# PRATHYUSHA ENGINEERING COLLEGE

# LECTURE NOTES

| | |
|---|---|
| **Course code** | **:CS8603** |
| **Name of the course** | **: DISTRIBUTED SYSTEM** |
| **Regulation** | **: 2017** |
| **Course faculty** | **: Ms.AnithaLakshmi.V** |

# CS8603 – DISTRIBUTED SYSTEMS UNIT 1

Introduction: Definition –Relation to computer system components –Motivation –Relation to parallel systems – Message-passing systems versus shared memory systems –Primitives for distributed communication –Synchronous versus asynchronous executions –Design issues and challenges. A model of distributed computations: A distributed program –A model of distributed executions –Models of communication networks –Global state – Cuts –Past and future cones of an event –Models of process communications. Logical Time: A framework for a system of logical clocks –Scalar time –Vector time – Physical clock synchronization: NTP.

## 1. Introduction

### 1.1 Definition – Distributed Systems

- A **distributed system** is a **system** whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.
- A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.
- Autonomous processors communicating over a communication network

**Characteristics of Distributed Systems**

1. **No common physical clock** -> "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.

2. **No shared memory** -> distributed system may still provide the abstraction of a common address space via the distributed shared memory abstraction.

3. **Geographical separation** -> The geographically wider apart that the processors are, the more representative is the system of a distributed system network/cluster of workstations (NOW/COW) configuration connecting processors. The Google search engine is based on the NOW architecture.

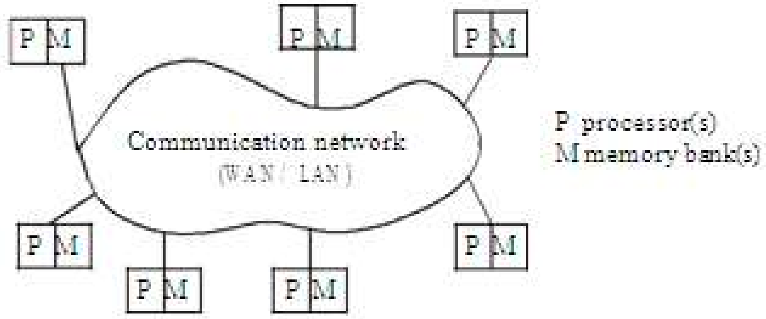
4. **Autonomy and heterogeneity** -> The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system.

### 1.2 Relation to computer system components

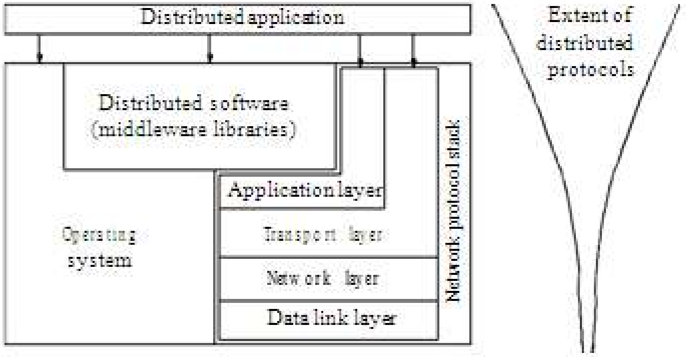
Each computer has a memory-processing unit and the computers are connected by a communication network. Figure shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning.

The distributed software is also termed as *middleware*. A *distributed execution* is the execution of processes across the distributed system to collaboratively achieve a common goal. An execution is also sometimes termed a *computation* or a *run*.

A distributed system connects processors by a communication network.



**Interaction of the software components at each process**



- The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level.
- There are several standards such as Object Management Group's (OMG) common object request broker architecture (CORBA) [36], and the remote procedure call (RPC) mechanism

## 1.3 Motivation

The motivation for using a distributed system is some or all of the following requirements:

1. **Inherently distributed computations**

The computation is inherently distributed

Eg., money transfer in banking

2. **Resource sharing**

**Resources** such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites. Further, they cannot be placed at a single site. Therefore, such resources are typically distributed across the system.

For example, distributed databases such as DB2 partition the data sets across several servers

### 3. Access to geographically remote data and resources

In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated.
For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site.

### 4. Enhanced reliability

A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances. Reliability entails several aspects:

a. **availability**, i.e., the resource should be accessible at all times;
b. **integrity**, i.e., the value/state of the resource should be correct
c. **fault-tolerance**, i.e., the ability to recover from system failures

### 5. Increased performance/cost ratio

By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased.

### 6. Scalability

As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network.

### 7. Modularity and incremental expandability

Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms. Similarly, existing processors may be easily replaced by other processors.

## 1.4 Relation to parallel multiprocessor/multicomputer systems

A parallel system may be broadly classified as belonging to one of three types:

1. Multiprocessor system
2. Multicomputer parallel system
3. Array processors

### ***1.4.1 Characteristics of parallel systems***

1. A *multiprocessor system* is a parallel system in which the multiple processors have *direct access to shared memory* which forms a common address space.

The architecture is shown in Figure (a). Such processors usually do not have a common clock.

A multiprocessor system *usually* corresponds to a uniform memory access (UMA) architecture in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same. The processors are in very close physical proximity and are connected by an interconnection network. Inter process communication across processors is traditionally through read and write operations on the shared memory, although the use of message-passing primitives such as those provided by

Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

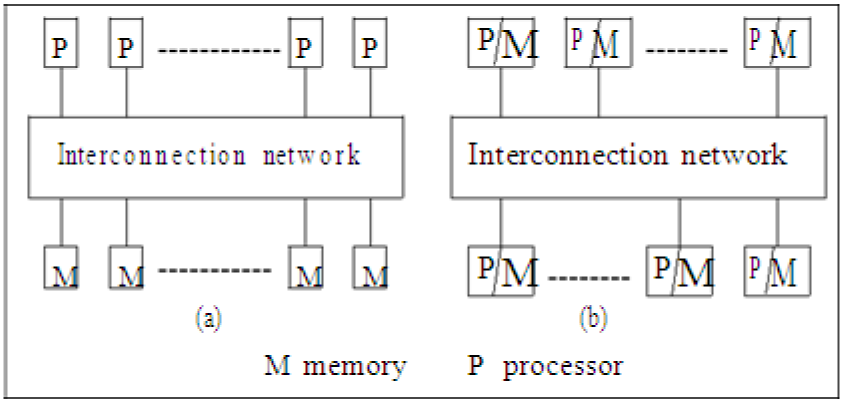
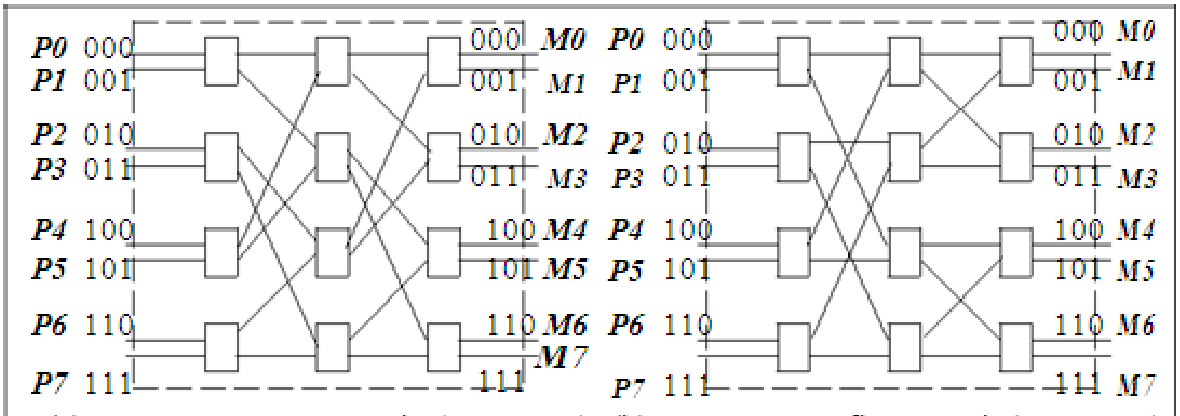


***Figure : Interconnection networks for shared memory multiprocessor systems. (a) Omega network [4] for n = 8 processors P0–P7 and memory banks M0–M7. (b) Butterfly network [10] for n = 8 processors P0–P7 and memory banks M0–M7.***

***Figure shows two popular interconnection networks – the Omega network and the Butterfly network, each of which is a multi-stage network formed of 2 ×2 switching elements. Each 2 ×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire.***

- Each 2 × 2 switch is represented as a rectangle in the figure. Further-more, a n-input and n-output network uses log n stages and log n bits for addressing.
- Omega interconnection function The Omega network which connects n processors to n memory units has $n/2\log_2 n$ switching elements of size 2 × 2 arranged in $\log_2 n$ stages.

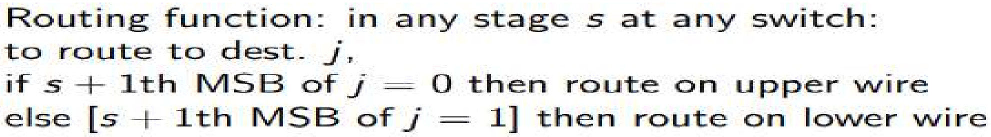


(a) 3−stage Omega network $(N=8, M=4)$ (b) 3−stage Butterfly network $(N=8, M=4)$

***Interconnection function:*** Output i of a stage connected to input j of next stage:

$$j = \begin{cases} 2i & \text{for } 0 \leq i \leq n/2 - 1 \\ 2i + 1 - n & \text{for } n/2 \leq i \leq n - 1 \end{cases}$$

- Consider any stage of switches. Informally, the upper (lower) input lines for each switch come in sequential order from the upper (lower) half of the switches in the earlier stage.
- With respect to the Omega network in Figure(a), n = 8. Hence, for any stage, for the outputs i, where 0 ≤ i ≤ 3, the output i is connected to input 2i of the next stage. For 4 ≤ i ≤ 7, the output i of any stage is connected to input 2i + 1 − n of the next stage.



Routing function: in any stage $s$ at any switch:
to route to dest. $j$,
if $s + 1$th MSB of $j = 0$ then route on upper wire
else $[s + 1$th MSB of $j = 1]$ then route on lower wire

***Omega routing function***

- The routing function from input line i to output line j considers only j and the stage number s, where s ∈ 0 $\log_2 n - 1$. In a stage s switch, if the s + 1th MSB (most significant bit) of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.
- The Butterfly and the Omega networks, the paths from the different inputs to any one output form a spanning tree. This implies that collisions will occur when data is destined to the same output line. However, the advantage is that data can be combined at the switches if the application semantics (e.g., summation of numbers) are known.

## 2. Multicomputer parallel system

A *multicomputer parallel system* is a parallel system in which the multiple processors *do not have direct access to shared memory.* The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.

Non-uniform memory access (NUMA) architecture

**Examples of parallel multicomputers are**: the NYU Ultracomputer and the Sequent shared memory machines, the CM* Connection machine and processors configured in regular and symmetrical topologies such as an array or mesh, ring, torus, cube, and hypercube (message-passing machines).

**(a) Wrap-around 2D-mesh, also known as torus. (b) Hypercube of dimension 4.**

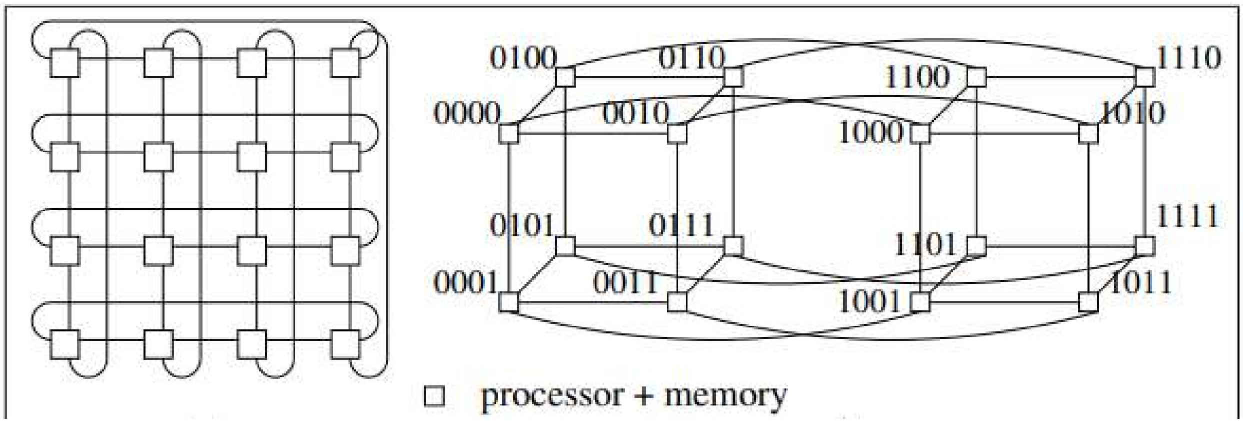


***Figure (a) shows a wrap-around 4 × 4 mesh. For a k × k mesh which will contain $k^2$ processors, the maximum path length between any two processors is 2 k/2 − 1 . Routing can be done along the Manhattan grid.***

***Figure (b) shows a four-dimensional hypercube. A k-dimensional hyper-cube has $2^k$ processor-and-memory units. Each such unit is a node in the hypercube, and has a unique k-bit label.***

***<u>Hamming distance</u>***

- The processors are labelled such that the shortest path between any two processors is the *Hamming distance* (defined as the number of bit positions in which the two equal sized bit strings differ) between the processor labels.
- Example Nodes 0101 and 1100 have a Hamming distance of 2. The shortest path between them has length 2.

3. **<u>Array processors</u>**

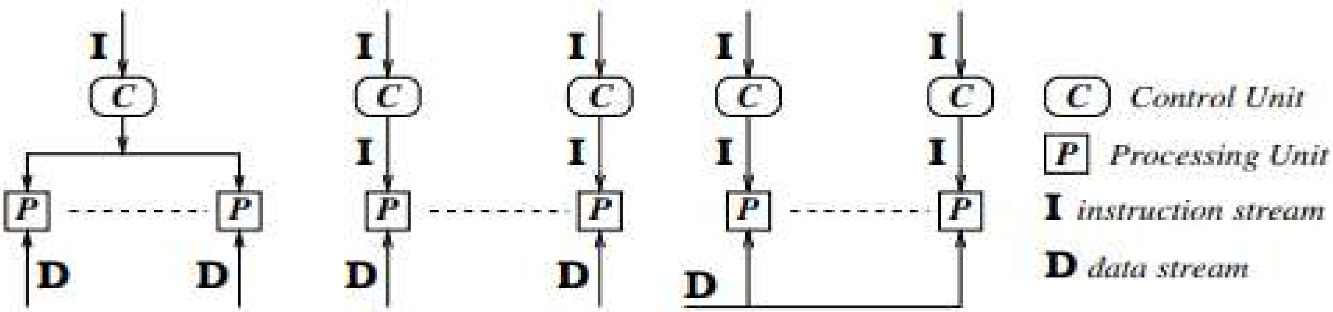
• **<u>Array processors</u>** belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock (but may not share memory and communicate by passing data using messages).

• Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing belong to this category.

• These applications usually involve a large number of iterations on the data. This class of parallel systems has a very niche market.

### *1.4.2 Flynn's Taxonomy*

Flynn identified four processing modes, based on whether the processors execute the same or different instruction streams at the same time, and whether or not the processors processed the same (identical) data at the same time.



(a) SIMD (b) MIMD (c) MISD

**SISD: Single Instruction Stream Single Data Stream (traditional)**
This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

**SIMD: Single Instruction Stream Multiple Data Stream**
This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items.

- scientific applications, applications on large arrays
- vector processors, systolic arrays, Pentium/SSE, DSP chips

**MISD: Multiple Instruction Stream Single Data Stream**
This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications

- E.g., visualization

**MIMD: Multiple Instruction Stream Multiple Data Stream**

➢ In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems.

➢ There is no common clock among the system processors.

Eg. Sun Ultra servers, multicomputer PCs, and IBM SP machines

### *1.4.3 Coupling, parallelism, concurrency, and granularity*

- **Coupling**

➢ The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules.

➢ When the degree of coupling is high (low), the modules are said to be tightly (loosely) coupled.

➢ SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.

➢ Various MIMD architectures in terms of coupling:

- Tightly coupled multiprocessors (with UMA shared memory). These may be either switch-based
- Tightly coupled multiprocessors (with NUMA shared memory or that communicate by message passing).
- Loosely coupled multi computers (without shared memory) physically co-located. These may be bus-based
- and the processors may be heterogeneous
- Loosely coupled multi computers (without shared memory and without common clock) that are physically remote.

**Parallelism or speedup of a program on a specific system**

➢ This is a measure of the relative speedup of a specific program, on a given machine.

➢ The speedup depends on the number of processors and the mapping of the code to the processors.

➢ It is expressed as the ratio of the time T(1) with a single processor, to the time T(n) with n processors.

➢ Parallelism within a parallel/distributed program

➢ This is an aggregate measure of the percentage of time that all the proces-sors are executing CPU instructions productively, as opposed to waiting for communication (either via shared memory or message-passing) operations to complete.

**Concurrency of a program**

The *parallelism/concurrency* in a parallel/distributed program can be measured by the ratio of the number of local (non-communication and non-shared memory access) operations to the total number of operations, including the communication or shared memory access operations.

**Granularity of a program**

➢ The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as *granularity*.

➢ Programs with fine-grained parallelism are best suited for tightly coupled systems. Eg. SIMD and MISD architectures

## 1.5 Message-passing vs. Shared Memory

➢ Shared memory systems are those in which there is a (common) shared address space throughout the system.

➢ Communication among processors takes place via shared data variables, and control variables for synchronization among the processors.

➢ Semaphores and monitors that were originally designed for shared memory uni processors and multiprocessors

- The abstraction called *shared memory* is sometimes provided to simulate a shared address space. For a distributed system, this abstraction is called *distributed shared memory*. Implementing this abstraction has a certain cost but it simplifies the task of the application programmer.
- The communication via message-passing can be simulated by communication via shared memory and vice-versa. Therefore, the two paradigms are equivalent.

### *1.5.1. Emulating message-passing on a shared memory system (MP → SM)*

- Partition shared address space
- Send/Receive emulated by writing/reading from special mailbox per pair of processes
- A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox.
- The write and read operations need to be controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received.

### *1.5.2. Emulating shared memory on a message-passing system (SM → MP)*

- This involves the use of “send” and “receive” operations for “write” and “read” operations.
- Model each shared object as a process
- Write to shared object emulated by sending message to owner process for the object
- Read from shared object emulated by sending query to owner of shared object
- In a MIMD message-passing multicomputer system, each “processor” may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory. Between two computers, the communication is by message passing.

## 1.6 Primitives for distributed communication

### *1.6.1. Blocking/non-blocking, synchronous/asynchronous primitives*

- A Send primitive has at least two parameters – the destination, and the buffer in the user space, containing the data to be sent.
- Similarly, a Receive primitive has at least two parameters – the source from which the data is to be received (this could be a wildcard), and the user buffer into which the data is to be received.
- There are two ways of sending data when the Send primitive is invoked – the buffered option and the unbuffered option. The buffered option which is the standard option copies the data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network. In the unbuffered option, the data gets copied directly from the user buffer onto the network.
- For the Receive primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.

**Synchronous primitive(send/receive)**

- Handshake between sender and receiver
- Send completes when Receive completes
- Receive completes when data copied into buffer

**Asynchronous primitive (send)**

- A Send primitive is said to be asynchronous if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.

**Blocking primitive (send/receive)**

- A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.

**Nonblocking primitive (send/receive)**

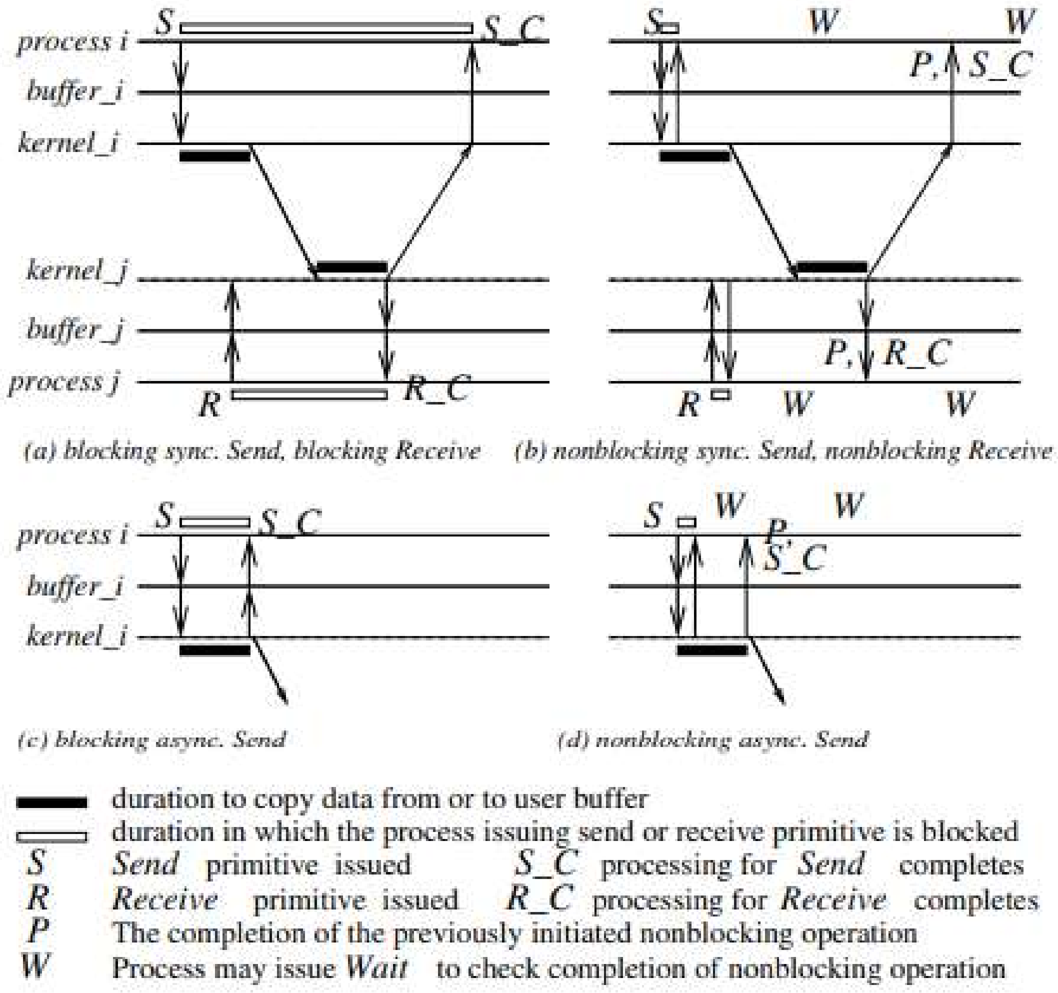
- A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.
- Send: even before data copied out of user buffer
- Receive: even before data may have arrived from sender

A non-blocking send primitive. When the Wait call returns, at least one of its parameters is posted.

| | |
|---|---|
| ***Send(X, destination, handle$_k$)*** | ***// handle$_k$ is a return parameter*** |
| | |
| ***Wait(handle$_1$, handle$_2$, …, handle$_k$, …, handle$_m$)*** | ***// Wait always blocks*** |

- Return parameter returns a system-generated handle
- Use later to check for status of completion of call
- Keep checking (loop or periodically) if handle has been posted
- Issue Wait(handle1, handle2, : : :) call with list of handles
- Wait call blocks until one of the stipulated handles is posted

Blocking/nonblocking; Synchronous/asynchronous; send/receive primities


process i
buffer_i
kernel_i
kernel_j
buffer_j
process j
S
S_C
R
R_C
W
P,
duration to copy data from or to user buffer
duration in which the process issuing send or receive primitive is blocked
S Send primitive issued S_C processing for Send completes
R Receive primitive issued R_C processing for Receive completes
P The completion of the previously initiated nonblocking operation
W Process may issue Wait to check completion of nonblocking operation

(a) blocking sync. Send, blocking Receive (b) nonblocking sync. Send, nonblocking Receive

(c) blocking async. Send (d) nonblocking async. Send

### *1.6.2. Processor synchrony*

- ***Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.***
- It is used to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

### *1.6.3. Libraries and standards*

- The message-passing interface (MPI) library and the PVM (parallel virtual machine) library
- Commercial software is often written using the remote procedure calls (RPC) mechanism for example, Sun RPC, and distributed computing environ-ment (DCE) RPC
- “Messaging” and “streaming” are two other mechanisms for communication, (RMI) and remote object invocation (ROI)

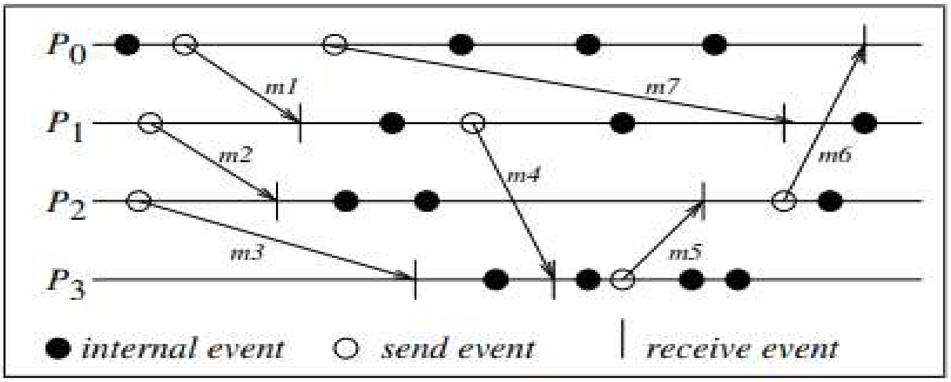
- CORBA (common object request broker architecture) and DCOM (distributed component object model) are two other standardized architectures with their own set of primitives

## 1.7 Synchronous versus asynchronous executions

An ***asynchronous execution*** is an execution in which

- There is no processor synchrony and there is no bound on the drift rate of processor clocks,
- Message delays (transmission + propagation times) are finite but unbounded, and
- There is no upper bound on the time taken by a process to execute a step.

*An example of an asynchronous execution in a message-passing system. A timing diagram is used to illustrate the execution*



An example asynchronous execution with four processes P0 to P3 is shown in Figure. The arrows denote the messages; the tail and head of an arrow mark the send and receive event for that message, denoted by a circle and vertical line, respectively. Non-communication events, also termed as internal events, are shown by shaded circles.

A ***synchronous execution*** is an execution in which

(i) processors are synchronized and the clock drift rate between any two processors is bounded,

(ii)message delivery (transmission + delivery) times are such that they occur in one logical step or round, and

(iii) there is a known upper bound on the time taken by a process to execute a step.

***There is a hurdle to having a truly synchronous execution***

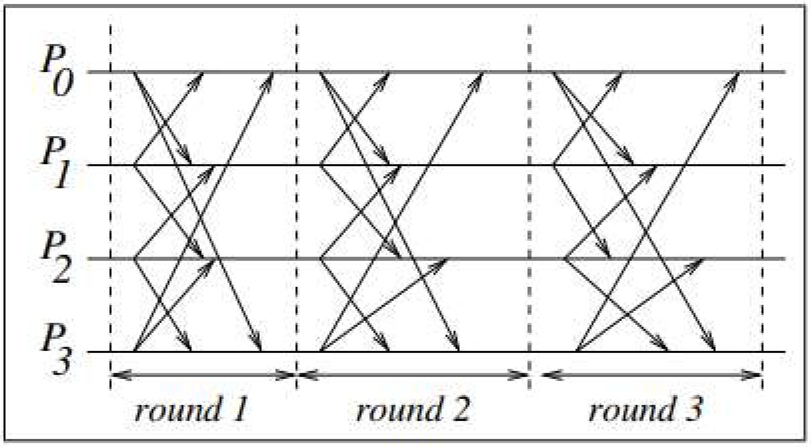
- It is practically difficult to build a completely synchronous system, and have the messages delivered within a bounded time.

- Therefore, this synchrony has to be simulated under the covers, and will inevitably involve delaying or blocking some processes for some time durations.
- Thus, synchronous execution is an abstraction that needs to be provided to the programs.
- When implementing this abstraction, observe that the fewer the steps or "synchronizations" of the processors, the lower the delays and costs.

***Virtual Synchrony***

- If processors are allowed to have an asynchronous execution for a period of time and then they synchronize, then the granularity of the synchrony is coarse. This is really a ***virtually synchronous execution***, and the abstraction is sometimes termed as ***virtual synchrony***.
- Ideally, many programs want the processes to execute a series of instructions in rounds (also termed as steps or phases) asynchronously, with the requirement that after each round/step/phase, all the processes should be synchronized and all messages sent should be delivered.
- This is the commonly understood notion of a synchronous execution. Within each round/phase/step, there may be a finite and bounded number of sequential sub-rounds (or sub-phases or sub-steps) that processes execute. Each sub-round is assumed to send at most one message per process; hence the message(s) sent will reach in a single message hop.

An example of a synchronous execution in a message-passing system. All the messages sent in a round are received within that same round.



In this system, there are four nodes $P_0$ to $P_3$. In each round, process Pi sends a message to $P_{i+1 \bmod 4}$ and $P_{i-1 \bmod 4}$ and calculates some application-specific function on the received values.

Synchronous execution in a message-passing system
In any round/step/phase: (send j internal) (receive j internal)

**Sync vs async executions**

- Async execution
  - ➢ No processor synchrony, no bound on drift rate of clocks
  - ➢ Message delays finite but unbounded
  - ➢ No bound on time for a step at a process
- Sync execution
  - ➢ Processors are synchronized; clock drift rate bounded
  - ➢ Message delivery occurs in one logical step/round
  - ➢ Known upper bound on time to execute a step at a process

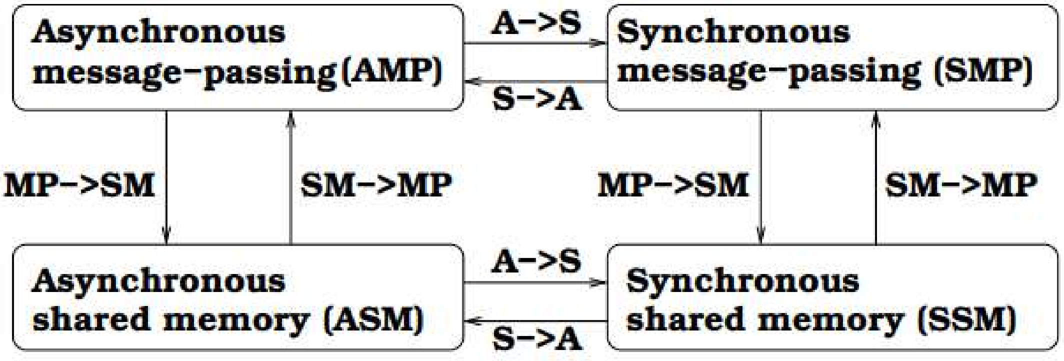
- Difficult to build a truly synchronous system; can simulate this abstraction
- Virtual synchrony:
  - async execution, processes synchronize as per application requirement;
  - execute in rounds/steps
- Emulations:
  - Async program on sync system: trivial (A is special case of S)
  - Sync program on async system: tool called synchronizer

**System Emulations**

➢ The shared memory system could be emulated by a message-passing system, and vice-versa

➢ If system A can be emulated by system B, denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A. Likewise, if a problem is solvable in A, it is also solvable in B. Hence, in a sense, all four classes are equivalent in terms of "computability" – what can and cannot be computed – in failure-free systems.

***Emulations among the principal system classes in a failure-free system.***


Asynchronous message-passing (AMP)
A->S
S->A
Synchronous message-passing (SMP)
MP->SM
SM->MP
MP->SM
SM->MP
Asynchronous shared memory (ASM)
A->S
S->A
Synchronous shared memory (SSM)

- Assumption: *failure-free system*
- System A emulated by system B:
  - If not solvable in B, not solvable in A

- If solvable in A, solvable in B

## 1.8 Design issues and challenges

- ❖ Distributed systems challenges from a system perspective
- ❖ Algorithmic challenges in distributed computing
- ❖ Applications of distributed computing and newer challenges

The categorization of design issues and challengesm as (i) having a greater component related to systems design and operating systems design, or (ii) having a greater component related to algorithm design, or (iii) emerging from recent technology advances and/or driven by new applications.

### ***1.8.1 Distributed systems challenges from a system perspective***

The following functions must be addressed when designing and building a distributed system:

**Communication mechanisms:** E.g., Remote Procedure Call (RPC), remote object invocation (ROI), message-oriented vs. stream-oriented communication

**Processes:** Code migration, process/thread management at clients and servers, design of software and mobile agents

**Naming:** Easy to use identifiers needed to locate resources and processes transparently and scalable.

**Synchronization**

Mechanisms for synchronization or coordination among the processes are essential. Mutual exclusion is the classical example of synchronization

**Data storage and access**

- Schemes for data storage, search, and lookup should be fast and scalable across network
- Revisit file system design

**Consistency and replication**

- Replication for fast access, scalability, avoid bottlenecks
- Require consistency management among replicas
- Fault-tolerance: correct and efficient operation despite link, node, process failures

**Distributed systems security**

- Secure channels, access control, key management (key generation and key distribution), authorization, secure group management
- Scalability and modularity of algorithms, data, services • Some experimental systems: Globe, Globus, Grid

**API for communications, services: ease of use**

Transparency: hiding implementation policies from user

- Access: hide differences in data rep across systems, provide uniform operations to access resources
- Location: locations of resources are transparent
- Migration: relocate resources without renaming
- Relocation: relocate resources as they are being accessed
- Replication: hide replication from the users
- Concurrency: mask the use of shared resources
- Failure: reliable and fault-tolerant operation

**Scalability and modularity**

- Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

### *1.8.2 Algorithmic challenges in distributed computing*

**Useful execution models** and frameworks: to reason with and design correct distributed programs

- Interleaving model
- Partial order model
- Input/Output automata
- Temporal Logic of Actions

**Dynamic distributed graph algorithms and routing algorithms**

- System topology: distributed graph, with only local neighborhood knowledge
- Graph algorithms: building blocks for group communication, data dissemination, object location
- Algorithms need to deal with dynamically changing graphs
- Algorithm efficiency: also impacts resource consumption, latency, traffic, congestion

**Time and global state**

- The processes in the system are spread across three-dimensional physical space. Another dimension, time, has to be superimposed uniformly across space.
- The challenges pertain to providing accurate physical time, and to providing a variant of time, called logical time
- Logical time captures inter-process dependencies and tracks relative time progression
- Global state observation: inherent distributed nature of system
- Concurrency measures: concurrency depends on program logic, execution speeds within logical threads, communication speeds

**Synchronization/coordination mechanisms**

Some examples of problems requiring synchronization:

- Physical clock synchronization: hardware drift needs correction
- Leader election: select a distinguished process, due to inherent symmetry
- Mutual exclusion: coordinate access to critical resources

- Distributed deadlock detection and resolution: need to observe global state; avoid duplicate detection, unnecessary aborts
- Termination detection: global state of quiescence; no CPU processing and no in-transit messages
- Garbage collection: Reclaim objects no longer pointed to by any process

**Group communication, multicast, and ordered message delivery**

- A group is a collection of processes that share a common context and collab-orate on a common task within an application domain.
- Multiple joins, leaves, fails
- Concurrent sends: semantics of delivery order

**Monitoring distributed events and predicates**

- Predicate: condition on global system state
- An important paradigm for monitoring distributed events is that of event streaming, wherein streams of relevant events reported from different processes are examined collectively to detect predicates.

**Distributed program design and verification tools**

- Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering.

**Debugging distributed programs**

- Debugging sequential programs is hard; debugging distributed programs is that much harder because of the concurrency in actions

**Data replication, consistency models, and caching**

- Fast, scalable access;
- coordinate replica updates;
- optimize replica placement

**World Wide Web design: caching, searching, scheduling**

- Global scale distributed system; end-users
- Read-intensive; prefetching over caching
- Object search and navigation are resource-intensive
- User-perceived latency

**Distributed shared memory abstraction**

- Wait-free algorithm design: process completes execution, irrespective of
    - actions of other processes, i.e., n - 1 fault-resilience
- Mutual exclusion
- Bakery algorithm, semaphores, based on atomic hardware primitives, fast algorithms when contention-free access
- Register constructions
- Revisit assumptions about memory access

**Consistency models:**

- For multiple copies of a variable/object, varying degrees of consistency among the replicas can be allowed.
- These represent a trade-off of coherence versus cost of implementation.

- Weaker models than strict consistency of uniprocessors

**Reliable and fault-tolerant distributed systems**

Consensus algorithms: processes reach agreement in spite of faults (under various fault models)

**Replication and replica management**

Replication (as in having backup servers) is a classical method of providing fault-tolerance. The triple modular redundancy (TMR) technique has long been used in software as well as hardware installations.

- Voting and quorum systems
- Distributed databases, commit: ACID properties
- Self-stabilizing systems: "illegal" system state changes to "legal" state; requires built-in redundancy
- Check pointing and recovery algorithms: roll back and restart from earlier "saved" state
- Failure detectors:
- Difficult to distinguish a "slow" process/message from a failed process/ never sent message algorithms that "suspect" a process as having failed and converge on a determination of its up/down status

**Load balancing**: to reduce latency, increase throughput, dynamically. E.g., server farms

- Computation migration: relocate processes to redistribute workload
- Data migration: move data, based on access patterns
- Distributed scheduling: across processors

**Real-time scheduling:** difficult without global view, network delays make task harder

**Performance modeling and analysis:** Network latency to access resources must be reduced

- Metrics: theoretical measures for algorithms, practical measures for systems
- Measurement methodologies and tools

### ***1.8.3 Applications of distributed computing and newer challenges***

**Mobile systems**

- Wireless communication: unit disk model; broadcast medium (MAC), power management etc.
- CS perspective: routing, location management, channel allocation, localization and position estimation, mobility management
- Base station model (cellular model)
- Ad-hoc network model (rich in distributed graph theory problems)

**Sensor networks**: Processor with electro-mechanical interface • Ubiquitous or pervasive computing

- Processors embedded in and seamlessly pervading environment

- Wireless sensor and actuator mechanisms; self-organizing; network-centric, resource-constrained
- E.g., intelligent home, smart workplace
- Peer-to-peer computing
- No hierarchy; symmetric role; self-organizing; efficient object storage and lookup; scalable; dynamic reconfiguration
- all processors are equal and play a symmetric role in the computation.

**Publish/subscribe, content distribution**

- Filtering information to extract that of interest

**Distributed agents**

- Processes that move and cooperate to perform specific tasks; coordination, controlling mobility, software design and interfaces

**Distributed data mining**

- Extract patterns/trends of interest
- Data not available in a single repository

**Grid computing**

- Grid of shared computing resources; use idle CPU cycles
- Issues: scheduling, QOS guarantees, security of machines and jobs

**Security**

- Confidentiality, authentication, availability in a distributed setting
- Manage wireless, peer-to-peer, grid environments
- Issues: e.g., Lack of trust, broadcast media, resource-constrained, lack of structure

## 1.9 A Model of Distributed Computations

### 1.9.1 A Distributed Program

- A distributed program is composed of a set of *n* asynchronous processes, $p_1$, $p_2$, ..., $p_i$ , ..., $p_n$.
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let $C_{ij}$ denote the channel from process $p_i$ to process $p_j$ and let $m_{ij}$ denote a message sent by $p_i$ to $p_j$.
- The message transmission delay is finite and unpredictable.

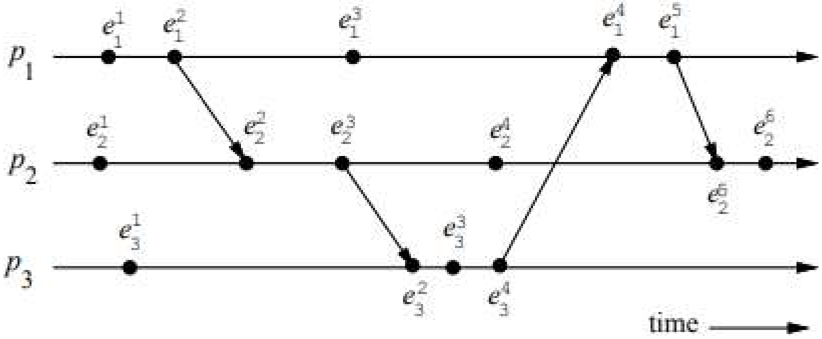
## 1.10 A Model of Distributed Executions

- The execution of a process consists of a sequential execution of its actions.
- The actions are atomic and the actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events.
- Let $e_i^x$ denote the *x* th event at process $p_i$ . For a message *m*, let *send*(*m*) and *rec*(*m*)

denote its send and receive events, respectively.

- The occurrence of events changes the states of respective processes and channels. An internal event changes the state of the process at which it occurs. A send event changes the state of the process that sends the message and the state of the channel on which the message is sent. A receive event changes the state of the process that receives the message and the state of the channel on which the message is received. The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process.
- A relation $\rightarrow_{msg}$ that captures the causal dependency due to message exchange, is defined as follows. For every message $m$ that is exchanged between two processes, we have $send(m) \rightarrow_{msg} rec(m)$.
- Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events.
- The evolution of a distributed execution is depicted by a space-time diagram.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.
- Since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to denote it as a dot on a process line.
- In the Figure, for process $p_1$, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

***<u>Figure : The space-time diagram of a distributed execution.</u>***



## ***Causal Precedence Relation***

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let $H=\cup_i h_i$ denote the set of events executed in a distributed computation.
- Define a binary relation → on the set $H$ as follows that expresses causal dependencies between events in the distributed execution.

$$\forall e_i^x, \forall e_j^y \in H, \quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \quad i.e., (i=j) \wedge (x<y) \\ or \\ e_i^x \rightarrow_{msg} e_j^y \\ or \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge \ e_k^z \rightarrow e_j^y \end{cases}$$

The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as H=(H, →).

- Note that the relation → is nothing but Lamport's "happens before" relation.
- For any two events $e_i$ and $e_j$ , if $e_i \rightarrow e_j$ , then event $e_j$ is directly or transitively dependent on event $e_i$. (Graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space-time diagram that starts at $e_i$ and ends at $e_j$.)
- For example, in Figure 2.1, $e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$.
- The relation → denotes flow of information in a distributed computation and $e_i \rightarrow e_j$ dictates that all the information available at $e_i$ is potentially accessible at $e_j$ .
- For example, in Figure 2.1, event $e_2^6$ has the knowledge of all other events shown in the figure.

- For any two events $e_i$ and $e_j$, $e_i \not\rightarrow e_j$ denotes the fact that event $e_j$ does not directly or transitively dependent on event $e_i$. That is, event $e_i$ does not causally affect event $e_j$.
- In this case, event $e_j$ is not aware of the execution of $e_i$ or any event executed after $e_i$ on the same process.
- For example, in Figure 2.1, $e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$.

Note the following two rules:

For any two events $e_i$ and $e_j$, $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$.

For any two events $e_i$ and $e_j$, $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.

Concurrent Events

- For any two events $e_i$ and $e_j$, if $e_i \not\rightarrow e_j$ and $e_j \not\rightarrow e_i$, then events $e_i$ and $e_j$ are said to be concurrent (denoted as $e_i \parallel e_j$).
- In the execution of Figure 2.1, $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$.
- The relation $\parallel$ is not transitive; that is, $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$.
- For example, in Figure 2.1, $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$, however, $e_3^3 \not\parallel e_1^5$.
- For any two events $e_i$ and $e_j$ in a distributed execution, $e_i \rightarrow e_j$ or $e_j \rightarrow e_i$, or $e_i \parallel e_j$.

## Logical vs. Physical Concurrency

- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other.
- Physical concurrency, on the other hand, has a connotation that the events occur at the same

instant in physical time.

- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time.
- However, if processor speed and message delays would have been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not, does not change the outcome of the computation.
- Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, we can assume that these events occured at the same instant in physical time.

## 1.11 Models of communication networks

- There are several models of the service provided by communication networks, namely, FIFO, Non-FIFO, and causal ordering.
- In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- The "causal ordering" model is based on Lamport's "happens before" relation.
- A system that supports the causal ordering model satisfies the following property:

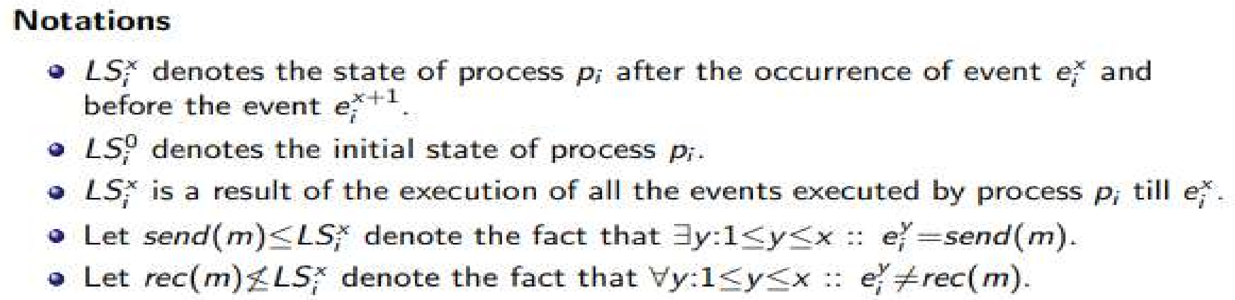
***CO:* For any two messages $m_{ij}$ and $m_{kj}$ ,if send ($m_{ij}$ )→send ($m_{kj}$ ), then rec ($m_{ij}$ ) → rec ($m_{kj}$ ).**

- This property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that CO $\subset$ FIFO $\subset$ Non-FIFO.)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

## 1.12 Global State of a Distributed System

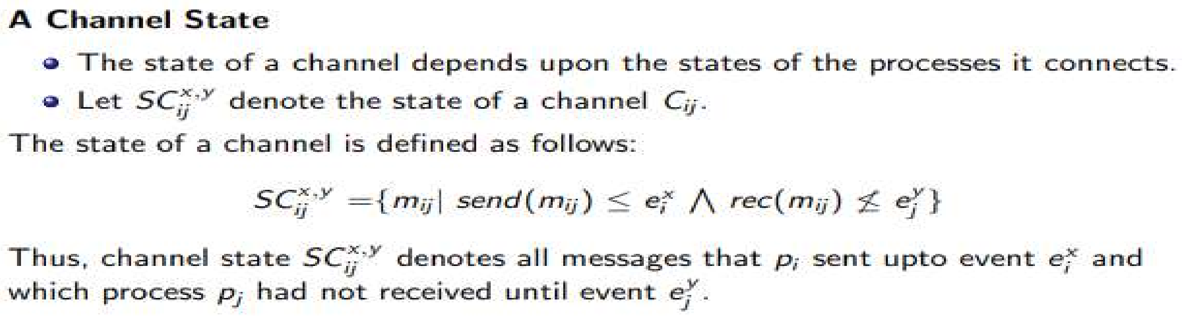
"The global state of a distributed system is a collection of the local states of its components, namely, the processes and the communication channels."

- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that or receives the message and the state of the channel on which the message is received.



**Notations**

- $LS_i^x$ denotes the state of process $p_i$ after the occurrence of event $e_i^x$ and before the event $e_i^{x+1}$.
- $LS_i^0$ denotes the initial state of process $p_i$.
- $LS_i^x$ is a result of the execution of all the events executed by process $p_i$ till $e_i^x$.
- Let $send(m) \leq LS_i^x$ denote the fact that $\exists y{:}1 \leq y \leq x :: e_i^y = send(m)$.
- Let $rec(m) \not\leq LS_i^x$ denote the fact that $\forall y{:}1 \leq y \leq x :: e_i^y \neq rec(m)$.



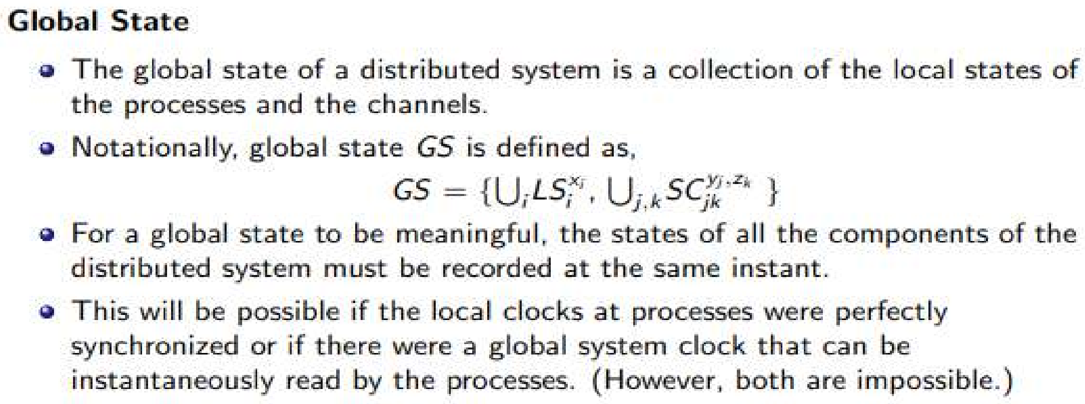
**A Channel State**

- The state of a channel depends upon the states of the processes it connects.
- Let $SC_{ij}^{x,y}$ denote the state of a channel $C_{ij}$.

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq e_i^x \wedge rec(m_{ij}) \not\leq e_j^y\}$$

Thus, channel state $SC_{ij}^{x,y}$ denotes all messages that $p_i$ sent upto event $e_i^x$ and which process $p_j$ had not received until event $e_j^y$.

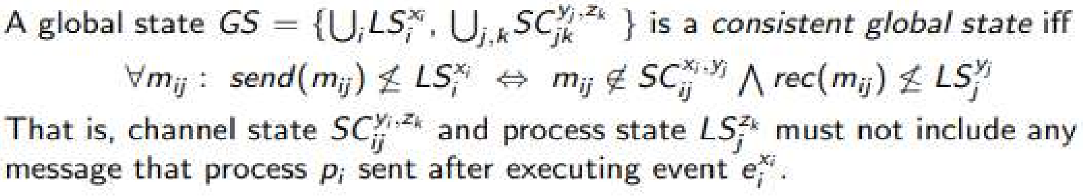


**Global State**

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state $GS$ is defined as,
$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$$
- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)

## A Consistent Global State

- Even if the state of all the components is not recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent.
- Basic idea is that a state should not violate causality – an effect should not be present without its cause. A message cannot be received if it was not sent.
- Such states are called *consistent global states* and are meaningful global states.



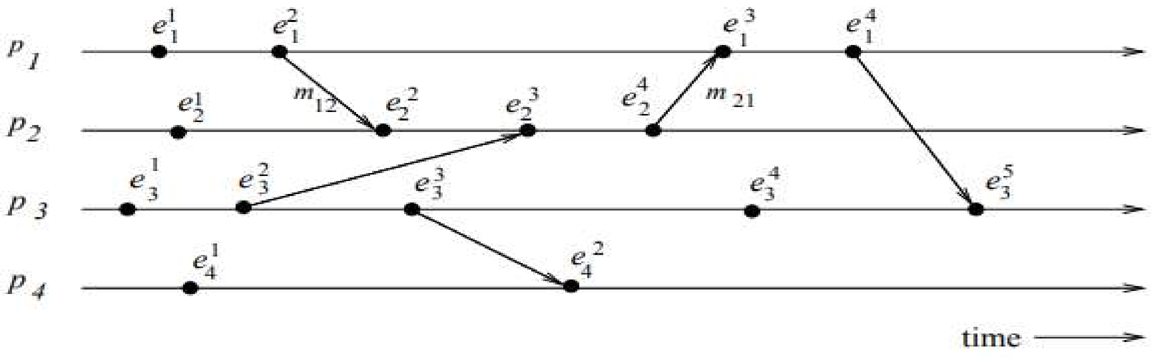
A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

$$\forall m_{ij} : send(m_{ij}) \not\leq LS_i^{x_i} \Leftrightarrow m_{ij} \notin SC_{ij}^{x_i,y_j} \wedge rec(m_{ij}) \not\leq LS_j^{y_j}$$

That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process $p_i$ sent after executing event $e_i^{x_i}$.

**An Example**

Consider the distributed execution of Figure



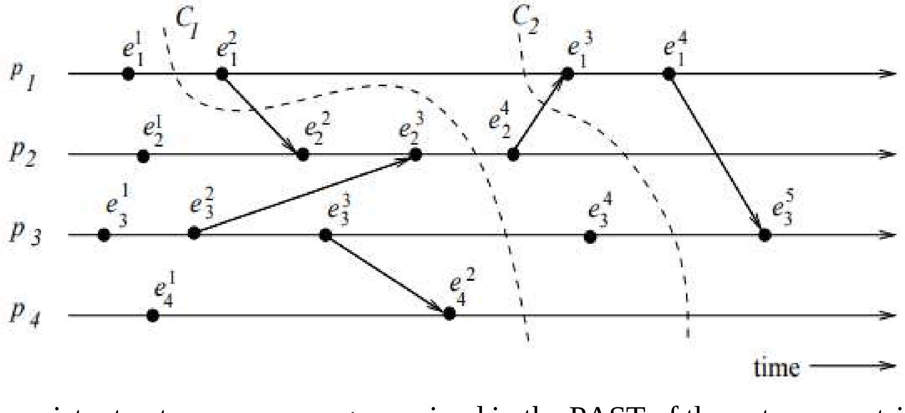
- A global state $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent because the state of $p_2$ has recorded the receipt of message $m_{12}$, however, the state of $p_1$ has not recorded its send.
- A global state $GS_2$ consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except $C_{21}$ that contains message $m_{21}$.

## 1.13 Cuts of a Distributed Computation

"In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line."

- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut $C$, let PAST($C$) and FUTURE($C$) denote the set of events in the PAST and FUTURE of $C$, respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space-time diagram.
- Cuts in a space-time diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

**Figure: Illustration of cuts in a distributed execution.**



- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure, cut $C_2$ is a consistent cut.)

- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is *inconsistent* if a message crosses the cut from the FUTURE to the PAST. (In Figure, cut $C_1$ is an inconsistent cut.)

## 1.14 Past and Future Cones of an Event

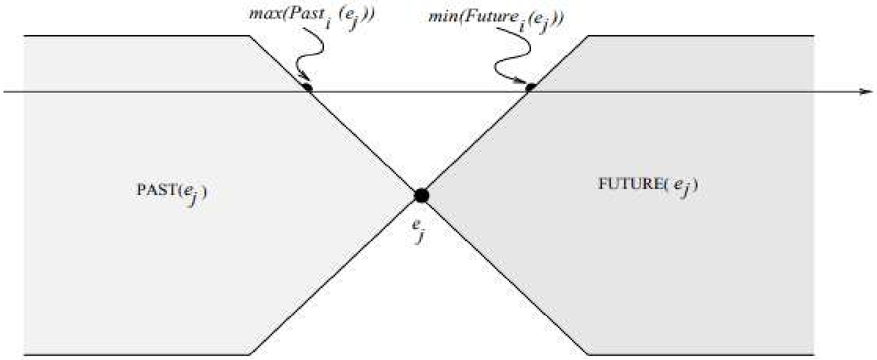
Past Cone of an Event

- An event $e_j$ could have been affected only by all events $e_i$ such that $e_i \rightarrow e_j$ .
- In this situtaion, all the information available at $e_i$ could be made accessible at $e_j$ .
- All such events $e_i$ belong to the past of $e_j$ .

