- Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving it.
- Ordering by Lamport is based on the happened-before relation (denoted by →) as:
- a → b, if a and b are events in the same process and a occurred before b.

LAMPORT CLOCK METHOD



- P1, P2, P3 are processes and a, b, c, d, e, f are events.
- a→b, c→d, e→f, b→c, d →f
- a→c, a→d, a→f, b→d, b →f, ...

LAMPORT CLOCK METHOD

- a → b, if a is the event of sending a message m in a process, and b is the event of the same message m being received by another process.
- If a → b and b → c, then a → c (the relation is transitive).
- If a → b, we say that event a causally affects event b. The two events are causally related.
- There are events which are not related by the happened-before relation.
- If both a →e and e →a are false, then a and e are concurrent events; we write a || e.

IMPLEMENTATION OF LAMPORT CLOCK

- There is a logical clock CP_i at each process P_i in the system.
- The value of the logical clock is used to assign timestamps to events.
- $CP_i(a)$ is the timestamp of event a in process Pi.
- There is no relationship between a logical clock and any physical clock.
- To capture the happened-before relation, logical clocks have to be implemented so that if a →b, then C(a) < C(b)

IMPLEMENTATION OF LAMPORT CLOCK

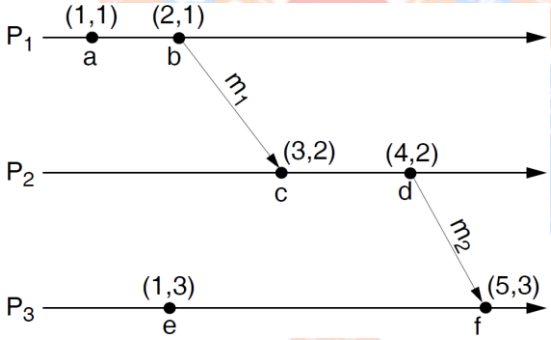
- Implementation of logical clocks is performed using the following rules for updating the clocks and transmitting their values in messages:
- **Rule 1**: CP_i is incremented before each event is issued at Process P_i.
  - $CP_i := CP_i + 1$.
- **Rule 2**:
- When a is the event of sending a message m from process P_i, then the timestamp $t_m = CP_i(a)$ is included in 'm' ($CP_i(a)$ is the logical clock value obtained after applying Rule 1).

IMPLEMENTATION OF LAMPORT CLOCK

- On receiving message 'm' by process P_j, its logical clock CP_j is updated as follows:
  - $CP_j := \max(CP_j, tm)$.
- The new value of CP_j is used to timestamp the event of receiving message m by P_j (applying Rule 1).
- If a and b are events in the same process and a oc-curred before b, then $a \rightarrow b$, and (by Rule 1) $C(a) < C(b)$.
- If a is the event of sending a message m in a process, and b is the event of the same message 'm' being received by another process, then $a \rightarrow b$, and (by Rule 2) $C(a) < C(b)$.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$, and (by induction) $C(a) < C(c)$.

DRAWBACK-1

- Lamport's logical clocks impose only a partial order on the set of events; pairs of distinct events generated by different processes can have identical timestamp.

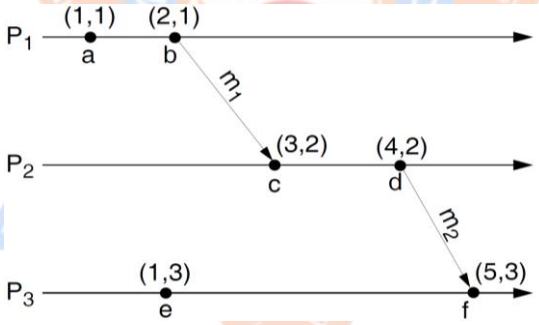


DRAWBACK-1

- For certain applications a total ordering is needed; they consider that no two events can occur at the same time.
- In order to enforce total ordering a global logical timestamp is introduced:
- The global logical timestamp of an event a occurring at process P_i, with logical timestamp $CP_i(a)$, is a pair $(CP_i(a), i)$, where i is an identifier of process P_i.
- We define $(CP_i(a), i) < (CP_j(b), j)$ if and only if $CP_i(a) < CP_j(b)$, or $CP_i(a) = CP_j(b)$ and $i < j$.

DRAWBACK-2

- Lamport's logical clocks are not powerful enough to perform a causal ordering of events.



DRAWBACK-2

- If a → b, then C(a) < C(b). However, the reverse is not always true if the events occurred in different processes.
- If C(a) < C(b), then a→b is not necessarily true. It is only guaranteed that b →a is not true.
- C(e) < C(b), however there is no causal relation from event e to event b.
- By just looking at the timestamps of the events, we cannot say whether two events are causally related or not.

VECTOR CLOCK METHOD

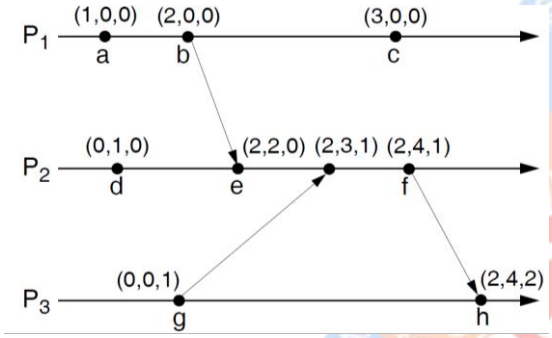
- Vector clocks give the ability to decide whether two events are causally related or not by simply looking at their timestamp.
- Each process P_i has a clock CP_i^v, which is an integer vector of length n where n is the number of processes.
- The value of CP_i^v is used to assign timestamps to events in process P_i.

VECTOR CLOCK METHOD

- $CP_i^v(a)$ is the timestamp of event a in process P_i.
- $CP_i^v[i]$, the i^{th} entry of CP_i^v, corresponds to own logical time of P_i.
- $CP_i^v[j]$, $j \neq i$, is Pi's "best guess" of the logical time at P_j. $CP_i^v[j]$ indicates the logical time of occurrence of the last event at P_j which is in a happened-before relation to the current event at P_i.

IMPLEMENTATION OF VECTOR CLOCK

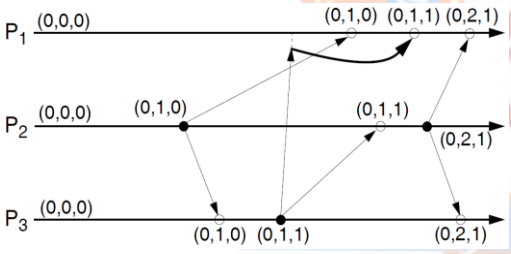
- **Rule 1:** C^v is incremented before each event is issued at P_i
- **Rule 2:**
- When 'a' is the event of sending a message 'm' from process Pi, then the timestamp $t_m = CP_i^v(a)$ is included in 'm'. $CP_i^v(a)$ is the vector clock value obtained after applying Rule 1.
- On receiving message 'm' by process P_j, its vector clock CP_j^v is updated as follows:
- $\forall_k \in \{1,2,..,n\}$, $CP_j^v[k] := \max(CP_j^v[k], t_m[k])$.
- The new value of CP_j^v is used to timestamp the event of receiving message m by P_j (Applying Rule 1).



- For any two vector timestamps u and v, we have:
  u = v if and only if ∀ i, u[i] = v[i]
  u ≤ v if and only if ∀ i, u[i] ≤ v[i]
  u < v if and only if (u ≤ v ∧ u ≠ v)
  u || v if and only if ¬(u < v) ∧ ¬(v < u)

- Two events a and b are causally related if and only if $C^v(a) < C^v(b)$ or $C^v(b) < C^v(a)$. Otherwise the events are concurrent.
- With vector clocks we get the property which we missed for Lamport's logical clocks: $a \rightarrow b$ if and only if $C^v(a) < C^v(b)$.
- Thus, by just looking at the timestamps of the events, we can say whether two events are causally related or not.

CAUSAL ORDERING OF MESSAGES IN VECTOR CLOCK METHOD



- For the problem in sending messages in Logical Clock, we would like messages to be processed according to their causal order which can be solved by Vector Clock as:
- If Send(M_1) → Send(M_2), then every recipient of both messages M_1 and M_2 must receive M_1 before M_2.

CAUSAL ORDERING OF MESSAGES IN VECTOR CLOCK METHOD

- A message delivery protocol which performs causal ordering based on vector clocks as:
- A message is delivered to a process only if the message immediately preceding it (considering the causal ordering) has been already delivered to the process. Otherwise, the message is buffered.
- We assume that processes communicate using broadcast messages. There exist similar protocols for non-broadcast communication too.
- For the events of sending of messages vector clocks will be incremented only for message sending.

IMPLEMENTATION

- **Rule 1:**
- Before broadcasting a message m, a process P_i increments the vector clock: $C^vP_i[i] := C^vP_i[i] + 1$.
- The timestamp $t_m = CP_i^v$ is included in 'm'.
- **Rule 2:**
- The receiving side, at process P_j, delays the delivery of message m coming from Pi until both the following conditions are satisfied:
- $CP_j^v[i] = t_m[i] - 1$
- Here, $t_m[i] - 1$ indicates how many messages originating from P_i precede m.

IMPLEMENTATION

- It ensures that process P_j has received all the messages originating from Pi that precede m.
- $\forall\ k \in \{1,2,..,n\} - \{i\}$, $CP_j^v[k] \geq t_m[k]$
- It ensures that P_j has received all those messages received by P_i before sending m.
- Delayed messages are queued at each process in a queue that is sorted by their vector timestamp. Concurrent messages are ordered by the time of their arrival.
- **Rule 3:**
- When a message is delivered at process P_j, its vector clock CP_j^v is updated according to rule (Rule 2b) for vector clock implementation.