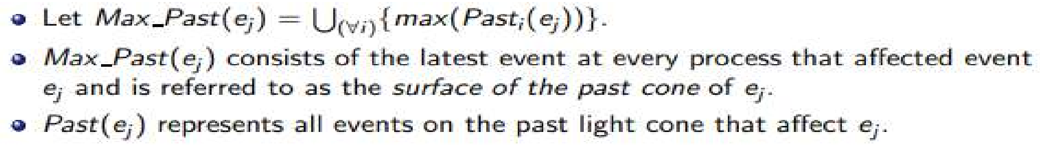
Let $Past(e_j)$ denote all events in the past of $e_j$ in a computation ($H$, →). Then,

$Past(e_j) = \{e_i \mid \forall e_i \in H, e_i \rightarrow e_j\}$.

Figure: Illustration of past and future cones.

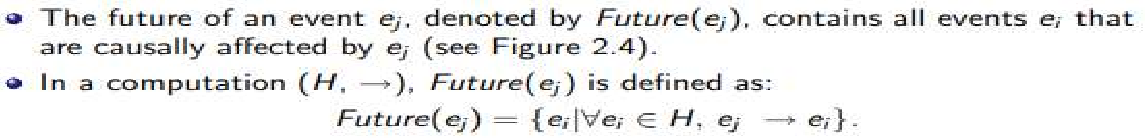


- Let $Past_i(e_j)$ be the set of all those events of $Past(e_j)$ that are on process $p_i$.
- $Past_i(e_j)$ is a totally ordered set, ordered by the relation $\rightarrow_i$ , whose maximal element is denoted by $max(Past_i(e_j))$.
- $max(Past_i(e_j))$ is the latest event at process $p_i$ that affected event $e_j$



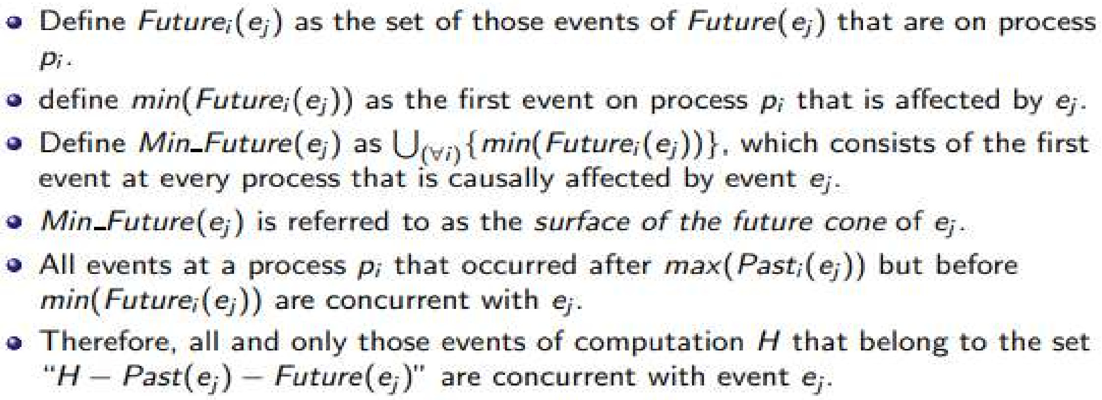
- Let $Max\_Past(e_j) = \bigcup_{(\forall i)}\{max(Past_i(e_j))\}$.
- $Max\_Past(e_j)$ consists of the latest event at every process that affected event $e_j$ and is referred to as the *surface of the past cone* of $e_j$.
- $Past(e_j)$ represents all events on the past light cone that affect $e_j$.

### Future cone of an Event



- The future of an event $e_j$, denoted by $Future(e_j)$, contains all events $e_i$ that are causally affected by $e_j$ (see Figure 2.4).
- In a computation ($H$, →), $Future(e_j)$ is defined as:

$$Future(e_j) = \{e_i \mid \forall e_i \in H, e_j \rightarrow e_i\}.$$



- Define $Future_i(e_j)$ as the set of those events of $Future(e_j)$ that are on process $p_i$.
- define $min(Future_i(e_j))$ as the first event on process $p_i$ that is affected by $e_j$.
- Define $Min\_Future(e_j)$ as $\bigcup_{(\forall i)}\{min(Future_i(e_j))\}$, which consists of the first event at every process that is causally affected by event $e_j$.
- $Min\_Future(e_j)$ is referred to as the *surface of the future cone* of $e_j$.
- All events at a process $p_i$ that occurred after $max(Past_i(e_j))$ but before $min(Future_i(e_j))$ are concurrent with $e_j$.
- Therefore, all and only those events of computation $H$ that belong to the set "$H - Past(e_j) - Future(e_j)$" are concurrent with event $e_j$.

## 1.15 Models of Process Communications

- There are two of basic models process communications – **synchronous and asynchronous.**
- The ***synchronous* communication** model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process. The sender process resumes execution only after it learns that the receiver process has accepted the message.
- Thus, the sender and the receiver processes must synchronize to exchange a message. On the other hand, *asynchronous* communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
- After having sent a message, the sender process does not wait for the message to be delivered to the receiver process. The message is buffered by the system and is delivered to the receiver process when it is ready to accept the message. Neither of the communication models is superior to the other.
- **Asynchronous communication** provides higher parallelism because the sender process can execute while the message is in transit to the receiver.
- However, A buffer overflow may occur if a process sends a large number of messages in a burst to another process. Thus, an implementation of asynchronous communication requires more complex buffer management.
- In addition, due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications.
- Synchronous communication is simpler to handle and implement.
- However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks.

## 1.16 Logical Time

### Introduction

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have a global physical time.
- As asynchronous distributed computations make progress in spurts, the logical time is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.
- This chapter discusses three ways to implement logical time - scalar time, vector time, and matrix time.
- Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.
- The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems, such as distributed algorithms design, tracking of dependent events, knowledge about the progress of a computation, and concurrency measures.

## 1.17 A Framework for a System of Logical Clocks

### 1.17.1 Definition

- A system of logical clocks consists of a time domain *T* and a logical clock *C* . Elements of *T* form a partially ordered set over a relation <.
- Relation < is called the ***happened before* or *causal precedence***. Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time.
- The logical clock *C* is a function that maps an event *e* in a distributed system to an element in the time domain *T*, denoted as C(*e*) and called the timestamp of *e*, and is defined as follows:

  $$\boldsymbol{C : H \rightarrow T}$$

  such that the following property is satisfied:

  for two events $e_i$ and $e_j$ , $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$.

  This monotonicity property is called the *clock consistency condition*. When *T* and *C* satisfy the following condition,
- for two events $e_i$ and $e_j$ , $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$

  the system of clocks is said to be *strongly consistent*.

### 1.17.2 Implementing Logical Clocks

- Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol to update the data structures to ensure the consistency condition.
- Each process $p_i$ maintains data structures that allow it the following two capabilities:

A *local logical clock*, denoted by $lc_i$ , that helps process $p_i$ measure its own progress.

A *logical global clock*, denoted by $gc_i$, that is a representation of process $p_i$ ’s local view of the logical global time. Typically, $lc_i$ is a part of $gc_i$ .

- The protocol ensures that a process’s logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

*R1*: This rule governs how the local logical clock is updated by a process when it executes an event.

*R2*: This rule governs how a process updates its global logical clock to update its view of the global time and global progress.

- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.

## 1.18 Scalar Time

- The scalar time representation was proposed by Lamport in 1978 [9] as an attempt to totally order events in a distributed system. Time domain in this representation is the set of non-negative integers.
- The logical local clock of a process $p_i$ and its local view of the global time are squashed into one integer variable $C_i$ .
- Rules *R1* and *R2* to update the clocks are as follows:

*R1*: Before executing an event (send, receive, or internal), process $p_i$ executes the following:
$C_i := C_i + d \quad (d > 0)$ In general, every time *R1* is executed, $d$ can have a different value; however, typically $d$ is kept at 1.

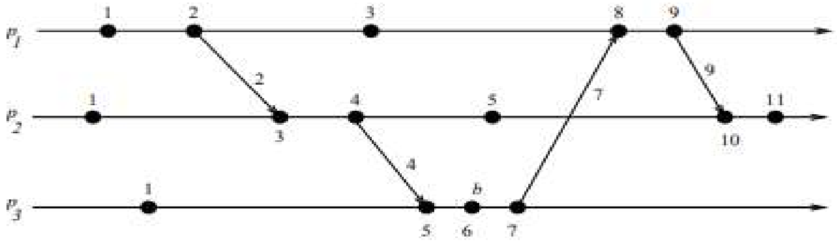
*R2*: Each message piggybacks the clock value of its sender at sending time. When a process $p_i$ receives a message with timestamp $C_{msg}$, it executes the following actions:

1. $C_i := max\ (C_i\ , C_{msg}\ )$
2. Execute *R1*.
3. Deliver the message.

- Figure shows evolution of scalar time.

### Evolution of scalar time:

Figure : The space-time diagram of a distributed execution.



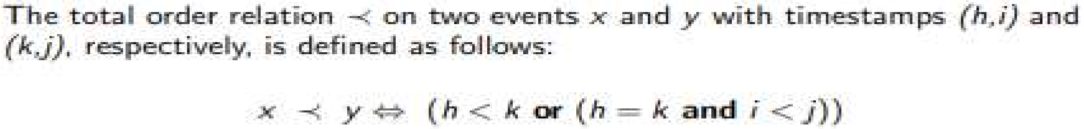
### Basic Properties

### Consistency Property

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

### Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure, the third event of process $P_1$ and the second event of process $P_2$ have identical scalar timestamp.
- A tie-breaking mechanism is needed to order such events. A tie is broken as follows:
- Process identifiers are linearly ordered and tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple $(t, i)$ where $t$ is its time of occurrence and $i$ is the identity of the process where it occurred.



The total order relation $\prec$ on two events $x$ and $y$ with timestamps $(h,i)$ and $(k,j)$, respectively, is defined as follows:

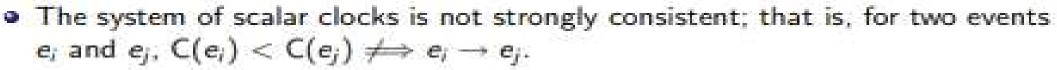
$$x \prec y \Leftrightarrow (h < k \textbf{ or } (h = k \textbf{ and } i < j))$$

## Event counting

- If the increment value $d$ is always 1, the scalar time has the following interesting property: if event $e$ has a timestamp $h$, then $h$-$1$ represents the minimum logical duration, counted in units of events, required before producing the event $e$;
- We call it the height of the event $e$.
- In other words, $h$-$1$ events have been produced sequentially before the event $e$ regardless of the processes that produced these events.

For example, in Figure, five events precede event b on the longest causal path ending at b.

## No Strong Consistency



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

- For example, in Figure, the third event of process $P_1$ has smaller scalar timestamp than the third event of process $P_2$. However, the former did not happen before the latter.
- The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.
- For example, in Figure, when process P2 receives the first message from process P1, it updates its clock to 3, forgetting that the timestamp of the latest event at P1 on which it depends is 2.

## 1.19 Vector Time

- The system of vector clocks was developed independently by Fidge, Mattern and Schmuck.
- In the system of vector clocks, the time domain is represented by a set of $n$-dimensional non-negative integer vectors.
- Each process $p_i$ maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of $p_i$ and

describes the logical time progress at process $p_i$.

$vt_i\,[j]$ represents process $p_i$ 's latest knowledge of process $p_j$ local time.

If $vt_i[j]=x$, then process $p_i$ knows that local time at process $p_j$ has progressed till $x$.

The entire vector $vt_i$ constitutes $p_i$'s view of the global logical time and is used to timestamp events.

- Process $p_i$ uses the following two rules *R1* and *R2* to update its clock:

*R1*: Before executing an event, process $p_i$ updates its local logical time as follows:
$vt_i[i] := vt_i[i] + d \qquad\qquad (d > 0)$

*R2*: Each message *m* is piggybacked with the vector clock *vt* of the sender process at sending time. On the receipt of such a message *(m,vt)*, process $p_i$ executes the following sequence of actions:

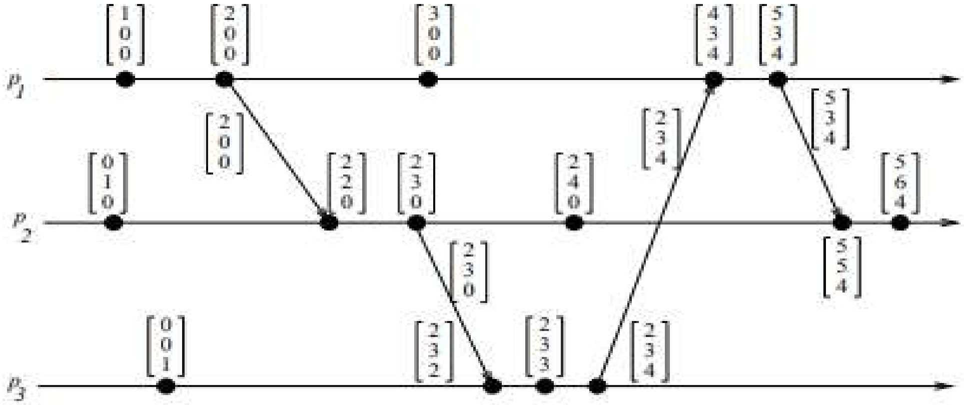
1. ***Update its global logical time as follows:***
   ***$1 \le k \le n$ : $vt_i\,[k] := \max\,(vt_i\,[k], vt[k])$***
2. ***Execute R1.***
3. ***Deliver the message m.***

The timestamp of an event is the value of the vector clock of its process when the event is executed.

Figure shows an example of vector clocks progress with the increment value *d=1*.

Initially, a vector clock is [0,0,0,........,0].

## An Example of Vector Clocks


$p_1$
$p_2$
$p_3$
$\begin{bmatrix}1\\0\\0\end{bmatrix}$
$\begin{bmatrix}2\\0\\0\end{bmatrix}$
$\begin{bmatrix}3\\0\\0\end{bmatrix}$
$\begin{bmatrix}4\\3\\4\end{bmatrix}$
$\begin{bmatrix}5\\3\\4\end{bmatrix}$
$\begin{bmatrix}2\\0\\0\end{bmatrix}$
$\begin{bmatrix}2\\3\\4\end{bmatrix}$
$\begin{bmatrix}5\\3\\4\end{bmatrix}$
$\begin{bmatrix}0\\1\\0\end{bmatrix}$
$\begin{bmatrix}2\\2\\0\end{bmatrix}$
$\begin{bmatrix}2\\3\\0\end{bmatrix}$
$\begin{bmatrix}2\\4\\0\end{bmatrix}$
$\begin{bmatrix}5\\6\\4\end{bmatrix}$
$\begin{bmatrix}2\\3\\0\end{bmatrix}$
$\begin{bmatrix}5\\5\\4\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\end{bmatrix}$
$\begin{bmatrix}2\\3\\2\end{bmatrix}$
$\begin{bmatrix}2\\3\\3\end{bmatrix}$
$\begin{bmatrix}2\\3\\4\end{bmatrix}$

## Comparing Vector Timestamps

The following relations are defined to compare two vector timestamps, *vh* and *vk* :

$$
\begin{array}{rcl}
vh = vk & \Leftrightarrow & \forall x : vh[x] = vk[x] \\
vh \leq vk & \Leftrightarrow & \forall x : vh[x] \leq vk[x] \\
vh < vk & \Leftrightarrow & vh \leq vk \text{ and } \exists x : vh[x] < vk[x] \\
vh \parallel vk & \Leftrightarrow & \neg(vh < vk) \land \neg(vk < vh)
\end{array}
$$

If the process at which an event occurred is known, the test to compare two timestamps can be simplified as follows: If events *x* and *y* respectively occurred at processes $p_i$ and $p_j$ and are assigned timestamps *vh* and *vk*, respectively, then

$$x \rightarrow y \quad \Leftrightarrow \quad vh[i] \leq vk[i]$$
$$x \parallel y \quad \Leftrightarrow \quad vh[i] > vk[i] \wedge vh[j] < vk[j]$$

## Basic Properties of Vector Time

### Isomorphism

- If events in a distributed system are time stamped using a system of vector clocks, we have the following property.
- If two events *x* and *y* have timestamps *vh* and *vk*, respectively, then

$$x \rightarrow y \;\; \Leftrightarrow \;\; vh < vk \; x \parallel y \Leftrightarrow vh \parallel vk.$$

- Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps

### Strong Consistency

- The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.
- However, Charron-Bost showed that the dimension of vector clocks cannot be less than n, the total number of processes in the distributed computation, for this property to hold.

### Event Counting

- If $d$=1 (in rule *R1*), then the $i^{th}$ component of vector clock at process $p_i$, $vt_i[i]$, denotes the number of events that have occurred at $p_i$ until that instant.
- So, if an event e has timestamp vh,

  vh[j] denotes the number of events executed by process pj that causally precede e. Clearly, vh[j] – 1 represents the total number of events that causally precede *e* in the distributed computation.

### Applications

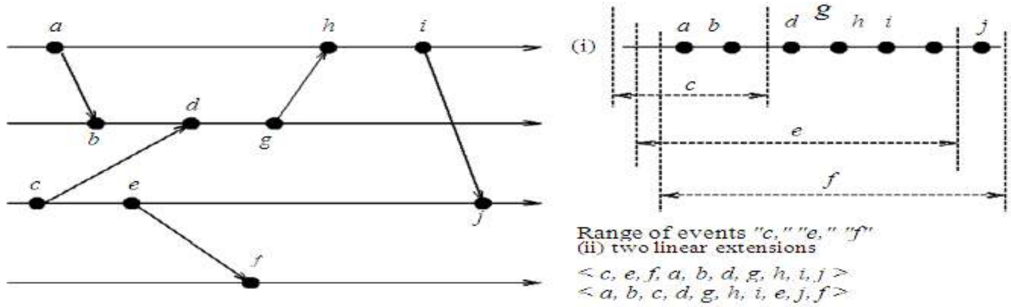
- Distributed debugging,
- Implementations of causal ordering,
- Communication and causal distributed shared memory,
- Establishment of global breakpoints
- Determining the consistency of checkpoints in optimistic recovery

### Size of vector clocks

A linear extension of a partial order E ≺ is a linear ordering of E that is consistent with the partial order, i.e., if two events are ordered in the partial order, they are also ordered in the linear order. A linear extension can be viewed as projecting all the events from the different processes on a single time axis. However, the linear order will necessarily introduce ordering between each pair of events, and some of these orderings are not in the partial order.

Now consider an execution on processes P1 and P2 such that each sends a message to the other before receiving the other’s message. The two send events are concurrent, as are the two receive events. To determine the causality between the send events or between the receive events, it is not sufficient to use a single integer; a vector clock of size n = 2 is necessary. This execution exhibits the ***graphical property called a crown,*** wherein there are some messages m0 mn−1 such that

Send mi < Receive mi+1 mod n−1 for all i from 0 to n − 1. A crown of n messages has dimension n


a
h
i
b
d
g
c
e
j
f
(i)
a b d g h i j
c
e
f
Range of events "c," "e," "f"
(ii) two linear extensions
< c, e, f, a, b, d, g, h, i, j >
< a, b, c, d, g, h, i, e, j, f >

***Dimension of a execution  For n = 4 processes, the dimension is 2.***

## 1.20 Physical Clock Synchronization: NTP

### Motivation

In centralized systems, there is only single clock. A process gets the time by simply issuing a system call to the kernel. In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time. These clocks can easily drift seconds per day, accumulating significant errors over time. Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start. This clearly poses serious problems to applications that depend on a synchronized notion of time.

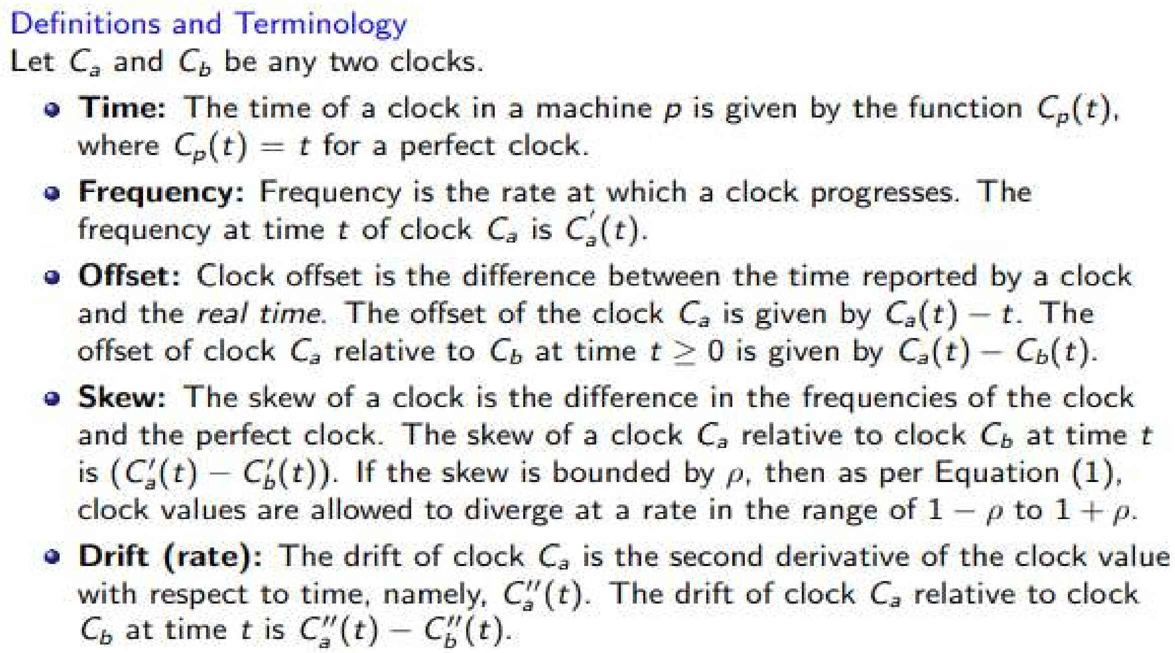
For most applications and algorithms that run in a distributed system, we need to know time in one or more of the following contexts:

- The time of the day at which an event happened on a specific machine in the network.
- The time interval between two events that happened on different machines in the network.
- The relative ordering of events that happened on different machines in the network.

Unless the clocks in each machine have a common notion of time, time-based queries cannot be answered. Clock synchronization has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values.

- Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time.
- Due to different clocks rates, the clocks at various sites may diverge with time and periodically a clock synchronization must be performed to correct this clock skew in distributed systems.
- Clocks are synchronized to an accurate real-time standard like **UTC (Universal Coordinated Time).**

Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed *physical clocks*.



## Definitions and Terminology

Let $C_a$ and $C_b$ be any two clocks.

- **Time:** The time of a clock in a machine $p$ is given by the function $C_p(t)$, where $C_p(t) = t$ for a perfect clock.
- **Frequency:** Frequency is the rate at which a clock progresses. The frequency at time $t$ of clock $C_a$ is $C_a'(t)$.
- **Offset:** Clock offset is the difference between the time reported by a clock and the *real time*. The offset of the clock $C_a$ is given by $C_a(t) - t$. The offset of clock $C_a$ relative to $C_b$ at time $t \geq 0$ is given by $C_a(t) - C_b(t)$.
- **Skew:** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock $C_a$ relative to clock $C_b$ at time $t$ is $(C_a'(t) - C_b'(t))$. If the skew is bounded by $\rho$, then as per Equation (1), clock values are allowed to diverge at a rate in the range of $1 - \rho$ to $1 + \rho$.
- **Drift (rate):** The drift of clock $C_a$ is the second derivative of the clock value with respect to time, namely, $C_a''(t)$. The drift of clock $C_a$ relative to clock $C_b$ at time $t$ is $C_a''(t) - C_b''(t)$.

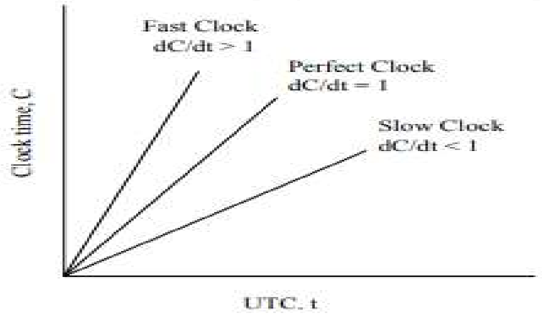
## Clock Inaccuracies

Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).

However, due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if (where constant ρ is the maximum skew rate specified by the manufacturer.)

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

*Figure illustrates the behavior of fast, slow, and perfect clocks with respect to UTC.*



## Offset delay estimation method

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

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

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

## Clock offset and delay estimation:

In practice, a source node cannot accurately estimate the local time on the target node due to varying message or network delays between the nodes. This protocol employs a common practice of performing several trials and chooses the trial with the minimum delay.

Figure shows how NTP timestamps are numbered and exchanged between peers $A$ and $B$.

Let $T_1, T_2, T_3, T_4$ be the values of the four most recent timestamps as shown. Assume clocks $A$ and $B$ are stable and running at the same speed.

## Offset and delay estimation.

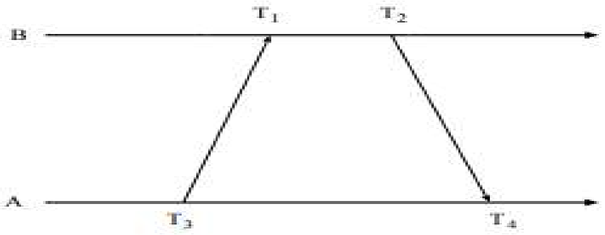
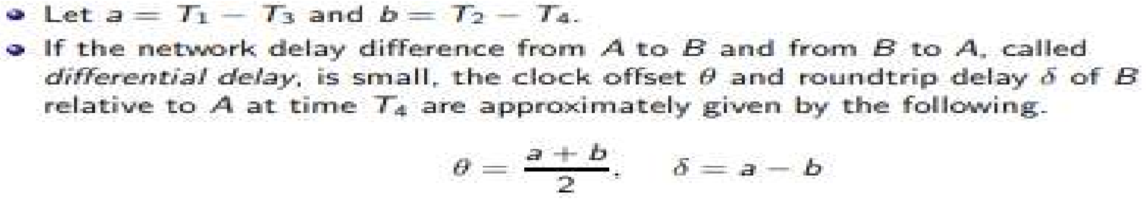


Figure 3.6: Offset and delay estimation.



- Let $a = T_1 - T_3$ and $b = T_2 - T_4$.
- If the network delay difference from $A$ to $B$ and from $B$ to $A$, called *differential delay*, is small, the clock offset $\theta$ and roundtrip delay $\delta$ of $B$ relative to $A$ at time $T_4$ are approximately given by the following.

$$\theta = \frac{a+b}{2}, \quad \delta = a - b$$

Each NTP message includes the latest three timestamps $T_1$, $T_2$ and $T_3$, while $T_4$ is determined upon arrival. Thus, both peers $A$ and $B$ can independently calculate delay and offset using a single bidirectional message stream as shown in Figure.

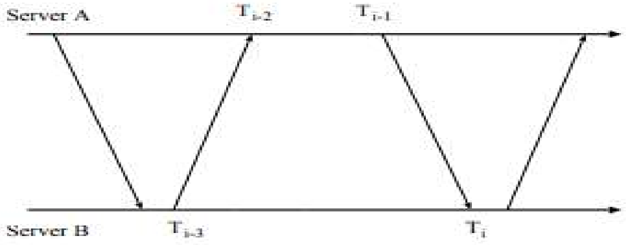


Figure 3.7: Timing diagram for the two servers.

## PART A

1. **What Is Distributed system?**

- A **distributed system** is a **system** whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.
- A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.
- Autonomous processors communicating over a communication network

2. **Listout the Characteristics of Distributed Systems**

- **No common physical clock** -> "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.
- **No shared memory** -> distributed system may still provide the abstraction of a common address space via the distributed shared memory abstraction.
- **Geographical separation** -> The geographically wider apart that the processors are, the more representative is the system of a distributed system network/cluster of workstations (NOW/COW) configuration connecting processors. The Google search engine is based on the NOW architecture.
- **Autonomy and heterogeneity** -> The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system.

3. **What is distributed execution.**
A *distributed execution* is the execution of processes across the distributed system to collaboratively achieve a common goal. An execution is also sometimes termed a *computation* or a *run*.

4. **Listout the motivation of distributed systems.**

8. Inherently distributed computations
9. Resource sharing
10. Access to geographically remote data and resources
11. Enhanced reliability
12. Increased performance/cost ratio

5. **List out two standard architectures for parallel systems.**
Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

6. **What is multicomputer parallel system.**
A *multicomputer parallel system* is a parallel system in which the multiple processors *do not have direct access to shared memory*. The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.

7. **What is Hamming distance**

- The processors are labelled such that the shortest path between any two processors is the *Hamming distance* (defined as the number of bit positions in which the two equal sized bit strings differ) between the processor labels.
- Example Nodes 0101 and 1100 have a Hamming distance of 2. The shortest path between them has length 2.

8. **What is Array processors?**
Array processors belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock (but may not share memory and communicate by passing data using messages).
   - Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing belong to this category.

9. **Describe about flynns classification.**
Flynn identified four processing modes, based on whether the processors execute the same or different instruction streams at the same time, and whether or not the processors processed the same (identical) data at the same time.
SISD: Single Instruction Stream Single Data Stream (traditional)
SIMD: Single Instruction Stream Multiple Data Stream
MISD: Multiple Instruction Stream Single Data Stream
MIMD: Multiple Instruction Stream Multiple Data Stream

10. **Define Coupling**

The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules.

**11. List out MIMD architectures in terms of coupling:**

- Tightly coupled multiprocessors (with UMA shared memory). These may be either switch-based
- Tightly coupled multiprocessors (with NUMA shared memory or that communicate by message passing).
- Loosely coupled multi computers (without shared memory) physically co-located. These may be bus-based and the processors may be heterogeneous
- Loosely coupled multi computers (without shared memory and without common clock) that are physically remote.

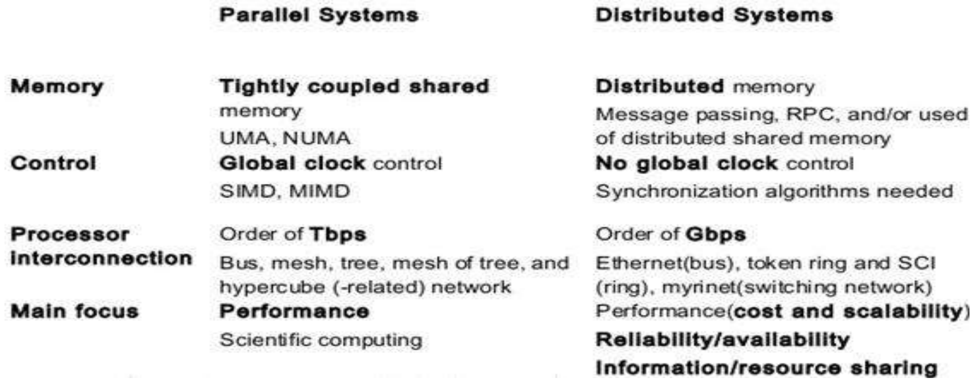
**12. Define Concurrency of a program**

The *parallelism/concurrency* in a parallel/distributed program can be measured by the ratio of the number of local (non-communication and non-shared memory access) operations to the total number of operations, including the communication or shared memory access operations.

**13. Define *granularity*.**

- The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as *granularity*.
- Programs with fine-grained parallelism are best suited for tightly coupled systems. Eg. SIMD and MISD architectures

**14. Differentiate pararall systems and distributed systems.**



| | **Parallel Systems** | **Distributed Systems** |
|---|---|---|
| **Memory** | **Tightly coupled shared** memory<br>UMA, NUMA | **Distributed** memory<br>Message passing, RPC, and/or used of distributed shared memory |
| **Control** | **Global clock** control<br>SIMD, MIMD | **No global clock** control<br>Synchronization algorithms needed |
| **Processor interconnection** | Order of **Tbps**<br>Bus, mesh, tree, mesh of tree, and hypercube (-related) network | Order of **Gbps**<br>Ethernet(bus), token ring and SCI (ring), myrinet(switching network) |
| **Main focus** | **Performance**<br>Scientific computing | Performance(**cost and scalability**)<br>**Reliability/availability**<br>**Information/resource sharing** |

**15. Identify some distributed applications in the scientific and commercial application areas. For each application, determine which of the motivating factors are important for building the application over a distributed system.**

- Scientific: Cosmology@Home

a. Inherently distributed
b. Resource sharing: CPU time

- Commercial: HDFS

a. Access
b. Reliability
c. Scalability
d. Modularity and Expandability

**16. Explain why a Receive call cannot be asynchronous.**

Async is about copying out. But the Receive is about copying in user-buffer. After copying-in, we can continue the user code immediately. This is different from sending -- which takes OS-time and other time out of user-code.

**17. What are the three aspects of reliability? Is it possible to order them in different ways in terms of importance, based on different applications' requirements? Justify your answer by giving examples of different applications.**

- Availability
- Integrity
- Fault-tolerance

Yes.
For banking service, fault-tolerance is of the top-most importance.
But for web service, availability is the most important one.

**18. The emulations among the principal system classes in a failure-free system. 1. Which of these emulations are possible in a failure-prone system? Explain. 2. Which of these emulations are not possible in a failure-prone system? Explain.**

1. Impossible
- MP -> SM: If there is no previously sent messages, any read will cause error
- S -> A: If the process is out of synchronization, there will be error

2. Possible
- SM -> MP
- A -> S

**19. Listout Primitives for distributed communication**

- Nonblocking primitive
- Blocking primitive
- Asynchronous primitive
- Synchronous primitive

**20. What is processor synchrony**

- ***Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.***
- It is used to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

**21. Differentiate asynchronous execution and Synchronous execution.**

An ***asynchronous execution*** is an execution in which

- There is no processor synchrony and there is no bound on the drift rate of processor clocks,
- Message delays (transmission + propagation times) are finite but unbounded, and
- There is no upper bound on the time taken by a process to execute a step.

A ***synchronous execution*** is an execution in which

(iv) processors are synchronized and the clock drift rate between any two processors is bounded,
(v) message delivery (transmission + delivery) times are such that they occur in one logical step or round, and
(vi) there is a known upper bound on the time taken by a process to execute a step.

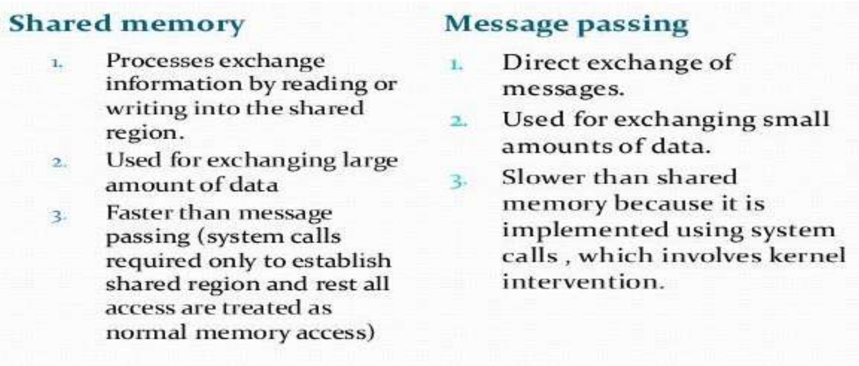
**22. Define Virtual Synchrony**

- If processors are allowed to have an asynchronous execution for a period of time and then they synchronize, then the granularity of the synchrony is coarse. This is really a ***virtually synchronous execution***, and the abstraction is sometimes termed as ***virtual synchrony***.

23. Define a Distributed Program

- A distributed program is composed of a set of *n* asynchronous processes, $p_1$, $p_2$, ..., $p_i$ , ..., $p_n$.
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.

**24. Differentiate Shared memory and message passing**



| Shared memory | Message passing |
|---|---|
| 1. Processes exchange information by reading or writing into the shared region. | 1. Direct exchange of messages. |
| 2. Used for exchanging large amount of data | 2. Used for exchanging small amounts of data. |
| 3. Faster than message passing (system calls required only to establish shared region and rest all access are treated as normal memory access) | 3. Slower than shared memory because it is implemented using system calls , which involves kernel intervention. |

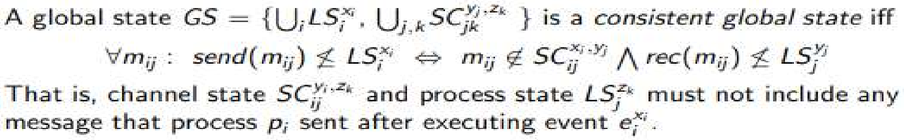
**25. List out the models of communication networks**

- There are several models of the service provided by communication networks, namely, FIFO, Non-FIFO, and causal ordering.
- In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- The "causal ordering" model is based on Lamport's "happens before" relation.

**26. Define global state of a distributed system**

- "The global state of a distributed system is a collection of the local states of its components, namely, the processes and the communication channels."
- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.

**27. What is Consistent global state**



A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

$$\forall m_{ij} : \ send(m_{ij}) \not\preceq LS_i^{x_i} \ \Leftrightarrow \ m_{ij} \notin SC_{ij}^{x_i,y_j} \wedge rec(m_{ij}) \not\preceq LS_j^{y_j}$$

That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process $p_i$ sent after executing event $e_i^{x_i}$.

**28. What is CUT**

"In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line."

- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.

**29. What is physical clocks?**

Coordinating **physical clocks** among several **systems** is possible, but it can never be exact. In**distributed systems**, we must be willing to accept some drift away from the "real" time on each **clock**. A typical real-time **clock** within a computer has a relative error of approxmiately $10^{-5}$.

**30. Define the following terms.**

Skew: Disagreement in the reading of two clocks
Drift: Difference in the rate at which two clocks count the time
Due to physical differences in crystals, plus heat, humidity, voltage, etc.
Accumulated drift can lead to significant skew
Clock drift rate: Difference in precision between a prefect reference clock and a physical clock

# CS8603 – DISTRIBUTED SYSTEMS

## PART B

1. Define distributed system. Listout the characteristics of distributed systems. How to relate the computer system components in distributed environment. (1.1 & 1.2)
2. Describe the motivations of implementing distributed systems. (1.3)
3. Describe the parallel systems with examples. (1.4)
4. Differentiate message passing and shared memory and how they emulate (1.5)
5. Describe the primitives of distributed computing (1.6)
6. Differentiate sync and async execution with example. (1.7)
7. Explain the Design issues and challenges of distributed computing. (1.8)
8. Discuss the model of distributed execution. (1.10)
9. Explain global states with example. (1.12)
10. What is cut and past, future cones of an event in distributed systems (1.13 &1.14)
11. Explain Logical clocks with example.(1.16 &1.17)
12. Discuss scalar time and its properties. (1.18)
13. Discuss Vector time(1.19)
14. Explain physical clock synchronization with example (1.20)

## UNIT II -Message ordering and group communication

**Message ordering paradigms –Asynchronous execution with synchronous communication –Synchronous program order on an asynchronous system –Group communication – Causal order (CO) – Total order. Global state and snapshot recording algorithms: Introduction –System model and definitions –Snapshot algorithms for FIFO channels**

## 2.1 MESSAGE ORDERING PARADIGMS

### Notations

We model the distributed system as a graph (N, L). The following notation is used to refer to messages and events:

- When referring to a message without regard for the identity of the sender and receiver processes, we use $m^i$. For message $m^i$, its send and receive events are denoted as $s^i$ and $r^i$, respectively.
- More generally, send and receive events are denoted simply as s and r. When the relationship between the message and its send and receive events is to be stressed, we also use M, send(M) , and receive(M) respectively.

For any two events a and b, where each can be either a send event or a receive event, the notation a ~ b denotes that a and b occur at the same process, i.e., $a \in E_i$ and $b \in E_i$ for some process i.

The send and receive event pair for a message is said to be **a pair of *corresponding* events**. The send event corresponds to the receive event, and vice-versa. For a given execution E, let the set of all send–receive event pairs be denoted as $T = \{(s,r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$.

### Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program.

Several orderings on messages have been defined:

(i) non-FIFO,

(ii) FIFO,

(iii) causal order, and

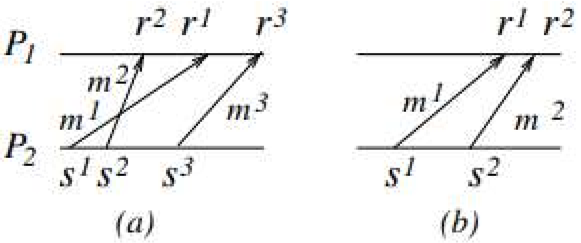
(iv) synchronous order.

### 2.1.1 Asynchronous and FIFO Executions

**Definition (A-execution) : An asynchronous execution (or A-execution) is an execution E≺ for which the causality relation is a partial order.**

- On any logical link between two nodes in the system, messages may be delivered in any order, *not necessarily* first-in first-out. Such executions are also known as *non-FIFO executions*. , e.g., network layer IPv4 connectionless service
- **All physical links obey FIFO**

(a) A-execution that is not FIFO (b) A-execution that is FIFO

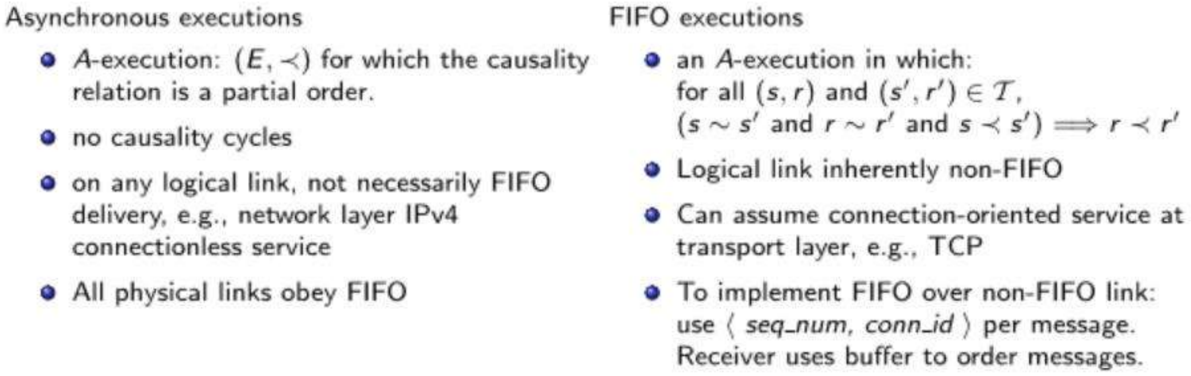


### 2.1.2 FIFO executions

**Definition (FIFO executions) A FIFO execution is an A-execution in which, for all (s, r) and (s', r')∈ *T*, (s ~ s'and r ~ r' and s ≺ s' ) => r ≺ r' .**

- Logical link inherently non-FIFO
- Can assume connection-oriented service at transport layer, e.g., TCP
- To implement FIFO over non-FIFO link: use < seq num, conn id > per message. Receiver uses buffer to order messages.

**Difference between Asynchronous and FIFO executions.**



Asynchronous executions

- *A*-execution: $(E, \prec)$ for which the causality relation is a partial order.
- no causality cycles
- on any logical link, not necessarily FIFO delivery, e.g., network layer IPv4 connectionless service
- All physical links obey FIFO

FIFO executions

- an *A*-execution in which: for all $(s, r)$ and $(s', r') \in \mathcal{T}$, $(s \sim s'$ and $r \sim r'$ and $s \prec s') \Longrightarrow r \prec r'$
- Logical link inherently non-FIFO
- Can assume connection-oriented service at transport layer, e.g., TCP
- To implement FIFO over non-FIFO link: use ⟨ *seq_num, conn_id* ⟩ per message. Receiver uses buffer to order messages.

### 2.1.3 Causal order (CO)

**A CO execution is an a execution in which, for all (s,r) and (s' ,r' ) ∈ T , (r ∼ r' and s ≺ s' ) =⇒ r ≺ r'**

- **If send events s and s' are related by causality ordering (not physical time ordering), their corresponding receive events r and r' occur in the same order at all common destinations.**
- **If s and s' are not related by causality, then CO is vacuously satisfied.**

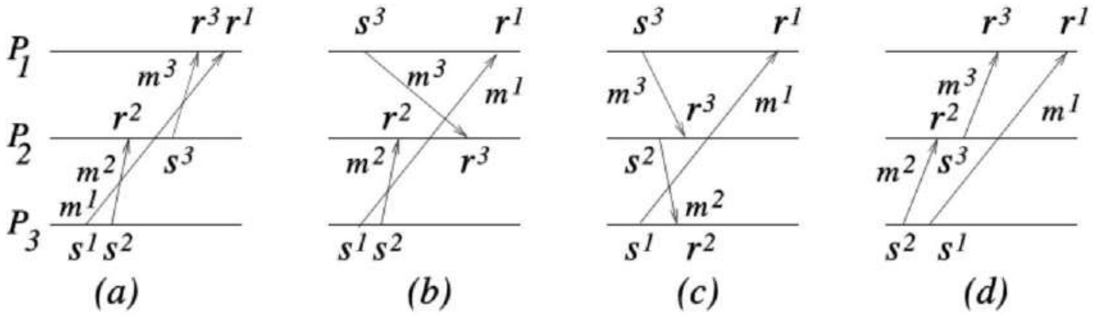


***Fig (6.2) (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.***

**Examples**

- Figure (a) shows an execution that violates CO because $s^1 < s^3$ and at the common destination $P_1$, we have $r^3 < r^1$.
- Figure (b) shows an execution that satisfies CO. Only $s^1$ and $s^2$ are related by causality but the destinations of the corresponding messages are different.
- Figure (c) shows an execution that satisfies CO. No send events are related by causality.
- Figure (d) shows an execution that satisfies CO. $s^2$ and $s^1$ are related by causality but the destinations of the corresponding messages are different. Similarly for $s^2$ and $s^3$.

**Definition: (Definition of causal order (CO) for implementations) If send($m^1$)< send($m^2$) then for each common destination d of messages $m^1$ and $m^2$, deliver$_d$($m^1$)< deliver$_{d(}$$m^2$) must be satisfied.**

**Message arrival vs. Delivery**

To implement CO, we distinguish between the arrival of a message and its delivery.

- A message m that arrives in the local OS buffer at $P_i$ may have to be delayed until the messages that were sent to $P_i$ causally before m was sent (the "overtaken" messages) have arrived and are processed by the application. The delayed message m is then given to the application for processing.
- The event of an application processing an arrived message is referred to as a *delivery* event (instead of as a *receive* event) for emphasis.
- No message overtaken by a chain of messages between the same (sender, receiver) pair. In Fig. (a), $m_1$ overtaken by chain <$m_2$, $m_3$ >
- CO degenerates to FIFO when m1, m2 sent by same process

**Listout the Uses of CO.**
Causal order is useful for applications requiring **updates to shared data, implementing distributed shared memory, and fair resource allocation** such as granting of requests for distributed mutual exclusion ,**collaborative applications, event notification systems, distributed virtual environments**

## Other Characterizations of Causal Order

**(i) Definition (Message order (MO)) A MO execution is an execution in which, for all (s,r) and (s',r') ∈ T , $s \prec s' \Rightarrow \neg(r' \prec r)$**

Example Consider any message pair, say $m^1$ and $m^3$ in Figure (a). $s^1 < s^3$ but $\neg\ r^3 < r^1$ is false. Hence, the execution does not satisfy MO.

**(ii)** Another characterization of a CO execution in terms of the partial order E < is known as **the empty-interval (EI) property.**

**Definition (Empty-interval execution) An execution E < is an empty-interval (EI) execution if for each pair of events s r ∈ *T*, the open interval set x ∈ E $s \prec x \prec r$ in the partial order is empty.**

- Example: Consider any message, say m2, in Figure (b). There does not exist any event x such that $s2 \prec x \prec r2$. This holds for all messages in the execution. Hence, the execution is EI.
- For EI <s,r> there exists some linear extension < such the corresp. interval {x ∈ E | s < x < r} is also empty. (A linear extension of a partial order E ≺ is any total order E < such that each ordering relation of the partial order is preserved.)
- An empty <s,r> interval in a linear extension implies s,r may be arbitrarily close; shown by vertical arrow in a timing diagram.
- An execution E is CO iff for each M, there exists some space-time diagram in which that message can be drawn as a vertical arrow.

**(iii) Common Past and Future**

Another characterization of CO executions is in terms of the causal past/future of a send event and its corresponding receive event.

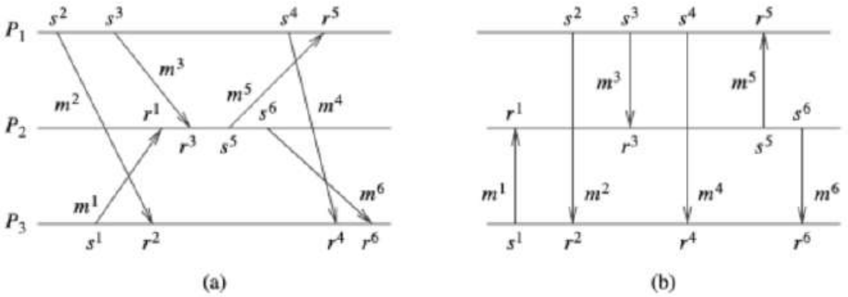
***An execution* E ≺ *is CO if and only if for each pair of events* s r ∈ *T and each event* e ∈ E,**

- ***weak common past:*** **$e \prec r = \neg\ s \prec e$ *;***
- ***weak common future:*** **$s \prec e = \neg\ e \prec r$ .**

If the past of both the s and r events are identical (and analogously for the future), viz., $e \prec r$ => $e \prec s$ and $s \prec e$ = $r \prec e$, we get a subclass of CO executions, called ***synchronous executions.***

### 2.1.4 Synchronous execution (SYNC)

**Figure 6.3** Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b) Equivalent instantaneous communication.



**Definition (Casuality in a synchronous execution) The synchronous causality relation on E is the smallest transitive relation that satisfies the following:**

S1. If $x$ occurs before $y$ at the same process, then $x \ll y$

S2. If $(s, r) \in \mathcal{T}$, then for all $x \in E$, $[(x \ll s \Longleftrightarrow x \ll r)$ and $(s \ll x \Longleftrightarrow r \ll x)]$

S3. If $x \ll y$ and $y \ll z$, then $x \ll z$

We can now formally define a synchronous execution.

**Synchronous execution (or S-execution).**

An execution $(E, \ll)$ for which the causality relation $\ll$ is a partial order.

**Timestamping a synchronous execution.**

An execution $(E, \prec)$ is synchronous iff there exists a mapping from $E$ to $T$ (scalar timestamps) |

- for any message $M$, $T(s(M)) = T(r(M))$
- for each process $P_i$, if $e_i \prec e_i'$ then $T(e_i) < T(e_i')$

### 2.2. Asynchronous Execution with Synchronous Communication

Will a program written for an asynchronous system (A-execution) run correctly if run with synchronous primitives?

| Process $i$ | Process $j$ |
|---|---|
| ... | ... |
| $Send(j)$ | $Send(i)$ |
| $Receive(j)$ | $Receive(i)$ |
| ... | ... |

## A-execution deadlocks when using synchronous primitives

**An A-execution that is realizable under synchronous communication is a realizable with synchronous communication (RSC) execution.**

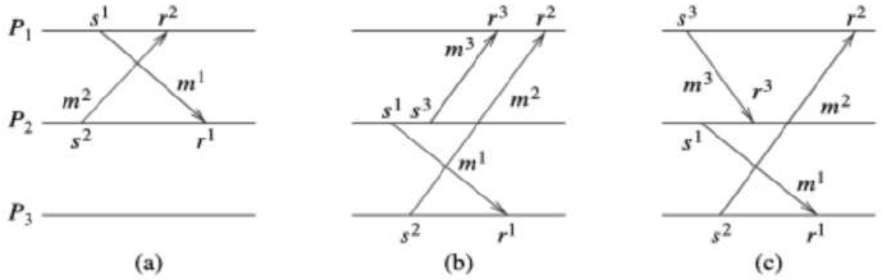


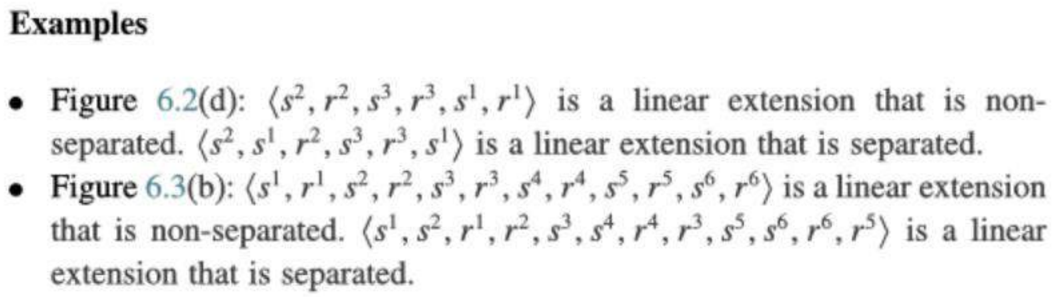
**Figure 6.5** Illustrations of asynchronous executions and of crowns. (a) Crown of size 2. (b) Another crown of size 2. (c) Crown of size 3.

## 2.2.1 RSC (Realizable with synchronous communication) Executions

Non-separated linear extension of (E, ≺)

**A linear extension of (E, ≺) such that for each pair (s,r) ∈ T , the interval { x ∈ E | s ≺ x ≺ r } is empty.**

*Exercise: Identify a non-separated and a separated linear extension in Figs 6.2(d) and 6.3(b)*



### Examples

- Figure 6.2(d): $\langle s^2, r^2, s^3, r^3, s^1, r^1\rangle$ is a linear extension that is non-separated. $\langle s^2, s^1, r^2, s^3, r^3, s^1\rangle$ is a linear extension that is separated.
- Figure 6.3(b): $\langle s^1, r^1, s^2, r^2, s^3, r^3, s^4, r^4, s^5, r^5, s^6, r^6\rangle$ is a linear extension that is non-separated. $\langle s^1, s^2, r^1, r^2, s^3, s^4, r^4, r^3, s^5, s^6, r^6, r^5\rangle$ is a linear extension that is separated.