MUTUAL EXCLUSION

MUTUAL EXCLUSION IN DISTRIBUTED SYSTEM

- Mutual exclusion is a mechanism that prevent interference and ensure consistency when accessing the resources by a collection of processes.
- Mutual exclusion ensures that concurrent processes make a serialized access to shared resources or data.
- Mutual exclusion is a concurrency control property which is introduced to prevent race conditions.
- It is the requirement that a process cannot enter its critical section while another concurrent process is currently present or executing in its critical section i.e. only one process is allowed to execute the critical section at any given instance of time.

MUTUAL EXCLUSION IN DISTRIBUTED SYSTEM

- In single computer system, memory and other resources are shared between different processes.
- The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.
- In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables.
- To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

REQUIREMENT FOR MUTUAL EXCLUSION

- **Safety:** At most one process may execute in the critical section (CS) at a time.
- **Liveness:** A process requesting entry to the CS is eventually granted it so long as any process executing the CS eventually leaves it.
- Liveness implies freedom of deadlock and starvation.

ALGORITHMS FOR MUTUAL EXCLUSION

- **Non-token-based Mutual Exclusion:**
  - Each process freely and equally competes for the right to use the shared resource; requests are arbitrated by a central control site or by distributed agreement.
- **Token-based Mutual Exclusion:**
  - A logical token representing the access right to the shared resource is passed in a regulated fashion among the processes; whoever holds the token is allowed to enter the critical section.

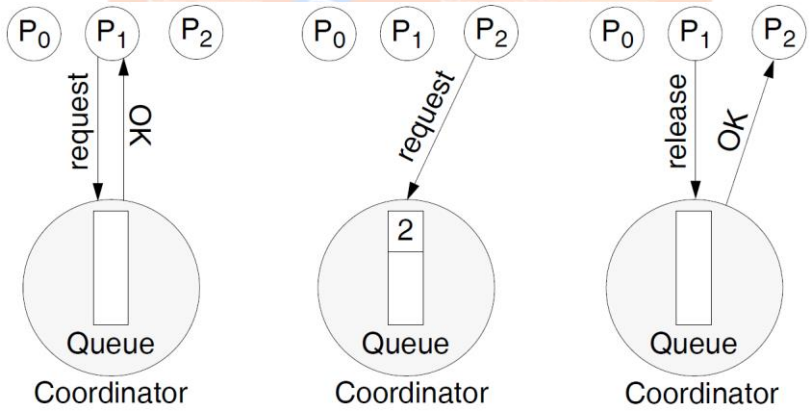
NON-TOKEN-BASED MUTUAL EXCLUSION

- Central Coordinator Algorithm
- Ricart-Agrawala Algorithm

CENTRAL COORDINATOR ALGORITHM

- A central coordinator grants permission to enter a CS.
- To enter a CS, a process sends a request message to the coordinator and then waits for a reply (during this waiting period the process can continue with other work).
- The reply from the coordinator gives the right to enter the CS.

CENTRAL COORDINATOR ALGORITHM



CENTRAL COORDINATOR ALGORITHM

- After finishing work in the CS the process notifies the coordinator with a release message.
- The scheme is simple and easy to implement.
- The strategy requires only three messages per use of a CS (request, OK, release).

DRAWBACKS

- The coordinator can become a performance bottleneck.
- The coordinator is a critical point of failure:
- If the coordinator crashes, a new coordinator must be created.
- The coordinator can be one of the processes competing for access; an election algorithm (see later) has to be run in order to choose one and only one new coordinator.

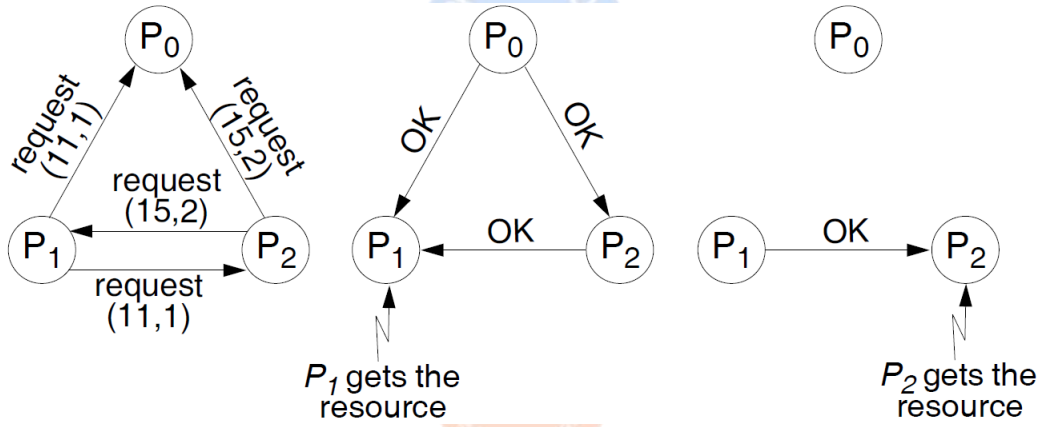
RICART-AGRAWALA ALGORITHM

- The process that requires entry to a CS multicasts the request message to all other processes competing for the same resource.
- It is allowed to enter the CS when all processes have replied to this message. The request message consists of the requesting process' timestamp (logical clock) and its identifier.
- Each process keeps its state with respect to the CS: *Released*, *Requested*, or *Held*.

ALGORITHM

- #Rule for process initialization
  - /* performed by each process Pi at initialization */
- [RI1]: statePi := RELEASED.
- #Rule for access request to CS
  - /* performed whenever process Pi requests an access to the CS */
- [RA1]: statePi := REQUESTED.
  - TPi := the value of the local logical clock corresponding to this request.
- [RA2]: Pi sends a request message to all processes; the message is of the form (TPi, i), where i is an identifier of Pi.
- [RA3]: Pi waits until it has received replies from all other n-1 processes.

RICART-AGRAWALA ALGORITHM



ALGORITHM

- #Rule for executing the CS
  - /* performed by Pi after it received the n-1 replies */
- [RE1]: statePi := HELD.
  - Pi enters the CS.
- #Rule for handling incoming requests
  - /* performed by Pi whenever it received a request (TPj, j) from Pj */
- [RH1]: if statePi = HELD or ((statePi = REQUESTED) and ((TPi, i) < (TPj, j))) then
  - Queue the request from Pj without replying.
- else
  - Reply immediately to Pj.
- end if.

ALGORITHM

- Rule for releasing a CS
  - /* performed by Pi after it finished work in a CS */
- [RR1]: statePi := RELEASED.
  - Pi replies to all queued requests.

- A request issued by a process Pj is blocked by another process Pi only if
- Pi is holding the resource or
- if it is requesting the resource with a higher priority (this means a smaller timestamp) then Pj.

DRAWBACKS

- The algorithm is expensive in terms of message traffic; it requires 2(n-1) messages for entering a CS: (n-1) requests and (n-1) replies.
- The failure of any process involved makes progress impossible if no special recovery measures are taken

TOKEN-BASED MUTUAL EXCLUSION

- Ricart-Agrawala Second Algorithm
- Token Ring Algorithm

RICART-AGRAWALA SECOND ALGORITHM

- A process is allowed to enter the critical section when it got the token.
- In order to get the token it sends a request to all other processes competing for the same resource.
- The request message consists of the requesting process' timestamp (logical clock) and its identifier.
- Initially the token is assigned arbitrarily to one of the processes.

RICART-AGRAWALA SECOND ALGORITHM

- When a process Pi leaves a critical section it passes the token to one of the processes which are waiting for it; this will be the first process Pj, where j is searched in order [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] for which there is a pending request.
- If no process is waiting, Pi retains the token (and is allowed to enter the CS if it needs); it will pass over the token as result of an incoming request.
- Each process Pi records the timestamp corresponding to the last request it got from process Pj, in requestPi[j].
- In the token itself, token[j] records the timestamp (logical clock) of Pj's last holding of the token. If requestPi[j] > token[j] then Pj has a pending request.

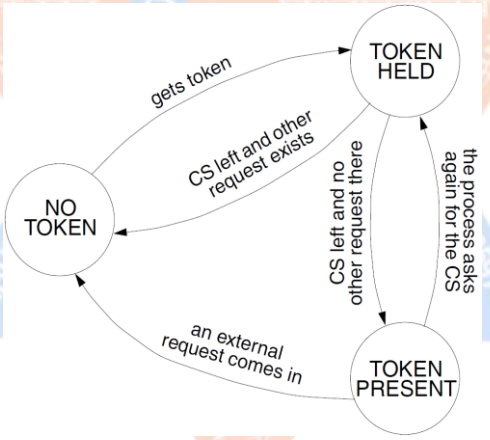
RICART-AGRAWALA SECOND ALGORITHM

- Each process keeps its state with respect to the token: NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.
- The complexity is reduced compared to the (first) Ricart-Agrawala algorithm: it requires n messages for entering a CS: (n-1) requests and one reply.
- The failure of a process, except the one which holds the token, doesn't prevent progress.