**Defn : RSC execution An A-execution (E, ≺) is an RSC execution iff there exists a non-separated linear extension of the partial order (E, ≺).**

- Checking for all linear extensions has exponential cost!
- Practical test using the crown characterization

## Crown: Definition

Let E be an execution. A crown of size k in E is a sequence $s^i$ $r^i$ , i ∈ 0 k − 1 of pairs of corresponding send and receive events such that: $s^0 \prec r^1$, $s^1 \prec r^2$, , $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

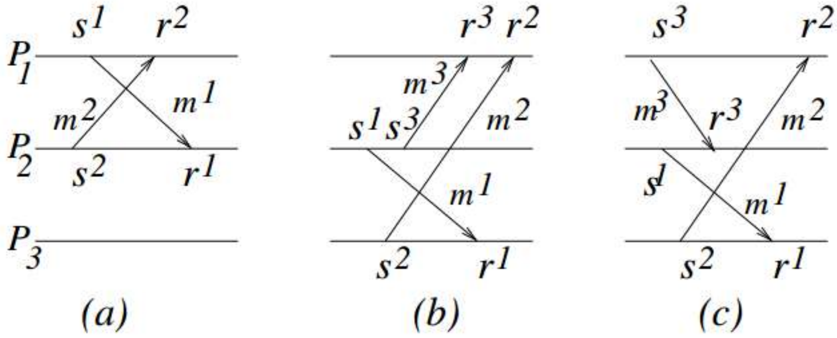


**Figure 6.5: Illustration of non-RSC A-executions and crowns. .**

(a) Crown of size 2.

(b) Another crown of size 2.

(c) Crown of size 3.

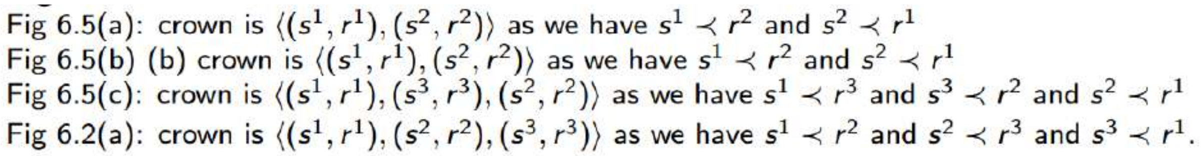
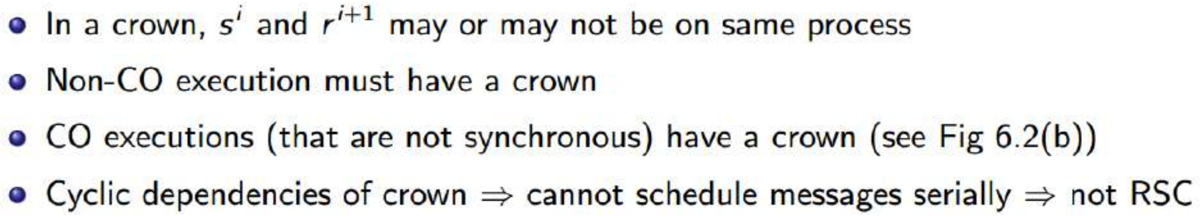
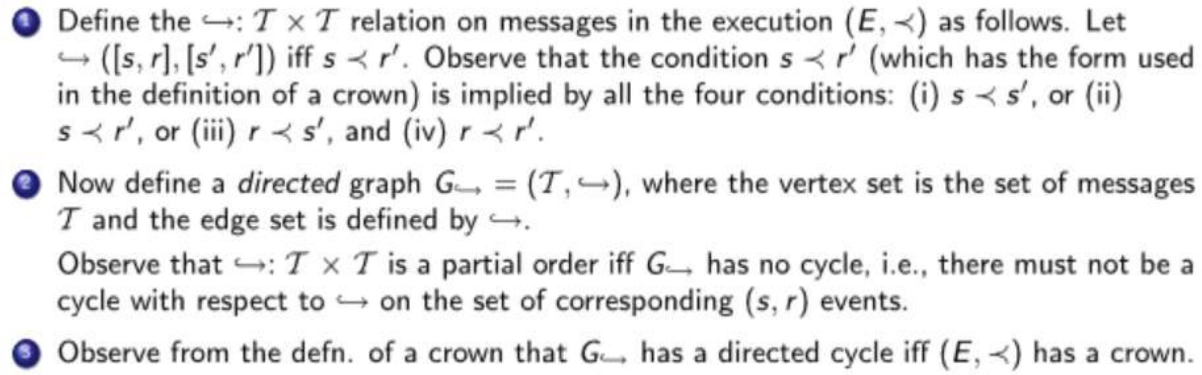


Fig 6.5(a): crown is $\langle(s^1,r^1),(s^2,r^2)\rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$
Fig 6.5(b) (b) crown is $\langle(s^1,r^1),(s^2,r^2)\rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$
Fig 6.5(c): crown is $\langle(s^1,r^1),(s^3,r^3),(s^2,r^2)\rangle$ as we have $s^1 \prec r^3$ and $s^3 \prec r^2$ and $s^2 \prec r^1$
Fig 6.2(a): crown is $\langle(s^1,r^1),(s^2,r^2),(s^3,r^3)\rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^3$ and $s^3 \prec r^1$.


- In a crown, $s^i$ and $r^{i+1}$ may or may not be on same process
- Non-CO execution must have a crown
- CO executions (that are not synchronous) have a crown (see Fig 6.2(b))
- Cyclic dependencies of crown ⇒ cannot schedule messages serially ⇒ not RSC

**Crown Test for RSC executions**


1. Define the $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ relation on messages in the execution $(E, \prec)$ as follows. Let $\hookrightarrow ([s,r],[s',r'])$ iff $s \prec r'$. Observe that the condition $s \prec r'$ (which has the form used in the definition of a crown) is implied by all the four conditions: (i) $s \prec s'$, or (ii) $s \prec r'$, or (iii) $r \prec s'$, and (iv) $r \prec r'$.
2. Now define a *directed* graph $G_{\hookrightarrow} = (\mathcal{T}, \hookrightarrow)$, where the vertex set is the set of messages $\mathcal{T}$ and the edge set is defined by $\hookrightarrow$.
   Observe that $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ is a partial order iff $G_{\hookrightarrow}$ has no cycle, i.e., there must not be a cycle with respect to $\hookrightarrow$ on the set of corresponding $(s,r)$ events.
3. Observe from the defn. of a crown that $G_{\hookrightarrow}$ has a directed cycle iff $(E, \prec)$ has a crown.

**Crown criterion**

An A-computation is RSC, i.e., it can be realized on a system with synchronous communication, iff it contains no crown.

**Crown test complexity: O(|E|) (actually, # communication events)**

**Timestamps for a RSC execution**

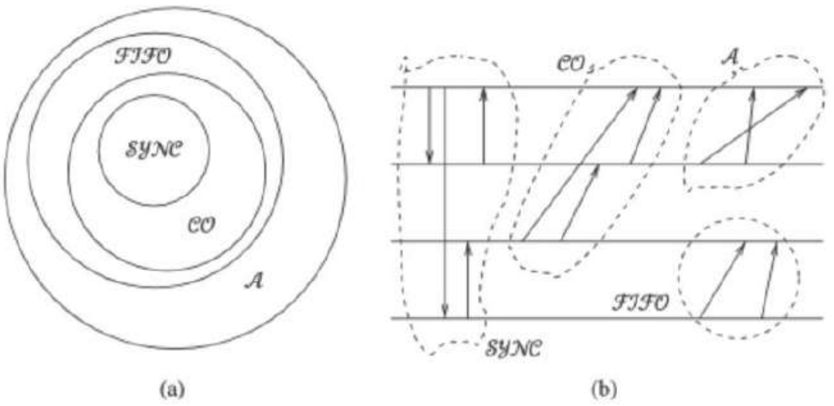
(E, ≺) is RSC iff there exists a mapping from E to T (scalar timestamps) such that

- for any message M, T(s(M)) = T(r (M))

- for each (a, b) in $(E \times E) \setminus T$ , $a \prec b \Longrightarrow T(a) < T(b)$

## 2.2.2 Hierarchy of Message Ordering Paradigms

**Figure 6.7** Hierarchy of execution classes. (a) Venn diagram. (b) Example executions.



- An A-execution is RSC iff A is an S-execution.
- RSC ⊂ CO ⊂ FIFO ⊂ A.
- More restrictions on the possible message orderings in the smaller classes. The degree of concurrency is most in A, least in SYN C.
- A program using synchronous communication easiest to develop and verify. A program using non-FIFO communication, resulting in an A-execution, hardest to design and verify.

## 2.2.3 Simulations:
## Async Programs on Sync Systems

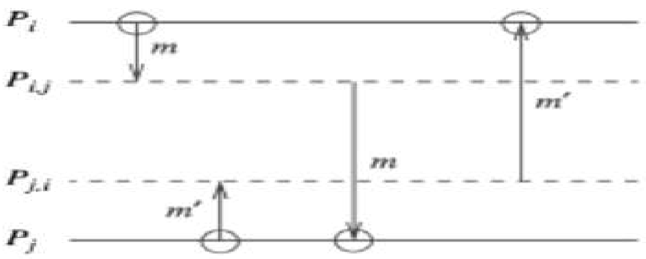
RSC execution: schedule events as per a non-separated linear extension

- adjacent (s,r) events sequentially
- partial order of original A-execution unchanged

If A-execution is not RSC:

- partial order has to be changed; or
- model each Ci,j by control process Pi,j and use sync communication (see Fig 6.8)
- Enables decoupling of sender from receiver.
- This implementation is expensive.

**Figure 6.8** Modeling channels as processes to simulate an execution using asynchronous primitives on an synchronous system.



## Simulations: Synch Programs on Async Systems

- Schedule msgs in the order in which they appear in S-program
- partial order of S-execution unchanged
- Communication on async system with async primitives

- When sync send is scheduled:
    - wait for ack before completion

## 2.3 Sync Program Order on Async Systems

**Deterministic program**: repeated runs produce same partial order

- Deterministic receive ⇒ deterministic execution ⇒ $(E, \prec)$ is fixed

**Nondeterminism** (besides due to unpredictable message delays):

- Receive call does not specify sender

Multiple sends and receives enabled at a process; can be executed in interchangeable order

Deadlock example of Fig 6.4

- If event order at a process is permuted, no deadlock!

How to schedule (nondeterministic) sync communication calls over async system?

- Match send or receive with corresponding event

**Binary rendezvous (implementation using tokens)**

- Token for each enabled interaction
- Schedule online, atomically, in a distributed manner
- Crown-free scheduling (safety); also progress to be guaranteed
- Fairness and efficiency in scheduling

### 2.3.1 Rendezvous

One form of group communication is called *multiway rendezvous*, which is a synchronous communication among an arbitrary number of asynchronous pro-cesses. All the processes involved "meet with each other," i.e., communicate "synchronously" with each other at one time. The solutions to this problem are fairly complex, and we will not consider them further as this model of syn-chronous communication is not popular. The **rendezvous between a pair of processes at a time, which is called *binary rendezvous* as opposed to the *multiway rendezvous*.**

Support for *binary rendezvous* communication was first provided by programming languages such as CSP and Ada. We consider here a subset of CSP. In these languages, the repetitive command (the ∗ operator) over the alternative command (the operator) on multiple guarded commands (each having the form $G_i \longrightarrow CL_i$) is used, as follows:

$$* [G_1 \longrightarrow CL_1 \quad G_2 \longrightarrow CL_2 \quad \cdots \quad G_k \longrightarrow CL_k]$$

Each communication command may be a part of a guard $G_i$, and may also appear within the statement block $CL_i$. A guard $G_i$ is a boolean expression. If a guard $G_i$ evaluates to true then $CL_i$ is said to be *enabled*, otherwise $CL_i$ is said to be *disabled*. A send command of local variable x to process $P_k$ is denoted as "x ! $P_k$." A receive from process $P_k$ into local variable x is denoted as "$P_k$ ? x." Some typical observations about synchronous communication under *binary rendezvous* are as follows:

- For the receive command, the sender must be specified. However, multiple recieve commands can exist. A type check on the data is implicitly performed.
- Send and received commands may be individually disabled or enabled. A command is disabled if it is guarded and the guard evaluates to *false*. The guard would likely contain an expression on some local variables.

- Synchronous communication is implemented by *scheduling* messages under the covers using asynchronous communication. Scheduling involves pairing of matching send and receive commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

The concept underlying *binary rendezvous*, which provides synchronous communication, differs from the concept underlying the classification of synchronous send and receive primitives as blocking or non-blocking. *Binary rendezvous* explicitly assumes that multiple send and receives are enabled. Any send or receive event that can be "matched" with the corresponding receive or send event can be scheduled. This is dynamically scheduling the ordering of events and the partial order of the execution.

**2.3.2 Algorithm for binary rendezvous**

These algorithms typically share the following features

- At each process, there is a set of tokens representing the current interactions that are enabled locally.
- If multiple interactions are enabled, a process chooses one of them and tries to "synchronize" with the partner process.

The problem reduces to one of scheduling messages satisfying the following constraints:

- Schedule on-line, atomically, and in a distributed manner, i.e., the schedul-ing code at any process does not know the application code of other processes.
- Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled.
- Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.
- Additional features of a good algorithm are: (i) symmetry or some form of fairness, i.e., not favoring particular processes over others during scheduling, and (ii) efficiency, i.e., using as few messages as possible, and involving as low a time overhead as possible.

We now outline a simple algorithm by Bagrodia that makes the following **assumptions:**

1. **Receive commands are forever enabled from all processes.**
2. **A send command, once enabled, remains enabled until it completes, i.e., it is not possible that a send command gets disabled (by its guard getting falsified) before the send is executed.**
3. **To prevent deadlock, process identifiers are used to introduce asymmetry to break potential crowns that arise.**
4. **Each process attempts to schedule only one send event at any time.**

The algorithm illustrates how crown-free message scheduling is achieved on-line.

**The message types used are: (i) M, (ii) *ack*(M), (iii) *request*(M), and (iv) *permission*(M).** A process blocks when it knows that it can successfully synchronize the current message with the partner process. Each process maintains a queue that is processed in FIFO order only when the process is unblocked. When a process is blocked waiting for a particular message that it is currently synchronizing, any other message that arrives is queued up.

Execution events in the synchronous execution are only the *send* of the message M and *receive* of the message M. The send and receive events for the other message types – *ack*(M), *request*(M), and *permission*(M) which are con-trol messages – are under the covers, and are not included in the synchronous execution. The messages *request*(M), *ack*(M), and *permission*(M) use M's unique tag; the message M is not included in these messages. We use cap-ital SEND(M) and RECEIVE(M) to denote the primitives in the application execution, the lower case send and receive are used for the control messages.

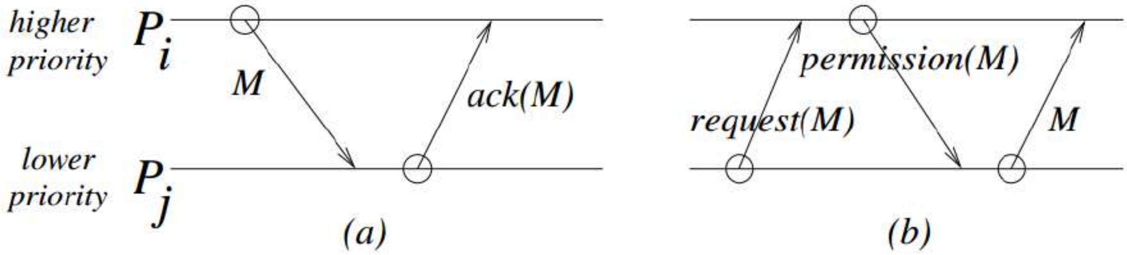
The algorithm to enforce synchronous order is given in Algorithm 6.1. The key rules to prevent cycles among the messages are summarized as follows and illustrated in Figure 6.9:

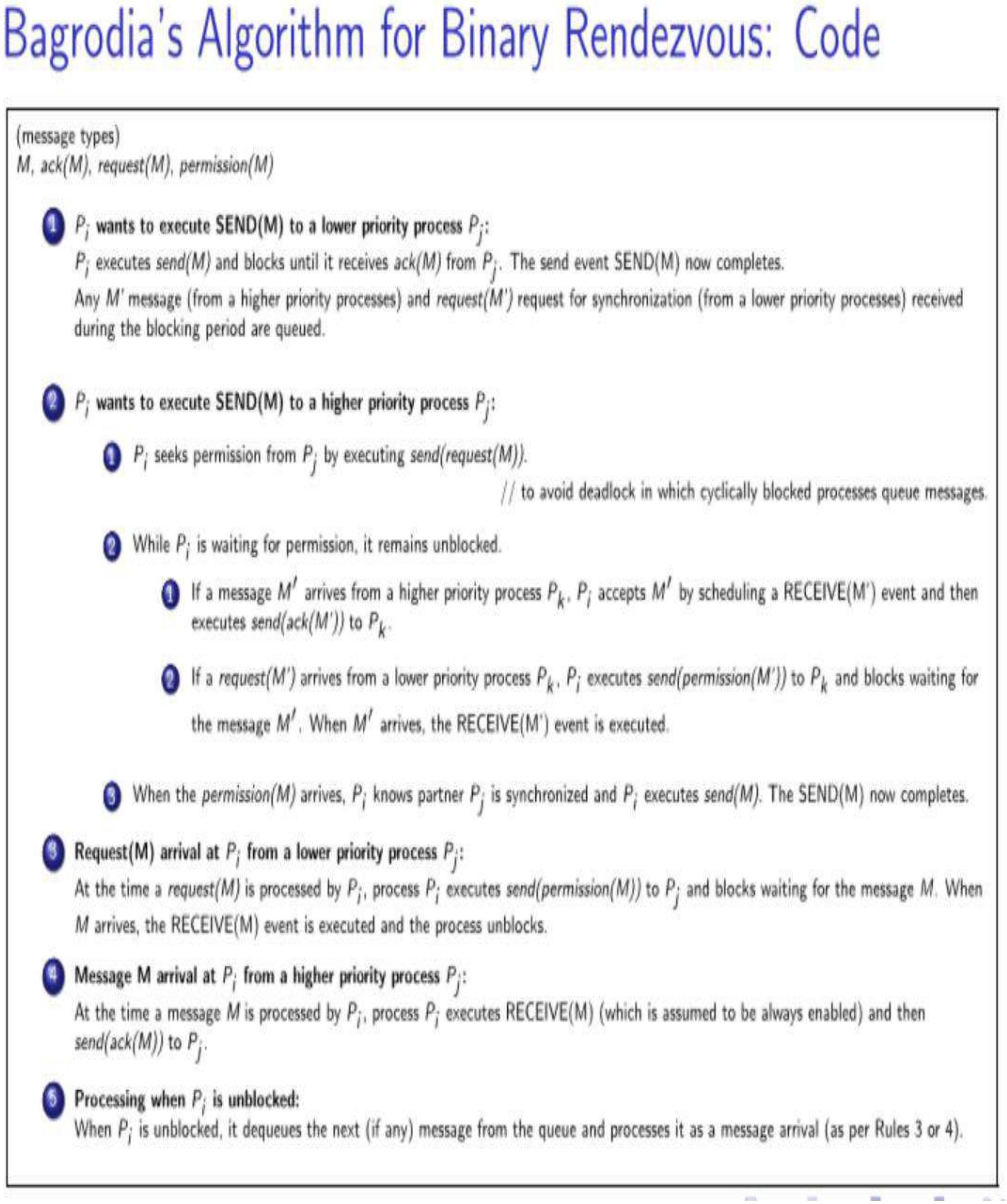
***To send to a lower priority process, messages M and ack(M) are involved in that order. The sender issues send(M) and blocks until ack(M) arrives. Thus, when sending to a lower priority process, the sender blocks waiting for the partner process to synchronize and send an acknowledgement.***

***To send to a higher priority process, messages request(M), permission(M), and M are involved, in that order. The sender issues send(request(M)), does not block, and awaits permission. When permission(M) arrives, the sender issues send(M).***

**Rules to prevent message cyles.**

**(a) High priority process blocks.** **(b) Low priority process does not block.**





# Bagrodia's Algorithm for Binary Rendezvous: Code

(message types)
*M*, *ack*(*M*), *request*(*M*), *permission*(*M*)

1. **$P_i$ wants to execute SEND(M) to a lower priority process $P_j$:**
   $P_i$ executes *send*(*M*) and blocks until it receives *ack*(*M*) from $P_j$. The send event SEND(M) now completes.
   Any $M'$ message (from a higher priority processes) and *request*($M'$) request for synchronization (from a lower priority processes) received during the blocking period are queued.

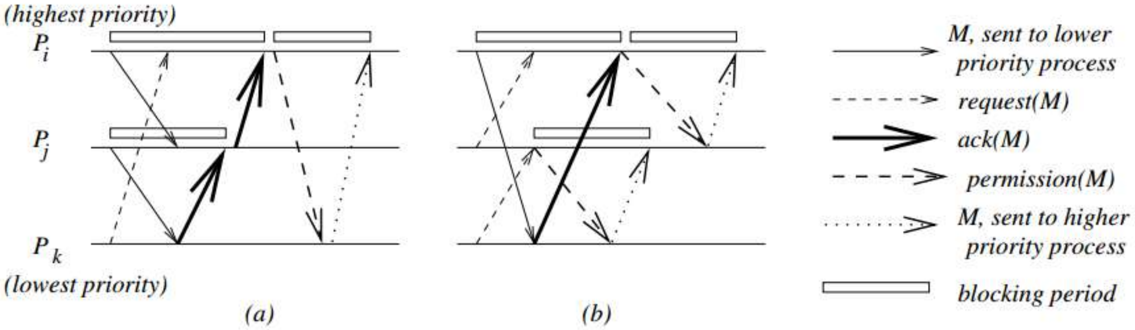
2. **$P_i$ wants to execute SEND(M) to a higher priority process $P_j$:**
   1. $P_i$ seeks permission from $P_j$ by executing *send*(*request*(*M*)).
      // to avoid deadlock in which cyclically blocked processes queue messages.
   2. While $P_i$ is waiting for permission, it remains unblocked.
      1. If a message $M'$ arrives from a higher priority process $P_k$, $P_i$ accepts $M'$ by scheduling a RECEIVE(M') event and then executes *send*(*ack*($M'$)) to $P_k$.
      2. If a *request*($M'$) arrives from a lower priority process $P_k$, $P_i$ executes *send*(*permission*($M'$)) to $P_k$ and blocks waiting for the message $M'$. When $M'$ arrives, the RECEIVE(M') event is executed.
   3. When the *permission*(*M*) arrives, $P_i$ knows partner $P_j$ is synchronized and $P_i$ executes *send*(*M*). The SEND(M) now completes.

3. **Request(M) arrival at $P_i$ from a lower priority process $P_j$:**
   At the time a *request*(*M*) is processed by $P_i$, process $P_i$ executes *send*(*permission*(*M*)) to $P_j$ and blocks waiting for the message *M*. When *M* arrives, the RECEIVE(M) event is executed and the process unblocks.

4. **Message M arrival at $P_i$ from a higher priority process $P_j$:**
   At the time a message *M* is processed by $P_i$, process $P_i$ executes RECEIVE(M) (which is assumed to be always enabled) and then *send*(*ack*(*M*)) to $P_j$.

5. **Processing when $P_i$ is unblocked:**
   When $P_i$ is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per Rules 3 or 4).

**Figure:6.10 Scheduling messages with sync communication.**



Higher prio $P_i$ blocks on lower prio $P_j$ to avoid cyclic wait (whether or not it is the intended sender or receiver of msg being scheduled)

- Before sending M to $P_i$ , $P_j$ requests permission in a nonblocking manner.

1. If a message M from a higher priority process arrives, it is processed by a receive (assuming receives are always enabled) and *ack*(M ) is returned. Thus, a cyclic wait is prevented.
2. Also, while waiting for this permission, if a *request*(M ) from a lower priority process arrives, a *permission*(M ) is returned and the process blocks until M actually arrives.

- Note: receive($M^0$) gets permuted with the send(M) event

## 6.4 Group communication

A *message broadcast* is the sending of a message to all members in the distributed system. The notion of a system can be confined only to those sites/processes participating in the joint application. Refining the notion of *broadcasting*, there is *multicasting* wherein a message is sent to a certain subset, identified as a *group*, of the processes in the system. At the other extreme is *unicasting*, which is the familiar point-to-point message communication.

Broadcast and multicast support can be provided by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information. However, the hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features such as the following:

- Application-specific ordering semantics on the order of delivery of messages.
- Adapting groups to dynamically changing membership.
- Sending multicasts to an arbitrary set of processes at each send event.
- Providing various fault-tolerance semantics.

If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a *closed group* algorithm. If the sender of the multicast can be outside the destination group, the multicast algorithm is said to be an *open group* algorithm. Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms. Closed group algorithms cannot be used in several scenarios such as in a large system (e.g., on-line reservation or Internet banking systems) where client processes are short-lived and in large numbers. It is also worth noting that, for multicast algorithms, the number of groups may be potentially exponential, i.e., O($2^n$), and algorithms that have to explicitly track the groups can incur this high overhead.

**Two popular orders for the delivery of messages were proposed in the context of group communication: *causal order* and *total order*.**

## 6.5 Causal order (CO)

Causal order has many applications such as updating replicated data, allo-cating requests in a fair manner, and synchronizing multimedia streams.

**The use of causal order in updating replicas of a data item in the system**.

Consider Figure 6.11(a), which shows two processes $P_1$ and $P_2$ that issue updates to the three replicas R1 d , R2 d , and R3 d of data item d. Message m creates a causality between send m1 and send m2 . If $P_2$ issues its update causally after $P_1$ issued its update, then $P_2$'s update should be seen by the replicas after they see $P_1$'s update, in order to preserve the semantics of the application. (In this case, CO is satisfied.) However, this may happen at some, all, or none of the replicas. Figure 6.11(b) shows that R1 sees $P_2$'s update first, while R2 and R3 see $P_1$'s update first. Here, CO is violated. Figure 6.11(c) shows that all replicas see $P_2$'s update first. However, CO is still violated. If message m did not exist as shown, then the executions shown in Figure 6.11(b) and (c) would satisfy CO.

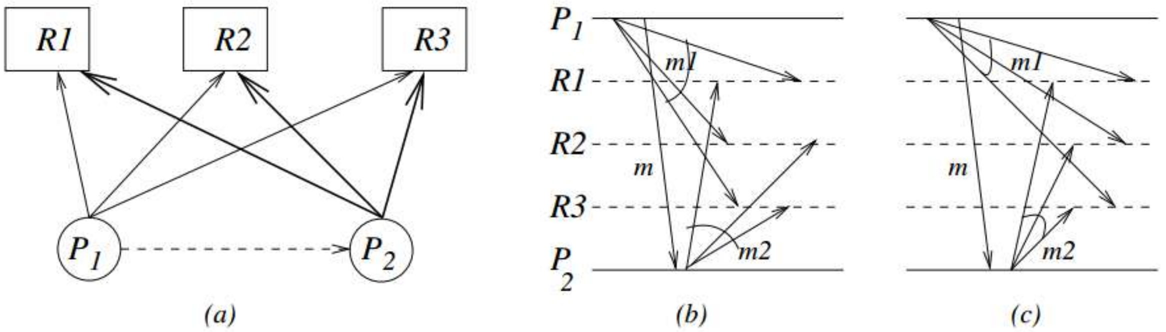


Figure 6.11: (a) Updates to 3 replicas. (b) Causal order (CO) and total order violated. (c) Causal order violated.

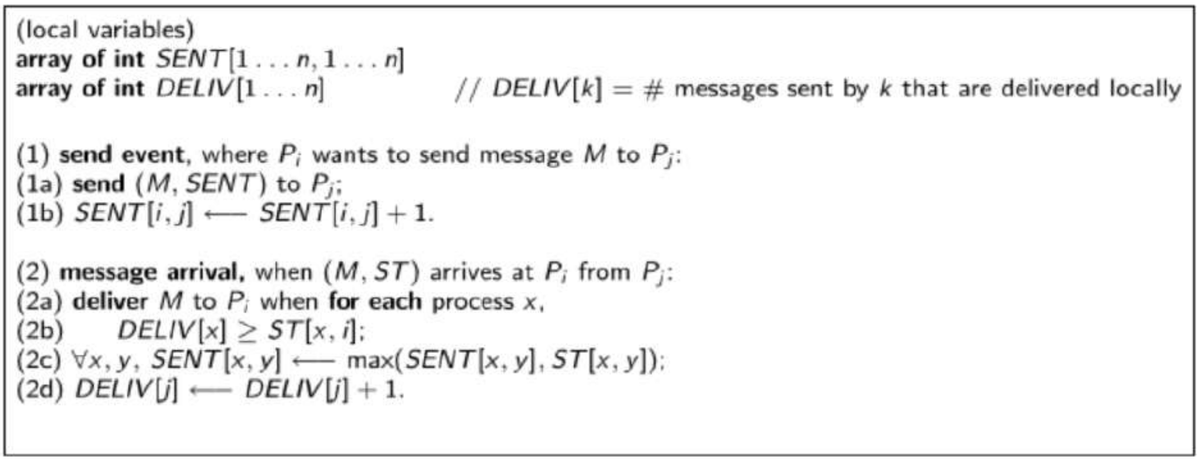
**The following two criteria must be met by a causal ordering protocol:**

- **Safety** In order to prevent causal order from being violated, a message M that arrives at a process may need to be buffered until all system wide messages sent in the causal past of the send M event to that same destination have already arrived.

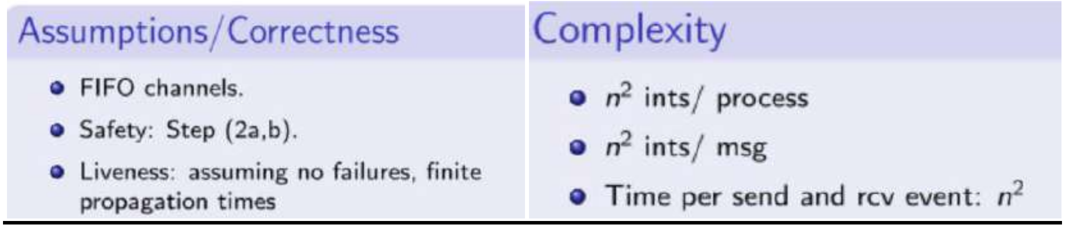
  Therefore, we distinguish between the arrival of a message at a process (at which time it is placed in a local system buffer) and the event at which the message is given to the application process (when the protocol deems it safe to do so without violating causal order). The arrival of a message is transparent to the application process. The delivery event corresponds to the *receive* event in the execution model.

- **Liveness** A message that arrives at a process must eventually be delivered to the process.

### 2.5.1 The Raynal–Schiper–Toueg algorithm (RST)



```
(local variables)
array of int $SENT[1 \ldots n, 1 \ldots n]$
array of int $DELIV[1 \ldots n]$        // $DELIV[k]$ = # messages sent by $k$ that are delivered locally

(1) send event, where $P_i$ wants to send message $M$ to $P_j$:
(1a) send $(M, SENT)$ to $P_j$;
(1b) $SENT[i, j] \longleftarrow SENT[i, j] + 1$.

(2) message arrival, when $(M, ST)$ arrives at $P_i$ from $P_j$:
(2a) deliver $M$ to $P_i$ when for each process $x$,
(2b)      $DELIV[x] \geq ST[x, i]$;
(2c) $\forall x, y$, $SENT[x, y] \longleftarrow \max(SENT[x, y], ST[x, y])$;
(2d) $DELIV[j] \longleftarrow DELIV[j] + 1$.
```



Assumptions/Correctness

- FIFO channels.
- Safety: Step (2a,b).
- Liveness: assuming no failures, finite propagation times

Complexity

- $n^2$ ints/ process
- $n^2$ ints/ msg
- Time per send and rcv event: $n^2$

### 2.5.2 Optimal KS Algorithm for CO: Principles

**Delivery Condition for correctness:**

Msg M that carries information "d ∈M.Dests", where message M was sent to d in the causal past of Send(M*), is not delivered to d if M has not yet been delivered to d .

**Necessary and Sufficient Conditions for Optimality:**

An optimal CO algorithm stores in local message logs and propagates on messages, information of the form "d is a destination of M" about a message sent in the causal past, *as long as* and *only as long as*:

***(Propagation Constraint I)*** it is not known that the message M is delivered to d, and

***(Propagation Constraint II)*** it is not known that a message has been sent to d in the causal future of Send M , and hence it is not guaranteed using a reasoning based on transitivity that the message M will be delivered to d in CO.

**The Propagation Constraints also imply that if either (I) or (II) is false, the information "d ∈ M Dests" must *not* be stored or propagated, even to remember that (I) or (II) has been falsified.**

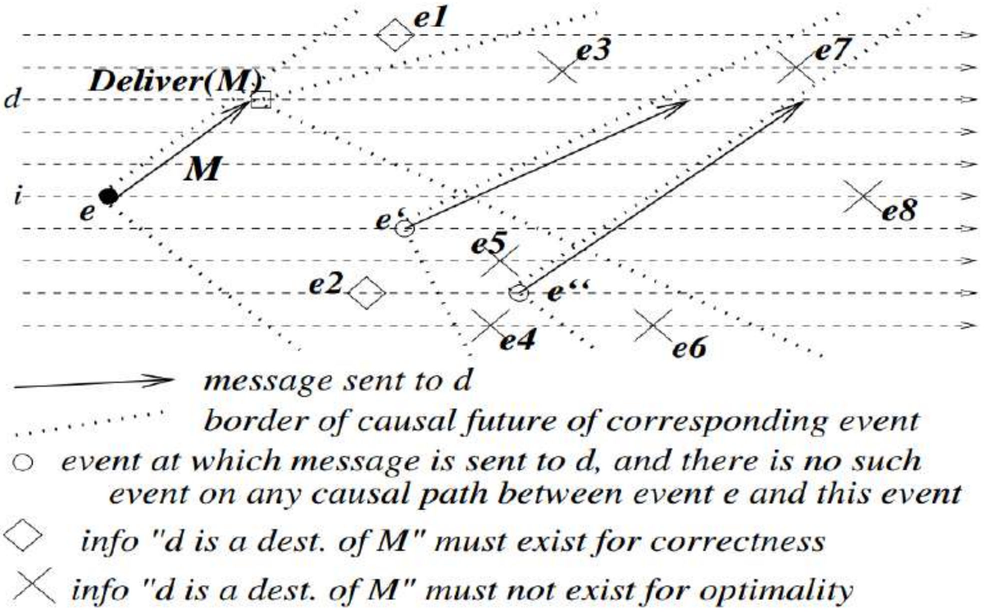
Stated differently, the information “d ∈ $M_{i,a}$ Dests” must be available in the causal future of event $e_{i\,a}$, but:

- not in the causal future of $\text{Deliver}_d$ $M_{i\,a}$ , and

- not in the causal future of $e_{k\,c}$, where d ∈ $M_{k,\,c}$ Dests and there is no other message sent causally between $M_{i,a}$ and $M_{k,\,c}$ to the same destination d.

In the causal future of $\text{Deliver}_d$ ($M_{i,a}$) , and Send($M_{k,c}$), the information is redundant; elsewhere, it is necessary. Additionally, to maintain optimality, no other information should be stored, including information about what messages have been delivered.

As information about what messages have been delivered (or are guaranteed to be delivered without violating causal order) is necessary for the Delivery Condition, this information is inferred using a set-operation based logic.

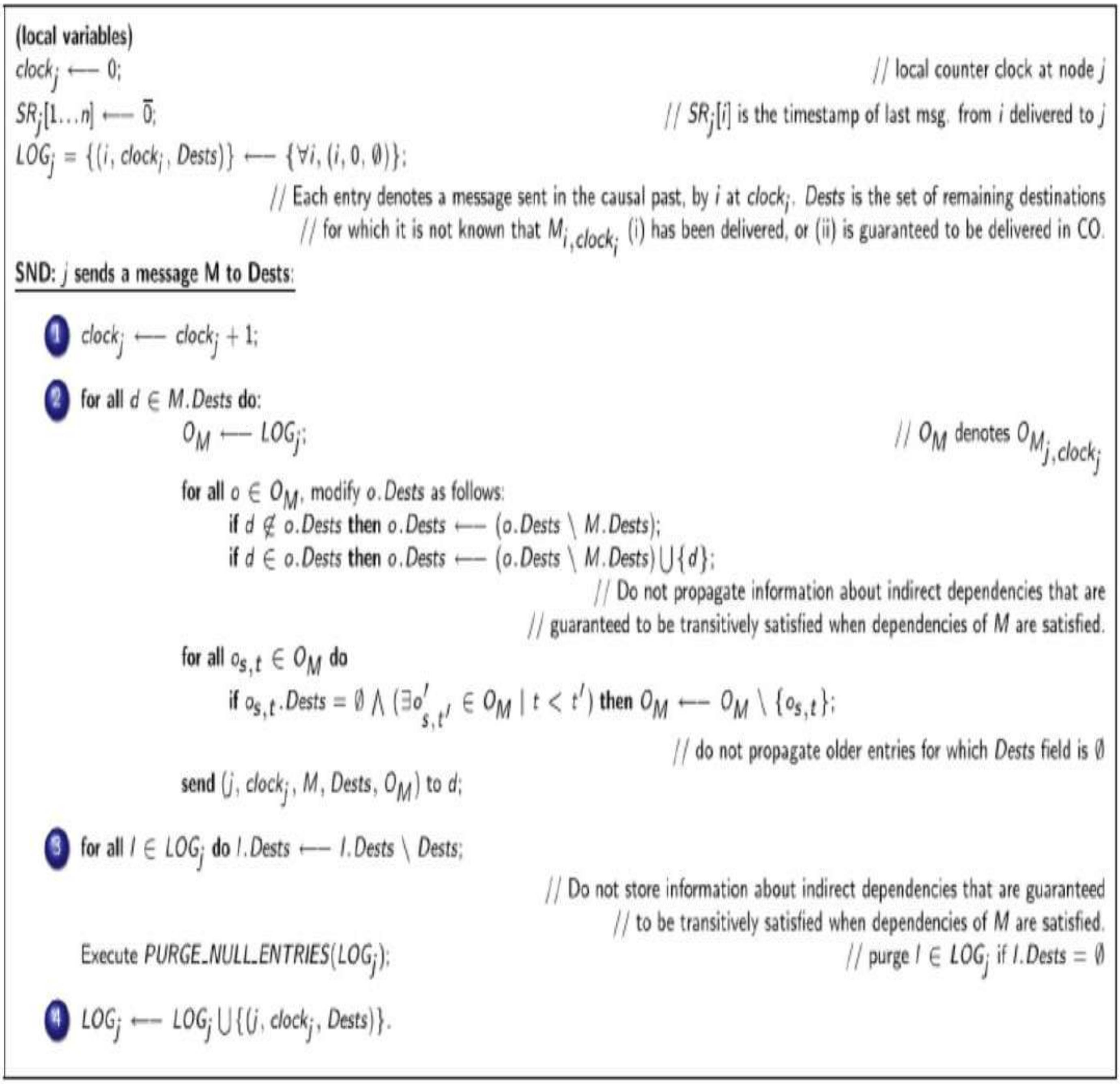

e1
e3
e7
Deliver(M)
d
M
i
e
e'
e8
e5
e2
e''
e4
e6
message sent to d
border of causal future of corresponding event
event at which message is sent to d, and there is no such event on any causal path between event e and this event
info "d is a dest. of M" must exist for correctness
info "d is a dest. of M" must not exist for optimality

The message M is sent by process i at event e to process d. The information **“d ∈ M Dests”:**

- **must exist at e1 and e2 because (I) and (II) are true;**
- **must not exist at e3 because (I) is false;**
- **must not exist at e4 e5 e6 because (II) is false;**
- **must not exist at e7 e8 because (I) and (II) are false.**

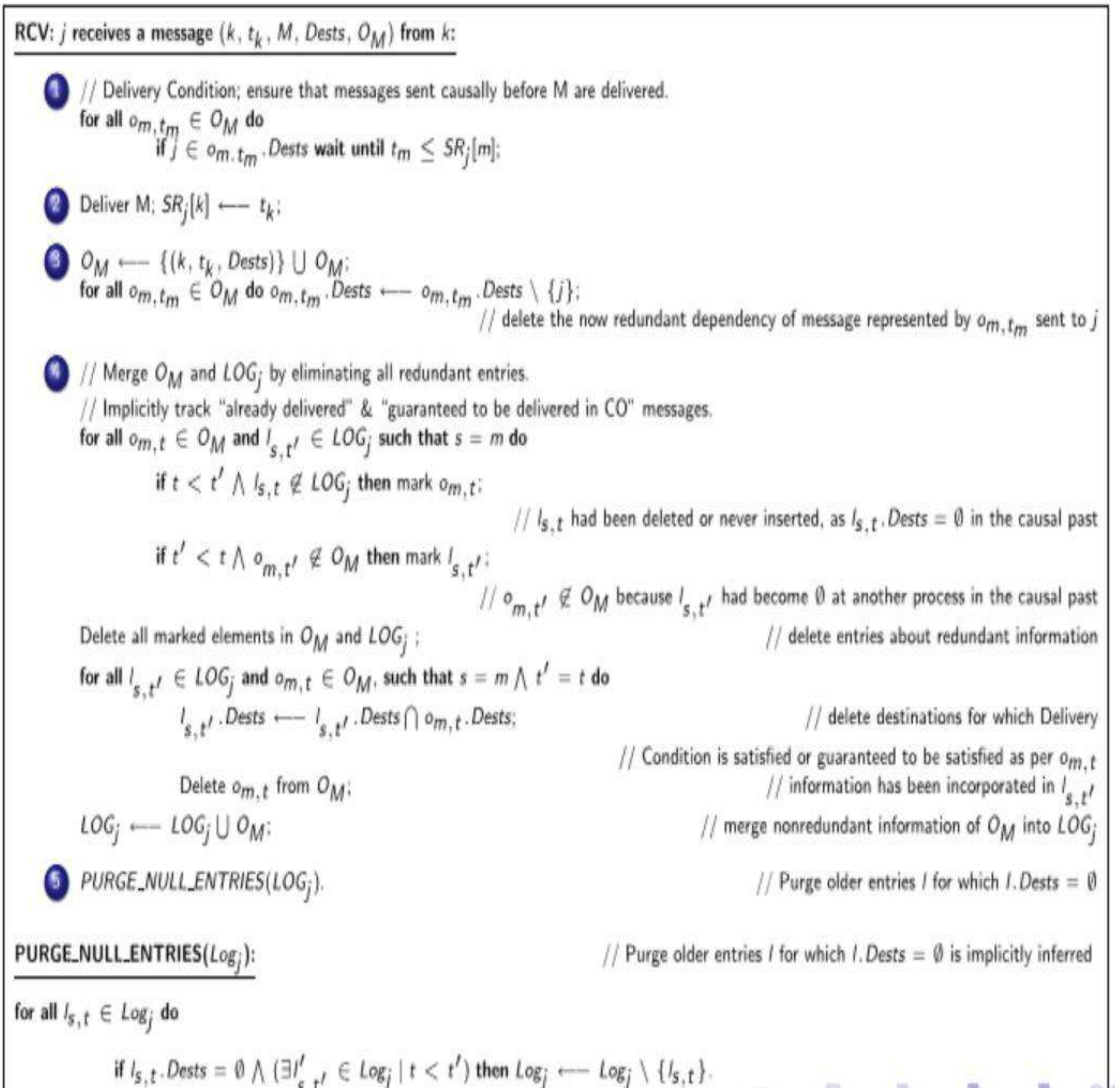
- Information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is ***explicitly* tracked** by the algorithm using (*source, timestamp, destination*) information.
- The information must be deleted as soon as either (i) or (ii) becomes false. The key problem in designing an optimal CO algorithm is to identify the events at which (i) or (ii) becomes false.
- Information about messages already delivered and messages guaranteed to be delivered in CO is ***implicitly* tracked** without storing or propagating it, and is derived from the explicit information.
- Such implicit information is used for determining when (i) or (ii) becomes false for the explicit information being stored or carried in messages.

## Optimal KS Algorithm for CO: Code (1)



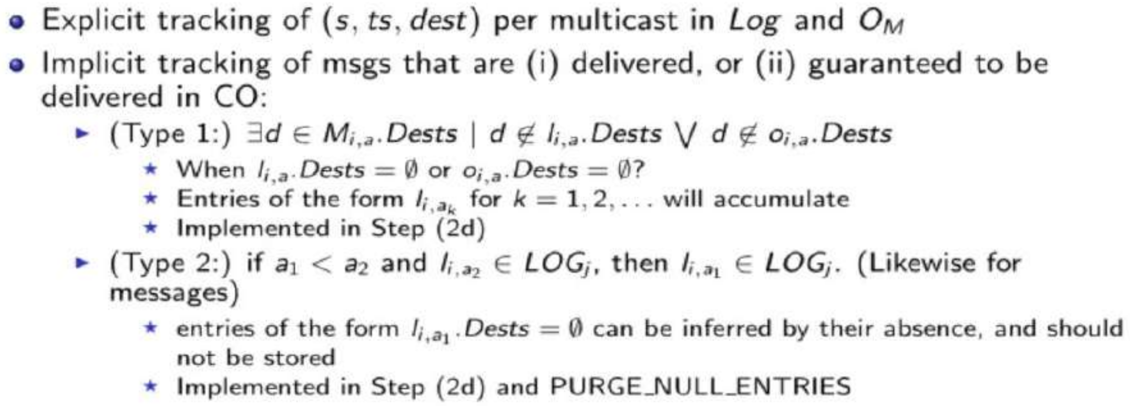
```
(local variables)
$clock_j \longleftarrow 0$;                                   // local counter clock at node $j$
$SR_j[1 \ldots n] \longleftarrow \overline{0}$;               // $SR_j[i]$ is the timestamp of last msg. from $i$ delivered to $j$
$LOG_j = \{(i, clock_i, Dests)\} \longleftarrow \{\forall i, (i, 0, \emptyset)\}$;
    // Each entry denotes a message sent in the causal past, by $i$ at $clock_i$. $Dests$ is the set of remaining destinations
    // for which it is not known that $M_{i,clock_i}$ (i) has been delivered, or (ii) is guaranteed to be delivered in CO.

SND: $j$ sends a message M to Dests:

1  $clock_j \longleftarrow clock_j + 1$;

2  for all $d \in M.Dests$ do:
        $O_M \longleftarrow LOG_j$;                           // $O_M$ denotes $O_{M_{j,clock_j}}$
        for all $o \in O_M$, modify $o.Dests$ as follows:
            if $d \notin o.Dests$ then $o.Dests \longleftarrow (o.Dests \setminus M.Dests)$;
            if $d \in o.Dests$ then $o.Dests \longleftarrow (o.Dests \setminus M.Dests) \bigcup \{d\}$;
                // Do not propagate information about indirect dependencies that are
                // guaranteed to be transitively satisfied when dependencies of $M$ are satisfied.
        for all $o_{s,t} \in O_M$ do
            if $o_{s,t}.Dests = \emptyset \bigwedge (\exists o'_{s,t'} \in O_M \mid t < t')$ then $O_M \longleftarrow O_M \setminus \{o_{s,t}\}$;
                // do not propagate older entries for which $Dests$ field is $\emptyset$
        send $(j, clock_j, M, Dests, O_M)$ to $d$;

3  for all $l \in LOG_j$ do $l.Dests \longleftarrow l.Dests \setminus Dests$;
        // Do not store information about indirect dependencies that are guaranteed
        // to be transitively satisfied when dependencies of $M$ are satisfied.
   Execute $PURGE\_NULL\_ENTRIES(LOG_j)$;                    // purge $l \in LOG_j$ if $l.Dests = \emptyset$

4  $LOG_j \longleftarrow LOG_j \bigcup \{(j, clock_j, Dests)\}$.
```

## Optimal KS Algorithm for CO: Code (2)



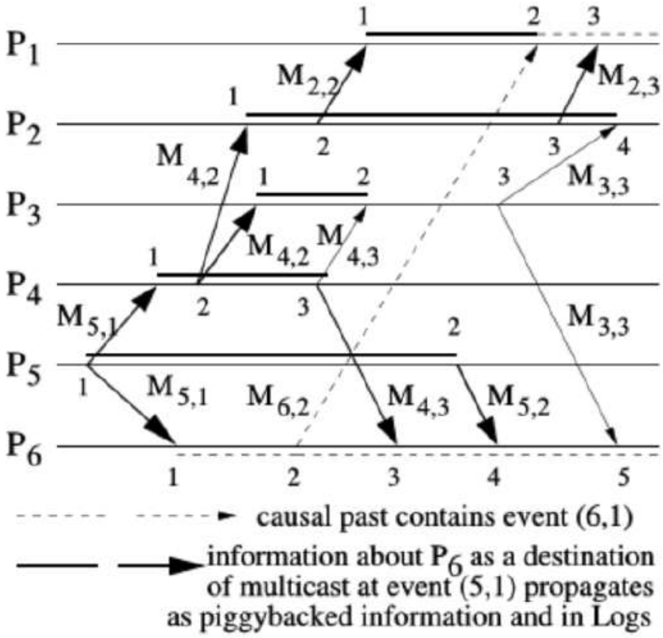
```
RCV: j receives a message $(k, t_k, M, Dests, O_M)$ from k:

(1) // Delivery Condition; ensure that messages sent causally before M are delivered.
    for all $o_{m,t_m} \in O_M$ do
        if $j \in o_{m,t_m}.Dests$ wait until $t_m \leq SR_j[m]$;

(2) Deliver M; $SR_j[k] \longleftarrow t_k$;

(3) $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
    for all $o_{m,t_m} \in O_M$ do $o_{m,t_m}.Dests \longleftarrow o_{m,t_m}.Dests \setminus \{j\}$;
                      // delete the now redundant dependency of message represented by $o_{m,t_m}$ sent to j

(4) // Merge $O_M$ and $LOG_j$ by eliminating all redundant entries.
    // Implicitly track "already delivered" & "guaranteed to be delivered in CO" messages.
    for all $o_{m,t} \in O_M$ and $l_{s,t'} \in LOG_j$ such that $s = m$ do
        if $t < t' \bigwedge l_{s,t} \not\in LOG_j$ then mark $o_{m,t}$;
                      // $l_{s,t}$ had been deleted or never inserted, as $l_{s,t}.Dests = \emptyset$ in the causal past
        if $t' < t \bigwedge o_{m,t'} \not\in O_M$ then mark $l_{s,t'}$;
                      // $o_{m,t'} \not\in O_M$ because $l_{s,t'}$ had become $\emptyset$ at another process in the causal past
    Delete all marked elements in $O_M$ and $LOG_j$ ;          // delete entries about redundant information
    for all $l_{s,t'} \in LOG_j$ and $o_{m,t} \in O_M$, such that $s = m \bigwedge t' = t$ do
        $l_{s,t'}.Dests \longleftarrow l_{s,t'}.Dests \bigcap o_{m,t}.Dests$;     // delete destinations for which Delivery
                      // Condition is satisfied or guaranteed to be satisfied as per $o_{m,t}$
        Delete $o_{m,t}$ from $O_M$;          // information has been incorporated in $l_{s,t'}$
    $LOG_j \longleftarrow LOG_j \bigcup O_M$;          // merge nonredundant information of $O_M$ into $LOG_j$

(5) $PURGE\_NULL\_ENTRIES(LOG_j)$.          // Purge older entries $l$ for which $l.Dests = \emptyset$

PURGE_NULL_ENTRIES($Log_j$):          // Purge older entries $l$ for which $l.Dests = \emptyset$ is implicitly inferred

for all $l_{s,t} \in Log_j$ do
    if $l_{s,t}.Dests = \emptyset \bigwedge (\exists l'_{s,t'} \in Log_j \mid t < t')$ then $Log_j \longleftarrow Log_j \setminus \{l_{s,t}\}$.
```

## Information Pruning



- Explicit tracking of $(s, ts, dest)$ per multicast in $Log$ and $O_M$
- Implicit tracking of msgs that are (i) delivered, or (ii) guaranteed to be delivered in CO:
  - (Type 1:) $\exists d \in M_{i,a}.Dests \mid d \not\in l_{i,a}.Dests \bigvee d \not\in o_{i,a}.Dests$
    - When $l_{i,a}.Dests = \emptyset$ or $o_{i,a}.Dests = \emptyset$?
    - Entries of the form $l_{i,a_k}$ for $k = 1, 2, \ldots$ will accumulate
    - Implemented in Step (2d)
  - (Type 2:) if $a_1 < a_2$ and $l_{i,a_2} \in LOG_j$, then $l_{i,a_1} \in LOG_j$. (Likewise for messages)
    - entries of the form $l_{i,a_1}.Dests = \emptyset$ can be inferred by their absence, and should not be stored
    - Implemented in Step (2d) and PURGE_NULL_ENTRIES

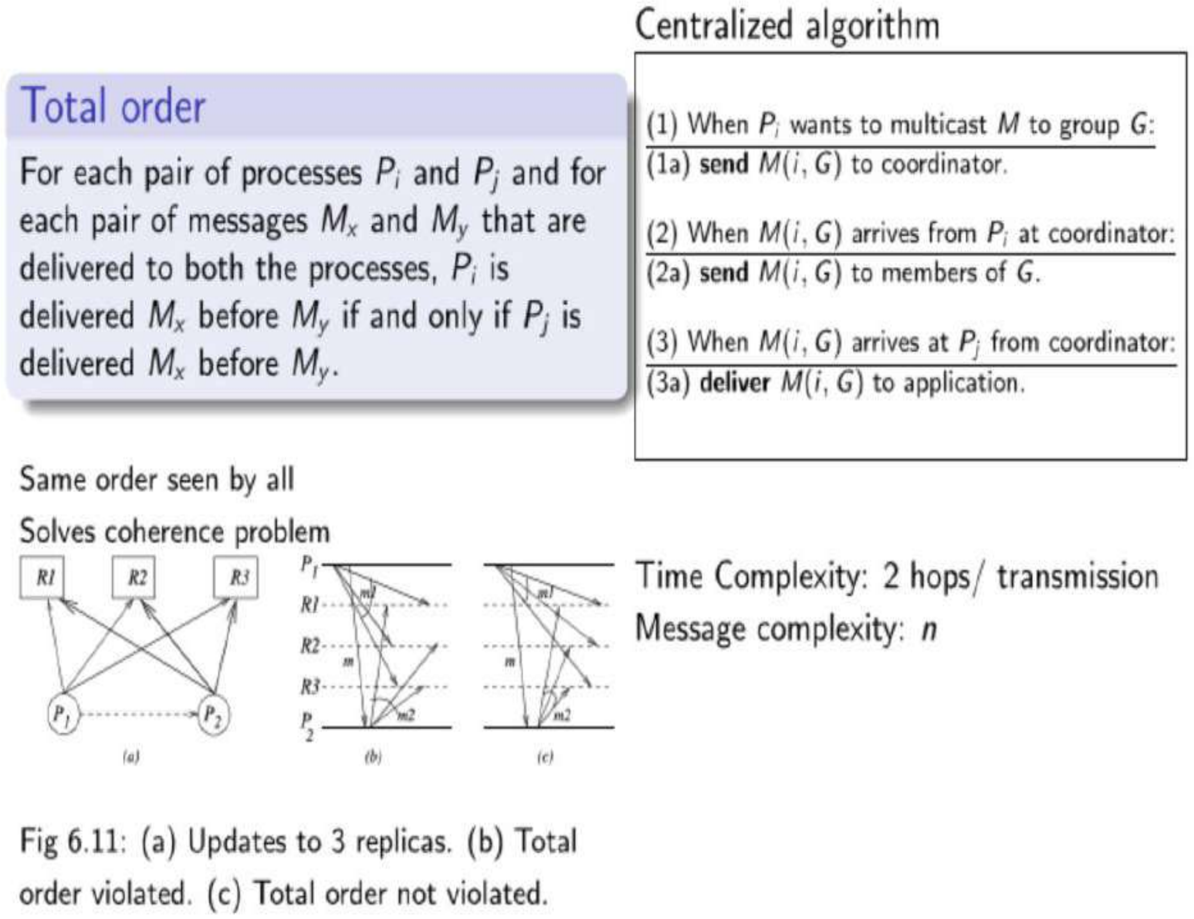
**Example**


P1
P2
P3
P4
P5
P6
M2,2
M2,3
M4,2
M3,3
M4,2
M4,3
M5,1
M3,3
M5,1
M6,2
M4,3
M5,2
1
2
3
1
2
3
4
1
2
3
1
2
3
2
1
1
2
3
4
5
causal past contains event (6,1)
information about P6 as a destination
of multicast at event (5,1) propagates
as piggybacked information and in Logs

| Message to dest. | piggybacked $M_{5,1}$.Dests |
|---|---|
| $M_{5,1}$ to $P_4$,$P_6$ | {$P_4$,$P_6$} |
| $M_{4,2}$ to $P_3$,$P_2$ | {$P_6$} |
| $M_{2,2}$ to $P_1$ | {$P_6$} |
| $M_{6,2}$ to $P_1$ | {$P_4$} |
| $M_{4,3}$ to $P_6$ | {$P_6$} |
| $M_{4,3}$ to $P_3$ | { } |
| $M_{5,2}$ to $P_6$ | {$P_4$,$P_6$} |
| $M_{2,3}$ to $P_1$ | {$P_6$} |
| $M_{3,3}$ to $P_2$,$P_6$ | { } |

Figure 6.13: Tracking of information about $M_{5,1}.Dests$

## 2.6 Total Message Order



### Total order

For each pair of processes $P_i$ and $P_j$ and for each pair of messages $M_x$ and $M_y$ that are delivered to both the processes, $P_i$ is delivered $M_x$ before $M_y$ if and only if $P_j$ is delivered $M_x$ before $M_y$.