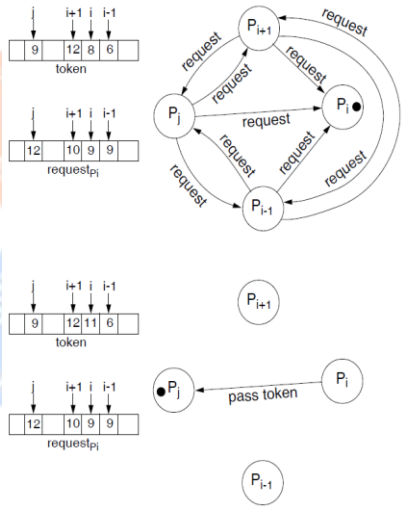
ALGORITHM

- #Rule for process initialization
  - /* performed at initialization */
- [RI1]: statePi := NO-TOKEN for all processes Pi, except one single process Px for which
  - statePx := TOKEN-PRESENT.
- [RI2]: token[k] initialized 0 for all elements k = 1... n. requestPi[k] initialized 0 for all processes Pi and all elements k=1.. n.
- #Rule for access request and execution of the CS
  - /* performed whenever process Pi requests an access to the CS and when it finally gets it; in particular Pi can already possess the token */
- [RA1]: if statePi = NO-TOKEN then
  - Pi sends a request message to all processes; the message is of the form (TPi, i), where
  - TPi = CPi is the value of the local logical clock, and i is an identifier of Pi.
  - Pi waits until it receives the token.
  - end if.
- statePi := TOKEN-HELD.
  - Pi enters the CS.

RICART-AGRAWALA SECOND ALGORITHM



RICART-AGRAWALA SECOND ALGORITHM



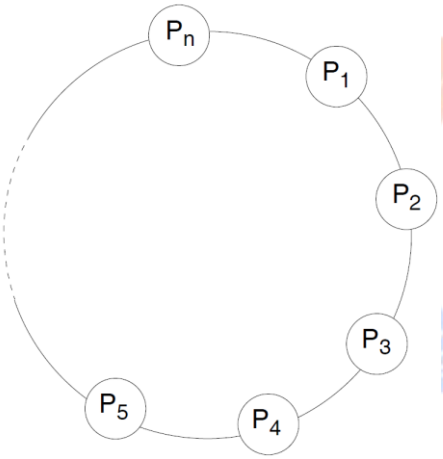
ALGORITHM

- Rule for handling incoming requests
  - /* performed by Pi whenever it received a request (TPj, j) from Pj */
- [RH1]: requestPi[j] := max(requestPi[j], TPj).
- [RH2]: if statePi = TOKEN-PRESENT then
  - Pi releases the resource (see rule RR2).
  - end if.
- Rule for releasing a CS
  - /* performed by Pi after it finished work in a CS or when it holds a token without using it and it got a request */
- [RR1]: statePi = TOKEN-PRESENT.
- [RR2]: for k = [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] do if requestPi[k] > token[k] then
  - statePi := NO-TOKEN.
  - token[i] := CPi, the value of the local logical clock.
  - Pi sends the token to Pk.
  - break.
  - /* leave the for loop */
  - end if.
- end for.

TOKEN RING ALGORITHM

- A very simple way to solve mutual exclusion where arrangement the n processes P1, P2, ... Pn in a logical ring is done.
- The logical ring topology is created by giving each process the address of one other process which is its neighbor in the clockwise direction.

TOKEN RING ALGORITHM



ALGORITHM

- The token is initially given to one process.
- The token is passed from one process to its neighbor round the ring.
- When a process requires to enter the CS, it waits until it receives the token from its left neighbor and then it retains it; after it got the token it enters the CS; after it left the CS it passes the token to its neighbor in clockwise direction.
- When a process receives the token but does not require to enter the critical section, it immediately passes the token over along the ring.

TOKEN RING ALGORITHM

- It can take from 1 to n-1 messages to obtain a token.
- Messages are sent around the ring even when no process requires the token i.e. additional load on the network.
- The algorithm works well in heavily loaded situations, when there is a high probability that the process which gets the token wants to enter the CS. It works poorly in lightly loaded cases.
- If a process fails, no progress can be made until a reconfiguration is applied to extract the process from the ring.
- If the process holding the token fails, a unique process has to be picked, which will regenerate the token and pass it along the ring; an election algorithm (see later) has to be run for this purpose.

DISTRIBUTED ELECTION

DISTRIBUTED ELECTION

- Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role. Examples with mutual exclusion:
- Central coordinator algorithm: at initialization or whenever the coordinator crashes, a new coordinator has to be elected.
- Token ring algorithm: when the process holding the token fails, a new process has to be elected which generates the new token
- We consider that it doesn't matter which process is elected; what is important is that one and only one process is chosen (we call this process the coordinator) and all processes agree on this decision.

DISTRIBUTED ELECTION

- We assume that each process has a unique number (identifier); in general, election algorithms attempt to locate the process with the highest number, among those which currently are up.
- Election is typically started after a failure occurs. The detection of a failure (e.g. the crash of the current coordinator is normally based on time-out i.e. a process that gets no response for a period of time suspects a failure and initiates an election process).
- An election process is typically performed in two phases:
  - Select a leader with the highest priority.
  - Inform all processes about the winner.