### Centralized algorithm

```
(1) When $P_i$ wants to multicast $M$ to group $G$:
(1a) send $M(i, G)$ to coordinator.

(2) When $M(i, G)$ arrives from $P_i$ at coordinator:
(2a) send $M(i, G)$ to members of $G$.

(3) When $M(i, G)$ arrives at $P_j$ from coordinator:
(3a) deliver $M(i, G)$ to application.
```

Same order seen by all

Solves coherence problem

Time Complexity: 2 hops/ transmission

Message complexity: $n$

Fig 6.11: (a) Updates to 3 replicas. (b) Total order violated. (c) Total order not violated.

### 6.6.2 Three-phase Algorithm

**A distributed algorithm to implement total order and causal order of messages**

The three phases of the algorithm are first described from the viewpoint of the sender, and then from the viewpoint of the receiver.

**Sender**

**Phase 1** In the first phase, a process multicasts (line 1b) the message M with a locally unique tag and the local timestamp to the group members.

**Phase 2** In the second phase, the sender process awaits a reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M . The await call in line 1d is non-blocking, i.e., any other messages received in the meanwhile are processed. Once all expected replies are received, the process computes the maximum of the proposed timestamps for M , and uses the maximum as the final timestamp.

**Phase 3** In the third phase, the process multicasts the final timestamp to the group in line (1f).

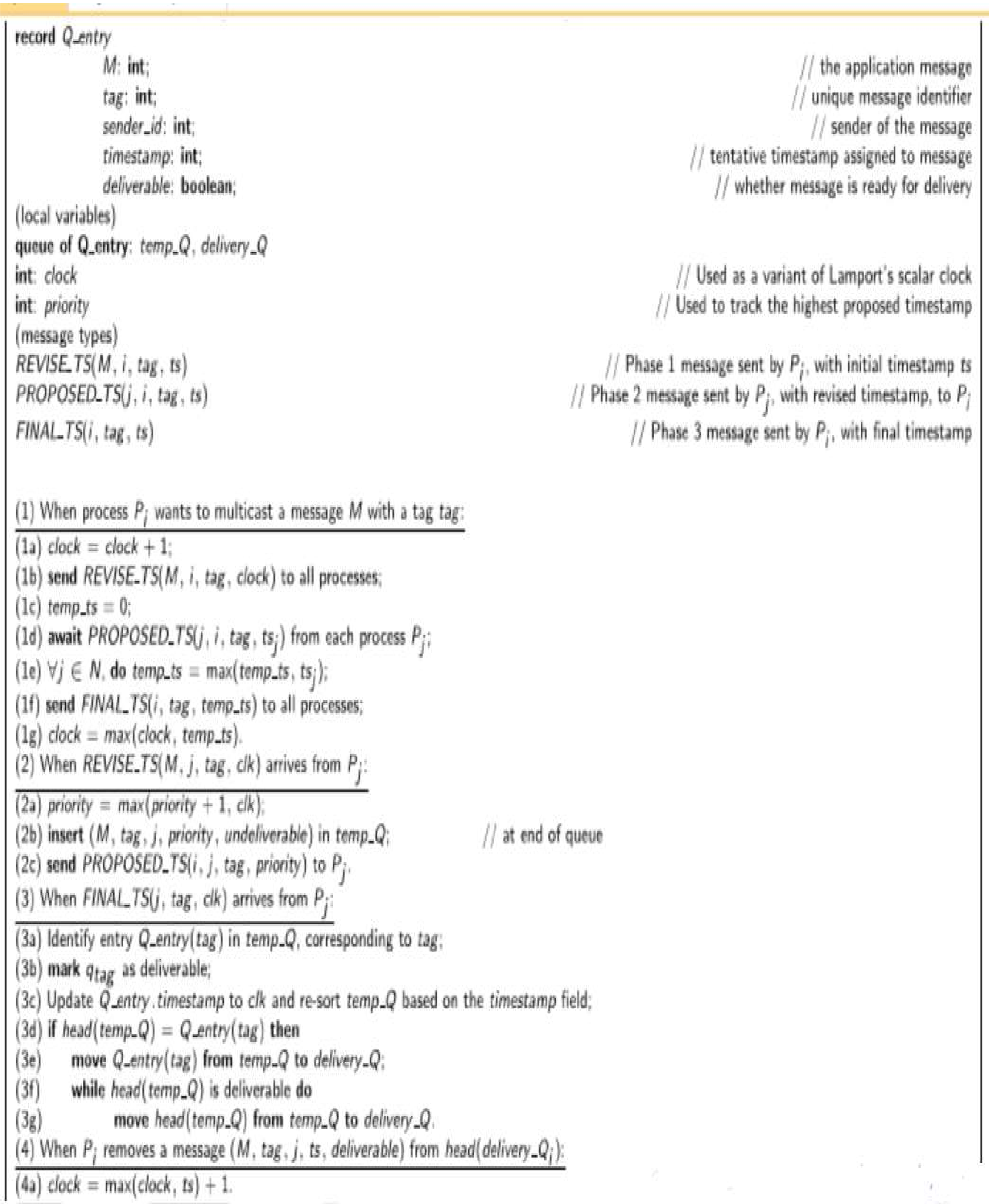
**Receivers**

**Phase 1** In the first phase, the receiver receives the message with a tentative/proposed timestamp. It updates the variable priority that tracks the highest proposed timestamp (line 2a), then revises the proposed timestamp to the priority, and places the message with its tag and the revised timestamp at the tail of the queue temp_Q (line 2b). In the queue, the entry is marked as undeliverable.

**Phase 2** In the second phase, the receiver sends the revised timestamp (and the tag) back to the sender (line 2c). The receiver then waits in a non-blocking manner for the final timestamp (correlated by the message tag).

**Phase 3** In the third phase, the final timestamp is received from the multicaster (line 3). The corresponding message entry in temp_Q is identified using the tag (line 3a), and is marked as deliverable (line 3b) after the revised timestamp is overwritten by the final timestamp (line 3c). The queue is then resorted using the timestamp field of the entries as the key (line 3c). As the queue is already sorted except for the modified entry for the message under consideration, that message entry has to be placed in its sorted position in the queue. If the message entry is at the head of the temp_Q, that entry, and all consecutive subsequent entries that are also marked as deliverable, are dequeued from temp_Q, and enqueued in deliver_Q in that order (the loop in lines 3d–3g).

## Three-phase Algorithm Code



```
record Q_entry
        M: int;                                   // the application message
        tag: int;                                 // unique message identifier
        sender_id: int;                           // sender of the message
        timestamp: int;                           // tentative timestamp assigned to message
        deliverable: boolean;                     // whether message is ready for delivery
(local variables)
queue of Q_entry: temp_Q, delivery_Q
int: clock                                        // Used as a variant of Lamport's scalar clock
int: priority                                     // Used to track the highest proposed timestamp
(message types)
REVISE_TS(M, i, tag, ts)                          // Phase 1 message sent by $P_i$, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts)                        // Phase 2 message sent by $P_j$, with revised timestamp, to $P_i$
FINAL_TS(i, tag, ts)                              // Phase 3 message sent by $P_i$, with final timestamp


(1) When process $P_i$ wants to multicast a message M with a tag tag:
(1a) clock = clock + 1;
(1b) send REVISE_TS(M, i, tag, clock) to all processes;
(1c) temp_ts = 0;
(1d) await PROPOSED_TS(j, i, tag, $ts_j$) from each process $P_j$;
(1e) $\forall j \in N$, do temp_ts = max(temp_ts, $ts_j$);
(1f) send FINAL_TS(i, tag, temp_ts) to all processes;
(1g) clock = max(clock, temp_ts).
(2) When REVISE_TS(M, j, tag, clk) arrives from $P_j$:
(2a) priority = max(priority + 1, clk);
(2b) insert (M, tag, j, priority, undeliverable) in temp_Q;     // at end of queue
(2c) send PROPOSED_TS(i, j, tag, priority) to $P_j$.
(3) When FINAL_TS(j, tag, clk) arrives from $P_j$:
(3a) Identify entry Q_entry(tag) in temp_Q, corresponding to tag;
(3b) mark $q_{tag}$ as deliverable;
(3c) Update Q_entry.timestamp to clk and re-sort temp_Q based on the timestamp field;
(3d) if head(temp_Q) = Q_entry(tag) then
(3e)     move Q_entry(tag) from temp_Q to delivery_Q;
(3f)     while head(temp_Q) is deliverable do
(3g)         move head(temp_Q) from temp_Q to delivery_Q.
(4) When $P_i$ removes a message (M, tag, j, ts, deliverable) from head(delivery_$Q_i$):
(4a) clock = max(clock, ts) + 1.
```

## Example and Complexity

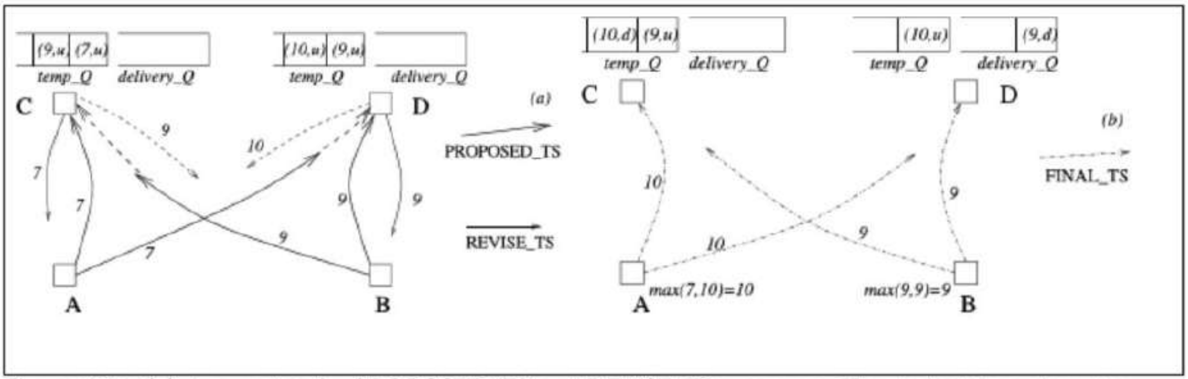


Figure 6.14: (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages.

### Complexity:

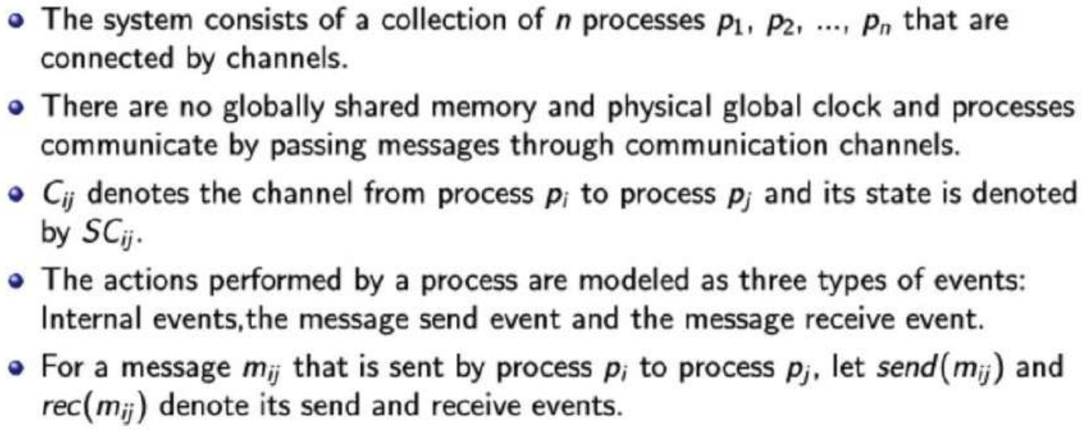
This algorithm uses three phases, and, to send a message to n− 1 processes, **it uses 3(n− 1) messages and incurs a delay of three message hops.**

## 6.7 Global state and snapshot recording algorithms

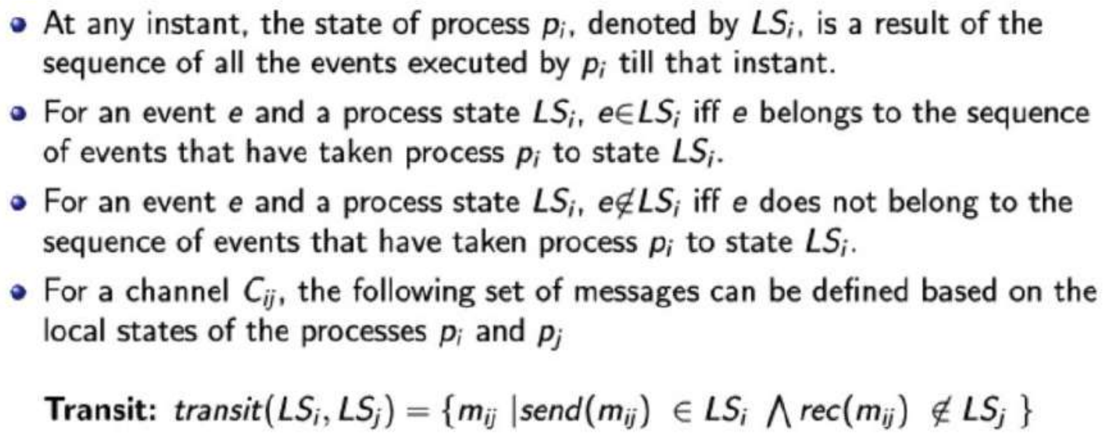
### Introduction

- **Recording the global state of a distributed system on-the-fly is an important**
- **paradigm.**
- **The lack of globally shared memory, global clock and unpredictable message delays in a distributed system make this problem non-trivial.**

### 6.7.1 System model



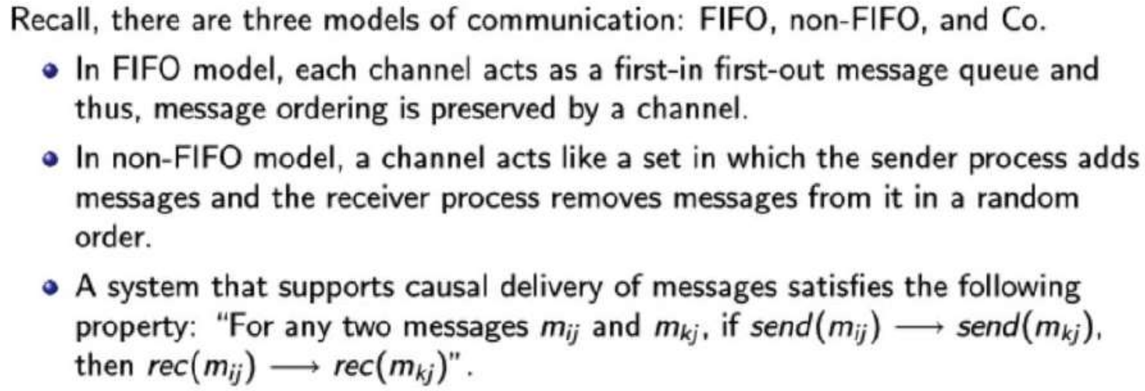
- The system consists of a collection of $n$ processes $p_1$, $p_2$, ..., $p_n$ that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- $C_{ij}$ denotes the channel from process $p_i$ to process $p_j$ and its state is denoted by $SC_{ij}$.
- The actions performed by a process are modeled as three types of events: Internal events,the message send event and the message receive event.
- For a message $m_{ij}$ that is sent by process $p_i$ to process $p_j$, let $send(m_{ij})$ and $rec(m_{ij})$ denote its send and receive events.



- At any instant, the state of process $p_i$, denoted by $LS_i$, is a result of the sequence of all the events executed by $p_i$ till that instant.
- For an event $e$ and a process state $LS_i$, $e \in LS_i$ iff $e$ belongs to the sequence of events that have taken process $p_i$ to state $LS_i$.
- For an event $e$ and a process state $LS_i$, $e \notin LS_i$ iff $e$ does not belong to the sequence of events that have taken process $p_i$ to state $LS_i$.
- For a channel $C_{ij}$, the following set of messages can be defined based on the local states of the processes $p_i$ and $p_j$

**Transit:** $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j \}$

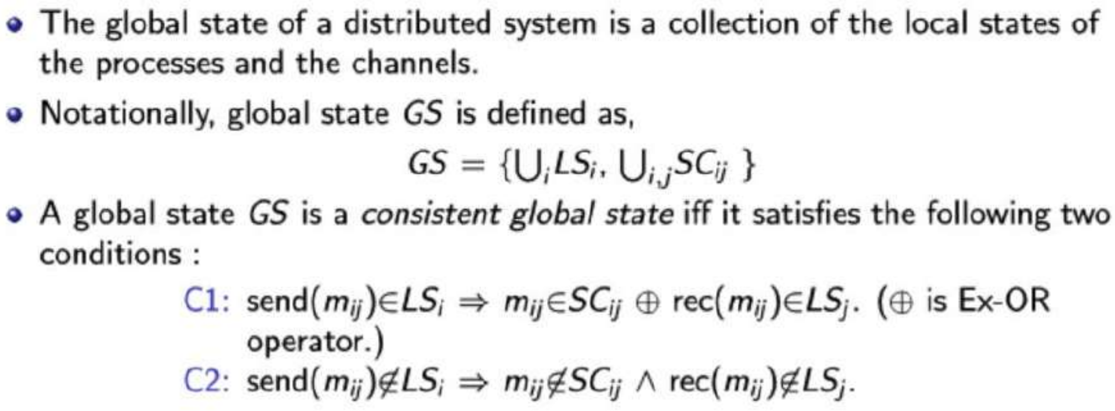
## Models of communication



Recall, there are three models of communication: FIFO, non-FIFO, and Co.

- In FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- A system that supports causal delivery of messages satisfies the following property: "For any two messages $m_{ij}$ and $m_{kj}$, if $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$".

## 6.7.2 Consistent global state



- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state $GS$ is defined as,

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij} \}$$

- A global state $GS$ is a *consistent global state* iff it satisfies the following two conditions :

C1: $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$. ($\oplus$ is Ex-OR operator.)

C2: $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$.

- In a consistent global state, every message that is recorded as received is also recorded as sent. Such a global state captures the notion of causality that a message cannot be received if it was not sent.
- Consistent global states are meaningful global states and inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

### 6.7.3 Interpretation in terms of cuts

- A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.
- Such a cut is known as a *consistent cut*.
- For example, consider the space-time diagram for the computation illustrated in Figure 4.1.
- Cut C1 is inconsistent because message m1 is flowing from the FUTURE to the PAST.
- Cut C2 is consistent and message m4 must be captured in the state of channel $C_{21}$.

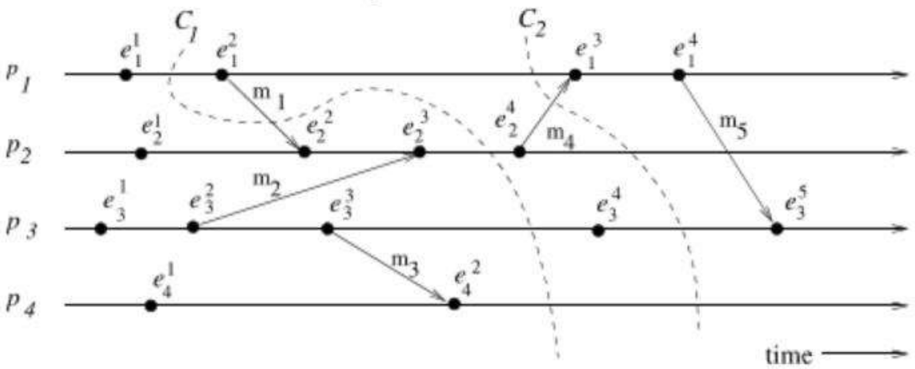


Figure 4.1: An Interpretation in Terms of a Cut.

### 6.7.4 Issues in recording a global state

The following two issues need to be addressed:

I1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

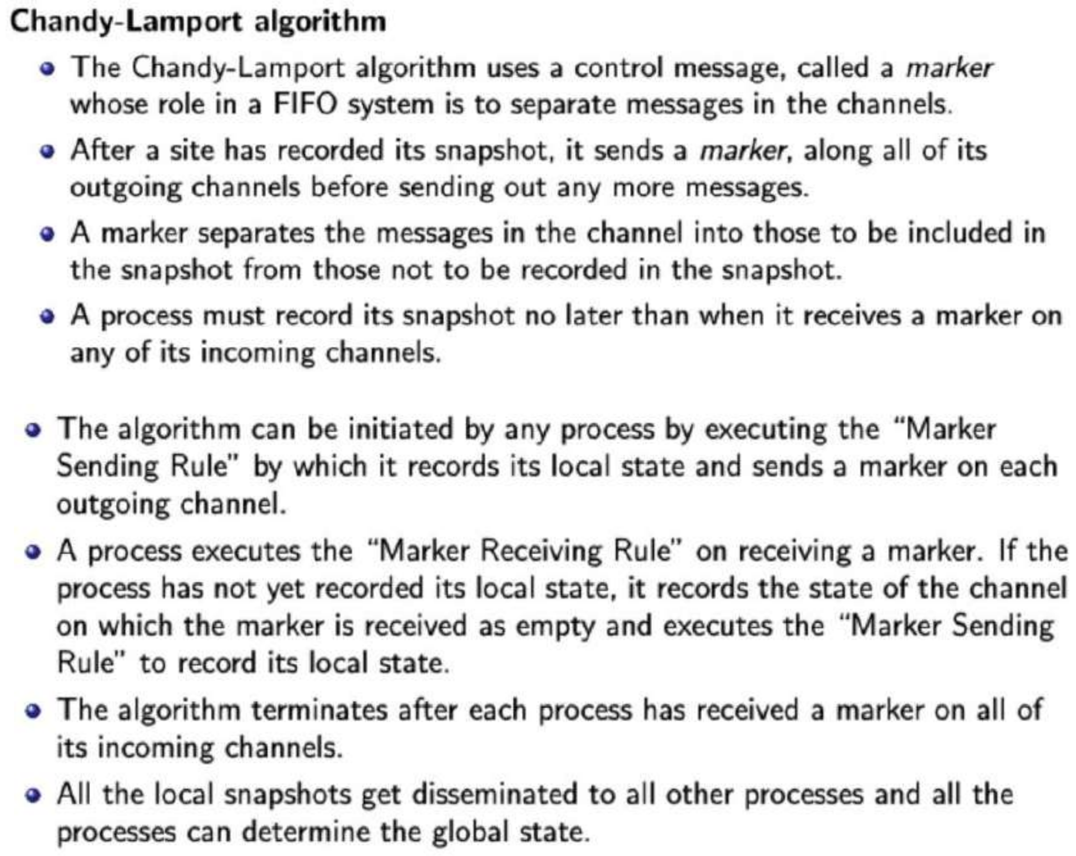
-Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
-Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

I2: How to determine the instant when a process takes its snapshot.

-A process $p_j$ must record its snapshot before processing a message $m_{ij}$ that was sent by process $p_i$ after recording its snapshot.

## 6.8 Snapshot algorithms for FIFO channels

### 6.8.1 Chandy Lamport Algorithm

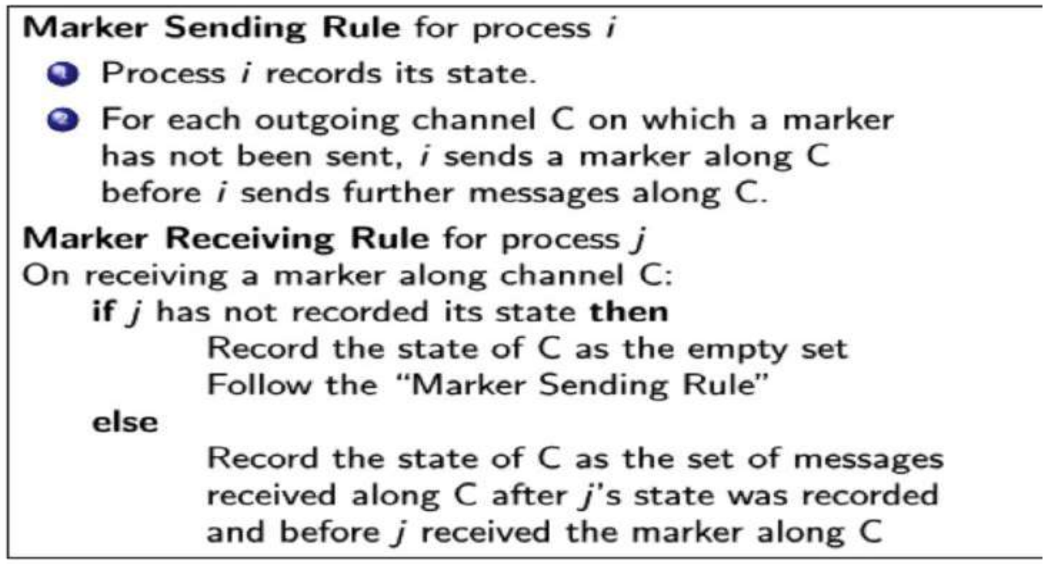


**Chandy-Lamport algorithm**

- The Chandy-Lamport algorithm uses a control message, called a *marker* whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a *marker*, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

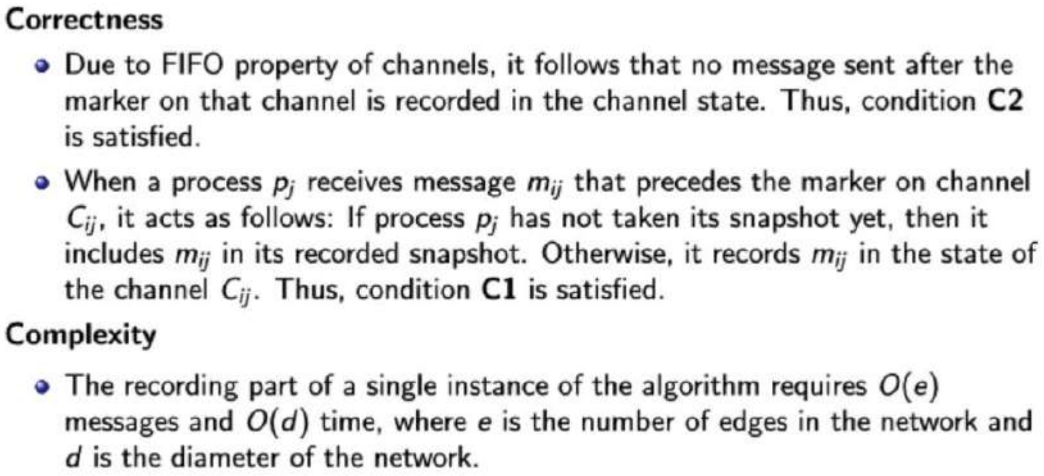
- The algorithm can be initiated by any process by executing the "Marker Sending Rule" by which it records its local state and sends a marker on each outgoing channel.
- A process executes the "Marker Receiving Rule" on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the "Marker Sending Rule" to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

### The algorithm



```
Marker Sending Rule for process i
  1 Process i records its state.
  2 For each outgoing channel C on which a marker
    has not been sent, i sends a marker along C
    before i sends further messages along C.
Marker Receiving Rule for process j
On receiving a marker along channel C:
    if j has not recorded its state then
            Record the state of C as the empty set
            Follow the "Marker Sending Rule"
    else
            Record the state of C as the set of messages
            received along C after j's state was recorded
            and before j received the marker along C
```

## Correctness and Complexity



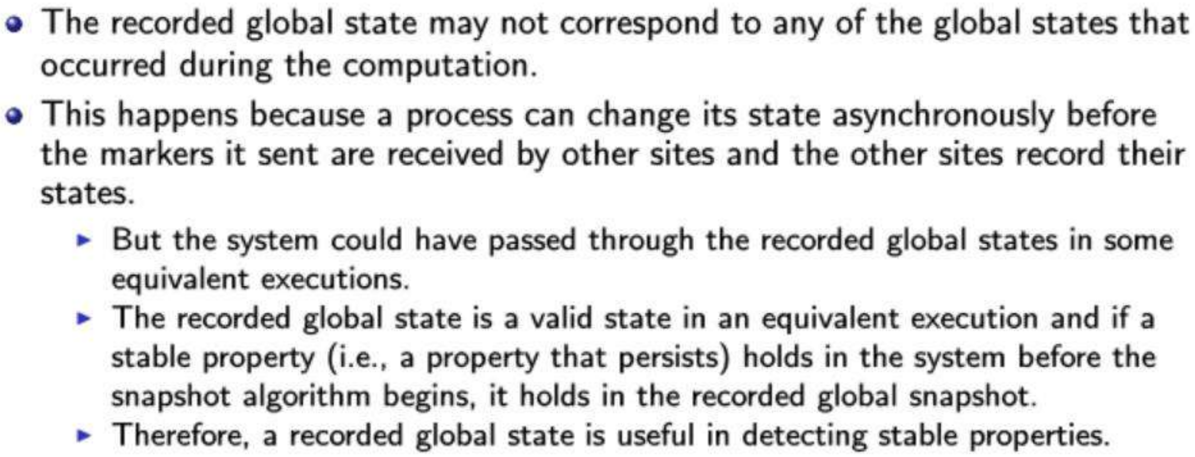
**Correctness**

- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied.
- When a process $p_j$ receives message $m_{ij}$ that precedes the marker on channel $C_{ij}$, it acts as follows: If process $p_j$ has not taken its snapshot yet, then it includes $m_{ij}$ in its recorded snapshot. Otherwise, it records $m_{ij}$ in the state of the channel $C_{ij}$. Thus, condition **C1** is satisfied.

**Complexity**

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

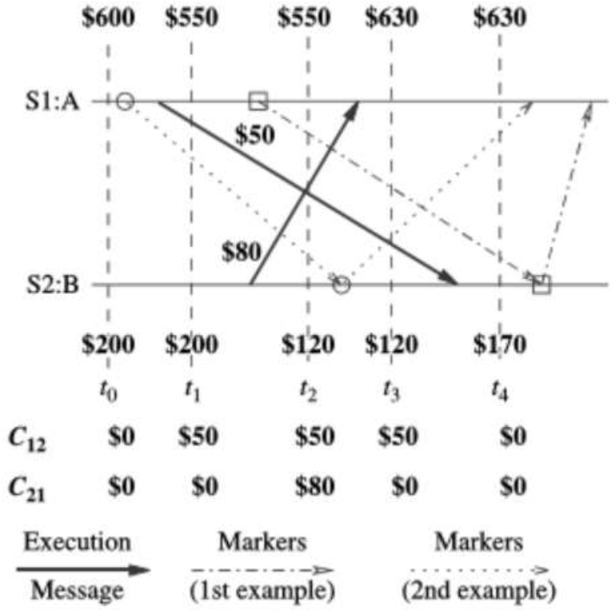
## 6.8.2 Properties of the recorded global state



- The recorded global state may not correspond to any of the global states that occurred during the computation.
- This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.
  - But the system could have passed through the recorded global states in some equivalent executions.
  - The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.
  - Therefore, a recorded global state is useful in detecting stable properties.

The recorded global state may not correspond to any of the global states that occurred during the computation. Consider two possible executions of the snapshot algorithm (shown in Figure 4.3) for the money transfer example of Figure 4.2:

**Figure 4.3** Timing diagram of two possible executions of the banking example.



(Markers shown using dashed-and-dotted arrows.) Let site S1 initiate the algorithm just after $t_1$. Site S1 records its local state (account A = $550) and sends a marker to site S2. The marker is received by site S2 after $t_4$. When site S2 receives the marker, it records its local state (account B = $170), the state of channel $C_{12}$ as $0, and sends a marker along channel $C_{21}$. When site S1 receives this marker, it records the state of channel $C_{21}$ as $80. The $800 amount in the system is conserved in the recorded global state,

A= $550 B = $170 $C_{12}$ = $0 $C_{21}$ = $80

(Markers shown using dotted arrows.) Let site S1 initiate the algorithm just after $t_0$ and before sending the $50 for S2. Site S1 records its local state (account A = $600) and sends a marker to site S2. The marker is received by site S2 between $t_2$ and $t_3$. When site S2 receives the marker, it records its local state (account B = $120), the state of channel $C_{12}$ as $0, and sends a marker along channel $C_{21}$. When site S1 receives this marker, it records the state of channel $C_{21}$ as $80. The $800 amount in the system is conserved in the recorded global state,

A= $600 B = $120 $C_{12}$ = $0 $C_{21}$ = $80

In both these possible runs of the algorithm, the recorded global states never occurred in the execution. This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.

A physical interpretation of the collected global state is as follows: consider the two instants of recording of the local states in the banking example. If the cut formed by these instants is viewed as being an elastic band and if the elastic band is stretched so that it is vertical, then recorded states of all processes occur simultaneously at one physical instant, and the recorded global state occurs in the execution that is depicted in this modified space– time diagram. This is called the **<u>rubber-band criterion</u>**.

## PART A

1. Listout different types of message ordering.

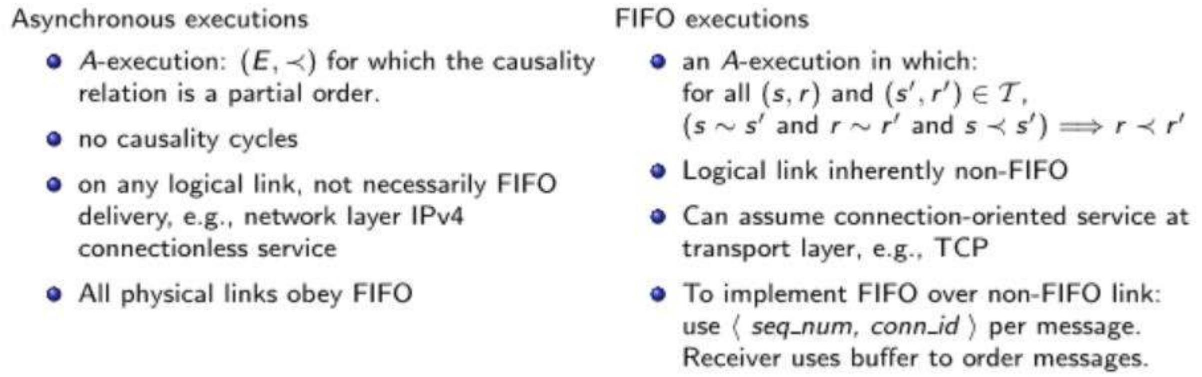
i) non-FIFO, (ii) FIFO,(iii) causal order, and

(iv) synchronous order.

2. What is **An asynchronous execution**

**An asynchronous execution (or A-execution) is an execution $E\prec$ for which the causality relation is a partial order.**

**Difference between Asynchronous and FIFO executions.**



Asynchronous executions

- *A*-execution: $(E, \prec)$ for which the causality relation is a partial order.
- no causality cycles
- on any logical link, not necessarily FIFO delivery, e.g., network layer IPv4 connectionless service
- All physical links obey FIFO

FIFO executions

- an *A*-execution in which: for all $(s, r)$ and $(s', r') \in \mathcal{T}$, $(s \sim s'$ and $r \sim r'$ and $s \prec s') \Longrightarrow r \prec r'$
- Logical link inherently non-FIFO
- Can assume connection-oriented service at transport layer, e.g., TCP
- To implement FIFO over non-FIFO link: use ⟨ *seq_num*, *conn_id* ⟩ per message. Receiver uses buffer to order messages.

3. Distinguish between **Message arrival vs. Delivery**

To implement CO, we distinguish between the arrival of a message and its delivery.

- A message m that arrives in the local OS buffer at $P_i$ may have to be delayed until the messages that were sent to $P_i$ causally before m was sent (the "overtaken" messages) have arrived and are processed by the application. The delayed message m is then given to the application for processing.
- The event of an application processing an arrived message is referred to as a *delivery* event (instead of as a *receive* event) for emphasis.
- No message overtaken by a chain of messages between the same (sender, receiver) pair. In Fig. (a), $m_1$ overtaken by chain <$m_2$, $m_3$ >
- CO degenerates to FIFO when m1, m2 sent by same process

**4. List out Uses of CO.**

- Causal order is useful for applications requiring **updates to shared data, implementing distributed shared memory, and fair resource allocation** such as granting of requests for distributed mutual exclusion ,**collaborative applications, event notification systems,**
- **distributed virtual environments**

**5. List out the Characterizations of Causal Order**

(iv) **Definition (Message order (MO)) A MO execution is an execution in which, for all (s,r) and (s’,r’) $\in$ T , s $\prec$ s’ $\Rightarrow \neg$(r’ $\prec$ r)**

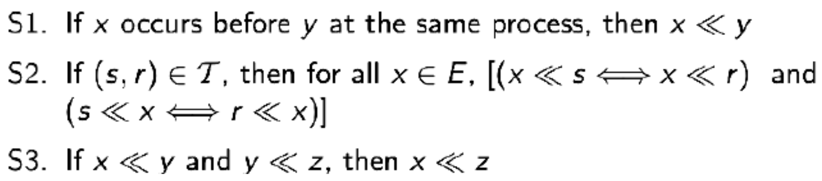
(v) Another characterization of a CO execution in terms of the partial order E $<$ is known as **<u>the empty-interval (EI) property.</u>**

(vi) **<u>Common Past and Future</u>**

Another characterization of CO executions is in terms of the causal past/future of a send event and its corresponding receive event.

**6. What are the conditions for Casuality in a synchronous execution?**

**(Casuality in a synchronous execution) The synchronous causality relation on E is the smallest transitive relation that satisfies the following:**

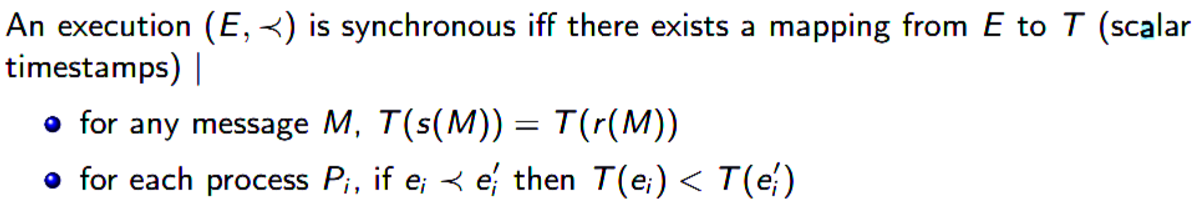


S1. If $x$ occurs before $y$ at the same process, then $x \ll y$

S2. If $(s, r) \in \mathcal{T}$, then for all $x \in E$, $[(x \ll s \Longleftrightarrow x \ll r)$ and $(s \ll x \Longleftrightarrow r \ll x)]$

S3. If $x \ll y$ and $y \ll z$, then $x \ll z$

**7. What is Timestamping a synchronous execution.**



An execution $(E, \prec)$ is synchronous iff there exists a mapping from $E$ to $T$ (scalar timestamps) |

- for any message $M$, $T(s(M)) = T(r(M))$
- for each process $P_i$, if $e_i \prec e_i'$ then $T(e_i) < T(e_i')$

8. What is RSC.

**RSC execution An A-execution (E, $\prec$) is an RSC execution iff there exists a non-separated linear extension of the partial order (E, $\prec$).**

9. What is crown?

Let E be an execution. A crown of size k in E is a sequence $s^i$ $r^i$ , i $\in$ 0 k $-$ 1 of pairs of corresponding send and receive events such that: $s^0 \prec r^1$, $s^1 \prec r^2$, , $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

10. What is ***binary rendezvous***

The **rendezvous between a pair of processes at a time, which is called *binary rendezvous* as opposed to the *multiway rendezvous*.**

11. What is group communication.

A *message broadcast* is the sending of a message to all members in the distributed system. The notion of a system can be confined only to those sites/processes participating in the joint application. Refining the notion of *broadcasting*, there is *multicasting* wherein a message is sent to a certain subset, identified as a *group*, of the processes in the system.

**12. List the two criteria for a causal ordering protocol:**

- **Safety** In order to prevent causal order from being violated, a message M that arrives at a process may need to be buffered until all system wide messages sent in the causal past of the send M event to that same destination have already arrived.
- **Liveness** A message that arrives at a process must eventually be delivered to the process.

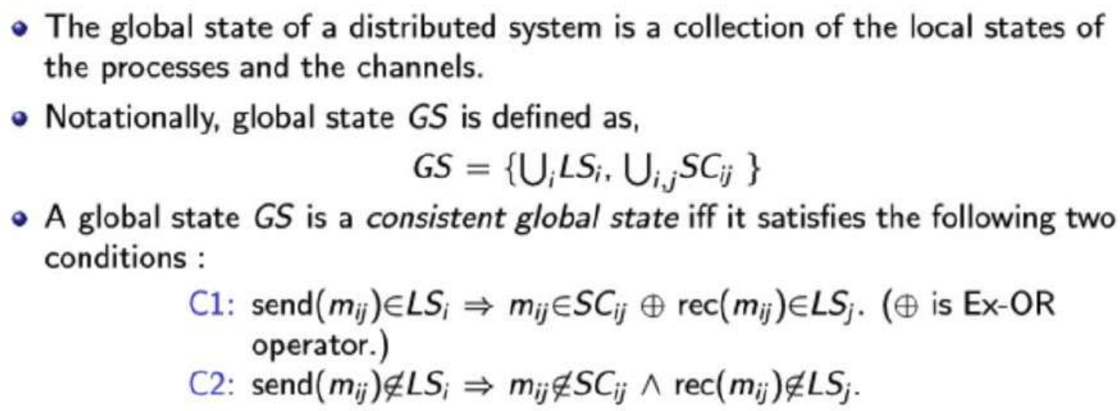
**13. Discuss the Necessary and Sufficient Conditions for Optimality:**

An optimal CO algorithm stores in local message logs and propagates on messages, information of the form "d is a destination of M" about a message sent in the causal past, *as long as* and *only as long as*:

(***Propagation Constraint I***) it is not known that the message M is delivered to d, and

(***Propagation Constraint II***) it is not known that a message has been sent to d in the causal future of Send M , and hence it is not guaranteed using a reasoning based on transitivity that the message M will be delivered to d in CO.

14. What is consistent global state.



- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state $GS$ is defined as,

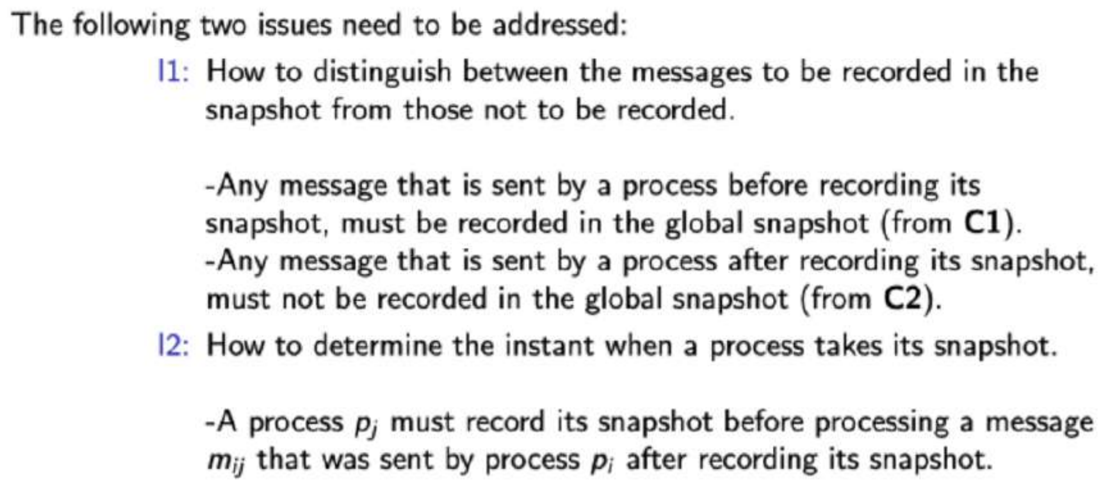
$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}$$

- A global state $GS$ is a *consistent global state* iff it satisfies the following two conditions :

C1: send($m_{ij}$)$\in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus$ rec($m_{ij}$)$\in LS_j$. ($\oplus$ is Ex-OR operator.)

C2: send($m_{ij}$)$\notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge$ rec($m_{ij}$)$\notin LS_j$.

**15. What are the Issues in recording a global state**



The following two issues need to be addressed:

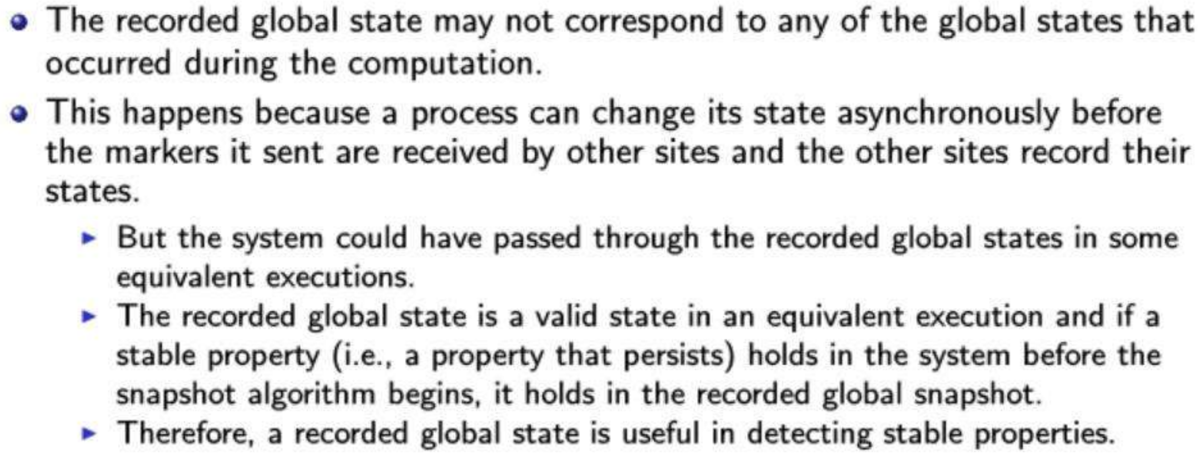
I1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

-Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
-Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

I2: How to determine the instant when a process takes its snapshot.

-A process $p_j$ must record its snapshot before processing a message $m_{ij}$ that was sent by process $p_i$ after recording its snapshot.

**16. Listout the Properties of the recorded global state**



- The recorded global state may not correspond to any of the global states that occurred during the computation.
- This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.
  - But the system could have passed through the recorded global states in some equivalent executions.
  - The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.
  - Therefore, a recorded global state is useful in detecting stable properties.

**PART B**

1. Explain Asynchronous execution with synchronous communication
2. Discuss Synchronous program order on an asynchronous system
3. Explain the Algorithm for binary rendezvous (or) Bagrodia’s Algorithm.
4. Dicuss the Raynal–Schiper–Toueg algorithm (RST) (2.5.1)
5. Explain group communication in detail.
6. Explain Optimal KS Algorithm for CO: (2.5.2)
7. Explain the distributed algorithm to implement total order and causal order of messages (or) Three-phase Algorithm
8. Explain Snapshot algorithms for FIFO channels or Chandy Lamport Algorithm

## UNIT III - DISTRIBUTED MUTEX & DEADLOCK

Distributed mutual exclusion algorithms: Introduction – Preliminaries – Lamport‘s algorithm – Ricart-Agrawala algorithm – Maekawa‘s algorithm – Suzuki–Kasami‘s broadcast algorithm. Deadlock detection in distributed systems: Introduction – System model – Preliminaries –Models of deadlocks – Knapp‘s classification –Algorithms for the single resource model, the AND model and the OR model.

## DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

**Mutual exclusion:** Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.

- Only one process is allowed to execute the critical section (CS) at any given time.
- In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.
- Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.

**Three basic approaches** for distributed mutual exclusion:

- Token based approach
- Non-token based approach
- Quorum based approach

## 3.1 INTRODUCTION

Token-based approach:

- A unique token is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Mutual exclusion is ensured because the token is unique.

Non-token based approach:

- Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.

Quorum based approach:

- Each site requests permission to execute the CS from a subset of sites (called a quorum).
- Any two quorums contain a common site.
- This common site is responsible to make sure that only one request executes the CS at any time.

## 3.2 PRELIMINARIES

### 3.2.1 System Model

- The system consists of N sites, $S_1$, $S_2$, ..., $S_N$.
- We assume that a single process is running on each site. The process at site
- $S_i$ is denoted by $p_i$ .
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the 'requesting the CS' state, the site is blocked and can not make further requests for the CS. In the 'idle' state, the site is executing outside the CS. In token-based algorithms, a site can also be in a state where a site holding
- the token is executing outside the CS (called the *idle token* state).
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

### 3.2.2. Requirements of Mutual Exclusion Algorithms

- **Safety Property**: At any instant, only one process can execute the critical section.
- **Liveness Property**: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
- **Fairness:** Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

The first property is absolutely necessary and the other two properties are considered important in mutual exclusion algorithms.

### 3.2.3. Performance Metrics

The performance is generally measured by the following **four metrics**:

- **Message complexity:** The number of messages required per CS execution by a site.
- **Synchronization delay**: After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1).

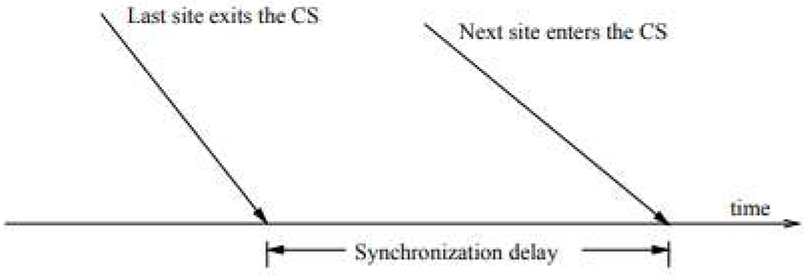


Figure 1: Synchronization Delay.

- **Response time:** The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).

- **System throughput**: The rate at which the system executes requests for the CS.

$$\boxed{\text{system throughput}=1/(SD+E)}$$

where *SD* is the synchronization delay and *E* is the average critical section execution time.

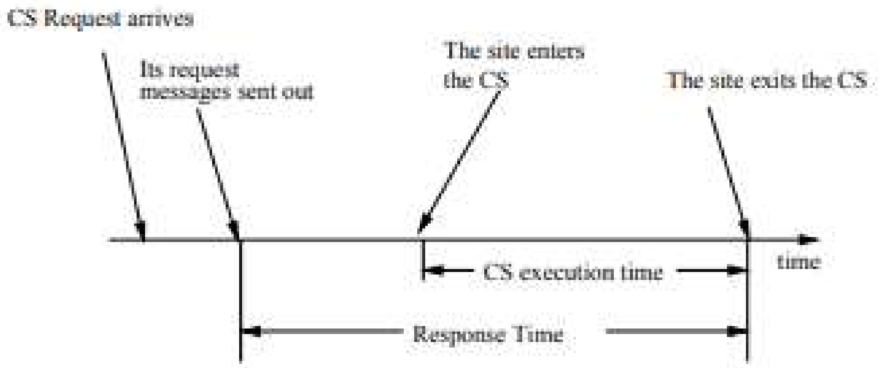


Figure 2: Response Time.

**Low and High Load Performance:**

- We often study the performance of mutual exclusion algorithms under two special loading conditions, viz., “low load” and “high load”.
- The load is determined by the arrival rate of CS execution requests.
- Under *low load* conditions, there is seldom more than one request for the critical section present in the system simultaneously.
- Under *heavy load* conditions, there is always a pending request for critical section at a site.

**Best and worst case performance**

- Generally, mutual exclusion algorithms have best and worst cases for the performance metrics. In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example, in most mutual exclusion algorithms the best value of the response time is a round-trip message delay plus the CS execution time, 2T + E.
- Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For examples, the best and worst values of the response time are achieved when load is, respectively, low and high; in some mutual exclusion algorithms the best and the worse message traffic is generated at low and heavy load conditions, respectively.

## 3.3 LAMPORT'S ALGORITHM

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site $S_i$ keeps a queue, *request queue$_i$*, which contains mutual exclusion requests ordered by their timestamps.
- This algorithm requires communication channels to deliver messages the FIFO order.

### The Algorithm

**Requesting the critical section:**

- When a site $S_i$ wants to enter the CS, it broadcasts a REQUEST($ts_i$ , $i$ ) message to all other sites and places the request on *request queue$_i$*. (($ts_i$, $i$) denotes the timestamp of the request.)
- When a site $S_j$ receives the REQUEST($ts_i$, $i$) message from site $S_i$,places site $S_i$ 's request on *request queue$_j$* and it returns a timestamped REPLY message to $S_i$.

**Executing the critical section:** Site $S_i$ enters the CS when the following two conditions hold:

L1: $S_i$ has received a message with timestamp larger than ($ts_i$, $i$) from all other sites.

L2: $S_i$ 's request is at the top of *request queue$_i$*.

**Releasing the critical section:**

- Site $S_i$ , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site $S_j$ receives a RELEASE message from site $S_i$ , it removes $S_i$ 's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

### Correctness

**Theorem: Lamport's algorithm achieves mutual exclusion.**

**Proof:**

- Proof is by contradiction. Suppose two sites $S_i$ and $S_j$ are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*.
- This implies that at some instant in time, say $t$, both $S_i$ and $S_j$ have their own requests at the top of their *request queues* and condition L1 holds at them. Without loss of generality, assume that $S_i$ 's request has smaller timestamp than the request of $S_j$.
- m condition L1 and FIFO property of the communication channels, it is clear that at instant $t$ the request of $S_i$ must be present in *request queue$_j$* when $S_j$ was executing its CS. This implies that $S_j$'s own request is at the top of its own *request queue* when a smaller timestamp request, $S_i$ 's request, is present in the *request queue$_j$* – a contradiction!

**Theorem: Lamport's algorithm is fair.**

Proof:

- The proof is by contradiction. Suppose a site $S_i$ 's request has a smaller timestamp than the request of another site $S_j$ and $S_j$ is able to execute the CS before $S_i$ .
- For $S_j$ to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, $S_j$ has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But *request queue* at a site is ordered by timestamp, and according to our assumption $S_i$ has lower timestamp. So $S_i$ 's request must be placed ahead of the $S_j$ 's request in the *request queue*$_j$. This is a contradiction!

**Performance**

- For each CS execution, Lamport's algorithm requires $(N - 1)$ REQUEST messages, $(N - 1)$ REPLY messages, and $(N - 1)$ RELEASE messages. Thus, Lamport's algorithm requires $3(N - 1)$ messages per CS invocation.
- Synchronization delay in the algorithm is $T$.

**An optimization**

- In Lamport's algorithm,REPLY messages can be omitted in certain situations. For example, if site $S_j$ receives a REQUEST message from site $S_i$ after it has sent its own REQUEST message with timestamp higher than the timestamp of site $S_i$ 's request, then site $S_j$ need not send a REPLY message to site $S_i$ .
- This is because when site $S_i$ receives site $S_j$ 's request with timestamp higher than its own, it can conclude that site $S_j$ does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between $3(N - 1)$ and $2(N - 1)$ messages per CS execution.

## 3.4 RICART-AGRAWALA ALGORITHM

- The Ricart-Agrawala algorithm assumes the communication channels are FIFO. The algorithm uses two types of messages: REQUEST and REPLY.
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section. A process sends a REPLY message to a process to give its permission to that process.
- Processes use Lamport-style logical clocks to assign a timestamp to critical section requests and timestamps are used to decide the priority of requests.
- Each process $p_i$ maintains the Request-Deferred array, $RD_i$ , the size of which is the same as the number of processes in the system.
- Initially, $\forall i \forall j : RD_i[j]=0$. Whenever $p_i$ defer the request sent by $p_j$ , it sets
- $RD_i$ [j]=1 and after it has sent a REPLY message to $p_j$ , it sets $RD_i$ [j]=0.

## Description of the Algorithm

### Requesting the critical section:

(a) When a site $S_i$ wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) When site $S_j$ receives a REQUEST message from site $S_i$, it sends a REPLY message to site $S_i$ if site $S_j$ is neither requesting nor executing the CS, or if the site $S_j$ is requesting and $S_i$ 's request's timestamp is smaller than site $S_j$ 's own request's timestamp. Otherwise, the reply is deferred and $S_j$ sets $RD_j[i]=1$

### Executing the critical section:

(c) Site $S_i$ enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

### Releasing the critical section:

(a) When site $S_i$ exits the CS, it sends all the deferred REPLY messages: $\forall j$ if $RD_i$ [j]=1, then send a REPLY message to $S_j$ and set $RD_i[j]=0$.

Notes:

- When a site receives a message, it updates its clock using the timestamp in the message.
- When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.

### Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.

### Proof:

- Proof is by contradiction. Suppose two sites $S_i$ and $S_j$ ‘ are executing the CS concurrently and $S_i$ 's request has higher priority than the request of $S_j$ .
- Clearly, $S_i$ received $S_j$ 's request after it has made its own request.
- Thus, $S_j$ can concurrently execute the CS with $S_i$ only if $S_i$ returns a REPLY to $S_j$

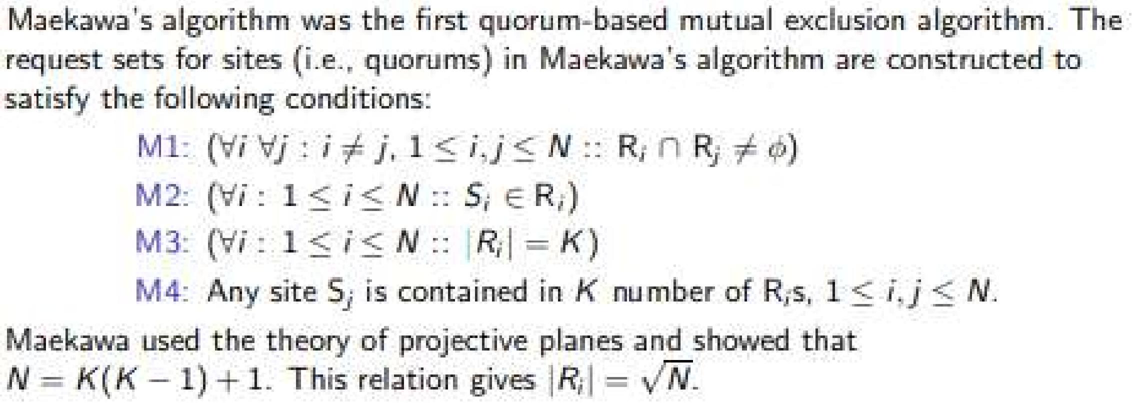
(in response to $S_j$ 's request) before $S_i$ exits the CS.

- However, this is impossible because $S_j$ 's request has lower priority.Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

**Performance**

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

## 3.5 MAEKAWA'S ALGORITHM



Maekawa's algorithm was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i \, \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$

M2: $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$

M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K)$

M4: Any site $S_j$ is contained in $K$ number of $R_i$s, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that $N = K(K - 1) + 1$. This relation gives $|R_i| = \sqrt{N}$.

- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm.
- Condition M3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion.
- Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have "equal responsibility" in granting permission to other sites.

**A site $S_i$ executes the following steps to execute the CS.**

**Requesting the critical section**

(a) A site $S_i$ requests access to the CS by sending REQUEST($i$) messages to all sites in its request set $R_i$.

(b) When a site $S_j$ receives the REQUEST($i$ ) message, it sends a REPLY($j$ ) message to $S_i$ provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST($i$ ) for later consideration.

**Executing the critical section**

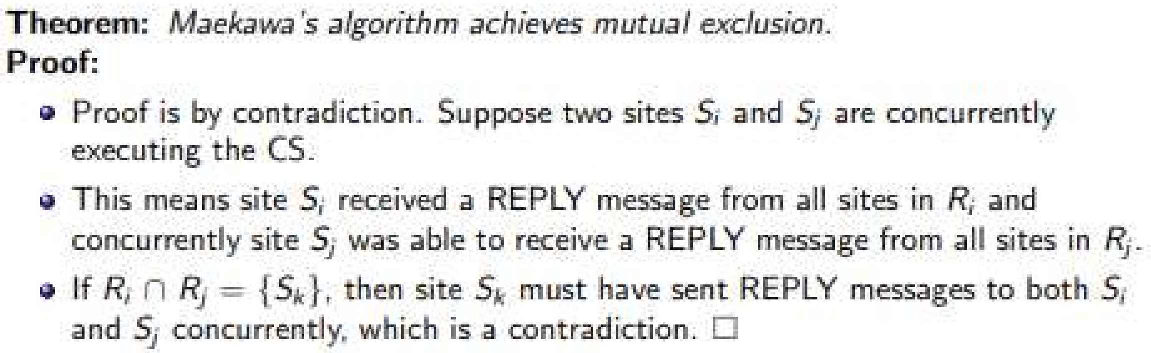
(c) Site $S_i$ executes the CS only after it has received a REPLY message from every site in $R_i$.

## Releasing the critical section

(a) After the execution of the CS is over, site $S_i$ sends a RELEASE($i$) message to every site in $R_i$.

(b) When a site $S_j$ receives a RELEASE($i$ ) message from site $S_i$ , it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

## Correctness

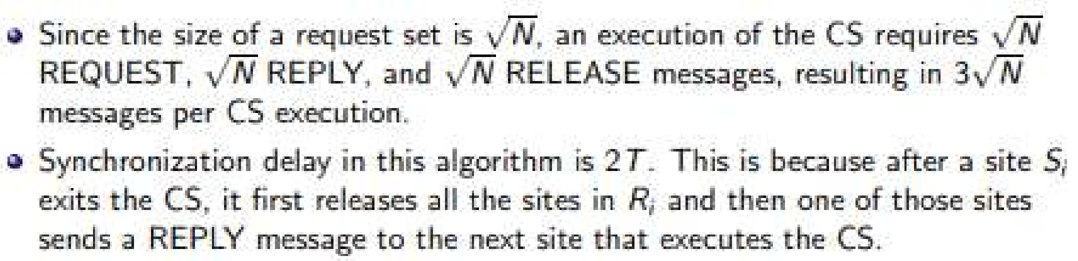


**Theorem:** *Maekawa's algorithm achieves mutual exclusion.*

**Proof:**

- Proof is by contradiction. Suppose two sites $S_i$ and $S_j$ are concurrently executing the CS.
- This means site $S_i$ received a REPLY message from all sites in $R_i$ and concurrently site $S_j$ was able to receive a REPLY message from all sites in $R_j$.
- If $R_i \cap R_j = \{S_k\}$, then site $S_k$ must have sent REPLY messages to both $S_i$ and $S_j$ concurrently, which is a contradiction. □

## Performance



- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution.
- Synchronization delay in this algorithm is $2T$. This is because after a site $S_i$ exits the CS, it first releases all the sites in $R_i$ and then one of those sites sends a REPLY message to the next site that executes the CS.

## 3.5.1 Problem of Deadlocks

- Maekawa's algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.
- Assume three sites $S_i$ , $S_j$ , and $S_k$ simultaneously invoke mutual exclusion. Suppose $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, and $R_k \cap R_i = \{S_{ki}\}$.
- Consider the following scenario:
  - $Sij$ has been locked by $Si$ (forcing $Sj$ to wait at $Sij$ ).
  - $S_{jk}$ has been locked by $S_j$(forcing $S_k$ to wait at $S_{jk}$).
  - $S_{ki}$ has been locked by $S_k$(forcing $S_i$ to wait at $S_{ki}$).
- This state represents a deadlock involving sites $S_i$ , $S_j$ , and $S_k$ .

## Handling Deadlocks

Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock.

- A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.

Deadlock handling requires three types of messages:

FAILED: A FAILED message from site $S_i$ to site $S_j$ indicates that $S_i$ can not grant $S_j$ 's request because it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from $S_i$ to $S_j$ indicates that $S_i$ would like to find out from $S_j$ if it has succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site $S_i$ to $S_j$ indicates that $S_i$ is returning the permission to $S_j$ (to yield to a higher priority request at $S_j$ ).

**Maekawa's algorithm handles deadlocks as follows:**

- When a REQUEST(*ts*, *i* ) from site $S_i$ blocks at site $S_j$ because $S_j$ has currently granted permission to site $S_k$ , then $S_j$ sends a FAILED(*j* ) message to $S_i$ if $S_i$ 's request has lower priority. Otherwise, $S_j$ sends an INQUIRE(*j* ) message to site $S_k$
- In response to an INQUIRE(*j* ) message from site $S_j$ , site $S_k$ sends a YIELD(*k* ) message to $S_j$ provided $S_k$ has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new GRANT from it.
- In response to a YIELD(*k* ) message from site $S_k$, site $S_j$ assumes as if it has been released by $S_k$ , places the request of $S_k$ at appropriate location in the request queue, and sends a GRANT(*j* ) to the top request's site in the queue. Maekawa's algorithm requires extra messages to handle deadlocks

**Maximum number of messages required per CS execution in this case is 5√ N.**

## Token-based algorithms

In token-based algorithms, a unique token is shared among the sites. A site is allowed to enter its CS if it possesses the token. A site holding the token can enter its CS repeatedly until it sends the token to some other site. Depending upon the way a site carries out the search for the token, there are numerous token-based algorithms. Next, we discuss two token-based mutual exclusion algorithms.

First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. (A primary function of the sequence numbers is to distinguish between old and current requests.) Second, the correctness proof of token-based algorithms, that they enforce mutual exclusion, is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS.

## 3.6 SUZUKI-KASAMI’S BROADCAST ALGORITHM

- If a site wants to enter the CS and it does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
- If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

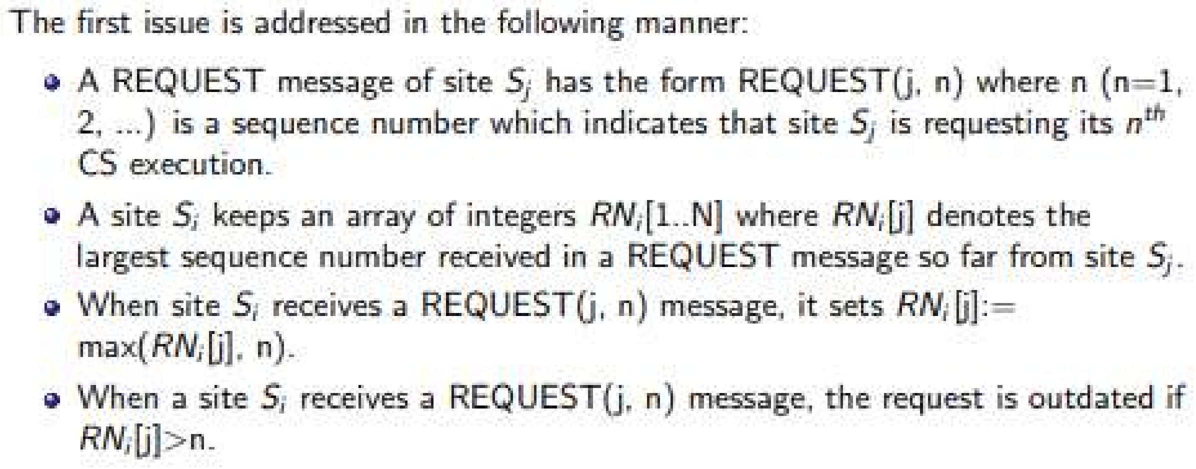
**This algorithm must efficiently address the following two design issues:**

(1) **How to distinguish an outdated REQUEST message from a current REQUEST message:**

- Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied.
- If a site can not determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it.
- This will not violate the correctness, however, this may seriously degrade the performance.

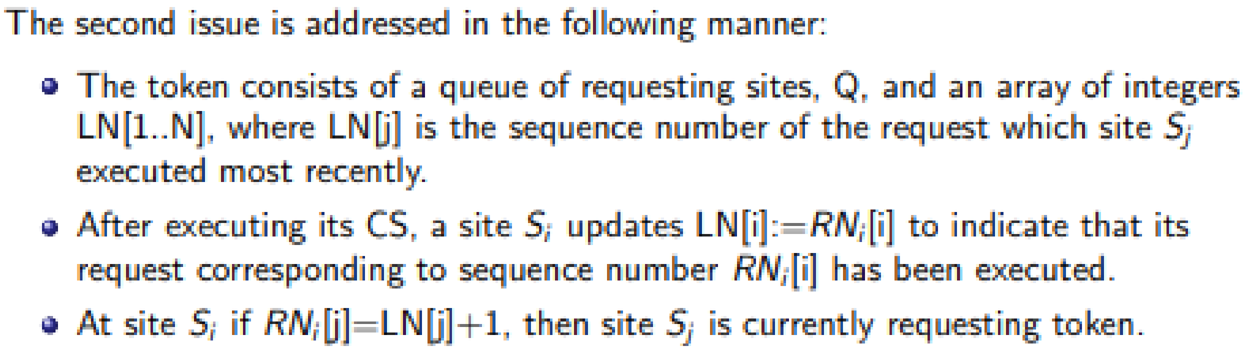
(2) **How to determine which site has an outstanding request for the CS:**

- After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them.



The first issue is addressed in the following manner:

- A REQUEST message of site $S_j$ has the form REQUEST(j, n) where n (n=1, 2, ...) is a sequence number which indicates that site $S_j$ is requesting its $n^{th}$ CS execution.
- A site $S_i$ keeps an array of integers $RN_i[1..N]$ where $RN_i[j]$ denotes the largest sequence number received in a REQUEST message so far from site $S_j$.
- When site $S_i$ receives a REQUEST(j, n) message, it sets $RN_i[j]:=$ max($RN_i[j]$, n).
- When a site $S_i$ receives a REQUEST(j, n) message, the request is outdated if $RN_i[j]>$n.



The second issue is addressed in the following manner:

- The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site $S_j$ executed most recently.
- After executing its CS, a site $S_i$ updates LN[i]:=$RN_i$[i] to indicate that its request corresponding to sequence number $RN_i$[i] has been executed.
- At site $S_i$ if $RN_i$[j]=LN[j]+1, then site $S_j$ is currently requesting token.

\

## The Algorithm

### Requesting the critical section

(a) If requesting site $S_i$ does not have the token, then it increments its sequence number, $RN_i$[i], and sends a REQUEST(i, *sn*) message to all other sites. ('*sn*' is the updated value of $RN_i$[i].)

(b) When a site $S_j$ receives this message, it sets $RN_j$[i] to max($RN_j$[i], *sn*). If $S_j$ has the idle token, then it sends the token to $S_i$ if $RN_j$[i]=LN[i]+1.

### Executing the critical section

(c) Site $S_i$ executes the CS after it has received the token.

### Releasing the critical section

Having finished the execution of the CS, site $S_i$ takes the following actions:

(a) It sets LN[i] element of the token array equal to $RN_i$[i].

(b) For every site $S_j$ whose id is not in the token queue, it appends its id to the token queue if $RN_i$[j]=LN[j]+1.

(c) If the token queue is nonempty after the above update, $S_i$ deletes the top site id from the token queue and sends the token to the site indicated by the id.

## Correctness

Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

Theorem: A requesting site enters the CS in finite time. Proof:

- Token request messages of a site $S_i$ reach other sites in finite time.
- Since one of these sites will have token in finite time, site $S_i$ 's request will be placed in the token queue in finite time.
- Since there can be at most $N - 1$ requests in front of this request in the token queue, site $S_i$ will get the token and execute the CS in finite time.

**Performance**

- No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.
- If a site does not hold the token when it makes a request, the algorithm requires $N$ messages to obtain the token. Synchronization delay in this algorithm is 0 or $T$ .

## 3.7 DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

- Deadlocks is a fundamental problem in distributed systems.
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.

## 3.8 SYSTEM MODEL

- A distributed program is composed of a set of n asynchronous processes $p_1$, $p_2$, . . . , $p_i$ , . . . , $p_n$ that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.

- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.

We make the following assumptions:

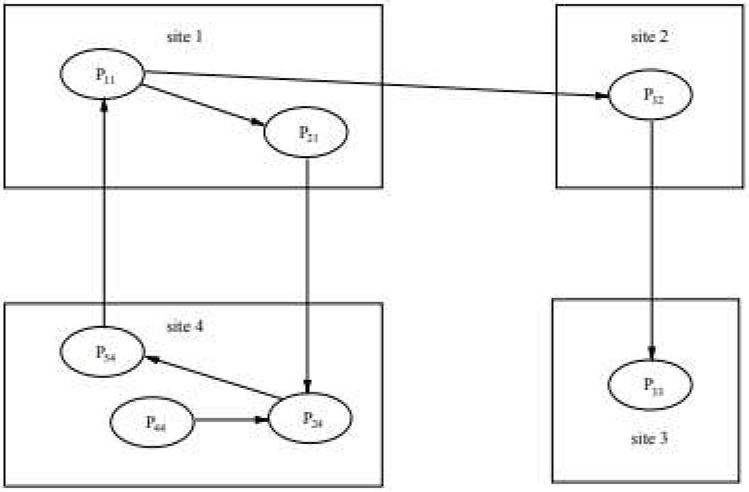
- The systems have only reusable resources.
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.

### 3.8.1 Wait for Graph (WFG)

- A process can be in two states: *running* or *blocked*.
- In the running state (also called *active* state), a process has all the needed resources and is either executing or is ready for execution.
- In the blocked state, a process is waiting to acquire some resource.

- The state of the system can be modeled by directed graph, called a *wait for graph* (WFG).
- In a WFG , nodes are processes and there is a directed edge from node $P_1$ to mode $P_2$ if $P_1$ is blocked and is waiting for $P_2$ to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.
- Figure 1 shows a WFG, where process $P_{11}$ of site 1 has an edge to process $P_{21}$ of site 1 and $P_{32}$ of site 2 is waiting for a resource which is currently held by process $P_{21}$.
- At the same time process $P_{32}$ is waiting on process $P_{33}$ to release a resource.
- If $P_{21}$ is waiting on process $P_{11}$, then processes $P_{11}$, $P_{32}$ and $P_{21}$ form a cycle and all the four processes are involved in a deadlock depending upon the request model.

**<u>An Example of  WFG</u>**



## 3.9 PRILIMINARIES

### 3.9.1 Deadlock Handling Strategies

- There are three strategies for handling deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection.
- Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay.
- Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.

- This approach is highly inefficient and impractical in distributed systems.
- In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system).
- However, due to several problems, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires examination of the status of process-resource interactions for presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

### 3.9.2 Issues in Deadlock Detection

- Deadlock handling using the approach of deadlock detection entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.

- Detection of deadlocks involves addressing two issues: Maintenance of the WFG and searching of the WFG for the presence of cycles (or knots).

**Correctness Criteria:** A deadlock detection algorithm must satisfy the following two conditions:

(i) Progress (No undetected deadlocks):

- The algorithm must detect all existing deadlocks in finite time.
- In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

(ii) Safety (No false deadlocks):

- The algorithm should not report deadlocks which do not exist (called *phantom or false* deadlocks).

### Resolution of a Detected Deadlock

- Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock.
- It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution.

## 3.10 MODELS OF DEADLOCKS

Distributed systems allow several kinds of resource requests.

### 3.10.1 The Single Resource Model

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

### 3.10.2 AND Model

- In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

Consider the example WFG described in the Figure 1.

- $P_{11}$ has two outstanding resource requests. In case of the AND model, $P_{11}$shall become active from idle state only after both the resources are granted.
- There is a cycle $P_{11}$->$P_{21}$->$P_{24}$->$P_{54}$->$P_{11}$ which corresponds to a deadlock situation.
- That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process $P_{44}$ in Figure 1.
- It is not a part of any cycle but is still deadlocked as it is dependent on $P_{24}$which is deadlocked.

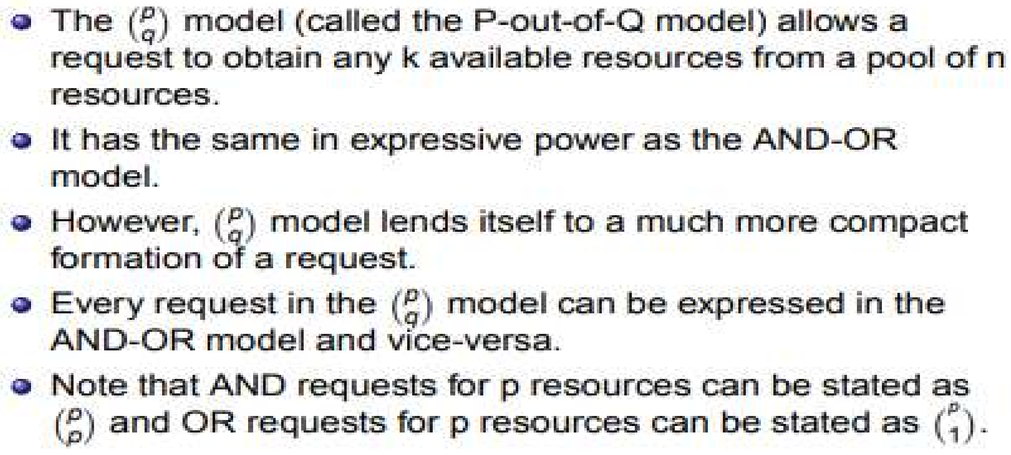
### 3.10.3 OR Model

- In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- Consider example in Figure 1: If all nodes are OR nodes, then process $P_{11}$ is not deadlocked because once process $P_{33}$ releases its resources, $P_{32}$ shall become active as one of its requests is satisfied.
- After $P_{32}$ finishes execution and releases its resources, process $P_{11}$ can continue with its processing.
- In the OR model, the presence of a knot indicates a deadlock.

### 3.10.4 AND – OR Model

- A generalization of the previous two models (OR model and AND model) is the AND-OR model.
- In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request.
- For example, in the AND-OR model, a request for multiple resources can be of the form x *and* (y *or* z).
- To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.
- Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

### 3. 10.5 The $\binom{p}{q}$ model (called the P-out-of-Q model)



- The $\binom{p}{q}$ model (called the P-out-of-Q model) allows a request to obtain any k available resources from a pool of n resources.
- It has the same in expressive power as the AND-OR model.
- However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request.
- Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as $\binom{p}{p}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

### 3.10.6 Unrestricted model

- In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests.
- Only one assumption that the deadlock is stable is made and hence it is the most general model.
- This model helps separate concerns: Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).

## 3.11 KNAPP'S CLASSIFICATION

Distributed deadlock detection algorithms can be divided into four classes:

- Path-Pushing
- Edge-Chasing
- Diffusion Computation
- Global State Detection.

### 3.11.1 Path-Pushing Algorithms

- In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG.
- The basic idea is to build a global WFG for each site of the distributed system.
- In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites.
- After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.
- This feature of sending around the paths of global WFG has led to the term path-pushing algorithms.

### 3.11.2 Edge-Chasing Algorithms

- In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is be verified by propagating special messages called probes, along the edges of the graph.
- These probe messages are different than the request and reply messages.
- The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
- Whenever a process that is executing receives a probe message, it discards this message and continues.
- Only blocked processes propagate probe messages along their outgoing edges.
- Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.

### 3.11.3 Diffusing Computations Based Algorithms

- In *diffusion computation* based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system.
- These algorithms make use of echo algorithms to detect deadlocks.
- This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock.
- To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
- These queries are successively propagated (i.e., diffused) through the edges of the WFG.

- When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent.
- For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message.
- The initiator of a deadlock detection detects a deadlock when it receives reply for every query it had sent out.

### 3.11.4 Global state detection based deadlock detection algorithms

- Global state detection based deadlock detection algorithms exploit the following facts:
- A consistent snapshot of a distributed system can be obtained without freezing the underlying computation and
- If a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.
- Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock.

## 3.12 MITCHELL AND MERRITT’S ALGORITHM FOR THE SINGLE-RESOURCE MODEL

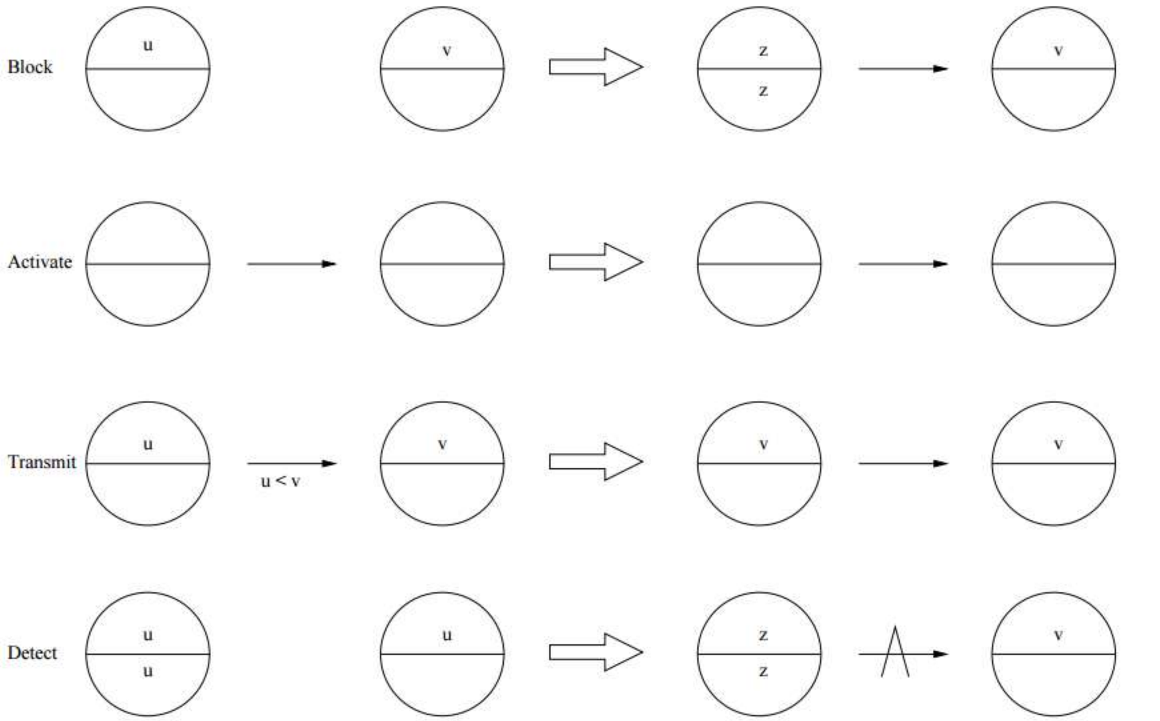
- Belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG.
- When a probe initiated by a process comes back to it, the process declares deadlock.
- Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock.

- Each node of the WFG has two local variables, called labels:
  - **a private label, which is unique to the node at all times, though it is not constant, and**
  - **a public label, which can be read by other processes and which may not be unique.**
- Each process is represented as u/v where u and u are the public and private labels, respectively.
- Initially, private and public labels are equal for each process.
- A global WFG is maintained and it defines the entire state of the system.

- The algorithm is defined by the four state transitions shown in Figure 2, where z = inc(u, v), and inc(u, v) yields a unique label greater than both u and v labels that are not shown do not change.

- Block creates an edge in the WFG.
- Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for.
- Activate denotes that a process has acquired the resource from the process it was waiting for.
- Transmit propagates larger labels in the opposite direction of the edges by sending a probe message.

**The four possible state transitions**



- Whenever a process receives a probe which is less then its public label, then it simply ignores that probe.
- Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
- The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted.

Message Complexity:

If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is s(s - 1)/2 Transmit steps, where s is the number of processes in the cycle.

## 3.13 CHANDY-MISRA-HAAS'S FOR AND MODEL

- Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model is based on edge-chasing.
- The algorithm uses a special message called *probe*, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process $P_i$ and it is being sent by the home site of process $P_j$ to the home site of process $P_k$ .
- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.
  - A process $P_j$ is said to be *dependent* on another process $P_k$ if there exists a sequence of processes $P_j$ , $P_{i1}$, $P_{i2}$, ..., $P_{im}$, $P_k$ such that each process except $P_k$ in the sequence is blocked and each process, except the $P_j$ , holds a resource for which the previous process in the sequence is waiting.
  - Process $P_j$ is said to be *locally dependent* upon process $P_k$ if $P_j$ is dependent upon $P_k$ and both the processes are on the same site.
  - Data Structures
  - Each process $P_i$ maintains a boolean array, *dependent*$_i$, where *dependent*$_i$(j) is true only if $P_i$ knows that $P_j$ is dependent on it.
  - Initially, *dependent*$_i$(j) is false for all i and j.

**Algorithm**

```
if P_i is locally dependent on itself
   then declare a deadlock
   else for all P_j and P_k such that
      (a)P_i is locally dependent upon P_j , and
      (b)P_j is waiting on P_k, and
      (c)P_j and P_k are on different sites,
       send a probe (i, j, k) to the home site of P_k

On the receipt of a probe (i, j, k), the site takes
   the following actions:

if

      (d)P_k is blocked, and
      (e)dependent_k i  is false, and
      (f)P_k has not replied to all requests P_j , then

         begin

            dependent_k i = true; if k = i
```

```
    then declare that $P_i$ is deadlocked else
  for all $P_m$ and $P_n$ such that

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

end.
```

- A probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

### Performance Analysis

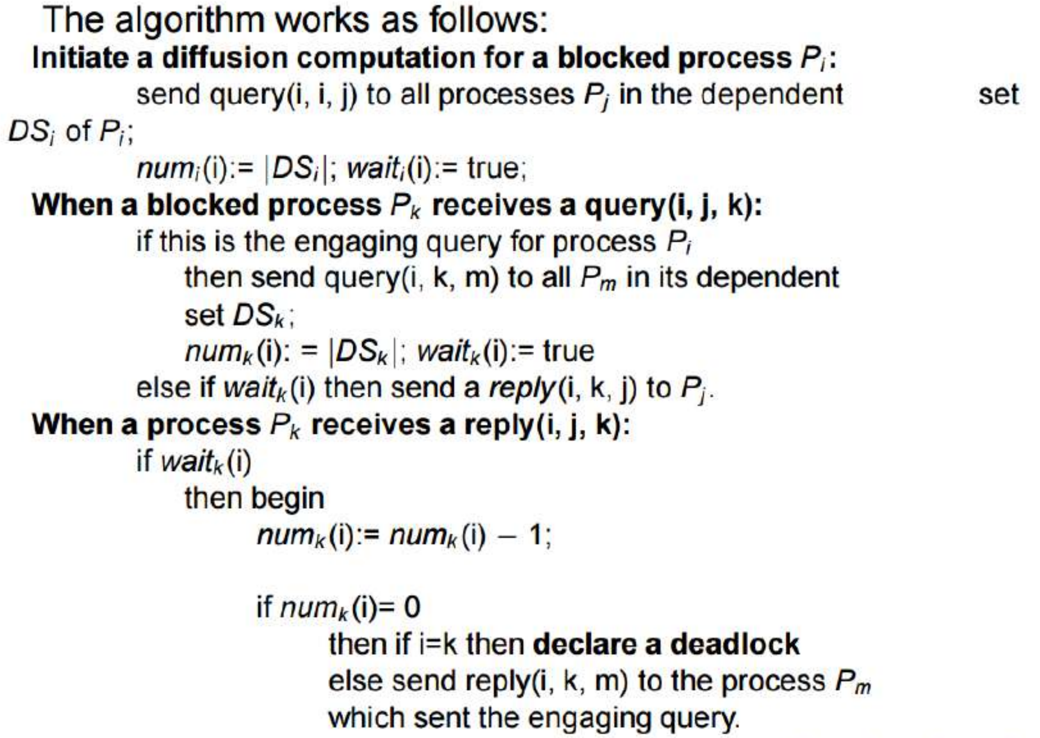
- One probe message (per deadlock detection initiation) is sent on every edge of the WFG which that two sites.
- Thus, the algorithm exchanges at most $m(n-1)/2$ messages to detect a deadlock that involves $m$ processes
- and that spans over $n$ sites.
- The size of messages is fixed and is very small (only 3 integer words).
- Delay in detecting a deadlock is $O(n)$.

## 3.14 CHANDY-MISRA-HAAS DISTRIBUTED DEADLOCK DETECTION ALGORITHM FOR OR MODEL

Chandy-Misra-Haas distributed deadlock detection algorithm for OR model is based on the approach of diffusion-computation.

- A blocked process determines if it is deadlocked by initiating a diffusion computation.
- **Two types of messages are used in a diffusion computation:**
- **query(i, j, k) and reply(i, j, k),** denoting that they belong to a diffusion computation initiated by a process $P_i$ and are being sent from process $P_j$ to process $P_k$ .
- A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set.
- If an active process receives a query or reply message, it discards it.
- When a blocked process $P_k$ receives a query(i, j, k) message, it takes the following actions:
  - If this is the first query message received by $P_k$ for the deadlock detection initiated by $P_i$ (called the *engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable $num_k$ (i) to the number of query messages sent.
  - If this is not the engaging query, then $P_k$ returns a reply message to it immediately provided $P_k$ has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

- Process $P_k$ maintains a boolean variable $wait_k$ (i) that denotes the fact that it has been continuously blocked since it received the last engaging query from process $P_i$ .
- When a blocked process $P_k$ receives a reply(i, j, k) message, it decrements $num_k$(i) only if $wait_k$(i) holds.
- A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query.
- The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.



The algorithm works as follows:

```
Initiate a diffusion computation for a blocked process $P_i$:
        send query(i, i, j) to all processes $P_j$ in the dependent            set
$DS_i$ of $P_i$;
        $num_i$(i):= |$DS_i$|; $wait_i$(i):= true;
When a blocked process $P_k$ receives a query(i, j, k):
        if this is the engaging query for process $P_i$
            then send query(i, k, m) to all $P_m$ in its dependent
            set $DS_k$;
            $num_k$(i): = |$DS_k$|; $wait_k$(i):= true
        else if $wait_k$(i) then send a reply(i, k, j) to $P_j$.
When a process $P_k$ receives a reply(i, j, k):
        if $wait_k$(i)
            then begin
                $num_k$(i):= $num_k$(i) − 1;

                if $num_k$(i)= 0
                    then if i=k then declare a deadlock
                    else send reply(i, k, m) to the process $P_m$
                    which sent the engaging query.
```

- In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process.
- However, messages for outdated diffusion computations may still be in transit.
- The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

**Performance Analysis**

For every deadlock detection, the algorithm exchanges e query messages and e reply messages, where e=n(n-1) is the number of edges.

## **PART** A

1. **Listout the basic approaches** for distributed mutual exclusion:

| |
|---|
| • Token based approach |
| • Non-token based approach |

- Quorum based approach

**2.** What is Token-based approach:

- A unique token is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Mutual exclusion is ensured because the token is unique.

**3.** What is Non-token based approach:

▲ Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.

**4.** What is the Quorum based approach:

▲ Each site requests permission to execute the CS from a subset of sites (called a quorum).

▲ Any two quorums contain a common site.

▲ This common site is responsible to make sure that only one request executes the CS at any time.

**5. Listout the Requirements of Mutual Exclusion Algorithms**

- **Safety Property**: At any instant, only one process can execute the critical section.
- **Liveness Property**: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
- **Fairness:** Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

**6. What are the Performance Metrics**

The performance is generally measured by the following **four metrics**:

- **Message complexity:** The number of messages required per CS execution by a site.
- **Synchronization delay**: After a site leaves the CS, it is the time required and before the next site enters the CS
- **Response time:** The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).
- **System throughput**: The rate at which the system executes requests for the CS.

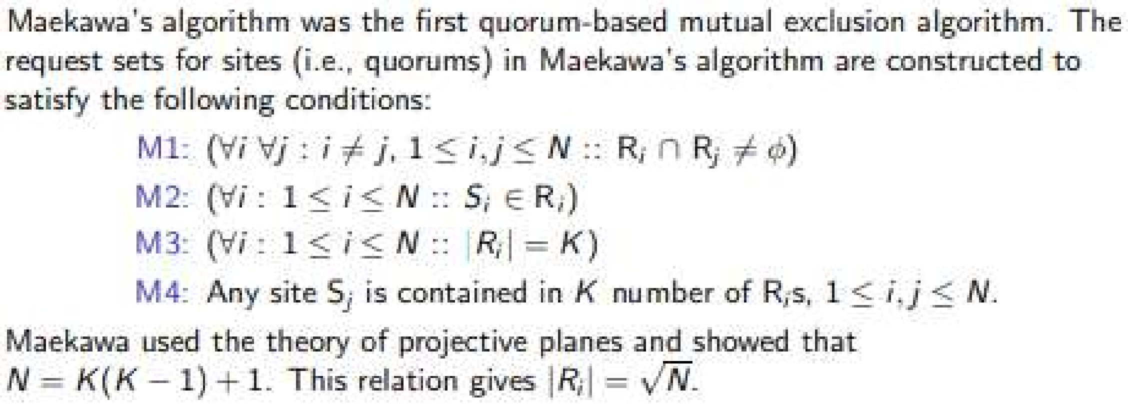
**7. Analyse the Performance of Lamport algorithm**

- For each CS execution, Lamport's algorithm requires $(N - 1)$ REQUEST messages, $(N - 1)$ REPLY messages, and $(N - 1)$ RELEASE messages. Thus, Lamport's algorithm requires $3(N - 1)$ messages per CS invocation.
- Synchronization delay in the algorithm is $T$.

**8.** How to optimize lamport algorithm

- In Lamport's algorithm,REPLY messages can be omitted in certain situations. For example, if site $S_j$ receives a REQUEST message from site $S_i$ after it has sent its own REQUEST message with timestamp higher than the timestamp of site $S_i$ 's request, then site $S_j$ need not send a REPLY message to site $S_i$ .
- This is because when site $S_i$ receives site $S_j$ 's request with timestamp higher than its own, it can conclude that site $S_j$ does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between $3(N - 1)$ and $2(N - 1)$ messages per CS execution.

**9.** List out the conditions of Maekawa's algorithm



Maekawa's algorithm was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$

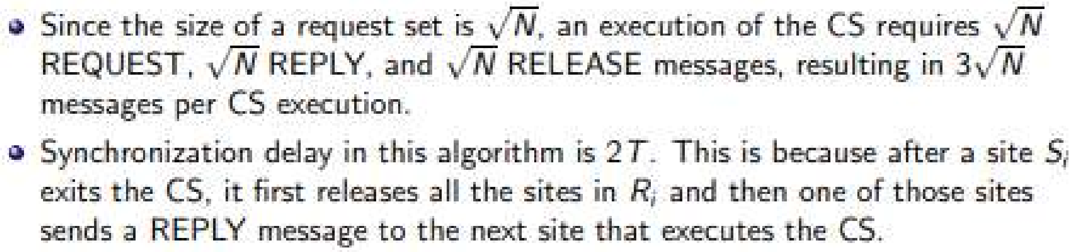
M2: $(\forall i :\ 1 \leq i \leq N :: S_i \in R_i)$

M3: $(\forall i :\ 1 \leq i \leq N :: |R_i| = K)$

M4: Any site $S_j$ is contained in $K$ number of $R_i$s, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that $N = K(K-1)+1$. This relation gives $|R_i| = \sqrt{N}$.

**10.** Analyse the performance of Maekawa's algorithm



- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution.
- Synchronization delay in this algorithm is $2T$. This is because after a site $S_i$ exits the CS, it first releases all the sites in $R_i$ and then one of those sites sends a REPLY message to the next site that executes the CS.

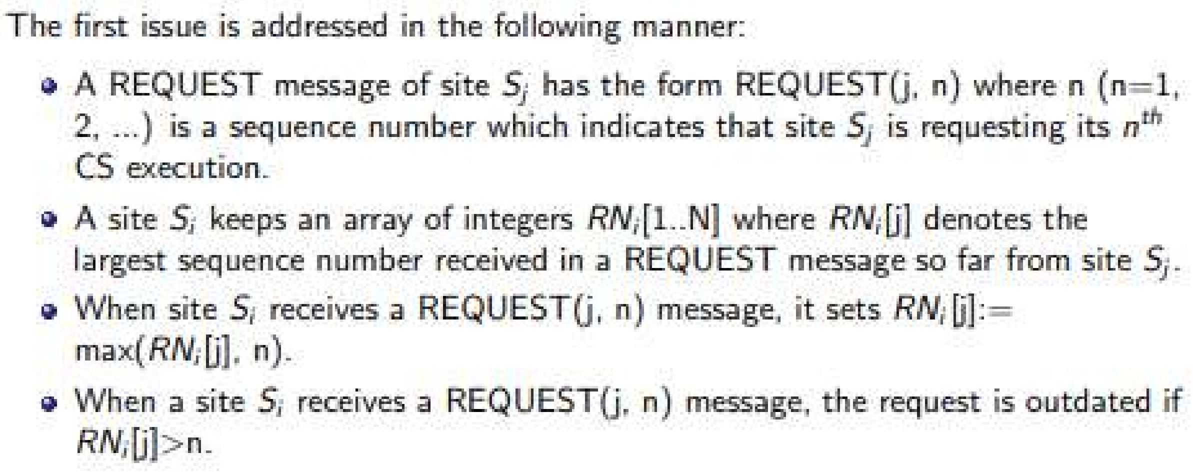
**11.** How to handle Deadlock in Maekawa's algorithm

FAILED: A FAILED message from site $S_i$ to site $S_j$ indicates that $S_i$ can not grant $S_j$ 's request because it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from $S_i$ to $S_j$ indicates that $S_i$ would like to find out from $S_j$ if it has succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site $S_i$ to $S_j$ indicates that $S_i$ is returning the permission to $S_j$ (to yield to a higher priority request at $S_j$ ).

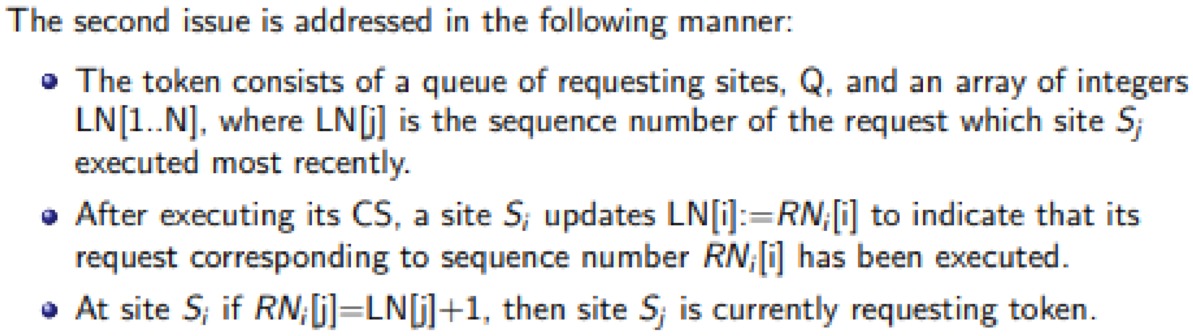
**12. How to distinguish an outdated REQUEST message from a current REQUEST message**



The first issue is addressed in the following manner:

- A REQUEST message of site $S_j$ has the form REQUEST(j, n) where n (n=1, 2, ...) is a sequence number which indicates that site $S_j$ is requesting its $n^{th}$ CS execution.
- A site $S_i$ keeps an array of integers $RN_i$[1..N] where $RN_i$[j] denotes the largest sequence number received in a REQUEST message so far from site $S_j$.
- When site $S_i$ receives a REQUEST(j, n) message, it sets $RN_i$[j]:= max($RN_i$[j], n).
- When a site $S_i$ receives a REQUEST(j, n) message, the request is outdated if $RN_i$[j]>n.

**13. How to determine which site has an outstanding request for the CS:**



The second issue is addressed in the following manner:

- The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site $S_j$ executed most recently.
- After executing its CS, a site $S_i$ updates LN[i]:=$RN_i$[i] to indicate that its request corresponding to sequence number $RN_i$[i] has been executed.
- At site $S_i$ if $RN_i$[j]=LN[j]+1, then site $S_j$ is currently requesting token.

**14. Listout the Deadlock Handling Strategies**

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

**15. What are the Issues in Deadlock Detection**

- Deadlock handling using the approach of deadlock detection entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.

- Detection of deadlocks involves addressing two issues: Maintenance of the WFG and searching of the WFG for the presence of cycles (or knots).

**16. What is Resolution of a Detected Deadlock**

- Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock.
- It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution.

**17.** Distributed deadlock detection algorithms can be divided into four classes:

- Path-Pushing

- Edge-Chasing
- Diffusion Computation
- Global State Detection.

**18.** Analyse the Message Complexity of single resource model

- If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is s(s - 1)/2 Transmit steps, where s is the number of processes in the cycle.

## PART B

**1.** Explain the Lamport‘s algorithm.

**2.** Discuss Ricart-Agrawala algorithm.

**3.** Explain Maekawa‘s algorithm.

**4.** Explain Suzuki–Kasami‘s broadcast algorithm.

**5.** How to detect Deadlock in distributed systems. Explain the system model

**6.** Discuss the Models of deadlocks

**7.** Explain the Knapp‘s classification.

**8.** Discuss the Algorithm for the single resource model.(**MITCHELL AND MERRITT’S ALGORITHM**

**9.** Discuss the Algorithm the AND model . (**CHANDY-MISRA-HAAS’S)**

**10.** Discuss the Algorithm OR model. (CHANDY-MISRA-HAAS)

## UNIT IV RECOVERY & CONSENSUS

Checkpointing and rollback recovery: Introduction – Background and definitions – Issues in failure recovery – Checkpoint-based recovery – Log-based rollback recovery – Coordinated checkpointing algorithm – Algorithm for asynchronous checkpointing and recovery. Consensus and agreement algorithms: Problem definition – Overview of results – Agreement in a failure – free system – Agreement in synchronous systems with failures.

## 4.1 Introduction

Rollback recovery treats a distributed system application as a collection of processes that communicate over a network. It achieves fault tolerance by periodically saving the state of a process during the failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work.

The saved state is called a ***checkpoint,*** and the procedure of restarting from a previously checkpointed state is called ***rollback recovery***. A checkpoint can be saved on either the stable storage or the volatile storage depending on the failure scenarios to be tolerated.

- In a distributed system, if each participating process takes its checkpoints independently, then the system is susceptible to the **domino effect**. This approach is called ***independent*** **or** ***uncoordinated checkpointing***.
- It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it. One such technique is ***coordinated check-pointing*** where processes coordinate their checkpoints to form a system-wide consistent state. In case of a process failure, the system state can be restored to such a consistent set of checkpoints, preventing the rollback propagation.
- Alternatively, ***communication-induced checkpointing*** forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes. Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

### ***Log-based*** **rollback recovery**

- The approaches discussed so far implement *checkpoint-based* rollback recovery, which relies only on checkpoints to achieve fault-tolerance**.** ***Log-based*** **rollback recovery** combines checkpointing with logging of non-deterministic events. Log-based rollback recovery relies on the ***piecewise deterministic*** **(PWD)** assumption, which postulates that all non-deterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's *determinant*.

- By logging and replaying the non-deterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed. Log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints. It is therefore particularly attractive for applications that frequently interact with the *outside world*, which consists of input and output devices that cannot roll back.

## Introduction

- **Rollback recovery protocols**

— restore the system back to a consistent state after a failure

— achieve fault tolerance by periodically saving the state of a process during the failure-free execution

— treats a distributed system application as a collection of processes that communicate over a network

**Checkpoints -> the saved states of a process**

**Why is rollback recovery of distributed systems complicated?**

messages induce inter-process dependencies during failure-free operation

**Rollback propagation**

The dependencies may force some of the processes that did not fail to roll back. This phenomenon is called "*domino effect*"

**If each process takes its checkpoints independently, then the system cannot avoid the domino effect**

This scheme is called independent or uncoordinated checkpointing

**Techniques that avoid domino effect**

➔ **Coordinated checkpointing rollback recovery**
processes coordinate their checkpoints to form a system-wide consistent state

➔ **Communication-induced checkpointing rollback recovery**
forces each process to take checkpoints based on information piggybacked on the application

➔ **Log-based rollback recovery**
combines checkpointing with logging of non-deterministic events relies on piecewise deterministic (PWD) assumption

## 4.2 Background and definitions

### 4.2.1 System model

Distributed system consists of a fixed number of processes, $P_1$, $P_2$ $P_N$ , which communicate only through messages. Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively. Figure shows a system consisting of three processes and interactions with the outside world.

Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication. Some protocols assume that the com-munication subsystem delivers messages reliably, in first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can

### 4.2.2 A local checkpoint

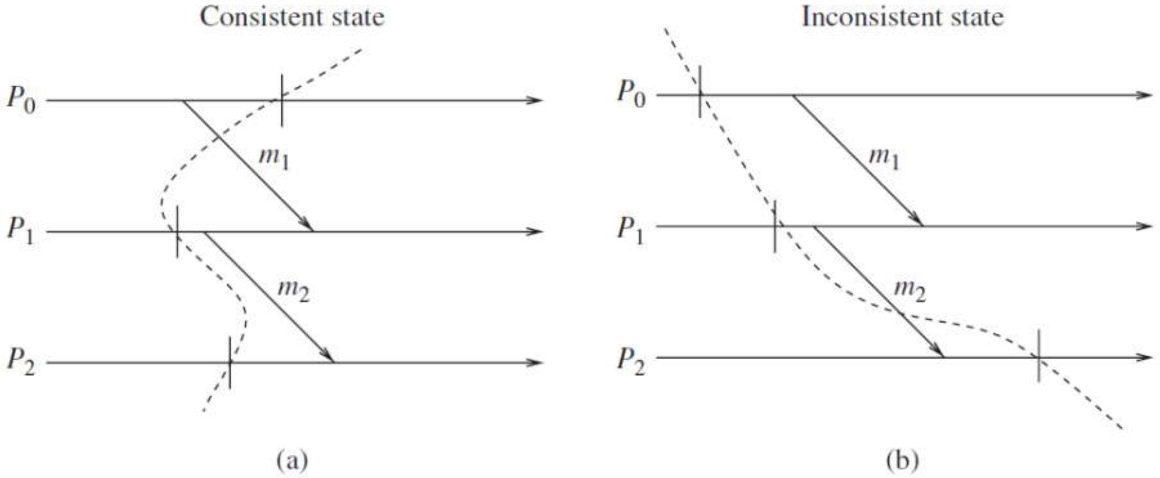
- In distributed systems, all processes save their local states at certain instants of time. This saved state is known as a local checkpoint.
- A local checkpoint is a snapshot of the state of the process at a given instance and the event of recording the state of a process is called local checkpointing.
- The contents of a checkpoint depend upon the application context and the checkpointing method being used.

Assumption
A process stores all local checkpoints on the stablestorage
A process is able to roll back to any of its existing localcheckpoints ,$k$
The $k$th local checkpoint at process is $C_{i,0}$
Aprocess $P_i$ takes a checkpoint $C_{i,0}$ before it startsexecution

### 4.2.3 Consistent system states

**A global state of a distributedsystem**
A global state of a distributed system is a collection of the individual states of all participating processes and the states of the communication channels.
**Consistent global state**

A consistent global state is one that may occur during a failure-free execution of a distributed computation. More precisely, a *consistent system state* is one in which a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of that message
**A globalcheckpoint**
a set of local checkpoints, one from eachprocess
**A consistent globalcheckpoint**
a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before takingits checkpoint
Consistent states – examples



- For instance, Figure shows two examples of global states. The state in Figure (a) is consistent and the state in Figure (b) is inconsistent. Note that the consistent state in Figure (a) shows message $m_1$ to have been sent but not yet received, but that is alright. The state in Figure (a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event.
- The state in Figure (b) is inconsistent because process $P_2$ is shown to have received $m_2$ but the state of process $P_1$ does not reflect having sent it. Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures. For instance, the situation shown in Figure (b) may occur if process $P_1$ fails after sending message $m_2$ to process $P_2$ and

then restarts at the state shown in Figure (b).

Thus, a local checkpoint is a snapshot of a local state of a process and a global checkpoint is a set of local checkpoints, one from each process. A consistent global checkpoint is a global checkpoint such that no message is sent by a process after taking its local checkpoint that is received by another process before taking its local checkpoint. The consistency of global checkpoints strongly depends on the flow of messages exchanged by processes and an arbitrary set of local checkpoints at processes may not form a consistent global checkpoint.

The fundamental goal of any rollback-recovery protocol is to bring the system to a consistent state after a failure. The reconstructed consistent state is not necessarily one that occurred before the failure. It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free execution, provided that it is consistent with the interactions that the system had with the outside world.