DISTRIBUTED ELECTION

- Bully Algorithm
- Ring Based Algorithm

BULLY ALGORITHM

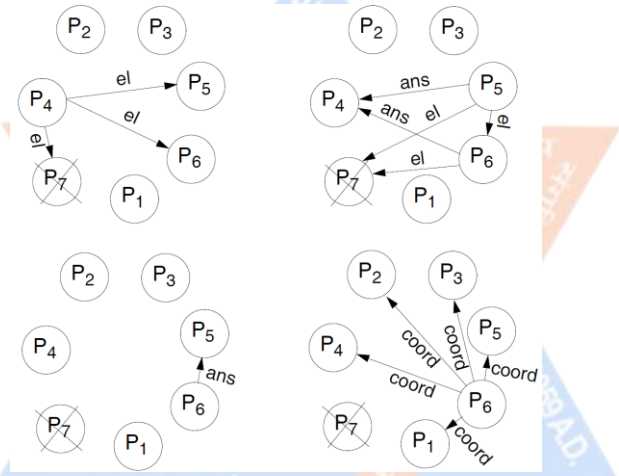
- A process has to know the identifier of all other processes; the process with the highest identifier, among those which are up, is selected.
- Any process could fail during the election procedure.
- When a process Pi detects a failure and a coordinator has to be elected, it sends an election message to all the processes with a higher identifier and then waits for an answer message:
- If no response arrives within a time limit, Pi becomes the coordinator (all processes with higher identifier are down) i.e. it broadcasts a coordinator message to all processes to let them know.

BULLY ALGORITHM

- If an answer message arrives, Pi knows that another process has to become the coordinator i.e. it waits in order to receive the coordinator message. If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the answer message) Pi resends the election message.
- When receiving an election message from Pi, a process Pj replies with an answer message to Pi and then starts an election procedure itself, unless it has already started one i.e. it sends an election message to all processes with higher identifier.
- Finally, all processes get an answer message, except the one which becomes the coordinator.

BULLY ALGORITHM

- By default, the state of a process is ELECTION-OFF
- Rule for election process initiator
  - /* performed by a process Pi, which triggers the election procedure, or which starts an election after receiving itself an election message */
- [RE1]: statePi := ELECTION-ON.
  - Pi sends an election message to all processes with a higher identifier.
  - Pi waits for answer message.
  - if no answer message arrives before time-out then
  - Pi is the coordinator and sends a coordinator message to all processes.
  - else
  - Pi waits for a coordinator message to arrive.
  - if no coordinator message arrives before time-out then
    - restart election procedure according to RE1
    - end if
  - end if.
- Rule for handling an incoming election message
  - /* performed by a process Pi at reception of an election message coming from Pj */
- [RH1]: Pi replies with an answer message to Pj.
- [RH2]: if statePi := ELECTION-OFF then
  - Start election procedure according to RE1 end if



- If P6 crashes before sending the coordinator message, P4 and P5 restart the election process.

BULLY ALGORITHM CASES

- **Best case:**
  - The process with the second highest identifier notices the coordinator's failure.
  - It can immediately select itself and then send n-2 coordinator messages.
- **Worst case:**
  - The process with the lowest identifier initiates the election
  - It sends n-1 election messages to processes which themselves initiate each one an election i.e. $O(n^2)$ messages.

RING-BASED ALGORITHM

- We assume that the processes are arranged in a logical ring; each process knows the address of one other process, which is its neighbor in the clockwise direction.
- The algorithm elects a single coordinator, which is the process with the highest identifier.
- Election is started by a process which has noticed that the current coordinator has failed. The process places its identifier in an election message that is passed to the following process.

RING-BASED ALGORITHM

- When a process receives an election message it compares the identifier in the message with its own.
- If the arrived identifier is greater, it forwards the received election message to its neighbor; if the arrived identifier is smaller it substitutes its own identifier in the election message before forwarding it.
- If the received identifier is that of the receiver itself i.e. this will be the coordinator. The new coordinator sends an elected message through the ring.

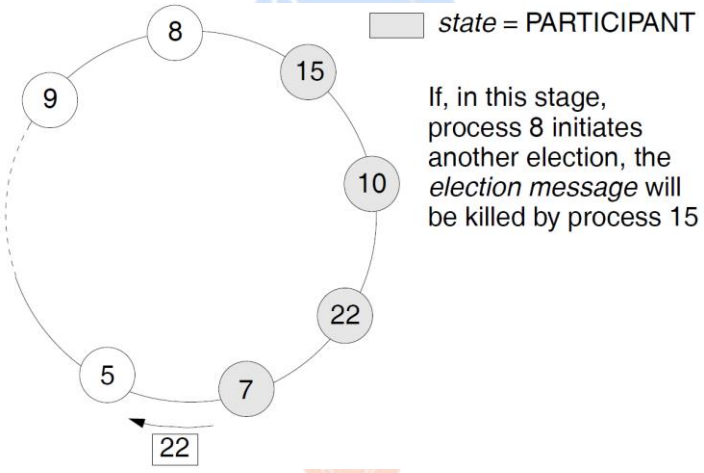
RING-BASED ALGORITHM (OPTIMIZATION)

- Several elections can be active at the same time. Messages generated by later elections should be killed as soon as possible.
- Processes can be in one of two states: participant and non-participant. Initially, a process is non-participant.
- The process initiating an election marks itself participant.