### 4.3.4 Interactions with outside world

**A distributed application often interacts with the outside world to receive input data or deliver the outcome of a computation**. If a failure occurs, the outside world cannot be expected to roll back. For example, a printer cannot roll back the effects of printing a character, and an automatic teller machine cannot recover the money that it dispensed to a customer.

- ➔ **A distributed system often interacts with the outside worldto receive input data or deliver the outcome of a computation**
- ➔ **Outside World Process (OWP)**
  a special process that interacts with the rest of the system through message passing

**A common approach**

save each input message on the stable storage before allowing the application program to process it

**Symbol "||"**

An interaction with the outside world to deliver the outcome of a computation

### 4.2.5 Different types of messages

i. **In-transit message ->messages that have been sent but not yet received**

   In Figure, the global state {$C_{1\,8}$ $C_{2\,9}$ $C_{3\,8}$ $C_{4\,8}$} shows that message $m_1$ has been sent but not yet received. We call such a message an *in-transit* message. Message $m_2$ is also an in-transit message.

ii. **Lost messages**

   Messages whose send is not undone but receive is undone due to rollback are called *lost* messages. This type of messages occurs when the process rolls back to a checkpoint prior to reception of the message while the sender does not rollback beyond the send operation of the message. In Figure, message $m_1$ is a lost message.

iii. **Delayed messages**

   Messages whose receive is not recorded because the receiving process was either down or the message arrived after the rollback of the receiving process, are called *delayed* messages. For example, messages $m_2$ and $m_5$ in Figure are delayed messages.

iv. ***orphan* messages**

   Messages with receive recorded but message send not recorded are called *orphan* messages. For example, a rollback might have undone the send of such messages, leaving the receive event intact at the receiving process. **Orphan messages do not arise if processes roll back to a consistent global state.**

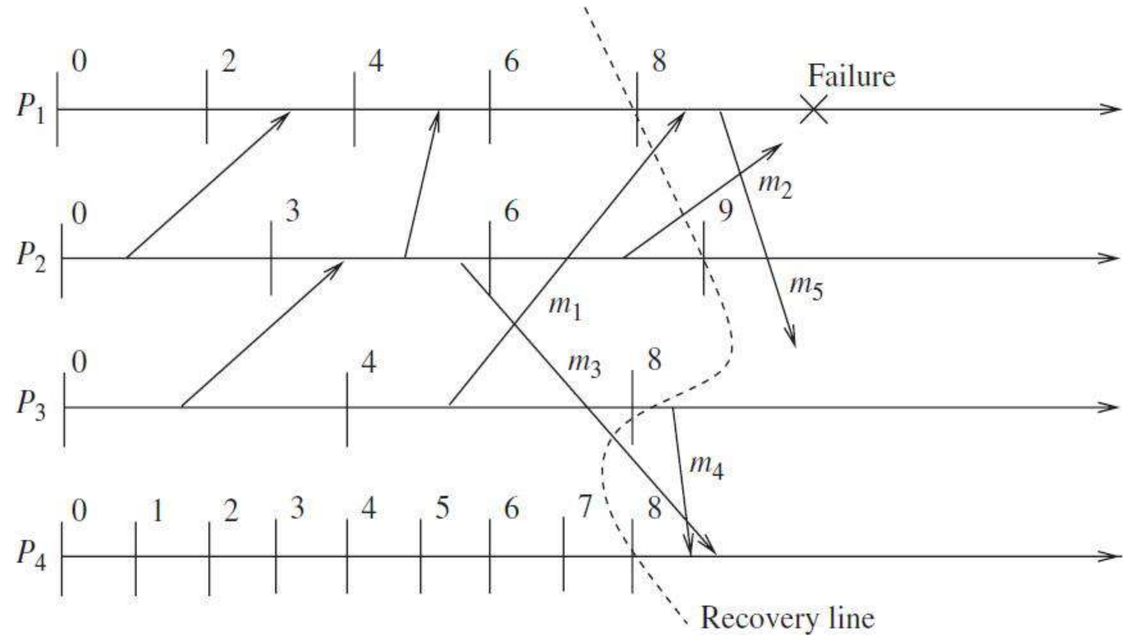
### v. Duplicate messages

Duplicate messages arise due to message logging and replaying during process recovery.

For example, in Figure, message $m_4$ was sent and received before the rollback. However, due to the rollback of process $P_4$ to $C_{4\,8}$ and process $P_3$ to $C_{3\,8}$, both send and receipt of message $m_4$ are undone. When process $P_3$ restarts from $C_{3\,8}$, it will resend message $m_4$. Therefore, $P_4$ should not replay message $m_4$ from its log. If $P_4$ replays message $m_4$, then message $m_4$ is called a *duplicate* message.

Message $m_5$ is an excellent example of a duplicate message. No matter what, the receiver of $m_5$ will receive a duplicate $m_5$ message.

Messages – example



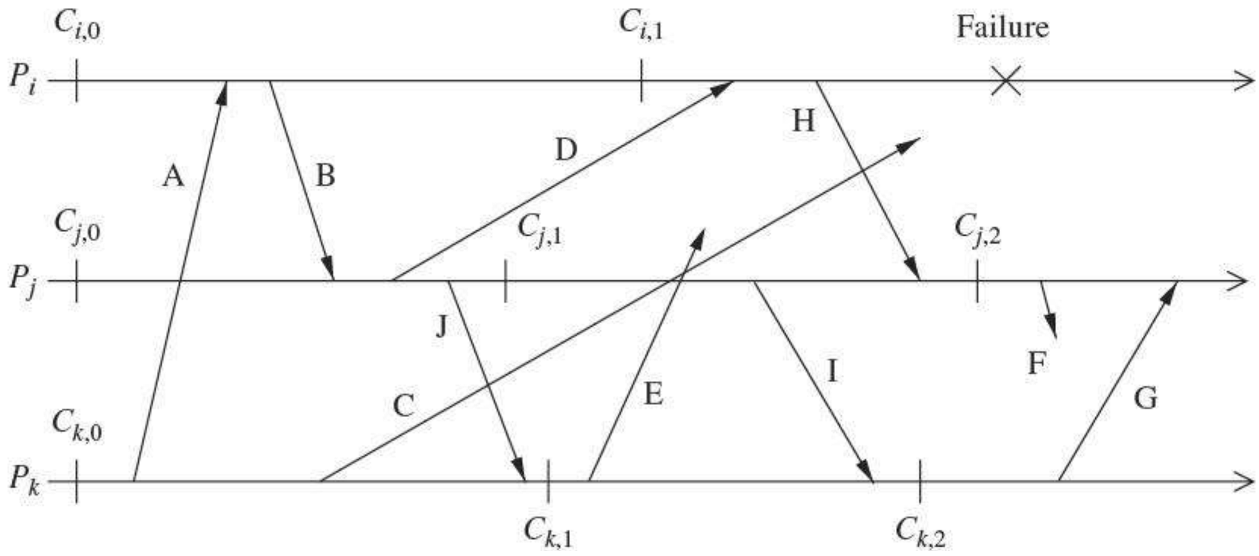
- In-transit – $m1$, $m2$
- Lost – $m1$
- Delayed – $m1$, $m5$
- Orphan – none
- Duplicated – $m4$, $m5$

## 4.3 Issues in failure recovery

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery.

The computation comprises of three processes $P_i$, $P_j$, and $P_k$, connected through a communication network. The processes communicate solely by exchanging messages over fault-free, FIFO communication channels. Processes $P_i$, $P_j$ , and $P_k$ have taken check-points $\{C_{i\,0}, C_{i\,1}\}$, $\{C_{j\,0}, C_{j\,1},$

$C_{j\ 2}$}, and {$C_{k\ 0}$, $C_{k\ 1}$}, respectively, and these processes have exchanged messages A to J as shown in Figure.



- Checkpoints : {$C_{i,0}$, $C_{i,1}$}, {$C_{j,0}$, $C_{j,1}$,$C_{j,2}$}, and {$C_{k,0}$, $C_{k,1}$,$C_{k,2}$}
- Messages : A -J
- The restored global consistent state : {$C_{i,1}$, $C_{j,1}$,$C_{k,1}$}

**The rollback of process to checkpoint $C_{i,1}$ created an orphan message H**

- Orphan message I is created due to the roll back of process $P_j$ to checkpoint $C_{j\ 1}$
- Messages C, D, E, and F are potentially problematic

— Message C: a delayed message

— Message D: a lost message since the send event for D is recorded in the restored state for process $P_j$ , but the receive event has been undone at process $P_i$.

- Lost messages can be handled by having processes keep a message log of all the sent messages

Messages E, F: delayed orphan messages. After resuming execution from their checkpoints, processes will generate both of these messages

## 4.4 Checkpoint-based recovery

In the checkpoint-based recovery approach, the state of each process and the communication channel is check pointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints. It does not rely on the PWD assumption, and so does not need to detect, log, or replay non-deterministic events. Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback recovery. However, checkpoint-based rollback recovery does not guarantee that pre-failure execution can be deterministically regenerated after a rollback. There-fore, checkpoint-based rollback recovery may not be suitable for applications that require frequent interactions with the outside world.

Checkpoint-based rollback-recovery techniques can be classified into three categories:

- *uncoordi-nated checkpointing*,
- *coordinated checkpointing*, and
- *communication-induced checkpointing*

### 4.4.1 Uncoordinated Checkpointing

**Each process has autonomy in deciding when to take checkpoints**

- Advantages

— The lower runtime overhead during normal execution

- **Disadvantages**

— Domino effect during a recovery

— Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints

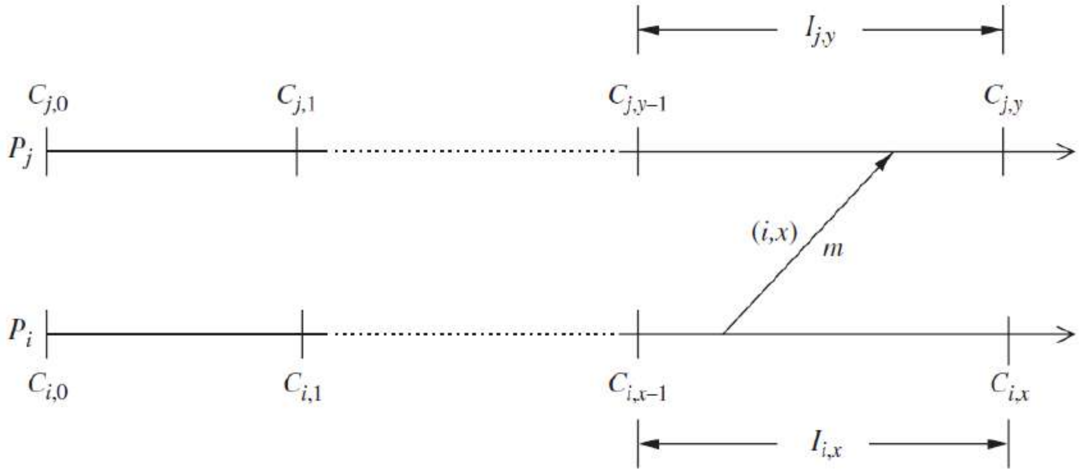
— Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm

— Not suitable for application with frequent output commits

- **The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation**

**Direct dependency tracking technique**

Let $C_{i\,x}$ be the *x*th checkpoint of process $P_i$, where i is the process i.d. and x is the checkpoint index (we assume each process $P_i$ starts its execution with an initial checkpoint $C_{i\,0}$). Let $I_{i\,x}$ denote the *checkpoint interval* or simply *interval* between checkpoints $C_{i\,x-1}$ and $C_{i\,x}$.



- When a failure occurs, the recovering process initiates rollback by broad-casting a *dependency request* message to collect all the dependency information maintained by each process. When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state.

- The initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line. Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

### 4.4.2 Coordinated checkpointing

- In coordinated checkpointing, processes orchestrate their checkpointing activ-ities so that all local checkpoints form a consistent global state. Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint.
- Also, coordinated checkpointing requires each process to maintain only one checkpoint on the stable storage, reducing the storage overhead and eliminating the need for garbage collection. The main disadvantage of this method is that large latency is involved in committing output, as a global checkpoint is needed before a message is sent to the OWP. Also, delays and overhead are involved everytime a new global checkpoint is taken.
- If perfectly synchronized clocks were available at processes, the following simple method can be used for checkpointing: all processes agree at what instants of time they will take checkpoints, and the clocks at processes trigger the local checkpointing actions at all processes. Since perfectly synchronized clocks are not available, the following approaches are used to guarantee checkpoint consistency: either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking.

**Blocking Checkpointing**

- A straightforward approach to coordinated checkpointing is to block commu-nications while the checkpointing protocol executes. After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete.
- The coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives this message, it stops its execution, flushes all the communication channels, takes a *tentative* checkpoint, and sends an acknowledgment message back to the coordinator. After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol.
- After receiving the commit message, a process removes the old permanent checkpoint and atomically makes the *tentative* checkpoint permanent and then resumes its execution and exchange of messages with other processes. A problem with this approach is that the computation is blocked during the checkpointing and therefore, non-blocking checkpointing schemes are preferable.

— After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete

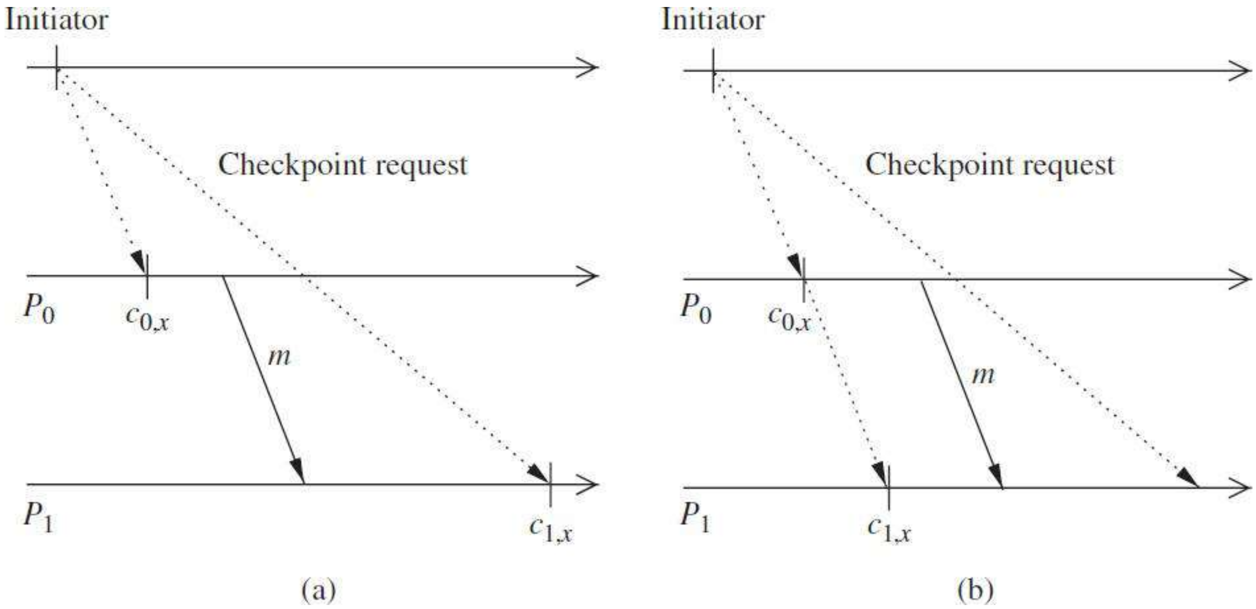
— **Disadvantages**

- the computation is blocked during the checkpointing

**Non-blocking Checkpointing**

In this approach the processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to pre-vent a process from receiving application messages that could make the checkpoint inconsistent.
Consider the example in Figure (a): message *m* is sent by $P_0$ *after* receiving a checkpoint request from the checkpoint coordinator. Assume *m* reaches $P_1$ *before* the checkpoint request. This situation results in an inconsistent checkpoint since checkpoint $c_{1\ x}$ shows the receipt of message *m* from $P_0$, while checkpoint $c_{0\ x}$ does not show *m* being sent from $P_0$.

**If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message, as illustrated in Figure 13.6(b).**



**Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) a solution with FIFO channels**

- The processes need not stop their execution while taking checkpoints
- A fundamental problem in coordinated check pointing is to preventa process from receiving application messages that could make the checkpoint inconsistent.

- **Example (a) : checkpoint inconsistency**

message *m* is sent by $P_0$ *after* receiving a checkpoint request from the checkpoint coordinator. Assume *m* reaches $P_1$ *before* the checkpoint request. This situation results in an inconsistent checkpoint since checkpoint $c_{1\ x}$ shows the receipt of message *m* from $P_0$, while checkpoint $c_{0\ x}$ does not show *m* being sent from $P_0$.

- **Example (b) : a solution with FIFO channels**

If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message

### 4.4.3 Impossibility of min-process non-blocking checkpointing

A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation. Clearly, such checkpointing algorithms will be very attractive. Cao and Singhal showed that it is impossible to design a min-process, non-blocking checkpointing algorithm.

The following type of min-process checkpointing algorithms are possible. The algorithm consists of two phases.

- During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request. Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified.
- During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes. In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

Based on a concept called "**Z-dependency**," Cao and Singhal proved that there does not exist a non-blocking algorithm that will allow a minimum number of processes to take their checkpoints. Here we give only a sketch of the proof and readers are referred to the original source for a detailed proof.

**Z-dependency is defined as follows**: if a process $P_p$ sends a message to process $P_q$ during its ith checkpoint interval and process $P_q$ receives the message during its jth checkpoint interval, then $P_q$ Z-depends on $P_p$ during $P_p$'s ith checkpoint interval and $P_q$ 's jth checkpoint interval, denoted by $P_p \rightarrow^{i}_{j} P_q$ . If $P_p \rightarrow^{i}_{j} P_q$ and $P_q \rightarrow^{j}_{k} P_r$ , then $P_r$ transitively Z-depends depends on $P_p$ during $P_r$ 's kth checkpoint interval and $P_p$'s ith checkpoint interval, and this is denoted as $P_p \,{}^{*}\!\!\rightarrow^{i}_{k} P_r$ .

A min process algorithm is one that satisfies the following condition: when a process $P_p$ initiates a new checkpoint and takes checkpoint $C_{p\,i}$, a process $P_q$ takes a checkpoint $C_{q\,j}$ associated with $C_{p\,i}$ if and only if $P_q \,{}^{*}\!\!\rightarrow^{j-1}_{i-1} P_p$. In a min-process non-blocking algorithm, process $P_p$ initiates a new checkpoint and takes a checkpoint $C_{p\,i}$ and if a process $P_r$ sends a message m to $P_q$ after it takes a new checkpoint associated with $C_{p\,i}$, then $P_q$ takes a checkpoint $C_{q\,i}$ before processing m if and only if $P_q \,{}^{*}\!\!\rightarrow^{j-1}_{i-1} P_p$. According to the min-process definition, $P_q$ takes checkpoint $C_{q\,j}$ if and only if $P_q \,{}^{*}\!\!\rightarrow^{j-1}_{i-1} P_p$, but $P_q$ should take $C_{q\,i}$ before processing m. If it takes $C_{q\,j}$ after processing m, m becomes an orphan. Therefore, when a process receives a message m, it must know if the initiator of a new checkpoint transitively Z-depends on it during the previous checkpoint interval. But it has been proved that there is not enough information at the receiver of a message to decide whether the initiator of a new checkpoint transitively Z-depends on the receiver. Therefore, no min-process, non-blocking algorithm exists.

### 4.4.4 Communication-induced Checkpointing

*Communication-induced checkpointing* is another way to avoid the domino effect, while allowing processes to take some of their checkpoints inde-pendently. Processes may be forced to take additional checkpoints (over and above their autonomous checkpoints), and thus process independence is constrained to guarantee the eventual progress of the recovery line. Communication-induced checkpointing reduces or completely eliminates the useless checkpoints.

- **Two types of checkpoints**
  - autonomous and forced checkpoints
- **Communication-induced checkpointing piggybacks protocol- related information on each application message**
- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line
- The forced checkpoint must be taken before the application may process the contents of the message
- In contrast with coordinated checkpointing, no special coordination messages are exchanged
- Two types of communication-induced checkpointing
  - **(i) model-based checkpointing and**
  - **(ii) index-based checkpointing.**

### Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and check-points that could result in inconsistent states among the existing checkpoints. \
- A process detects the potential for inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns.
- A forced checkpoint is generally used to prevent the undesirable patterns from occurring. No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggy-backed on application messages. The decision to take a forced checkpoint is done locally using the information available.

### Index-based checkpointing

- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.
- Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint.

## 4.5 Log-based rollback recovery

- **A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.**

### 4.5.1 Deterministic and Non-deterministic events

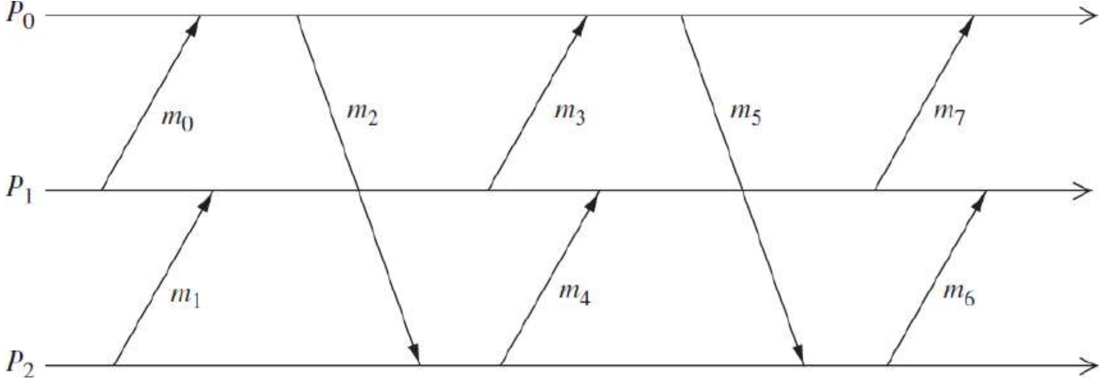
— Non-deterministic events can be the receipt of a message from another process or an event internal to the process

— a message send event is *not* a non-deterministic event.

The execution of process $P_0$ is a sequence of four deterministic intervals. The first one starts with the creation of the process, while the remaining three start with the receipt of messages $m_0$, $m_3$, and $m_7$, respectively. Send event of message $m_2$ is uniquely determined by the initial state of $P_0$ and by the receipt of message $m_0$, and is therefore not a non-deterministic event.

— Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage

— During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage



No-orphans consistency condition

- **Let *e* be a non-deterministic event that occurs at process *p***

***Depend*(*e*) ->** the set of processes that are affected by a non-deterministic event *e*.This set consists of *p*, and any process whose state depends on the event *e* according to Lamport's *happened before* relation

***Log*(*e*) ->** the set of processes that have logged a copy of *e*'s determinant in their volatile memory

***Stable*(*e*) ->** a predicate that is true if *e*'s determinant is logged on the stable storage

- ***always-no-orphans* condition**

$$\forall (e) : \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$$

### 4.5.2 Pessimistic Logging

**Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation**

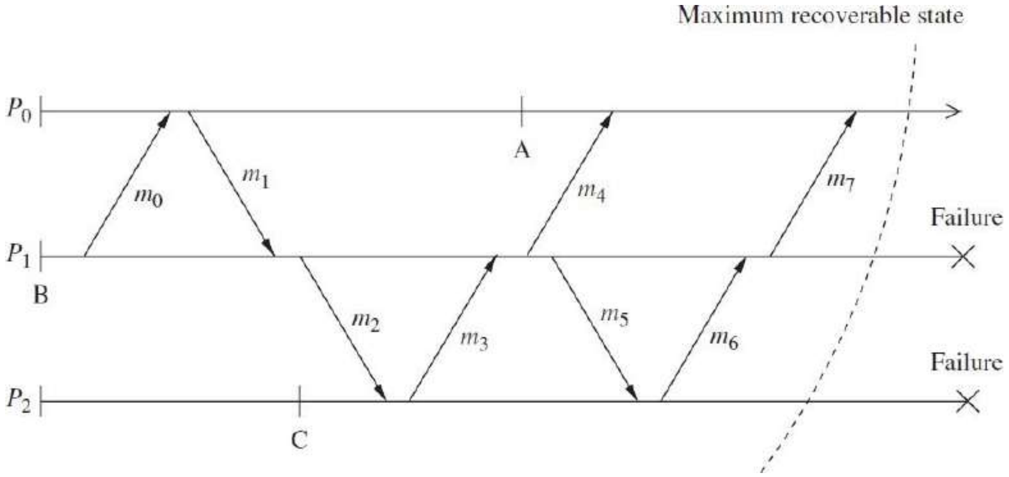
- However, in reality failures are rare

***synchronous logging***

$$\forall e: \neg \mathbf{Stable(e)} \Rightarrow |\mathbf{Depend(e)}| = 0$$

— if an event has not been logged on the stable storage, then no process ca n depend on it.

— stronger than the always-no-orphans condition



Suppose processes $P_1$ and $P_2$ fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to deliver again the same sequence of messages as in the pre-failure execution. This guarantees that $P_1$ and $P_2$ will repeat exactly their pre-failure execution and re-send the same messages. Hence, once the recovery is complete, both processes will be consistent with the state of $P_0$ that includes the receipt of message $m_7$ from $P_1$. In a pessimistic logging system, the observable state of each process is always recoverable.

## 4.5.3 Optimistic Logging

- In optimistic logging protocols, processes log determinants *asynchronously* to the stable storage . These protocols optimistically assume that logging will be complete before a failure occurs. Determinants are kept in a volatile log, and are periodically flushed to the stable storage. Thus, optimistic logging does not require the application to block waiting for the determinants to be written to the stable storage, and therefore incurs much less overhead during failure-free execution.
- However, the price paid is more complicated recovery, garbage collection, and slower output commit. If a process fails, the determinants in its volatile log are lost, and the state intervals that were started by the non-deterministic events corresponding to these determinants cannot be recovered.
- Furthermore, if the failed process sent a message during any of the state intervals that cannot be recovered, the receiver of the message becomes an orphan process and must roll back to undo the effects of receiving the message.

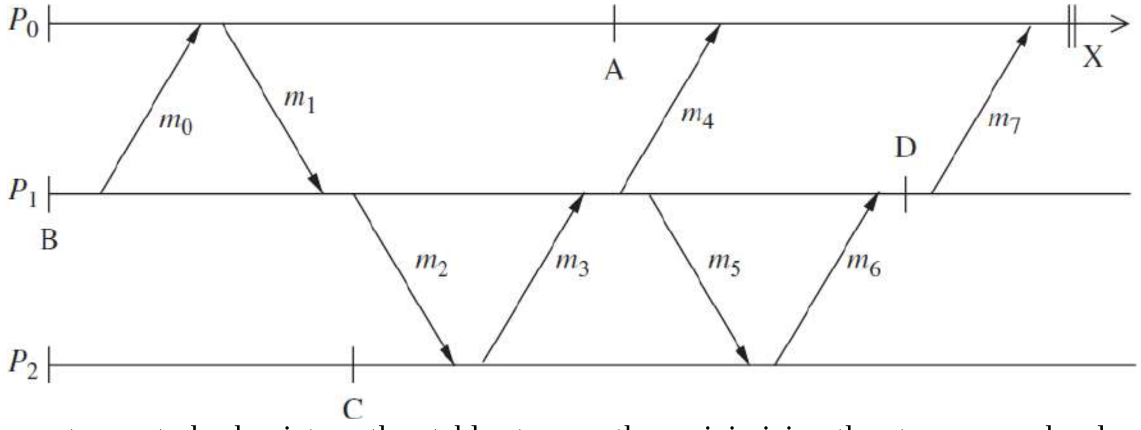
**To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution**

- Optimistic logging protocols require a non-trivial garbage collect ion scheme

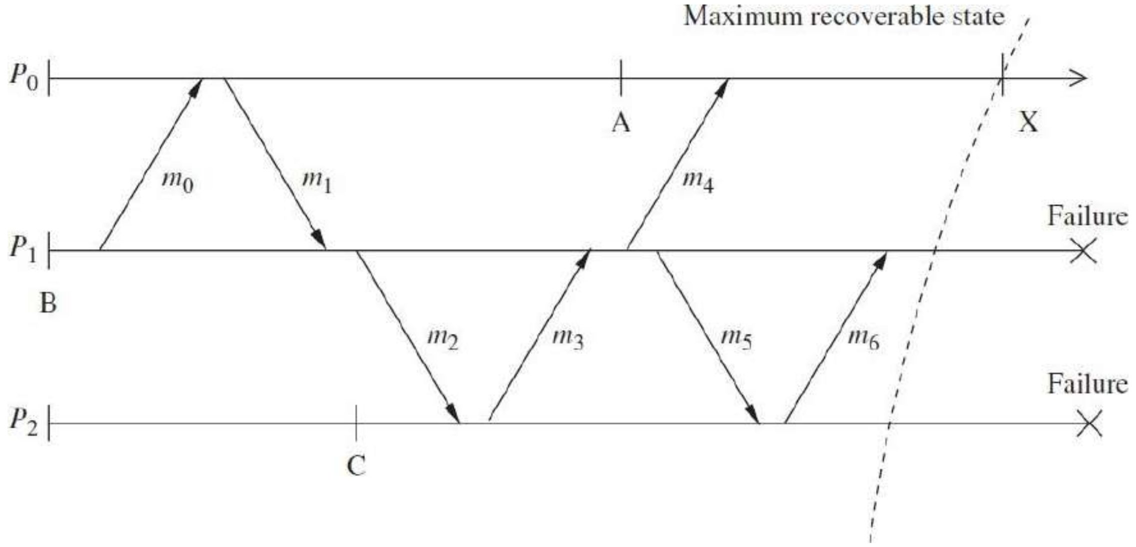
- Pessimistic protocols need only keep the most recent checkpoint of each process, whereas optimistic protocols may need to keep multiple checkpoints for each process

### 4.5.4 Causal Logging

Combines the advantages of both pessimistic and optimistic logging at the expense of a more complex recovery protocol. Like optimistic logging, it does not require synchronous access to the stable storage except during output commit. Like pessimistic logging, it allows each process to commit output independently and never creates orphans, thus isolating processes from the effects of failures at other processes. Moreover, causal logging limits the rollback of any failed process to the most recent checkpoint on the stable storage, thus minimizing the storage overhead and the amount of lost work.



- Make sure that the always-no-orphans property holds
- Each process maintains information about all the events that have causally affected its state



## 4.6 Koo-Toueg coordinated checkpointing algorithm

- A coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and avoids domino effect and livelock problems during the recovery

- Includes 2 parts: the checkpointing algorithm and the recovery algorithm

### 4.6.1 Checkpointing algorithm

– Assumptions: FIFO channel, end-to-end protocols, communication failures do not partition the network, single process initiation, no process fails during the execution of the algorithm

**Two kinds of checkpoints: permanent and tentative**

- Permanent checkpoint: local checkpoint, part of a consistent global checkpoint
- Tentative checkpoint: temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully

**Checkpointing algorithm**

***2 phases***

- The initiating process takes a tentative checkpoint and requests all other processes to take tentative checkpoints. Every process can not send messages after taking tentative checkpoint. All processes will finally have the single same decision: do or discard
- All processes will receive the final decision from initiating process and act accordingly

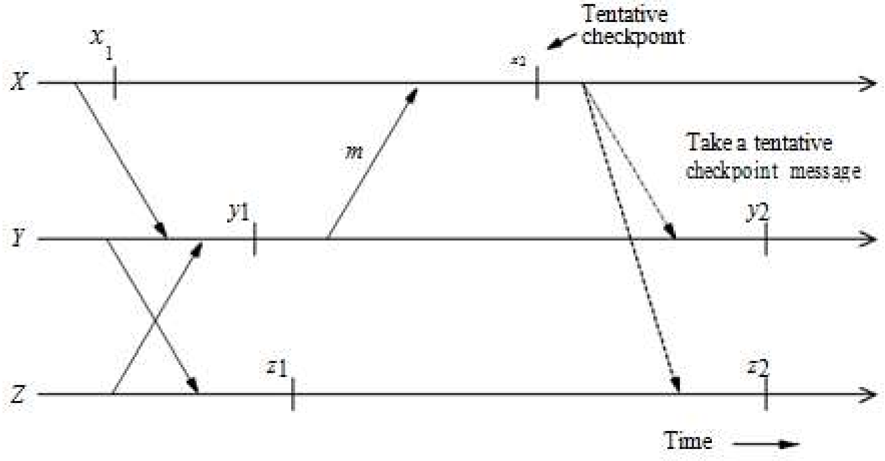
***Correctness: for 2 reasons***

- Either all or none of the processes take permanent checkpoint
- No process sends message after taking permanent checkpoint

***Optimization: maybe not all of the processes need to take checkpoints (if not change since the last checkpoint)***

**The rollback recovery algorithm**

- Restore the system state to a consistent state after a failure with assumptions: single initiator, checkpoint and rollback recovery algorithms are not invoked concurrently



Example of checkpoints taken unnecessarily

**2 phases**

*First phase*

An initiating process $P_i$ sends a message to all other processes to check if they all are willing to restart from their previous checkpoints. A process may reply "no" to a restart request due to any reason (e.g., it is already participating in a checkpoint or recovery process initiated by some other process). If $P_i$ learns that all processes are willing to restart from their previous checkpoints, $P_i$ decides that all processes should roll back to their previous checkpoints. Otherwise, $P_i$ aborts the rollback attempt and it may attempt a recovery at a later time.

*Second phase*

$P_i$ propagates its decision to all the processes. On receiving $P_i$'s decision, a process acts accordingly.

During the execution of the recovery algorithm, a process cannot send messages related to the underlying computation while it is waiting for $P_i$'s decision.

**Correctness**

All processes restart from an appropriate state because, if they decide to restart, they resume execution from a consistent state (the checkpointing algorithm takes a consistent set of checkpoints).

**An optimization**

The above recovery protocol causes all processes to roll back irrespective of whether a process needs to roll back or not. Consider the example shown in Figure. In the event of failure of process X, the above protocol will require processes X, Y, and Z to restart from checkpoints $x_2$, $y_2$, and $z_2$, respectively. However, note that process Z need not roll back because there has been no interaction between process Z and the other two processes since the last checkpoint at Z.

## 4.7 Juang-Venkatesan algorithm for asynchronous checkpointing and recovery

- Assumptions: communication channels are reliable, delivery messages in FIFO order, infinite buffers, message transmission delay is arbitrary but finite
- Underlying computation/application is event-driven: process P is at state s, receives message m, processes the message, moves to state s'and send messages out. So the triplet (*s, m, msgs_sent*) represents the state of P

**Two type of log storage are maintained:**

– **Volatile log**: short time to access but lost if processor crash. Move to stable log periodically.

– **Stable log**: longer time to access but remained if crashed

**Asynchronous checkpointing:**

- After executing an event, the triplet is recorded without any synchronization with other processes.
- Local checkpoint consist of set of records, first are stored in volatile log, then moved to stable log.

**Recovery algorithm**

**Notation and data structure**

The following notation and data structure are used by the algorithm:

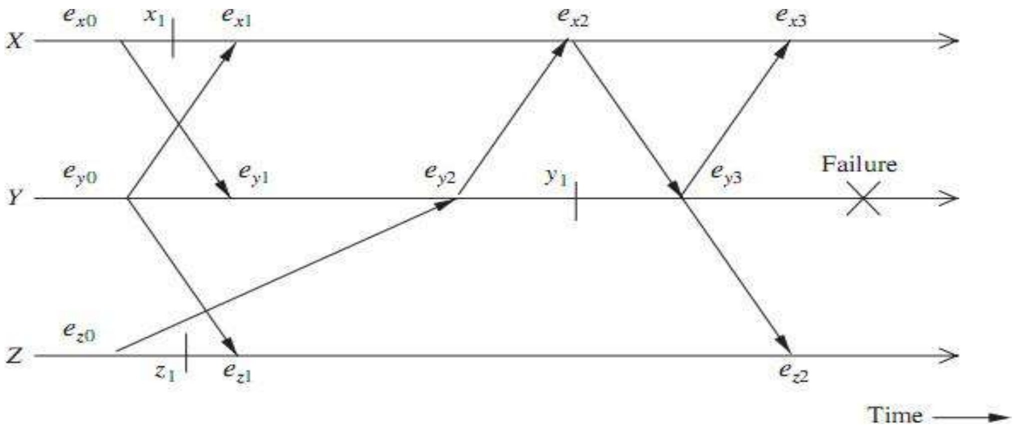
- **$RCVD_{i \leftarrow j}$ $CkPt_i$** represents the number of messages received by processor $p_i$ from processor $p_j$ , from the beginning of the computation until the checkpoint $CkPt_i$.

- **$SENT_{i \rightarrow j}$ $CkPt_i$** represents the number of messages sent by processor $p_i$ to processor $p_j$ , from the beginning of the computation until the checkpoint $CkPt_i$.

Idea:

- From the set of checkpoints, find a set of consistent checkpoints
- Doing that based on the number of messages sent and received

**Example**



**<u>Procedure RollBack_Recovery:</u>**

**processor $p_i$ executes the following:**

```
STEP (a)

if processor $p_i$ is recovering after a failure then $CkPt_i$ = latest event
   logged in the stable storage
else

   $CkPt_i$ = latest event that took place in $p_i$ {The latest event at $p_i$ can be either in stable or
   in volatile storage.}

end if

STEP (b)

for k = 1 to N {N is the number of processors in the system} do for each
   neighboring processor $p_j$ do

      compute $SENT_{i \rightarrow j}$  $CkPt_i$

      send a ROLLBACK i $SENT_{i \rightarrow j}$ $CkPt_i$ message to $p_j$ end for

   for every ROLLBACK j c  message received from a neighbor j do
```

```
      if $RCVD_{i \leftarrow j}$  $CkPt_i$  > c {Implies the presence of orphan messages}
        then
        find the latest event e such that $RCVD_{i \leftarrow j}$ e = c {Such an event e may be in the
        volatile storage or stable storage.}

        $CkPt_i$  = e
      end if

    end for

  end for{for k}
```

## 4.7 Consensus and Agreement

### 4.7.1 Assumptions

Assumptions underlying our study of agreement algorithms:

Failure models Among the n processes in the system, at most f processes can be faulty. A faulty process can behave in any manner allowed by the failure model assumed.

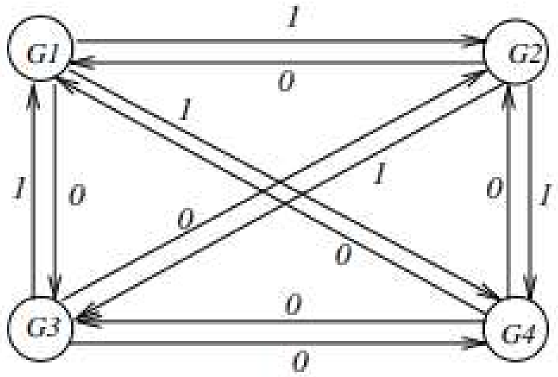
Synchronous/asynchronous communication If a failure-prone process chooses to send a message to process $P_i$ but fails, then $P_i$ cannot detect the non-arrival of the message in an asynchronous system because this scenario is indistinguishable from the scenario in which the message takes a very long time in transit.

- **Network connectivity** The system has full logical connectivity, i.e., each process can communicate with any other by direct message passing.

- **Sender identification** A process that receives a message always knows the identity of the sender process. This assumption is important – because even with Byzantine behavior, even though the payload of the message can contain fictitious data sent by a malicious sender, the underlying network layer protocols can reveal the true identity of the sender process.
- **Channel reliability** The channels are reliable, and only the processes may fail (under one of various failure models). This is a simplifying assumption in our study. As we will see even with this simplifying assumption, the agreement problem is either unsolvable, or solvable in a complex manner.

- **Authenticated vs. non-authenticated messages** In our study, we will be dealing only with *unauthenticated* messages. With unauthenticated mes-sages, when a faulty process relays a message to other processes, (i) it can forge the message and claim that it was received from another process, and (ii) it can also tamper with the contents of a received message before relaying it. An unauthenticated message is also called an *oral* message or an *unsigned* message.

- **Agreement variable** The agreement variable may be boolean or multi-valued, and need not be an integer. When studying some of the more complex algorithms, we will use a boolean variable. This simplifying assumption does not affect the results for other data types, but helps in the abstraction while presenting the algorithms.



## 4.7.2 Problem Specifications

### The Byzantine agreement and other problems

**Byzantine Agreement (single source has an initial value)**

**Agreement**:All non-faulty processes must agree on the same value.

**Validity:**If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.

**Termination:**Each non-faulty process must eventually decide on a value.

**Consensus Problem (all processes have an initial value)**

**Agreement:**All non-faulty processes must agree on the same (single) value.

**Validity:**If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.

**Termination:**Each non-faulty process must eventually decide on a value.

**Interactive Consistency (all processes have an initial value)**

**Agreement:**All non-faulty processes must agree on the same array of values $A[v_1 \ldots v_n]$.

**Validity:**If process $i$ is non-faulty and its initial value is $v_i$ , then all non-faulty processes agree on $v_i$ as the $i$ th element of the array $A$. If process $j$ is faulty, then the non-faulty processes can agree on any value for $A[j]$.

**Termination:**Each non-faulty process must eventually decide on the array $A$. These problems are equivalent to one another! Show using reductions.

## 4.8 Overview of Results

| Failure mode | Synchronous system (message-passing and shared memory) | Asynchronous system (message-passing and shared memory) |
|---|---|---|
| No failure | agreement attainable;<br>common knowledge also attainable | agreement attainable;<br>concurrent common knowledge attainable |
| Crash failure | agreement attainable<br>$f < n$ processes $\Omega(f + 1)$ rounds | agreement not attainable |
| Byzantine failure | agreement attainable<br>$f \leq |(n - 1)/3\int$ Byzantine processes<br>$\Omega(f + 1)$ rounds | agreement not attainable |

***Table:Overview of results on agreement. f denotes number of failure-prone processes. n is the total number of processes.***

## Agreement in a failure-free system (synchronous or asynchronous)

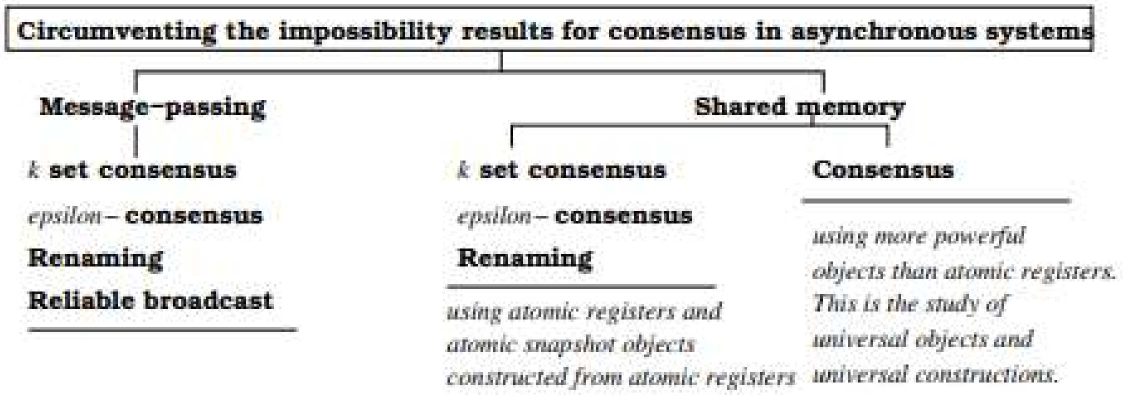
In a failure-free system, consensus can be attained in a straightforward manner

## Some Solvable Variants of the Consensus Problem in Async Systems

| Solvable Variants | Failure model and overhead | Definition |
|---|---|---|
| Reliable broadcast | crash failures, $n > f$ (MP) | Validity, Agreement, Integrity conditions |
| $k$-set consensus | crash failures. $f < k < n$. (MP and SM) | size of the set of values agreed upon must be less than $k$ |
| $s$-agreement | crash failures $n \geq 5f + 1$ (MP) | values agreed upon are within $s$ of each other |
| Renaming | up to $f$ fail-stop processes, $n \geq 2f + 1$ (MP)<br>Crash failures $f \leq n - 1$ (SM) | select a unique name from a set of names |

*Table:* ***Some solvable variants of the agreement problem in asynchronous system.*** The overhead bounds are for the given algorithms, and not necessarily tight bounds for the problem

## Solvable Variants of the Consensus Problem in Async Systems

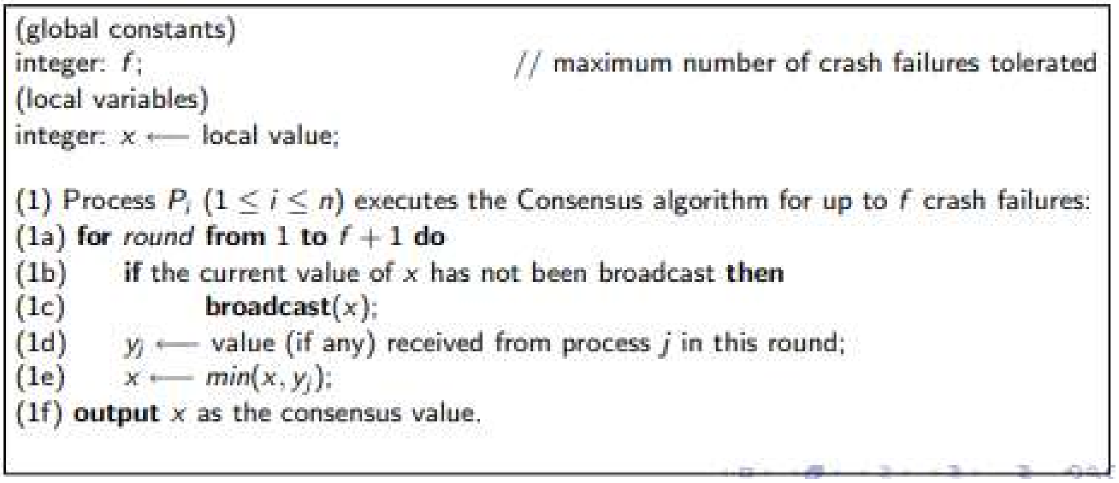

Circumventing the impossibility results for consensus in asynchronous systems
Message-passing
Shared memory
k set consensus
epsilon – consensus
Renaming
Reliable broadcast
k set consensus
epsilon – consensus
Renaming
using atomic registers and atomic snapshot objects constructed from atomic registers
Consensus
using more powerful objects than atomic registers. This is the study of universal objects and universal constructions.

## 14.9 Agreement in a failure-free system

## 14.9.1 Agreement in (message-passing) synchronous systems with failures

## 14.9.1.1. Consensus Algorithm for Crash Failures (MP, synchronous)

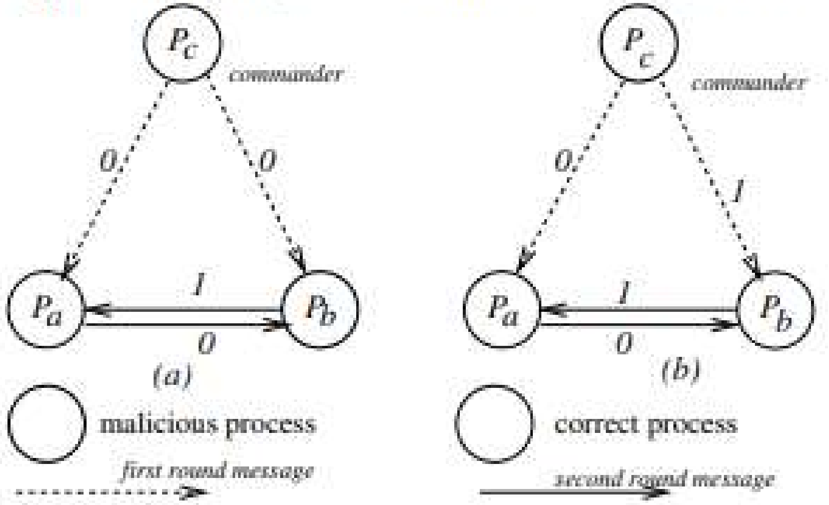
- Up to $f$ ($< n$) crash failures possible.
- In $f + 1$ rounds, at least one round has no failures.
- Now justify: agreement, validity, termination conditions are satisfied.
- Complexity: $O(f + 1)n^2$ messages
- $f + 1$ is lower bound on number of rounds



```
(global constants)
integer: $f$;                              // maximum number of crash failures tolerated
(local variables)
integer: $x \longleftarrow$ local value;

(1) Process $P_i$ $(1 \leq i \leq n)$ executes the Consensus algorithm for up to $f$ crash failures:
(1a) for round from 1 to $f + 1$ do
(1b)     if the current value of $x$ has not been broadcast then
(1c)             broadcast($x$);
(1d)     $y_j \longleftarrow$ value (if any) received from process $j$ in this round;
(1e)     $x \longleftarrow min(x, y_j)$;
(1f) output $x$ as the consensus value.
```

**Upper Bound on Byzantine Processes (sync)**

Agreement impossible when $f = 1, n = 3$.



Taking simple majority decision does not help because loyal commander $P_a$ cannot distinguish between the possible scenarios (a) and (b); hence does not know which action to take.

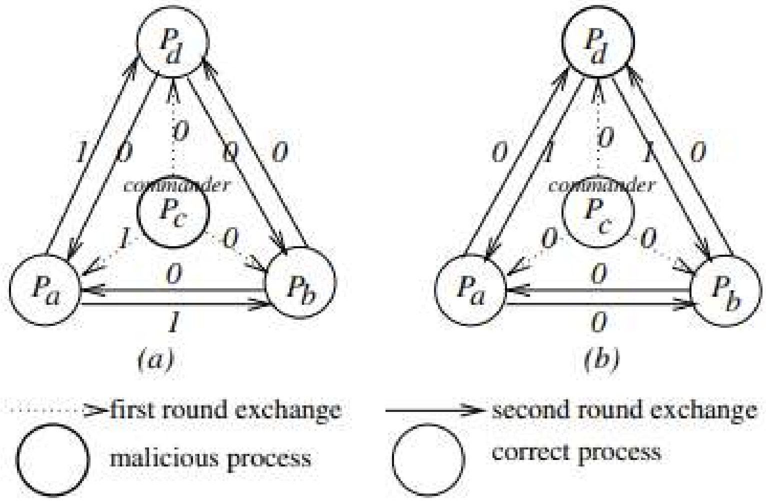
Proof using induction that problem solvable if $f \leq \lfloor \frac{n-1}{3} \rfloor$.

**4.9.2 Byzantine agreement tree algorithm: exponential (synchronous system)**

***Recursive formulation***

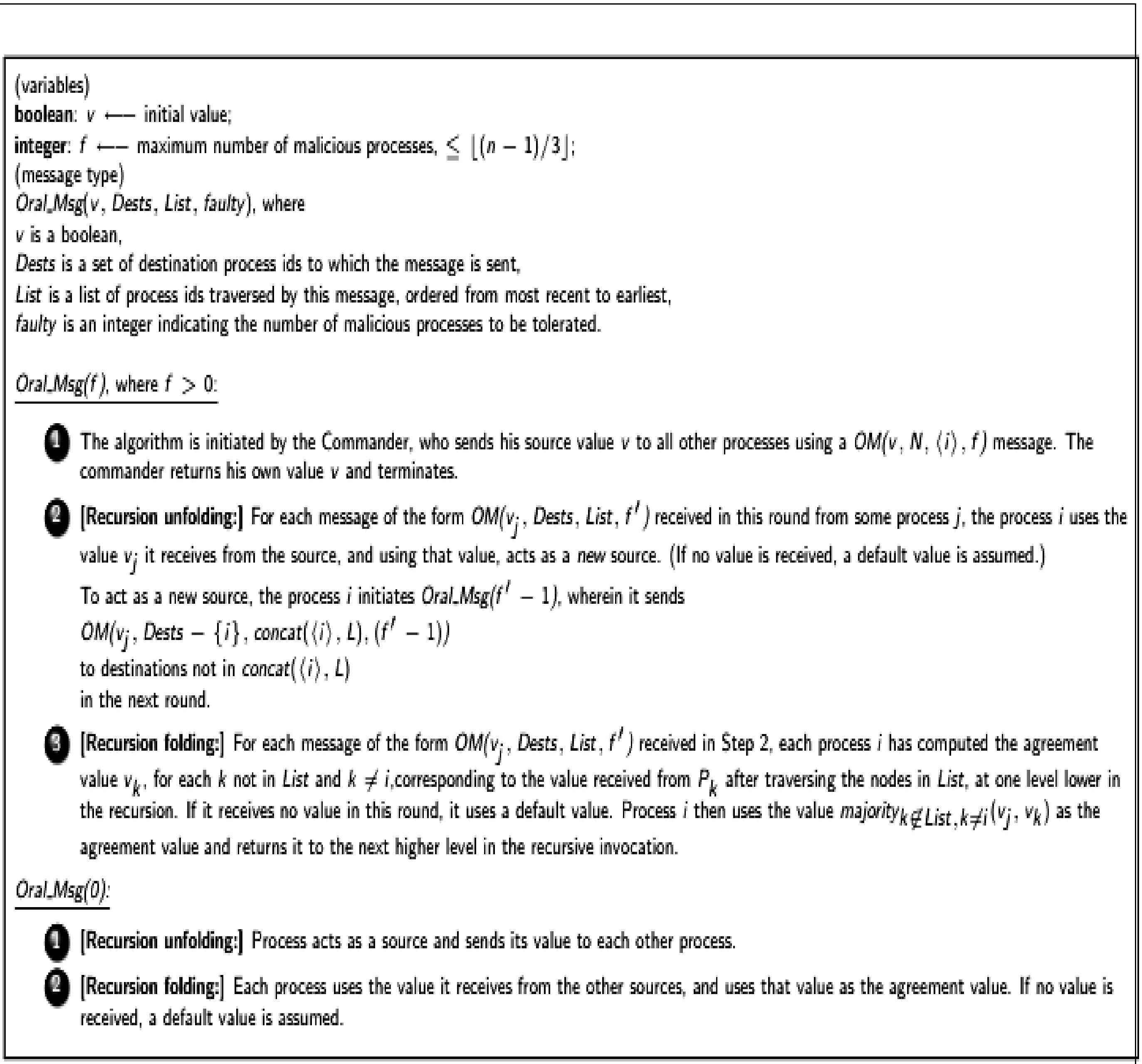
- In the first round, the commander $P_c$ sends its value to the other three lieutenants, as shown by dotted arrows.
- In the second round, each lieutenant relays to the other two lieutenants, the value it received from the commander in the first round. At the end of the second round, a lieutenant takes the majority of the values it received (i) directly from the commander in the first round, and (ii) from the other two lieutenants in the second round.
- The majority gives a correct estimate of the commander's value.

## Consensus Solvable when f = 1, n = 4



- There is no ambiguity at any loyal commander, when taking majority decision
- Majority decision is over 2nd round messages, and 1st round message received directly from commander-in-chief process.

## Byzantine Generals (recursive formulation), (sync, msg-passing)



(variables)
**boolean**: $v \longleftarrow$ initial value;
**integer**: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
*Oral_Msg*(*v*, *Dests*, *List*, *faulty*), where
*v* is a boolean,
*Dests* is a set of destination process ids to which the message is sent,
*List* is a list of process ids traversed by this message, ordered from most recent to earliest,
*faulty* is an integer indicating the number of malicious processes to be tolerated.

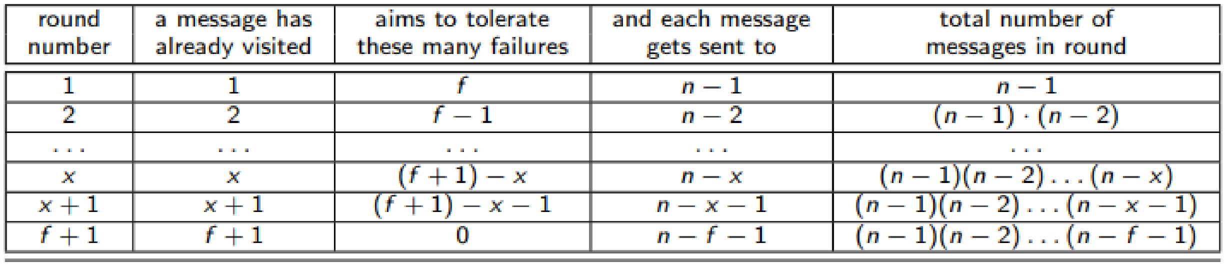
*Oral_Msg*($f$), where $f > 0$:

1. The algorithm is initiated by the Commander, who sends his source value $v$ to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value $v$ and terminates.
2. **[Recursion unfolding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process $j$, the process $i$ uses the value $v_j$ it receives from the source, and using that value, acts as a *new* source. (If no value is received, a default value is assumed.)
   To act as a new source, the process $i$ initiates *Oral_Msg*($f' - 1$), wherein it sends
   $OM(v_j, Dests - \{i\}, concat(\langle i \rangle, L), (f' - 1))$
   to destinations not in $concat(\langle i \rangle, L)$
   in the next round.
3. **[Recursion folding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in Step 2, each process $i$ has computed the agreement value $v_k$, for each $k$ not in *List* and $k \neq i$,corresponding to the value received from $P_k$ after traversing the nodes in *List*, at one level lower in the recursion. If it receives no value in this round, it uses a default value. Process $i$ then uses the value $majority_{k \notin List, k \neq i}(v_j, v_k)$ as the agreement value and returns it to the next higher level in the recursive invocation.

*Oral_Msg*(0):

1. **[Recursion unfolding:]** Process acts as a source and sends its value to each other process.
2. **[Recursion folding:]** Each process uses the value it receives from the other sources, and uses that value as the agreement value. If no value is received, a default value is assumed.

## Relationship between # Messages and Rounds

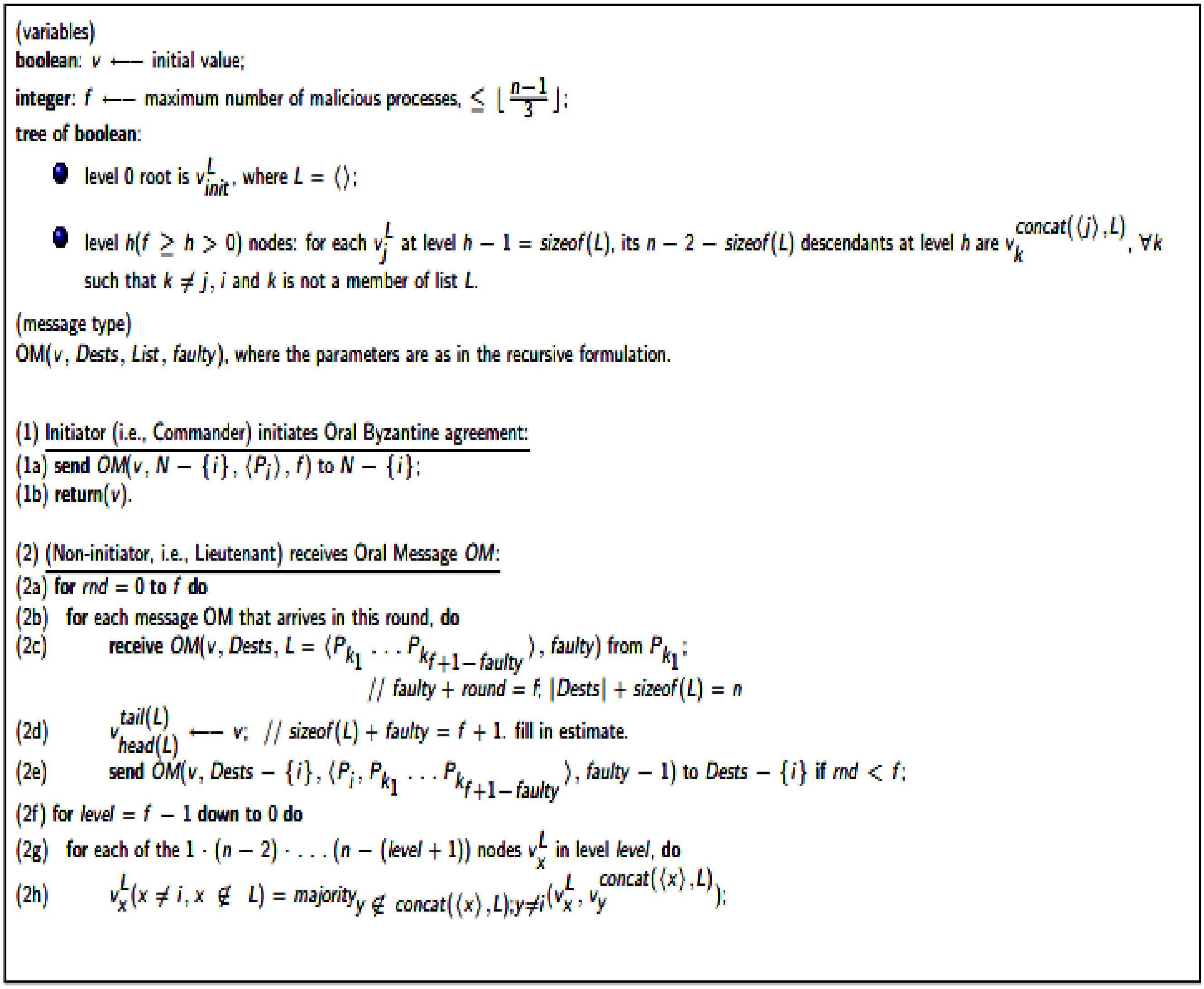


| round number | a message has already visited | aims to tolerate these many failures | and each message gets sent to | total number of messages in round |
|---|---|---|---|---|
| 1 | 1 | $f$ | $n-1$ | $n-1$ |
| 2 | 2 | $f-1$ | $n-2$ | $(n-1)\cdot(n-2)$ |
| ... | ... | ... | ... | ... |
| $x$ | $x$ | $(f+1)-x$ | $n-x$ | $(n-1)(n-2)\ldots(n-x)$ |
| $x+1$ | $x+1$ | $(f+1)-x-1$ | $n-x-1$ | $(n-1)(n-2)\ldots(n-x-1)$ |
| $f+1$ | $f+1$ | 0 | $n-f-1$ | $(n-1)(n-2)\ldots(n-f-1)$ |

**Relationships between messages and rounds in the Oral Messages algorithm for Byzantine agreement.**

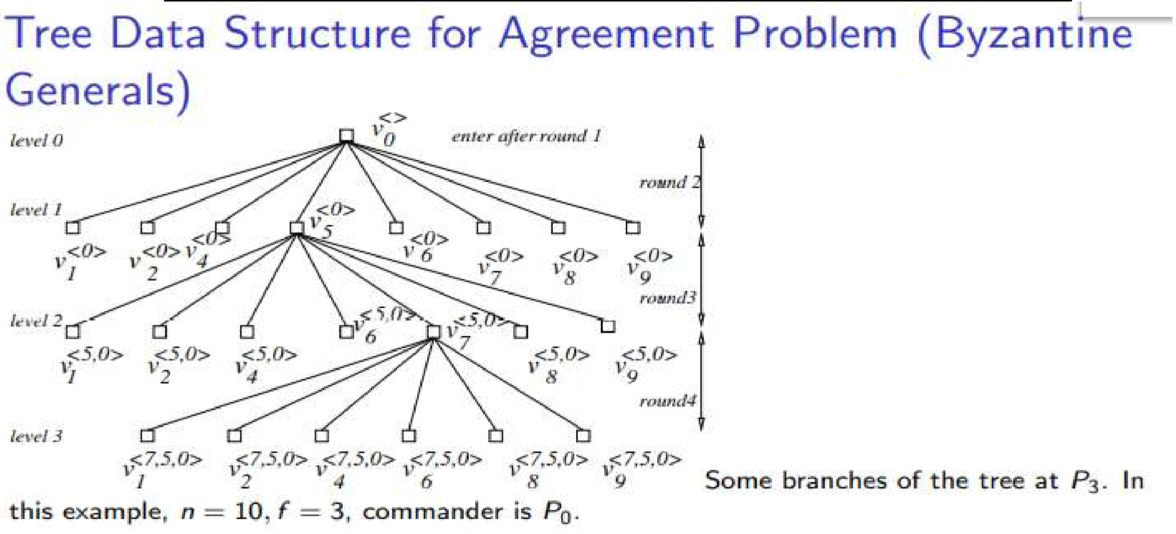
Complexity: f + 1 rounds, exponential amount of space, and (n − 1) + (n − 1)(n − 2) + . . . + (n − 1)(n − 2)..(n − f − 1)messages

## Bzantine Generals (iterative formulation), Sync, Msg-passing



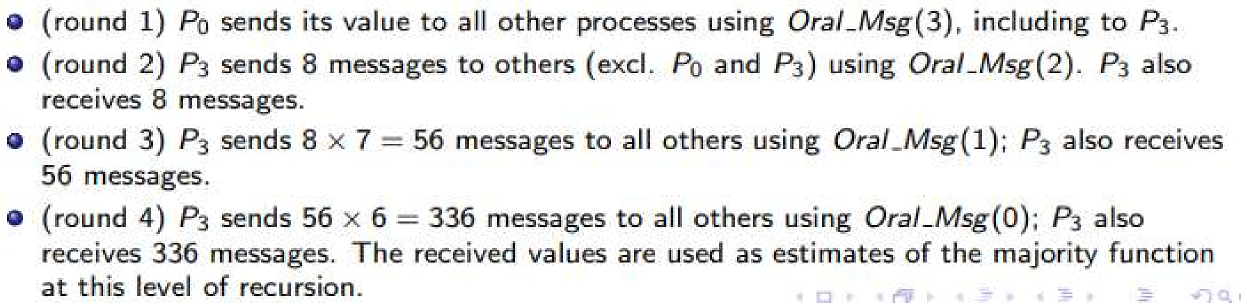
```
(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;
tree of boolean:
    ● level 0 root is $v^L_{init}$, where $L = \langle\rangle$;
    ● level $h(f \geq h > 0)$ nodes: for each $v^L_j$ at level $h - 1 = sizeof(L)$, its $n - 2 - sizeof(L)$ descendants at level $h$ are $v_k^{concat(\langle j \rangle, L)}$, $\forall k$
      such that $k \neq j, i$ and $k$ is not a member of list $L$.
(message type)
OM($v$, $Dests$, $List$, $faulty$), where the parameters are as in the recursive formulation.

(1) Initiator (i.e., Commander) initiates Oral Byzantine agreement:
(1a) send $OM(v, N - \{i\}, \langle P_i \rangle, f)$ to $N - \{i\}$;
(1b) return($v$).

(2) (Non-initiator, i.e., Lieutenant) receives Oral Message $OM$:
(2a) for $rnd = 0$ to $f$ do
(2b)   for each message OM that arrives in this round, do
(2c)       receive $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty)$ from $P_{k_1}$;
                                  // $faulty + round = f$; $|Dests| + sizeof(L) = n$
(2d)       $v^{tail(L)}_{head(L)} \longleftarrow v$;  // $sizeof(L) + faulty = f + 1$. fill in estimate.
(2e)       send $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty - 1)$ to $Dests - \{i\}$ if $rnd < f$;
(2f) for $level = f - 1$ down to 0 do
(2g)   for each of the $1 \cdot (n-2) \cdot \ldots (n - (level + 1))$ nodes $v^L_x$ in level $level$, do
(2h)       $v^L_x(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v^L_x, v_y^{concat(\langle x \rangle, L)})$;
```

## Tree Data Structure for Agreement Problem (Byzantine Generals)



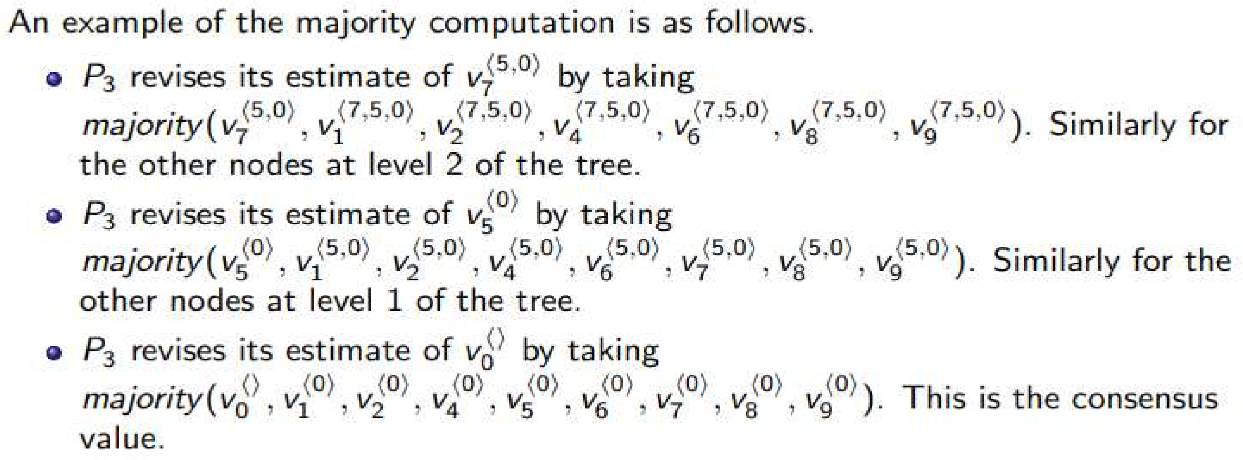
Tree Data Structure for Agreement Problem (Byzantine Generals)

Some branches of the tree at $P_3$. In this example, $n = 10$, $f = 3$, commander is $P_0$.



- (round 1) $P_0$ sends its value to all other processes using $Oral\_Msg(3)$, including to $P_3$.
- (round 2) $P_3$ sends 8 messages to others (excl. $P_0$ and $P_3$) using $Oral\_Msg(2)$. $P_3$ also receives 8 messages.
- (round 3) $P_3$ sends $8 \times 7 = 56$ messages to all others using $Oral\_Msg(1)$; $P_3$ also receives 56 messages.
- (round 4) $P_3$ sends $56 \times 6 = 336$ messages to all others using $Oral\_Msg(0)$; $P_3$ also receives 336 messages. The received values are used as estimates of the majority function at this level of recursion.

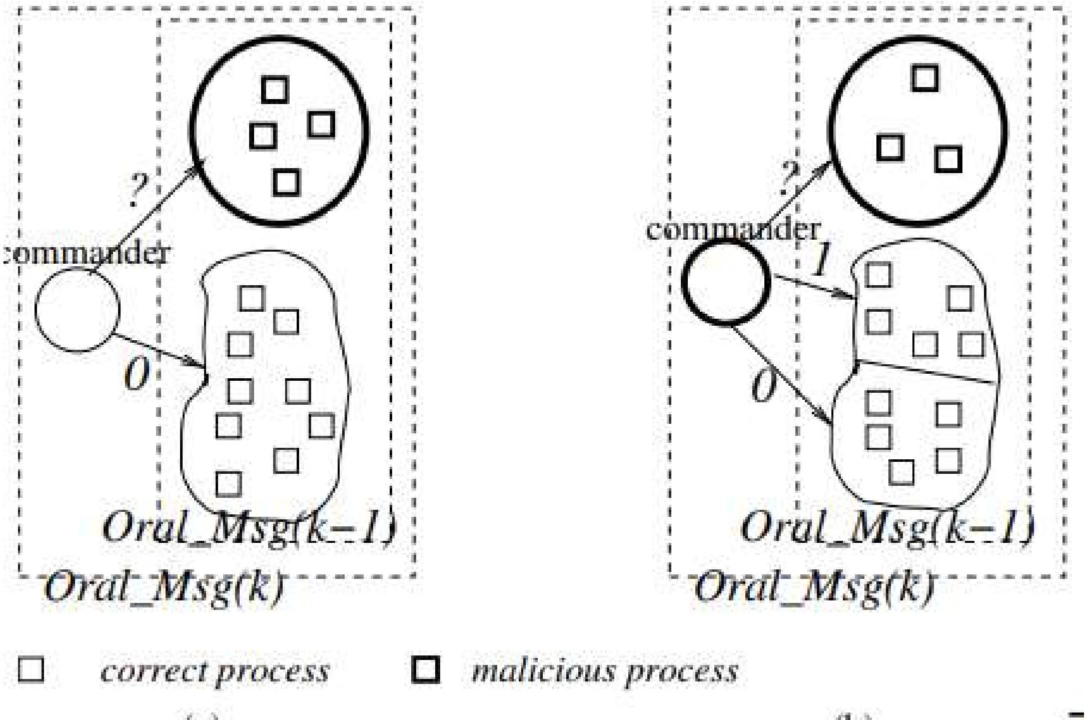
### 4.9.3 Exponential Algorithm: An example



An example of the majority computation is as follows.

- $P_3$ revises its estimate of $v_7^{\langle 5,0\rangle}$ by taking $majority(v_7^{\langle 5,0\rangle}, v_1^{\langle 7,5,0\rangle}, v_2^{\langle 7,5,0\rangle}, v_4^{\langle 7,5,0\rangle}, v_6^{\langle 7,5,0\rangle}, v_8^{\langle 7,5,0\rangle}, v_9^{\langle 7,5,0\rangle})$. Similarly for the other nodes at level 2 of the tree.
- $P_3$ revises its estimate of $v_5^{\langle 0\rangle}$ by taking $majority(v_5^{\langle 0\rangle}, v_1^{\langle 5,0\rangle}, v_2^{\langle 5,0\rangle}, v_4^{\langle 5,0\rangle}, v_6^{\langle 5,0\rangle}, v_7^{\langle 5,0\rangle}, v_8^{\langle 5,0\rangle}, v_9^{\langle 5,0\rangle})$. Similarly for the other nodes at level 1 of the tree.
- $P_3$ revises its estimate of $v_0^{\langle\rangle}$ by taking $majority(v_0^{\langle\rangle}, v_1^{\langle 0\rangle}, v_2^{\langle 0\rangle}, v_4^{\langle 0\rangle}, v_5^{\langle 0\rangle}, v_6^{\langle 0\rangle}, v_7^{\langle 0\rangle}, v_8^{\langle 0\rangle}, v_9^{\langle 0\rangle})$. This is the consensus value.

Impact of a Loyal and of a Disloyal Commander



(a) (b)

The effects of a loyal or a disloyal commander in a system with n = 14 and f = 4. The subsystems that need to tolerate k and k − 1 traitors are shown for two cases. (a) Loyal commander.
(b) No assumptions about commander.

(a) the commander who invokes Oral Msg(x) is loyal, so all the loyal processes have the same estimate. Although the subsystem of 3x processes has x malicious processes, all the loyal processes have the same view to begin with. Even if this case repeats for each nested invocation of Oral Msg, even after x rounds, among the processes, the loyal processes are in a simple majority, so the majority function works in having them maintain the same common view of the loyal commander's value.
(b) the commander who invokes Oral Msg(x) may be malicious and can send conflicting values to the loyal processes. The subsystem of 3x processes has x − 1 malicious processes, but all the loyal processes do not have the same view to begin with.

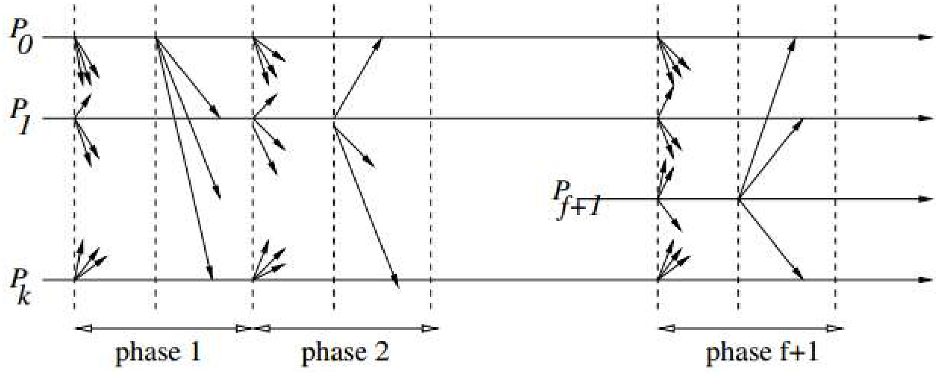
## 4.9.4 The Phase King Algorithm

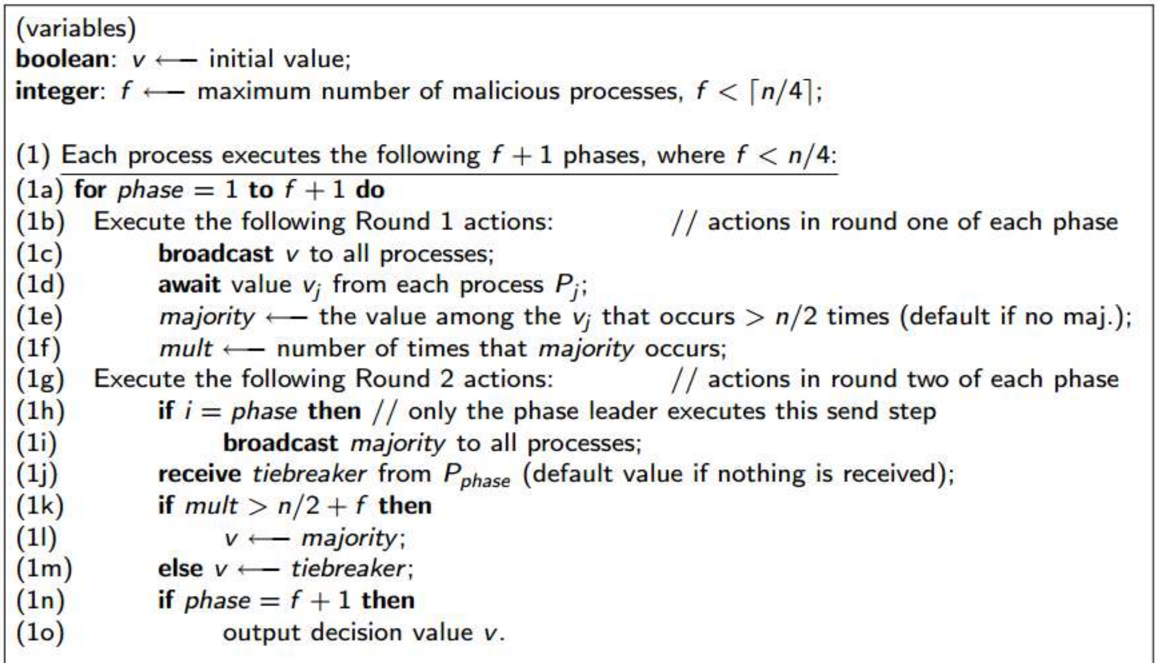
Operation

Each phase has a unique "phase king" derived, say, from PID. Each phase has two rounds:

in 1st round, each process sends its estimate to all other processes.

in 2nd round, the "Phase king" process arrives at an estimate based on the values it received in 1st round, and broadcasts its new estimate to all others.





```
(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $f < \lceil n/4 \rceil$;

Each process executes the following $f + 1$ phases, where $f < n/4$:
for phase = 1 to $f + 1$ do
  Execute the following Round 1 actions:          // actions in round one of each phase
      broadcast $v$ to all processes;
      await value $v_j$ from each process $P_j$;
      majority $\longleftarrow$ the value among the $v_j$ that occurs $> n/2$ times (default if no maj.);
      mult $\longleftarrow$ number of times that majority occurs;
  Execute the following Round 2 actions:          // actions in round two of each phase
      if $i = phase$ then // only the phase leader executes this send step
          broadcast majority to all processes;
      receive tiebreaker from $P_{phase}$ (default value if nothing is received);
      if $mult > n/2 + f$ then
          $v \longleftarrow$ majority;
      else $v \longleftarrow$ tiebreaker;
      if $phase = f + 1$ then
          output decision value $v$.
```

$(f + 1)$ phases, $(f + 1)[(n - 1)(n + 1)]$ messages, and can tolerate up to $f < |n/4|$ malicious processes

## Correctness Argument

**Among $f + 1$ phases, at least one phase $k$ where phase-king is non-malicious.**

**In phase $k$, all non-malicious processes $P_i$ and $P_j$ will have same estimate of consensus value as $P_k$ does.**

**$P_i$ and $P_j$ use their own majority values (Hint: $\Longrightarrow P_i$'s $mult > n/2 + f$ )**

**$P_i$ uses its majority value; $P_j$ uses phase-king's tie-breaker value. (Hint: $P_i$ ”s $mult > n/2 + f$ , $P_j$'s $mult > n/2$ for same value)**

**$P_i$ and $P_j$ use the phase-king's tie-breaker value. (Hint: In the phase in which $P_k$ is non-malicious, it sends same value to $P_i$ and $P_j$ )**

In all 3 cases, argue that $P_i$ and $P_j$ end up with same value as estimate

If all non-malicious processes have the value $x$ at the start of a phase, they will continue to have $x$ as the consensus value at the end of the phase.

# UNIT V P2P & DISTRIBUTED SHARED MEMORY

**Peer-to-peer computing and overlay graphs: Introduction – Data indexing and overlays – Chord – Content addressable networks – Tapestry. Distributed shared memory: Abstraction and advantages – Memory consistency models –Shared memory Mutual Exclusion**

## 5.1 Peer-to-peer Computing and Overlay Graphs

### Characteristics

- Peer-to-peer (P2P) network systems use an application-level organization of the network overlay for flexibly sharing resources (e.g., files and multimedia documents) stored across network-wide computers.
- All nodes are equal; communication directly between peers (no client-server) Allow location of arbitrary objects; no DNS servers required
- Large combined storage, CPU power, other resources, without scalability costs
- Dynamic insertion and deletion of nodes, as well as of resources, at low cost

| Features | Performance |
|---|---|
| self-organizing | large combined storage, CPU power, and resources |
| distributed control | fast search for machines and data objects |
| role symmetry for nodes | scalable |
| anonymity | efficient management of churn |
| naming mechanism | selection of geographically close servers |
| security, authentication, trust | redundancy in storage and paths |

***Table:Desirable characteristics and performance features of P2P systems.***

### 5.1.1. Napster

- One of the earliest popular P2P systems, Napster [25], used a server-mediated central index architecture organized around clusters of servers that store direct indices of the files in the system.
- Central server maintains a table with the following information of each registered client: (i) the client's address (IP) and port, and offered bandwidth, and (ii) information about the files that the client can allow to share.

1. A client connects to a meta-server that assigns a lightly-loaded server.
2. The client connects to the assigned server and forwards its query and identity.
3. The server responds to the client with information about the users connected to it and the files they are sharing.
4. On receiving the response from the server, the client chooses one of the users from whom to download a desired file. The address to enable the P2P connection between the client and the selected user is provided by the server to the client.

Users are generally anonymous to each other. The directory serves to provide the mapping from a particular host that contains the required content, to the IP address needed to download from it.

### 5.1.2 Application layer overlays

- ➢ A core mechanism in P2P networks is searching for data, and this mechanism depends on how (i) the data, and (ii) the network, are organized. Search algorithms for P2P networks tend to be data-centric, as opposed to the host-centric algorithms for traditional networks.
- ➢ P2P search uses the *P2P overlay*, which is a logical graph among the peers that is used for the object search and object storage and management algorithms. Note that above the P2P over-lay is the application layer overlay, where communication between peers is point-to-pont (representing a logical all-to-all connectivity) once a connection is established.
- ➢ The P2P overlay can be *structured* (e.g., hypercubes, meshes, butterfly networks, de Bruijn graphs) or *unstructured*

### Structured and Unstructured Overlays

- Search for data and placement of data depends on P2P overlay (which can be thought of as being below the application level overlay)
- Search is data-centric, not host-centric Structured P2P overlays:
  - ❯ E.g., hypercube, mesh, de Bruijn graphs
  - ❯ rigid organizational principles for object storage and object search
- Unstructured P2P overlays:
  - ❯ Loose guidelines for object search and storage
  - ❯ Search mechanisms are ad-hoc, variants of flooding and random walk
- Object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

## 5.2 Data indexing

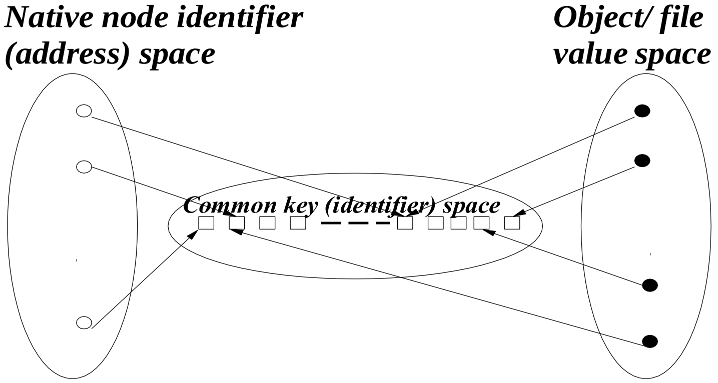
The data in a P2P network is identified by using indexing. Data indexing allows the physical data independence from the applications. Indexing mechanisms can be classified as being centralized, local, or distributed

- **Centralized indexing**, e.g., versions of Napster, DNS
- **Distributed indexing**. Indexes to data scattered across peers. Access data through mechanisms such as Distributed Hash Tables (DHT). These differ in hash mapping, search algorithms, diameter for lookup, fault tolerance, churn resilience.
- **Local indexing**. Each peer indexes only the local objects. Remote objects need to be searched for. Typical DHT uses flat key space. Used commonly in unstructured overlays (E.g., Gnutella) along with flooding search or random walk search.

An alternate way to classify indexing mechanisms is as being a *semantic index mechanism* or a *semantic-free* index mechanism.

- **Semantic indexing** - human readable, e.g., filename, keyword, database key. Supports keyword searches, range searches, approximate searches.
- **Semantic-free indexing**. Not human readable. Corresponds to index obtained by use of hash function.

## Simple Distributed Hash Table scheme



Mappings from node address space and object space in a simple DHT.

- Highly deterministic placement of files/data allows fast lookup. But file insertions/deletions under churn incurs some cost.
- Attribute search, range search, keyword search etc. not possible.

## 5.2.1 Distributed indexing

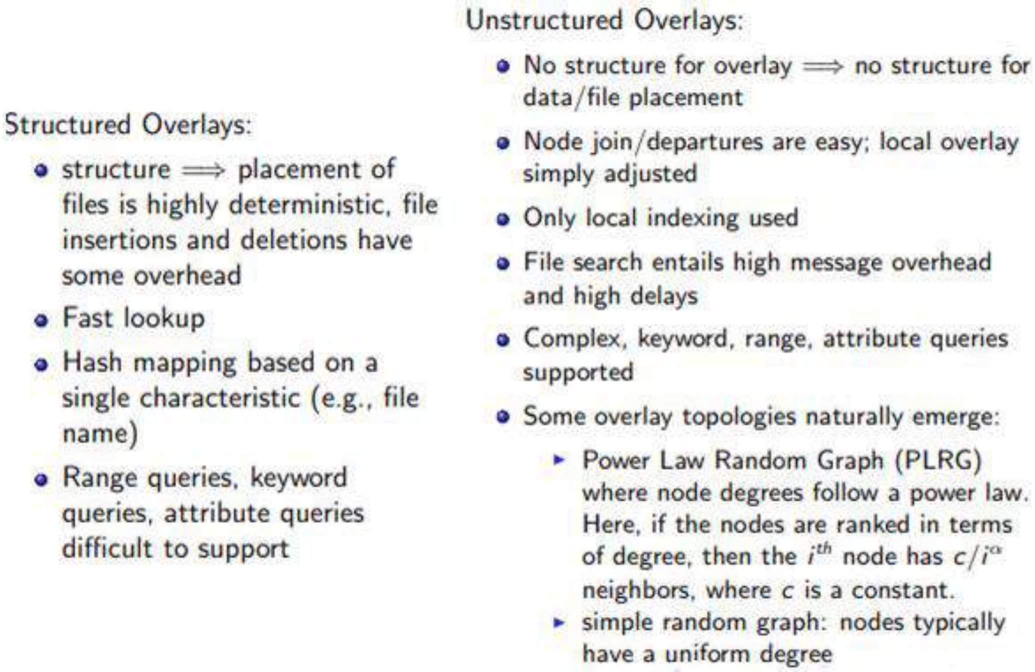
### Structured overlays

- The P2P network topology has a definite structure, and the placement of files or data in this network is highly deterministic as per some algorithmic mapping. (The placement of files can sometimes be “loose,” as in some earlier P2P systems like Freenet, where “hints” are used.)
- The objective of such a deterministic mapping is to allow a very fast and deterministic lookup to satisfy queries for the data. These systems are termed as lookup systems and typically use a hash table interface for the mapping.

### Unstructured overlays

- 
- The P2P network topology does not have any particular controlled structure, nor is there any control over where files/data is placed. Each peer typically indexes only its local data objects, hence, local indexing is used.
- Node joins and departures are easy – the local overlay is simply adjusted. File placement is not governed by the topology. Search for a file may entail high message overhead and high delays. However, complex queries are supported because the search criteria can be arbitrary.
- Although the P2P network topology does not have any controlled structure, some topologies naturally emerge.
  - Power law random graph (PLRG) This is a random graph where the node degrees follow the power law. Here, if the nodes are ranked in terms of their degree, then the ith node has c/i neighbors, where c is a constant.
  - Normal random graph This is a normal random graph where the nodes typically have a uniform degree.

### Structured vs. unstructured overlays



Structured Overlays:

- structure ⟹ placement of files is highly deterministic, file insertions and deletions have some overhead
- Fast lookup
- Hash mapping based on a single characteristic (e.g., file name)
- Range queries, keyword queries, attribute queries difficult to support

Unstructured Overlays:

- No structure for overlay ⟹ no structure for data/file placement
- Node join/departures are easy; local overlay simply adjusted
- Only local indexing used
- File search entails high message overhead and high delays
- Complex, keyword, range, attribute queries supported
- Some overlay topologies naturally emerge:
  - Power Law Random Graph (PLRG) where node degrees follow a power law. Here, if the nodes are ranked in terms of degree, then the $i^{th}$ node has $c/i^{\alpha}$ neighbors, where $c$ is a constant.
  - simple random graph: nodes typically have a uniform degree

## Unstructured Overlays: Properties

- Semantic indexing possible =⇒ keyword, range, attribute-based queries Easily accommodate high churn
- Efficient when data is replicated in network Good if user satisfied with ”best-effort” search
- Network is not so large as to cause high delays in search

### Gnutella features

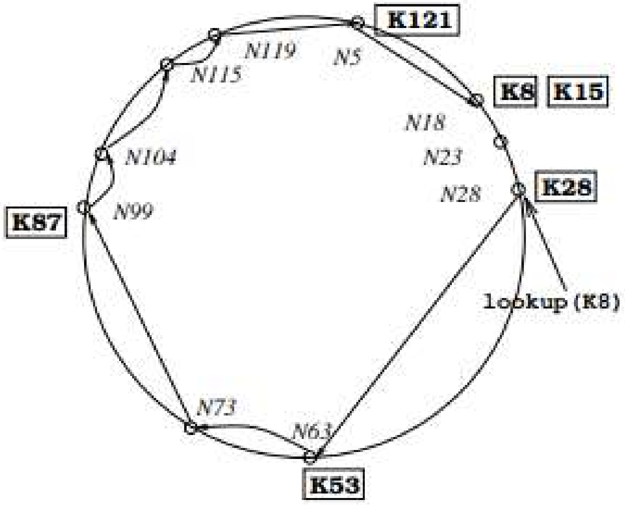
- A joiner connects to some standard nodes from Gnutella directory
- *Ping* used to discover other hosts; allows new host to announce itself
- *Pong* in response to *Ping* ; *Pong* contains IP, port #, max data size for download
- *Query* msgs used for flooding search; contains required parameters
- *QueryHit* are responses. If data is found, this message contains the IP, port #, file size, download rate, etc. Path used is reverse path of *Query*

## 5.3 Chord

The Chord protocol, uses a flat key space to associate the mapping between network nodes and data objects/files/values. The node address as well as the data object/file/value is mapped to a logical identifier in the common key space using a consistent hash function.

- When a node joins or leaves the network of *n* nodes, only 1/*n* keys have to moved.
- The Chord key space is flat, thus giving applications flexibility in map-ping their files/data to keys. Chord supports a single operation, lookup x , which maps a given key x to a network node. Specifically, Chord stores a file/object/value at the node to which the file/object/value’s key maps.
- Two steps involved.

- ❩ Map the object value to its key
- ❩ Map the key to the node in the native address space using *lookup*
- Common address space is a *m*-bit identifier ($2^m$ addresses), and this space is arranged on a logical ring *mod*($2^m$).
- A key *k* gets assigned to the first node such that the node identifier equals or is greater than the key identifier *k* in the logical space address.



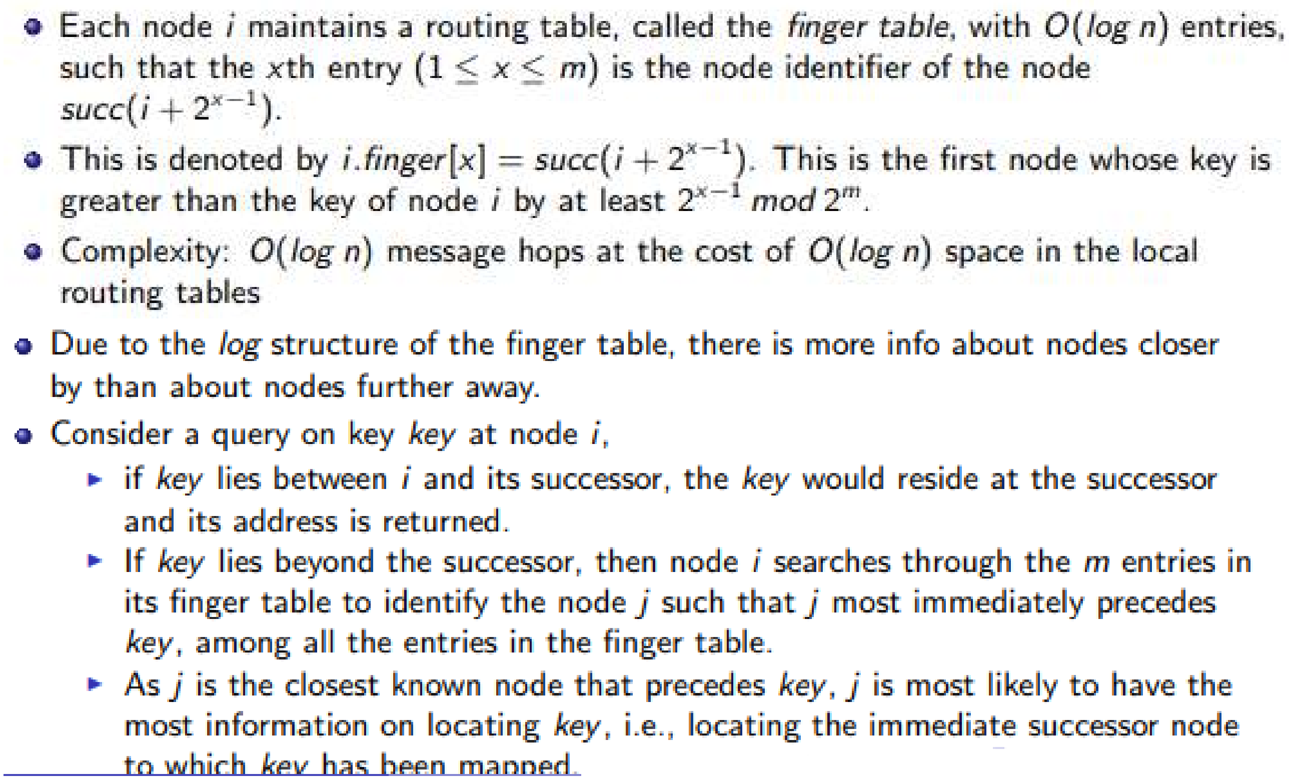
### 5.3.1 Chord: SimpleLookup

- A simple key lookup algorithm that requires each node to store only 1 entry in its routing table works as follows.
- Each node tracks its successor on the ring, in the variable successor; a query for key x is forwarded to the successors of nodes until it reaches the first node such that that node's identifier y is greater than the key x, modulo $2^m$.

- The result, which includes the IP address of the node with key *y*, is returned to the querying node along the reverse of the path that was followed by the query.

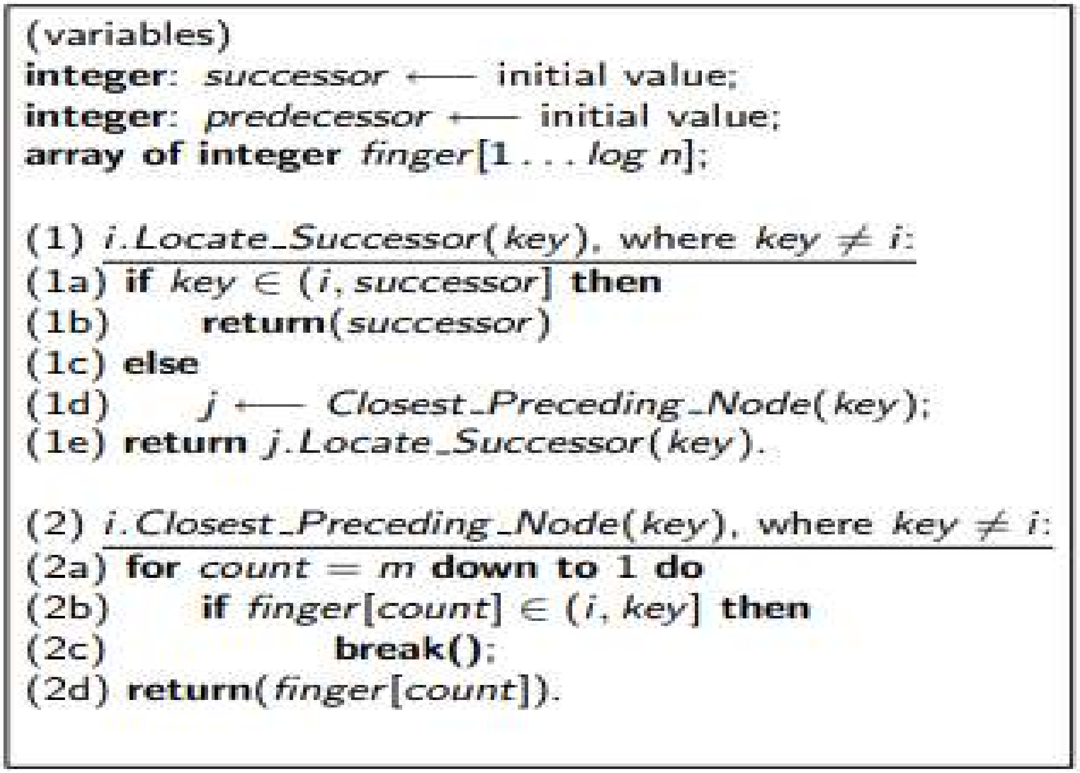
- This mechanism requires *O*(1) local space but *O*(*n*) hops.

### 5.3.2 Chord: Scalable Lookup



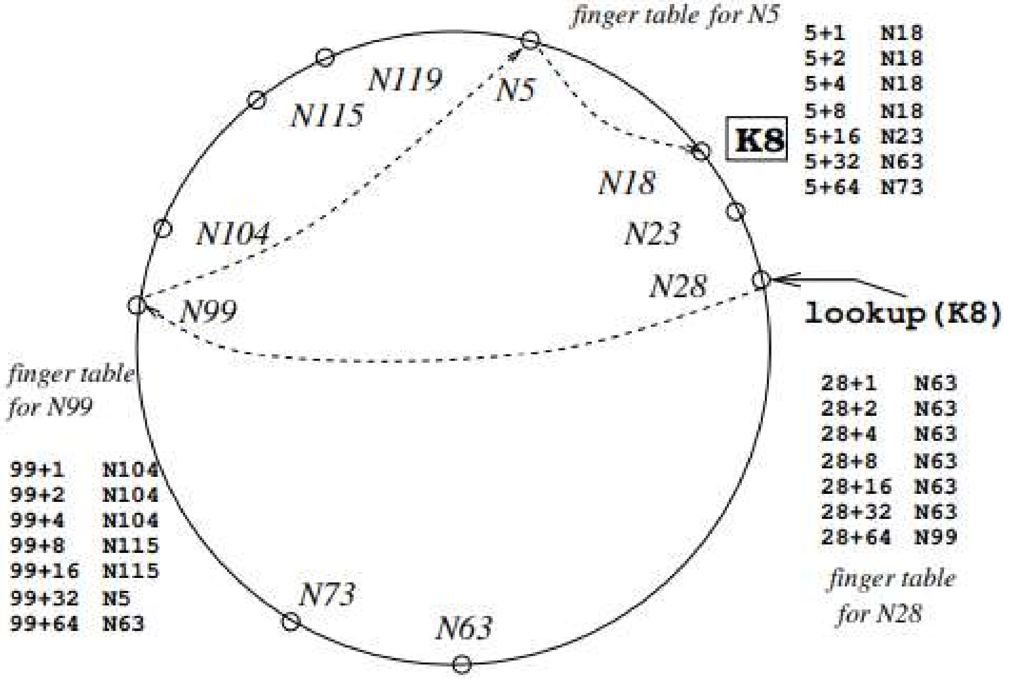
- Each node $i$ maintains a routing table, called the *finger table*, with $O(\log n)$ entries, such that the $x$th entry $(1 \leq x \leq m)$ is the node identifier of the node $succ(i + 2^{x-1})$.
- This is denoted by $i.finger[x] = succ(i + 2^{x-1})$. This is the first node whose key is greater than the key of node $i$ by at least $2^{x-1} \bmod 2^m$.
- Complexity: $O(\log n)$ message hops at the cost of $O(\log n)$ space in the local routing tables
- Due to the *log* structure of the finger table, there is more info about nodes closer by than about nodes further away.
- Consider a query on key *key* at node $i$,
  - if *key* lies between $i$ and its successor, the *key* would reside at the successor and its address is returned.
  - If *key* lies beyond the successor, then node $i$ searches through the $m$ entries in its finger table to identify the node $j$ such that $j$ most immediately precedes *key*, among all the entries in the finger table.
  - As $j$ is the closest known node that precedes *key*, $j$ is most likely to have the most information on locating *key*, i.e., locating the immediate successor node to which *key* has been mapped.

**Chord**



```
(variables)
integer: successor ⟵ initial value;
integer: predecessor ⟵ initial value;
array of integer finger[1 . . . log n];

(1) i.Locate_Successor(key), where key ≠ i:
(1a) if key ∈ (i, successor] then
(1b)      return(successor)
(1c) else
(1d)      j ⟵ Closest_Preceding_Node(key);
(1e) return j.Locate_Successor(key).

(2) i.Closest_Preceding_Node(key), where key ≠ i:
(2a) for count = m down to 1 do
(2b)      if finger[count] ∈ (i, key] then
(2c)           break();
(2d) return(finger[count]).
```

**Scalable Lookup - Example**



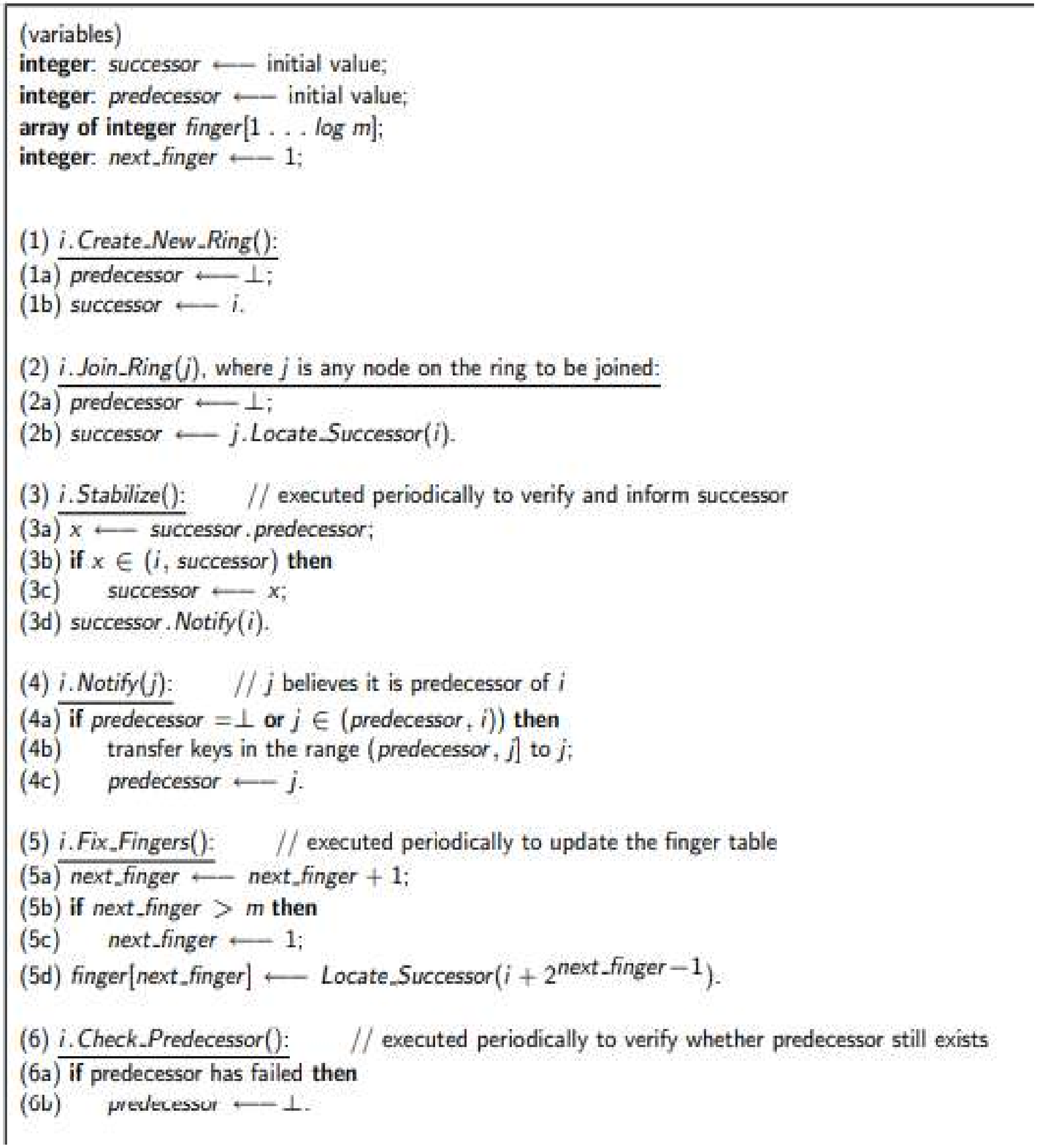
### 5.3.3 Chord: Managing Churn

The code to manage dynamic node joins, departures, and failures is given in Algorithm

**Node joins**

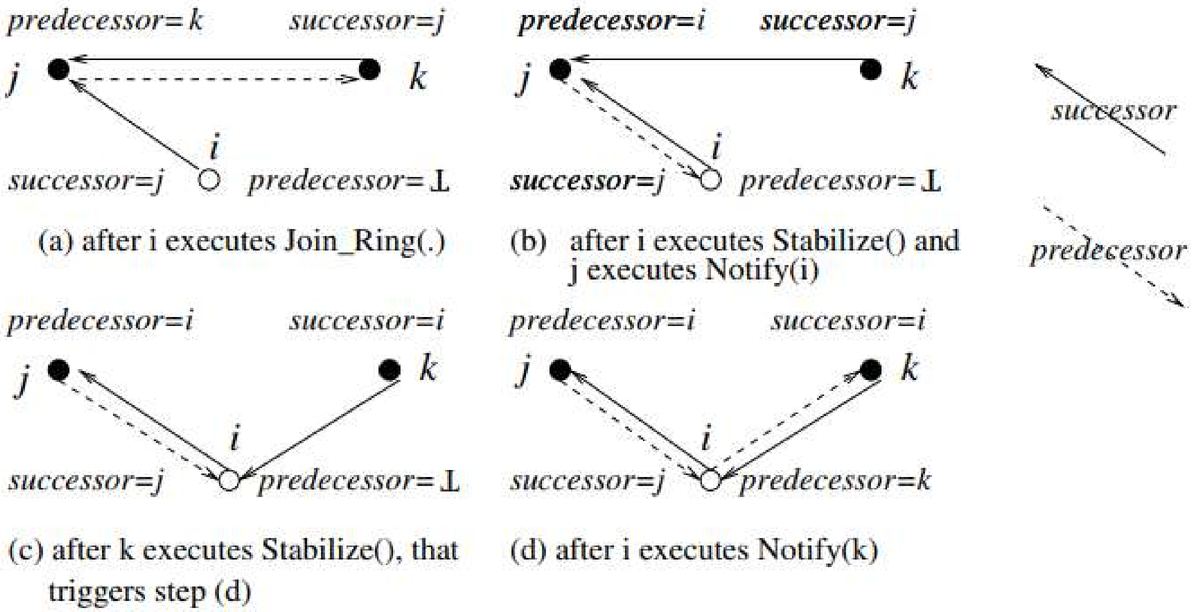
- To create a new ring, a node i executes Create_New_Ring which creates a ring with the singleton node.
- To join a ring that contains some node j, node i invokes Join_Ring j . Node j locates i's successor on the logical ring and informs i of its successor.
- Before i can participate in the P2P exchanges, several actions need to happen: i's successor needs to update its predecessor entry to i, i's predecessor needs to revise its successor field to i, i needs to identify its predecessor, the finger table at i needs to be built, and the finger tables of all nodes need to be updated to account for i's presence.
- This is achieved by procedures Stabilize , Fix_Fingers , and Check_Predecessor that are periodically invoked by each node.