RING-BASED ALGORITHM (OPTIMIZATION)

- A participant process, in the case that the identifier in the election message is smaller than the own, does not forward any message (it has already forwarded it, or a larger one, as part of another simultaneously ongoing election).
- When forwarding an election message, a process marks itself participant.
- When sending (forwarding) an elected message, a process marks itself non-participant.

RING-BASED ALGORITHM



RING-BASED ALGORITHM

- By default, the state of a process is NON-PARTICIPANT
- #Rule for election process initiator
- /* performed by a process Pi, which triggers the election procedure */
- [RE1]: statePi := PARTICIPANT.
- [RE2]: Pi sends an election message with message id := i to its neighbor.
- #Rule for handling an incoming election message
- /* performed by a process Pj, which receives an election message */
- [RH1]: if message.id > j then
- Pj forwards the received election message.
- statePj := PARTICIPANT.

RING-BASED ALGORITHM

- else if message.id < j then
- if statePj = NON-PARTICIPANT then
- Pj forwards an election message with
- message.id := j.
- statePj := PARTICIPANT
- end if
- else
- Pj is the coordinator and sends an elected message with message.id := j to its neighbor.
- statePj := NON-PARTICIPANT.
- end if.
- #Rule for handling an incoming elected message
  - /* performed by a process Pi, which receives an elected message */
- [RD1]: if message.id ≠ i then
- Pi forwards the received elected message.
- statePj := NON-PARTICIPANT.
- end if.

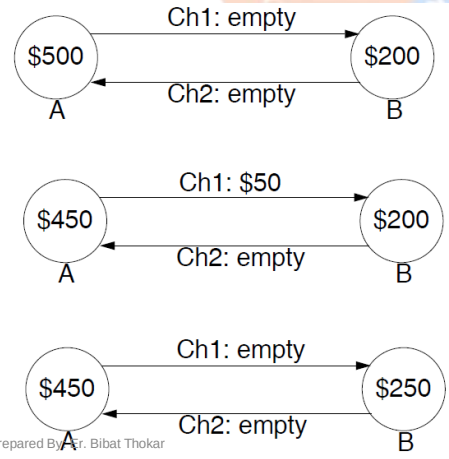
With one single election started:

- **On average:**
  - n/2 (election) messages needed to reach maximal node; n (election) messages to return to maximal node; n messages to rotate elected message.
  - Number of messages: 2n + n/2.
- **Worst case:**
  - n-1 messages needed to reach maximal node;
  - Number of messages: 3n - 1.
- The ring algorithm is more efficient on average than the bully algorithm.

GLOBAL STATE AND STATE RECORDING

- Global State
- In general, a global state consists of a set of local states and a set of states of the communication channels.
- The state of the communication channel in a consistent global state should be the sequence of messages sent along the channel before the sender's state was recorded, excluding the sequence of messages received along the channel before the receiver's state was recorded.
- It is difficult to record channel states to ensure these rule. Global states are very often recorded without using channel states. A certain global state can be consistent or not.

Consider a bank system with two accounts a and b at tow different sites, we transfer $50 between A and B global state



A	Ch1	B	C : consistent NC: not consistent
500	empty	200	C
500	50	200	NC
450	50	200	C
450	empty	200	NC
500	50	250	NC
450	50	250	NC
450	empty	250	C
500	empty	250	NC

GLOBAL STATE (FORMAL DEFINITION)

- LSi is the local state of process Pi. Beside other information, the local state also includes a record of all messages sent and received by the process.
- We consider the global state GS of a system, as the collection of the local states of its processes: GS = {LS1, LS2, ..., LSn}.
- send(mkij) denotes the event of sending message mkij from Pi to Pj;
- rec(mkij) denotes the event of receiving message mkij by Pj.

GLOBAL STATE AND STATE RECORDING

- send(mkij) ∈ LSi if and only if the sending event occurred before the local state was recorded;
- rec(mkij) ∈ LSj if and only if the receiving event occurred before the local state was recorded.
- transit(LSi,LSj) = {mkij | send(mkij) ∈ LSi ∧ rec(mkij) ∉ LSj}
  inconsistent(LSi, LSj) = {mkij | send(mkij) ∉ LSi ∧ rec(mkij) ∈ LSj}

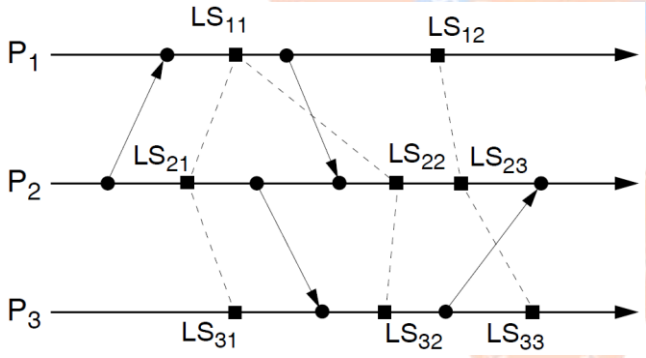
GLOBAL STATE AND STATE RECORDING

- A global state GS = {LS_1, LS_2, ..., LS_n} is consistent if and only if: ∀ i, ∀ j: 1 ≤ i, j ≤ n :: inconsistent(LS_i,LS_j) = ∅
  - In a consistent global state for every received message a corresponding send event is recorded in the global state.
  - In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded.

GLOBAL STATE AND STATE RECORDING

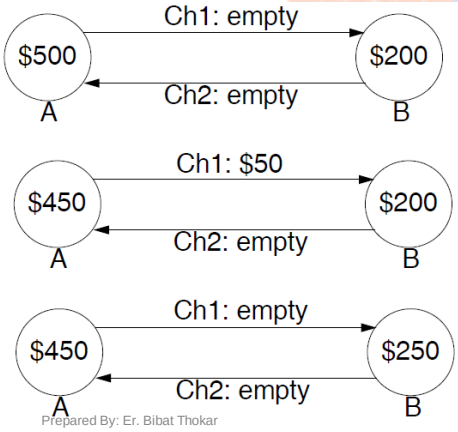
- A global state GS = {LS_1, LS_2, ..., LS_n} is transitless if and only if: ∀ i, ∀ j: 1 ≤ i, j ≤ n :: transit(LS_i, LS_j) = ∅
  - All messages recorded to be sent are also recorded to be received.
- A global state is strongly consistent if it is consistent and transitless.
  - A strongly consistent state corresponds to a consistent state in which all messages recorded as sent are also recorded as received.

Note: The global state, as defined here, is seen as a collection of the local states, without explicitly capturing the state of the channel.



- {LS_{11}, LS_{22}, LS_{32}} is "inconsistent"
- {LS12, LS23, LS_{33}} is "consistent"
- {LS_{11}, LS_{21}, LS_{31}} is "strongly consistent"

After registering of the receive events a consistent state becomes strongly consistent. It is considered to be a normal (transient) situation



A	B	C : consistent NC: not consistent
500	200	{A,B}: strongly C
450 ($mess_1$ sent)	200	{A,B}: C
500	250 ($mess_1$ received)	{A,B}: NC
450 ($mess_1$ sent)	250 ($mess_1$ received)	{A,B}: strongly C

GLOBAL STATE RECORDING
CHANDY-LAMPORT OR SNAPSHOT ALGORITHM

- The algorithm records a collection of local states which give a consistent global state of the system. In addition, it records the state of the channels which is consistent with the collected global state. Such a recorded "view" of the system is called a snapshot.
- We assume that processes are connected through one directional channels and message delivery is FIFO.
- We assume that the graph of processes and channels is strongly connected (there exists a path between any two processes).
- The algorithm is based on the use of a special message, snapshot token, in order to control the state collection process.

GLOBAL STATE RECORDING
CHANDY-LAMPORT OR SNAPSHOT ALGORITHM

- A process P_i records its local state LS_i and later sends a message m to P_j; LS_j has to be recorded at P_j before P_j has received 'm'.
- The state SCh_{ij} of the channel Ch_{ij} consists of all messages that process P_i sent before recording LS_i and which have not been received by P_j when recording LS_j.
- A snapshot is started at the request of a particular process Pi, for example, when it suspects a deadlock because of long delay in accessing a resource; P_i then records its state LS_i and, before sending any other message, it sends a token to every P_j that P_i communicates with.
- When P_j receives a token from P_i, and this is the first time it received a token, it must record its state before it receives the next message from P_i. After recording its state P_j sends a token to every process it communicates with, before sending them any other message.

ALGORITHM

- Rule for sender P_i:
  - /* performed by the initiating process and by any other process at the reception of the first token */
- [SR_1]: P_i records its state.
- [SR_2]: P_i sends a token on each of its outgoing channels.
- Rule for receiver P_j:
  - /* executed whenever Pj receives a token from another process P_i on channel Ch_{ij} */
- [RR1]: if P_j has not yet recorded its state then
- Record the state of the channel: $SCh_{ij} := \emptyset$.
- Follow the "Rule for sender".
- else
  - Record the state of the channel: SCh_{ij} := M, where M is the set of messages that P_j received from P_i after P_j recorded its state and before P_j received the token on Ch_{ij}.
- end if.