**Algorithm Managing churn in Chord. Code shown is for node**



```
(variables)
integer: successor ⟵ initial value;
integer: predecessor ⟵ initial value;
array of integer finger[1 . . . log m];
integer: next_finger ⟵ 1;


(1) i.Create_New_Ring():
(1a) predecessor ⟵ $\perp$;
(1b) successor ⟵ i.

(2) i.Join_Ring(j), where j is any node on the ring to be joined:
(2a) predecessor ⟵ $\perp$;
(2b) successor ⟵ j.Locate_Successor(i).

(3) i.Stabilize():        // executed periodically to verify and inform successor
(3a) x ⟵ successor.predecessor;
(3b) if $x \in (i, successor)$ then
(3c)     successor ⟵ x;
(3d) successor.Notify(i).

(4) i.Notify(j):        // j believes it is predecessor of i
(4a) if predecessor $= \perp$ or $j \in (predecessor, i)$ then
(4b)     transfer keys in the range (predecessor, j] to j;
(4c)     predecessor ⟵ j.

(5) i.Fix_Fingers():        // executed periodically to update the finger table
(5a) next_finger ⟵ next_finger + 1;
(5b) if next_finger > m then
(5c)     next_finger ⟵ 1;
(5d) finger[next_finger] ⟵ Locate_Successor($i + 2^{next\_finger - 1}$).

(6) i.Check_Predecessor():        // executed periodically to verify whether predecessor still exists
(6a) if predecessor has failed then
(6b)     predecessor ⟵ $\perp$.
```

**Figure illustrates the main steps of the joining process. A recent joiner node i that has executed Join_Ring · gets integrated into the ring by the following sequence:**



(a) after i executes Join_Ring(.)

(b) after i executes Stabilize() and j executes Notify(i)

(c) after k executes Stabilize(), that triggers step (d)

(d) after i executes Notify(k)

Node $i$ integrates into the ring, where $j > i > k$, as per the steps shown.

1. The configuration after a recent joiner node i has executed Join_Ring · .
2. Node i executes Stabilize , which allows its successor j to adjust j's variable predecessor to i. Specifically, when node i invokes Stabilize , it identifies the successor's predecessor k. If k ∈ i successor , then i updates its successor to k. In either case, i notifies its successor of itself via successor Notify i , so the successor has a chance to adjust its predecessor variable to i.
3. The earlier predecessor k of j (i.e., the predecessor in Step 1) executes Stabilize and adjusts its successor pointer from j to i.
4. Node i executes Fix_Fingers to build its finger table, and other nodes also execute the procedure to update their finger tables if necessary.

- How are node departures handled? or node failures?
- For a Chord network with $n$ nodes, each node is responsible for at most $(1 + s)$ $K/n$ keys, with "high probability", where $K$ is the total number of keys. Using consistent hashing, $s$ can be shown to be bounded by $O(\log n)$.
- The search for a successor in *Locate Successor* in a Chord network with $n$ nodes requires time complexity $O(\log n)$ with high probability.
- The size of the finger table is $\log(n) \leq m$. The average lookup time is $1/2 \log(n)$.

## 5.4 Content Addressable Network (CAN)

- An indexing mechanism that maps objects to locations in CAN
- object-location in P2P networks, large-scale storage management, wide-area name resolution services that decouple name resolution and the naming scheme
- Efficient, scalable addition of and location of objects using location-independent names or keys.
- 3 basic operations: insertion, search, deletion of (*key*, *value*) pairs
- *d*-dimensional logical Cartesian space organized as a *d*-torus logical topology, i.e.. *d* -dimensional mesh with wraparound.
- Space partitioned dynamically among nodes, i.e., node *i* has space *r* (*i* ). For object v , its key r (v ) is mapped to a point ˙p in the space. (v, key (v )) tuple stored at node which is the present owner containing the point ˙p.
- Analogously to retrieve object *v*.

**3 components of CAN**

- Set up CAN virtual coordinate space, partition among nodes
- Routing in virtual coordinate space to locate the node that is assigned the region corresponding to $\dot{p}$
- Maintain the CAN in spite of node departures and failures

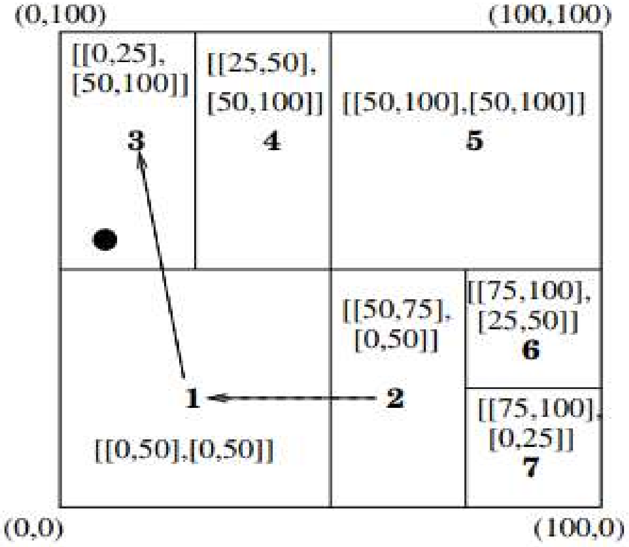
### 5.4.1 CAN Initialization

- Each CAN has a unique DNS name that maps to the IP address of a few bootstrap nodes. Bootstrap node: tracks a partial list of the nodes that it believes are currently in the CAN.
- A joiner node queries a bootstrap node via a DNS lookup. Bootstrap node replies with the IP addresses of some randomly chosen nodes that it believes are in the CAN.
- The joiner chooses a random point ˙p in the coordinate space. The joiner sends a request to one of the nodes in the CAN, of which it learnt in Step 2, asking to be assigned a region containing ˙p. The recipient of the request routes the request to the owner old owner (˙p) of the region containing ˙p, using CAN routing algorithm.
- The old owner (˙p) node splits its region in half and assigns one half to the joiner. The region splitting is done using an a priori ordering of all the dimensions. This also helps to methodically merge regions, if necessary. The (k, v ) tuples for which the key k now maps to the zone to be transferred to the joiner, are also transferred to the joiner.
- The joiner learns the IP addresses of its neighbours from old owner (˙p). The neighbors are old owner (˙p) and a subset of the neighbours of old owner (˙p). old owner (˙p) also updates its set of neighbours. The new joiner as well as old owner (˙p) inform their neighbours of the changes to the space allocation, In fact, each node has to send an immediate update of its assigned region, followed by periodic HEARTBEAT refresh messages, to all its neighbours.

When a node joins a CAN, only the neighbouring nodes in the coordinate space are required to participate. The overhead is thus of the order of the number of neighbours, which is $O(d)$ and independent of $n$.

### 5.4.2 CAN routing

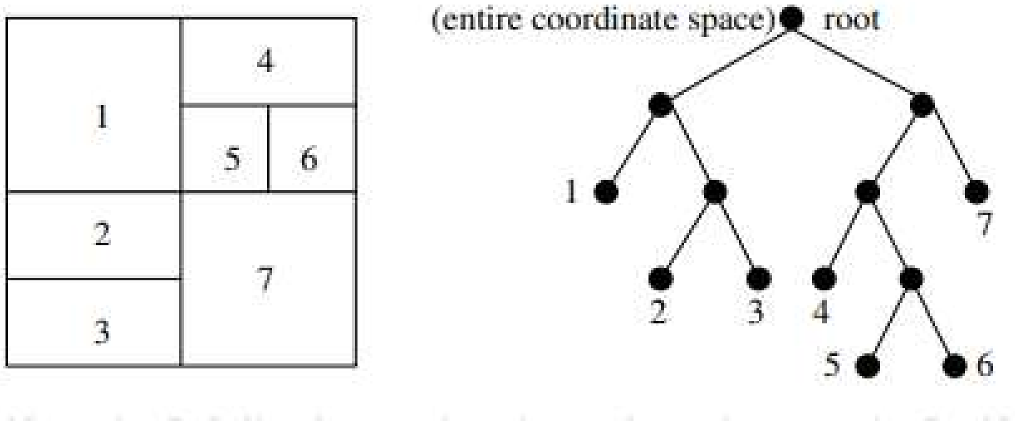
- CAN routing uses the straight-line path from the source to the destination in the logical Euclidean space.
- This routing is realized as follows. Each node maintains a routing table that tracks its neighbor nodes in the log-ical coordinate space. In d-dimensional space, nodes x and y are neigh-bors if the coordinate ranges of their regions overlap in d − 1 dimensions, and abut in one dimension.



- The routing table at each node tracks the IP address and the virtual coor-dinate region of each neighbor. To locate value v, its key k v is mapped to a point p- whose coordinates are used in the message header.
- Knowing the neighbors' region coordinates, each node follows simple greedy routing by forwarding the message to that neighbor having coordinates that are closest to the destination's coordinates

### 5.4.3 CAN Maintainence

- Voluntary departure: Hand over region and (*key, value*) tuples to a neighbor. Neighbor choice: formation of a convex region after merger of regions
- Otherwise, neighbor with smallest volume. However, regions are not merged and neighbor handles both regions until background reassignment protocol is run.
- Node failure detected when periodic HEARTBEAT message not received by neighbors. They then run a TAKEOVER protocol to decide which neighbor will own dead node's region. This protocol favors region with smallest volume.
- Despite TAKEOVER protocol, the (*key, value*) tuples remain lost until background region reassignment protocol is run.
- Background reassignment protocol: for 1-1 load balancing, restore 1-1 node to region assignment, and prevent fragmentation.



### 5.4.4 CAN Optimizations

Improve per-hop latency, path length, fault tolerance, availability, and load balancing. These techniques typically demonstrate a trade-off.

- **Multiple dimensions.** As the path length is $O(d \cdot n^{1/d})$, increasing the number of dimensions decreases the path length and increases routing fault tolerance at the expense of larger state space per node.
- **Multiple realities or coordinate spaces.** The same node will store different $(k, v)$ tuples belonging to the region assigned to it in each reality, and will also have a different neighbour set. The data contents $(k, v)$ get replicated, leading to higher availability. Furthermore, the multiple copies of each $(k, v)$ tuple offer a choice.
- Routing fault tolerance also improves.
- Use delay metric instead of Cartesian metric for routing
- Overloading coordinate regions by having multiple nodes assigned to each region. Path length and latency can reduce, fault tolerance improves, per-hop latency decreases.
- Use multiple hash functions. Equivalent to using multiple realities. Topologically sensitive overlay. This can greatly reduce per-hop latency.

**CAN Complexity: $O(d\cdot)$ for a joiner. $O(d/4\ log\ (n))$ for routing.
Node departure $O(d^2)$.**

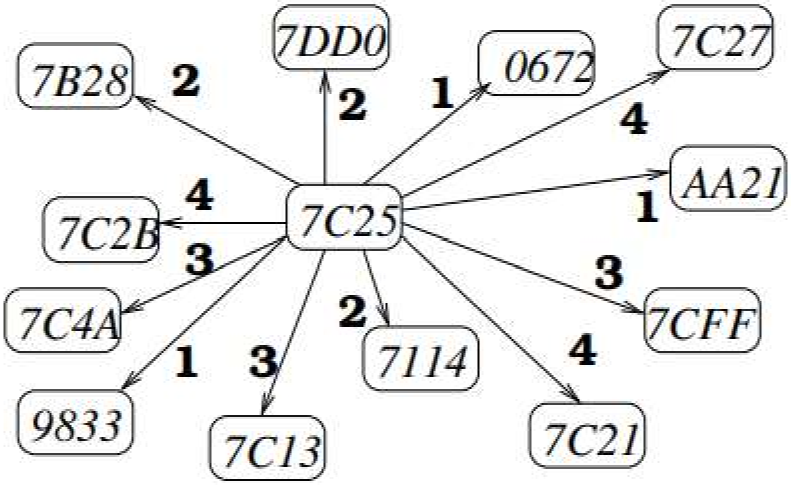
## 5.5 Tapestry

- The Tapestry P2P overlay network provides efficient scalable location-independent routing to locate objects distributed across the Tapestry nodes
- Nodes and objects are assigned IDs from common space via a distributed hashing.
- Hashed node ids are termed VIDs or $v_{id}$ . Hashed object identifiers are termed GUIDs or $O_G$ .
- ID space typically has $m$ = 160 bits, and is expressed in hexadecimal.
- If a node $v$ exists such that $v_{id} = O_G$ exists, then that $v$ become the root. If such a $v$ does not exist, then another unique node sharing the largest common prefix with $O_G$ is chosen to be ***the surrogate root.***
- The object $O_G$ is stored at the root, or the root has a direct pointer to the object.

- To access object $O$, reach the root (real or surrogate) using ***prefix routing*** Prefix routing to select the next hop is done by increasing the prefix match of the next hop's VID with the destination $O_{GR}$ . Thus, a message destined for
  $O_{GR} = 62C\ 35$ could be routed along nodes with VIDs 6****, then 62***,
  then 62C**, then 62C3*, and then to 62C35

## 5.5.1 Tapestry - Routing Table

- Let $M = 2^m$. The routing table at node $v_{id}$ contains $b \cdot log_b\ M$ entries, organized in
- $log_bM$ levels $i = 1 \ldots log_bM$. Each entry is of the form *($w_{id}$ , IP address)*.
- Each entry denotes some "neighbour" node VIDs with a $(i - 1)$-digit prefix match with $v_{id}$ – thus, the entry's $w_{id}$ matches $v_{id}$ in the $(i - 1)$-digit prefix. Further, in level $i$, for each digit $j$ in the chosen base (e.g., $0, 1, \ldots E, F$ when $b = 16$), there is an entry for which the $i^{th}$ digit position is $j$.
- For each forward pointer, there is a backward pointer.

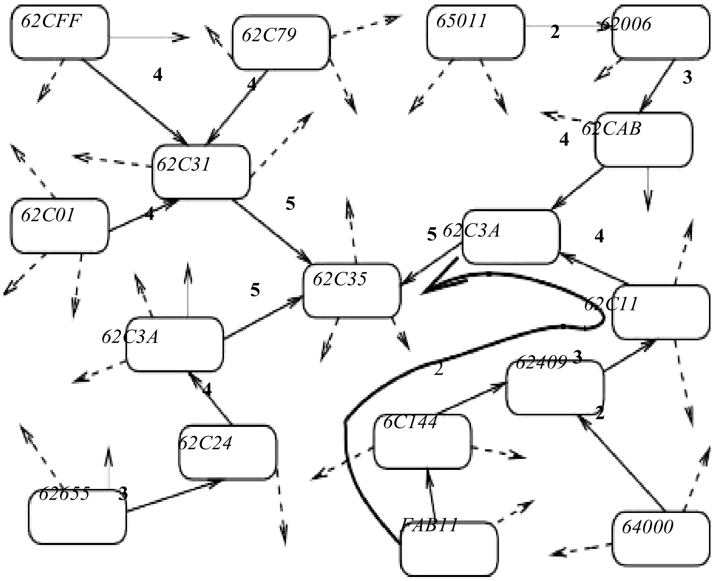


Some example links at node with identifier "7C25". Three links each of levels 1 through 4 are labeled.

## 5.5.2 Tapestry: Routing

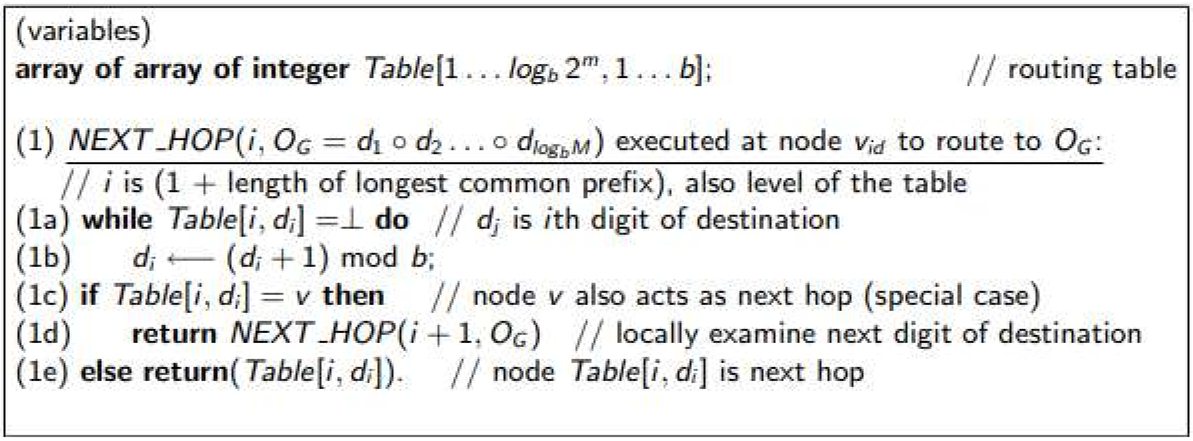
- The $j^{th}$ entry in level $i$ may not exist because no node meets the criterion. This is a *hole* in the routing table.
- ***Surrogate routing*** can be used to route around holes. If the $j^{th}$ entry in level $i$ should be chosen but is missing, route to the next non-empty entry in level $i$ , using wraparound if needed. All the levels from 1 to $log_b\ 2^m$ need to be considered in routing, thus requiring $log_b\ 2^m$ hops.

An example of routing from FAB11 to 62C35. The numbers on the arrows show the level of the routing table



### 5.5.3 Tapestry: RoutingAlgorithm

- Surrogate routing leads to a unique root.
- For each $v_{id}$ , the routing algorithm identifies a unique spanning tree rooted at $v_{id}$ .



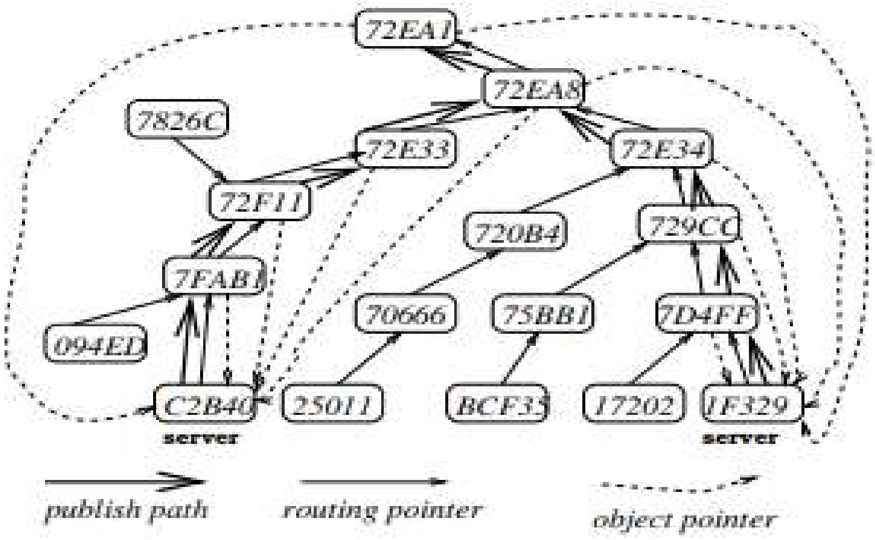
```
(variables)
array of array of integer Table[1 . . . log_b 2^m, 1 . . . b];              // routing table

(1) NEXT_HOP(i, O_G = d_1 ∘ d_2 . . . ∘ d_{log_b M}) executed at node v_id to route to O_G:
    // i is (1 + length of longest common prefix), also level of the table
(1a) while Table[i, d_i] =⊥ do    // d_j is ith digit of destination
(1b)      d_i ⟵ (d_i + 1) mod b;
(1c) if Table[i, d_i] = v then     // node v also acts as next hop (special case)
(1d)      return NEXT_HOP(i + 1, O_G)    // locally examine next digit of destination
(1e) else return(Table[i, d_i]).     // node Table[i, d_i] is next hop
```

### 5.5.4 Tapestry: Object Publication and Object Search

- The unique spanning tree used to route to $v_{id}$ is used to publish and locate an object whose unique root identifier $O_{GR}$ is $v_{id}$ .
- A server S that stores object *O* having GUID $O_G$ and root $O_{GR}$ periodically publishes the object by routing a *publish* message from *S* towards $O_{GR}$ .
- At each hop and including the root node $O_{GR}$ , the *publish* message creates a pointer to the object
- This is the directory info and is maintained in *soft-state*.
- To search for an object *O* with GUID $O_G$ , a client sends a query destined for the root $O_{GR}$ .

- ❯ Along the $log_b 2^m$ hops, if a node finds a pointer to the object residing on server $S$, the node redirects the query directly to $S$.
- ❯ Otherwise, it forwards the query towards the root $O_{GR}$ which is guaranteedto have the pointer for the location mapping.

- A query gets redirected directly to the object as soon as the query path overlaps the publish path towards the same root



An example showing publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40. A query for the object from 094ED will find the object pointer at 7FAB1. A query from 7826C will find the object pointer at 72F11. A query from BCF35 will find the object pointer at 729CC.

### 5.5.5 Tapestry: Node Insertions

- For any node *Y* on the path between a publisher of object *O* and the root
- $G_{OR}$ , node *Y* should have a pointer to *O*.
- Nodes which have a hole in their routing table should be notified if the insertion of node *X* can fill that hole.
- If *X* becomes the new root of existing objects, references to those objects should now lead to *X* .
- The routing table for node *X* must be constructed.
- The nodes near *X* should include *X* in their routing tables to perform more efficient routing.

The main steps in node insertion are as follows:

1. Node X uses some gateway node into the Tapestry network to route a message to itself. This leads to its “surrogate,” i.e., the root node with identifier closest to that of itself (which is

$X_{id}$). The surrogate Z identifies the length of the longest common prefix that $Z_{id}$ shares with $X_{id}$.

2. Node Z initiates a MULTICAST-CONVERGECAST on behalf of X by essentially creating a logical spanning tree as follows. Acting as a root,

   Z contacts all the j nodes, for all $j \in 0\ 1\ b-1$ (tree level 1). These are the nodes with prefix followed by digit j. Each such (level 1) node Z1 contacts all the prefix Z1 + 1 j nodes, for all $j \in 0\ 1\ b-1$ (tree level 2). This continues up to level $\log_b 2^m -$ and completes the MULTICAST. The nodes at this level are the leaves

### 5.5.6 Tapestry: Node Deletions and Failures

**Node deletion**

- Node *A* informs the nodes to which it has (routing) backpointers. It also provides them with replacement entries for each level from its routing table. This is to prevent holes in their routing tables. (The notified neighbours can periodically run the nearest neighbour algorithm to fine-tune their tables.)
- The servers to which *A* has object pointers are also notified. The notified servers send object republish messages.
- During the above steps, node *A* routes messages to objects rooted at itself to their new roots. On completion of the above steps, node *A* informs the nodes reachable via its backpointers and forward pointers that it is leaving, and then leaves.

Node failures: Repair the object location pointers, routing tables and mesh, using the redundancy in the Tapestry routing network. Refer to the book for the algorithms

**Complexity**

- A search for an object expected to take ($log_b 2^m$) hops. However, the routing tables are optimized to identify nearest neighbour hops (as per the space metric). Thus, the latency for each hop is expected to be small, compared to that for CAN and Chord protocols.
- The size of the routing table at each node is $c\ b\ log_b 2^m$, where *c* is the constant that limits the size of the neighbour set that is maintained for fault-tolerance.

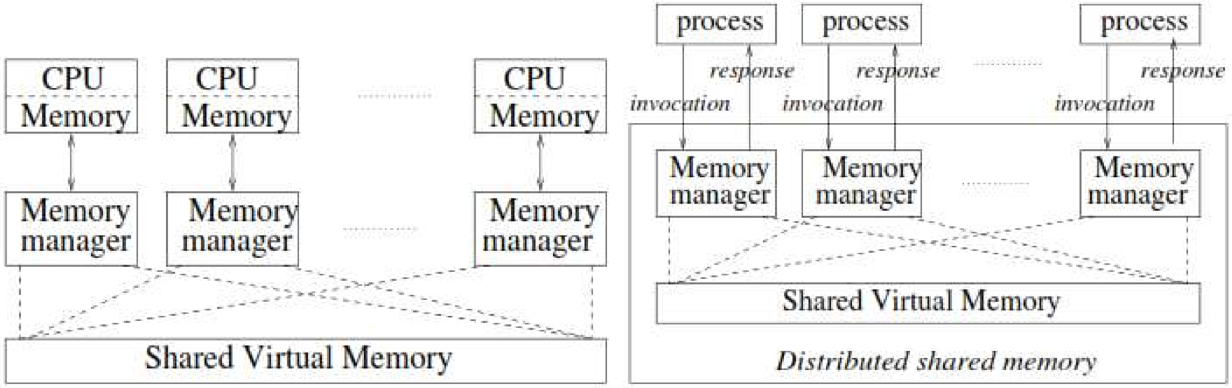
The larger the Tapestry network, the more efficient is the performance. Hence, better if different applications share the same overlay.

## 5.6 Distributed Shared Memory

### 5.6.1 Distributed Shared Memory Abstractions

Distributed shared memory (DSM) is an abstraction provided to the programmer of a distributed system. It gives the impression of a single monolithic memory, as in traditional von Neumann architecture. Programmers access the data across the network using only *read* and *write* primitives, as they would in a uniprocessor system. Programmers do not have to deal with *send* and *receive* communication primitives and the ensuing complexity of dealing explicitly with synchronization and consistency in the message-passing model.

- communicate with Read/Write ops in shared virtual space No Send and Receive primitives to be used by application
    - ❩ Under covers, Send and Receive used by DSM manager
- *Locking is too restrictive; need concurrent access*
- With replica management, problem of consistency arises!



## <u>5.6.2 Advantages/Disadvantages of DSM</u>

**Advantages:**

Shields programmer from Send/Receive primitives
Single address space; simplifies passing-by-reference and passing complex data structures
Exploit locality-of-reference when a block is moved
DSM uses simpler software interfaces, and cheaper off-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
No memory access bottleneck, as no single bus Large **<u>virtual memory space</u>**

- DSM programs portable as they use common DSM programming interface Disadvantages:
- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than msg-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

## <u>5.6.3 Issues in Implementing DSM Software</u>

- Semantics for concurrent access must be clearly specified Semantics – replication? partial? full? read-only? write-only? Locations for replication (for optimization)
- If not full replication, determine location of nearest data for access Reduce delays, # msgs to implement the semantics of concurrent access
- Data is replicated or cached Remote access by HW or SW
- Caching/replication controlled by HW or SW
- DSM controlled by memory management SW, OS, language run-time system

### 5.6.4 Comparison of Early DSM Systems

| Type of DSM | Examples | Management | Caching | Remote access |
|---|---|---|---|---|
| single-bus multiprocessor | Firefly, Sequent | by MMU | hardware control | by hardware |
| switched multiprocessor | Alewife, Dash | by MMU | hardware control | by hardware |
| NUMA system | Butterfly, CM* | by OS | software control | by hardware |
| Page-based DSM | Ivy, Mirage | by OS | software control | by software |
| Shared variable DSM | Midway, Munin | by language runtime system | software control | by software |
| Shared object DSM | Linda, Orca | by language runtime system | software control | by software |

## 5.7 Memory consistency models

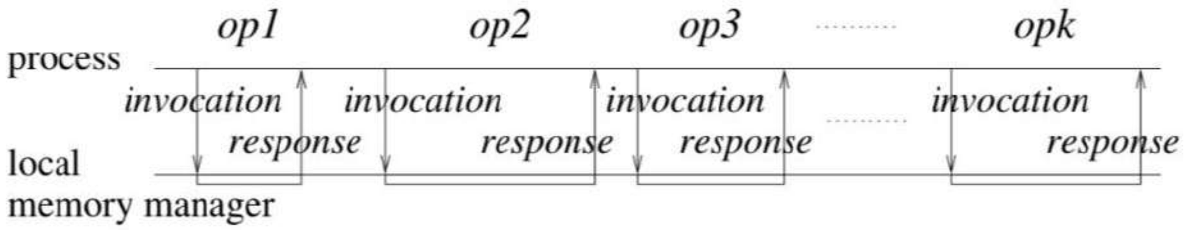
The *memory consistency model* defines the set of allowable memory access orderings.

**Memory Coherence**

*Memory coherence* is the ability of the system to execute memory operations correctly.

- *$s_i$ memory operations by $P_i$*
- $(s_1 + s_2 + \ldots s_n)!/(s_1!s_2! \ldots s_n!)$ possible interleavings
- *Memory coherence model defines which interleavings are permitted Traditionally, Read returns the value written by the most recent Write "Most recent" Write is ambiguous with replicas and concurrent accesses*

DSM consistency model is a *contract* between DSM system and application programmer



***Sequential invocations and responses in a DSM system, without any pipelining***

### 5.7.1 Strict Consistency/Linearizability/Atomic Consistency

### Strict consistency

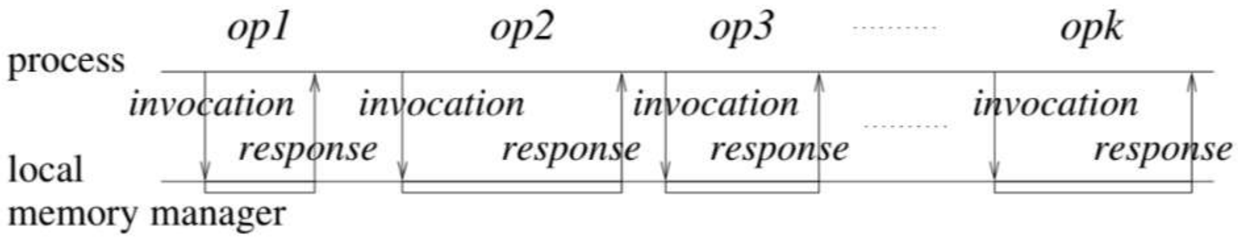
The strictest model, corresponding to the notion of correctness on the tradi-tional Von Neumann architecture or the uniprocessor machine, requires that any *Read* to a location (variable) should return the value written by the most recent *Write* to that location (variable).

Two salient features of such a system are the following: (i) a common global time axis is implicitly available in a uniprocessor system; (ii) each write is immediately visible to all processes.

1. A Read should return the most recent value written, per a global time axis. For operations that overlap per the global time axis, the following must hold.

2 All operations appear to be atomic and sequentially executed.

3 All processors see the same order of events, equivalent to the global time ordering of non-overlapping events.

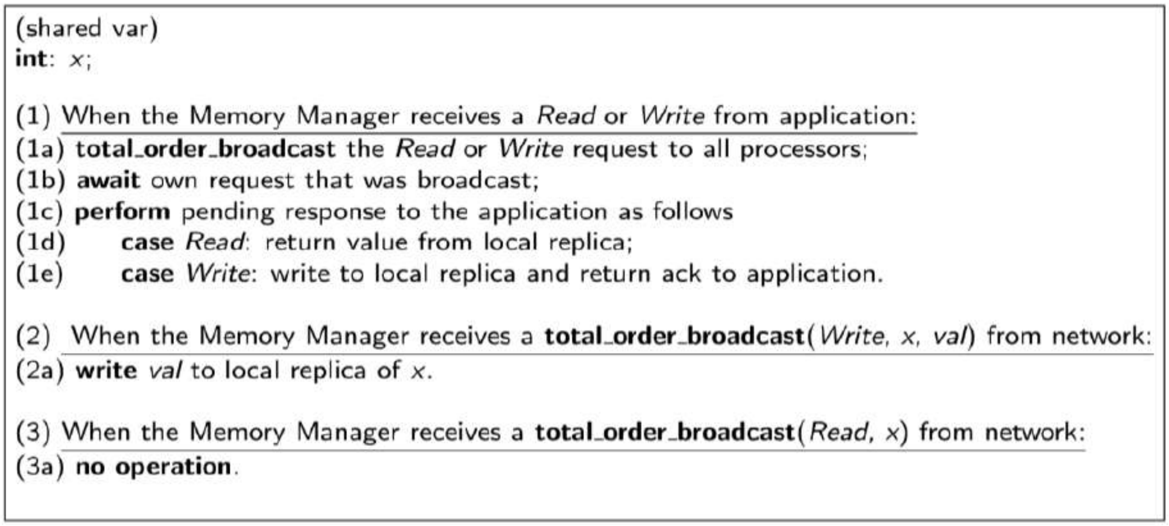


Sequential invocations and responses to each Read or Write operation.

Strict Consistency / Linearizability: Examples

Linearlzability: Implementation

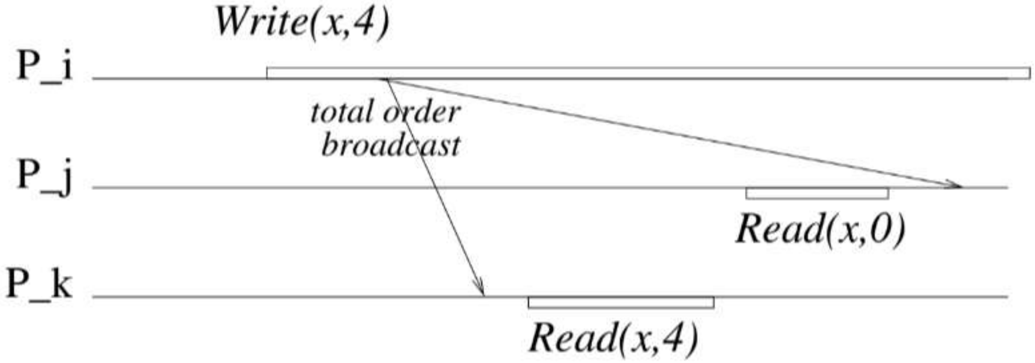
- Simulating global time axis is expensive.
- Assume full replication, and total order broadcast support.



```
(shared var)
int: x;

(1) When the Memory Manager receives a Read or Write from application:
(1a) total_order_broadcast the Read or Write request to all processors;
(1b) await own request that was broadcast;
(1c) perform pending response to the application as follows
(1d)       case Read: return value from local replica;
(1e)       case Write: write to local replica and return ack to application.

(2)  When the Memory Manager receives a total_order_broadcast(Write, x, val) from network:
(2a) write val to local replica of x.

(3) When the Memory Manager receives a total_order_broadcast(Read, x) from network:
(3a) no operation.
```

**Linearizability: Implementation**

When a Read in simulated at other processes, there is a no-op. Why do Reads participate in total order broadcasts?
Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



## 5.7.2 Sequential Consistency

Linearizability or strict/atomic consistency is difficult to implement because the absence of a global time reference in a distributed system necessitates that the time reference has to be simulated. This is very expensive. Programmers can deal with weaker models. The first weaker model, that of *sequential con-sistency* (SC) was proposed by Lamport and uses logical time reference instead of the global time reference.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

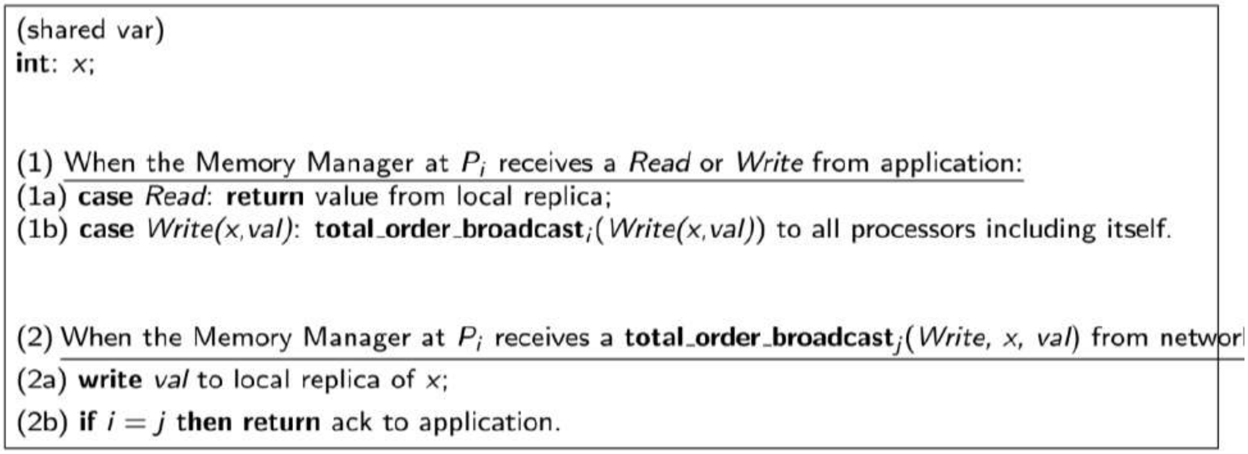
Any interleaving of the operations from the different processors is possible. But all processors must see *the same* interleaving. Even if two operations from different processors (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the *common* sequential order seen by all. See examples used for linearizability

Only Writes participate in total order BCs. Reads do not because:

- all consecutive operations by the same processor are ordered in that same order (no pipelining), and
- *Read* operations by different processors are independent of each other; to be ordered only with respect to the *Write* operations.

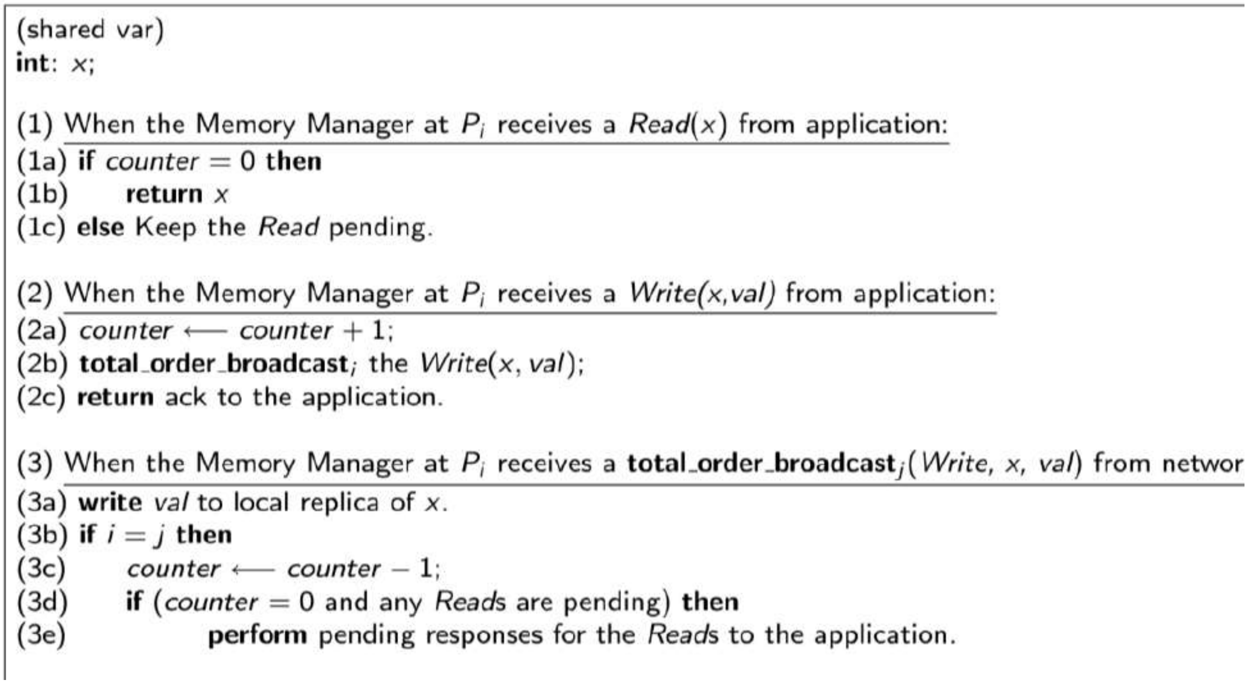
Direct simplification of the LIN algorithm. Reads executed atomically. Not so for Writes. Suitable for Read-intensive programs.

**Sequential Consistency using Local Read Algorithm**



```
(shared var)
int: x;


(1) When the Memory Manager at P_i receives a Read or Write from application:
(1a) case Read: return value from local replica;
(1b) case Write(x,val): total_order_broadcast_i(Write(x,val)) to all processors including itself.


(2) When the Memory Manager at P_i receives a total_order_broadcast_j(Write, x, val) from networ
(2a) write val to local replica of x;
(2b) if i = j then return ack to application.
```

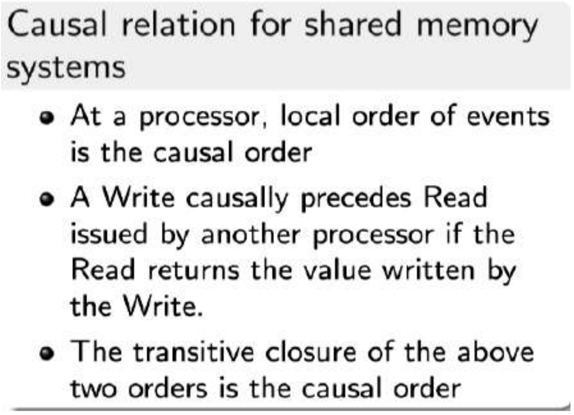
**Sequential Consistency using Local Write Algorithm**



```
(shared var)
int: $x$;

(1) When the Memory Manager at $P_i$ receives a Read($x$) from application:
(1a) if counter $= 0$ then
(1b)     return $x$
(1c) else Keep the Read pending.

(2) When the Memory Manager at $P_i$ receives a Write($x$,val) from application:
(2a) counter $\longleftarrow$ counter $+ 1$;
(2b) total_order_broadcast$_i$ the Write($x$, val);
(2c) return ack to the application.

(3) When the Memory Manager at $P_i$ receives a total_order_broadcast$_j$(Write, $x$, val) from networ
(3a) write val to local replica of $x$.
(3b) if $i = j$ then
(3c)     counter $\longleftarrow$ counter $- 1$;
(3d)     if (counter $= 0$ and any Reads are pending) then
(3e)         perform pending responses for the Reads to the application.
```

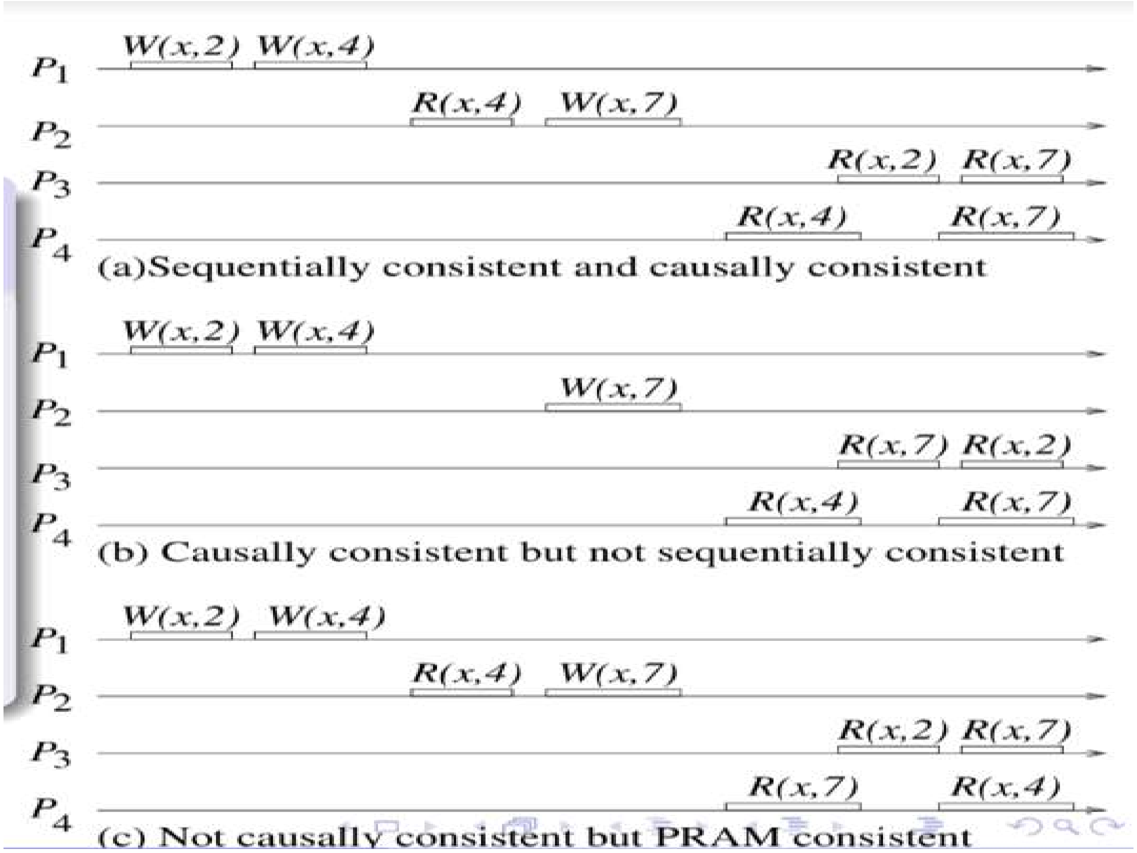
### 5.7.3 Causal Consistency

*In SC, all Write ops should be seen in common order.*
For *causal consistency*, only causally
*related Writes should be seen in common* P1
order.



Causal relation for shared memory systems

- At a processor, local order of events is the causal order
- A Write causally precedes Read issued by another processor if the Read returns the value written by the Write.
- The transitive closure of the above two orders is the causal order



(a)Sequentially consistent and causally consistent

(b) Causally consistent but not sequentially consistent

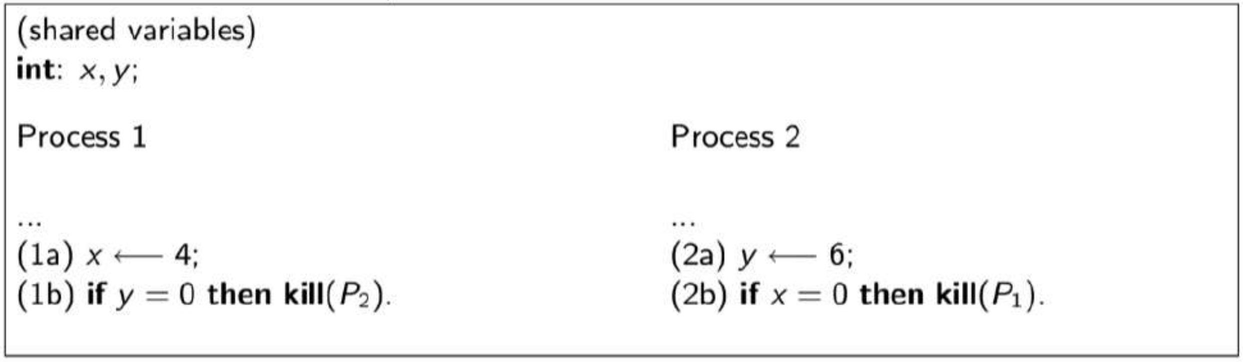
(c) Not causally consistent but PRAM consistent

### 5.7.4 Pipelined RAM or Processor Consistency

**PRAM memory**

- Only Write ops issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

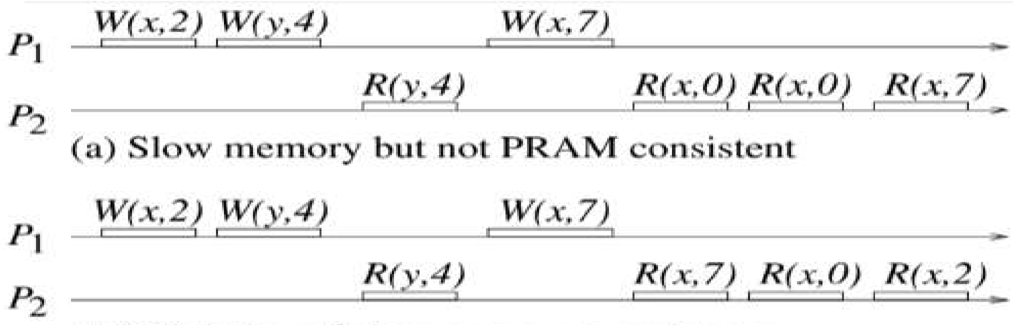
- PRAM can be implemented by FIFO broadcast? PRAM memory can exhibit counter-intuitive behavior, see below.



```
(shared variables)
int: x, y;

Process 1                              Process 2

...                                    ...
(1a) x ⟵ 4;                            (2a) y ⟵ 6;
(1b) if y = 0 then kill(P2).           (2b) if x = 0 then kill(P1).
```

### 5.7.5 Slow Memory

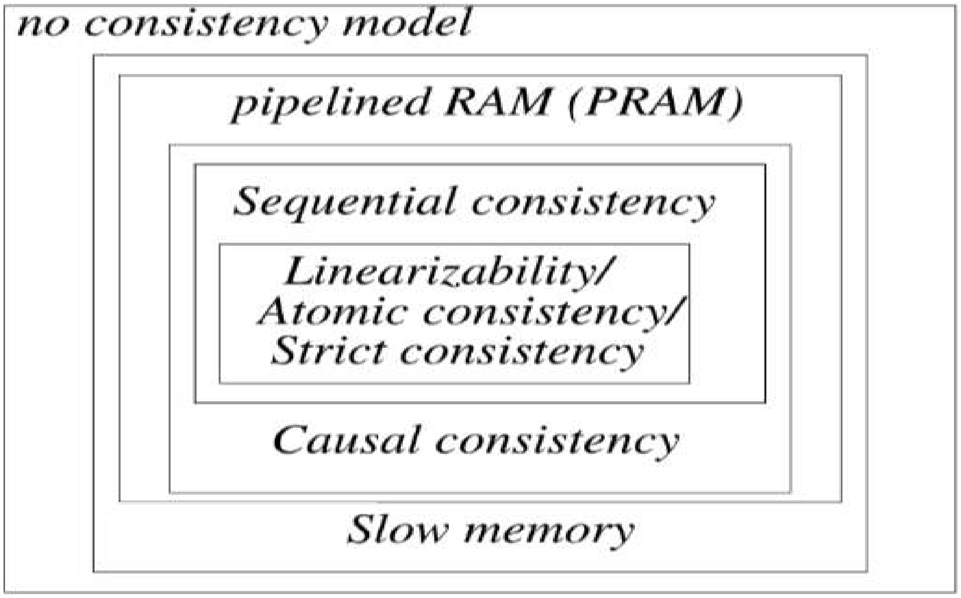
The next weaker consistency model is that of *slow memory*]. This model represents a location-relative weakening of the PRAM model. In this model, only all *Write* operations issued by the same processor and to the same memory location must be observed in the same order by all the processors.



(a) Slow memory but not PRAM consistent

(b) Violation of slow memory consistency

### 5.7.6 Hierarchy of Consistency Models



Synchronization-based Consistency Models: Weak Consistency
Consistency conditions apply only to special
synchronization” instructions, e.g.,

**barrier synchronization**

Non-sync statements may be executed in any order by various processors.
E.g.,weak consistency, release consistency, entry consistency

**Weak consistency:**

All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.

- Accesses to sync variables are sequentially consistent
- Access to sync variable is not permitted unless all Writes elsewhere have completed
- No data access is allowed until all previous synchronization variable accesses have

been performed

**Drawback**: cannot tell whether beginning access to shared variables (enter CS), or finished access to shared variables (exit CS).

**Synchronization based Consistency Models:**

Release Consistency and Entry Consistency

Two types of synchronization Variables: *Acquire* and *Release*

**Release Consistency**

*Acquire* indicates CS is to be entered. Hence all *Writes* from other processors should be locally reflected at this instruction

*Release* indicates access to CS is being completed. Hence, all Updates made locally should be propagated to the replicas at other processors.

*Acquire* and *Release* can be defined on a subset of the variables.

If no CS semantics are used, then Acquire and Release act as barrier synchronization variables.

Lazy release consistency: propagate updates on-demand, not the PRAM way.

**Entry Consistency**

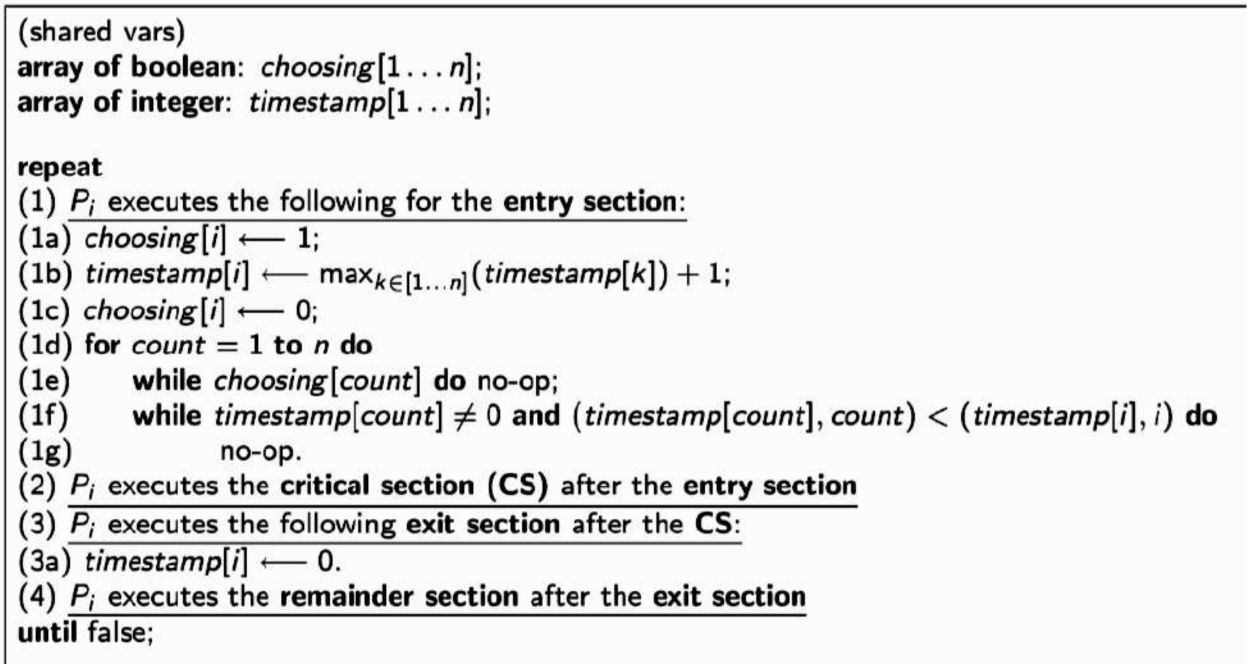
Each ordinary shared variable is associated with a synchronization variable (e.g., lock, barrier)

For Acquire /Release on a synchronization variable, access to only those ordinary variables guarded by the synchronization variables is performed.

## 5.8 Shared Memory Mutual Exclusion: Bakery Algorithm

### 5.8.1 Lamport's bakery algorithm

- Lamport proposed the classical *bakery algorithm* for n-process mutual exclusion in shared memory systems [18]. The algorithm is so called because it mimics the actions that customers follow in a bakery store. A process wanting to enter the critical section picks a token number that is one greater than the elements in the array choosing 1 n .
- Processes enter the critical section in the increasing order of the token numbers. In case of concurrent accesses to choosing by multiple processes, the processes may have the same token number. In this case, a unique *lexicographic order* is defined on the tuple token pid , and this dictates the order in which processes enter the critical section. The algorithm for process i is given in Algorithm.
- The algorithm can be shown to satisfy the three requirements of the critical section problem: (i) mutual exclusion, (ii) bounded waiting, and (iii) progress.



```
(shared vars)
array of boolean: choosing[1 . . . n];
array of integer: timestamp[1 . . . n];

repeat
(1) P_i executes the following for the entry section:
(1a) choosing[i] ⟵ 1;
(1b) timestamp[i] ⟵ max_{k∈[1...n]}(timestamp[k]) + 1;
(1c) choosing[i] ⟵ 0;
(1d) for count = 1 to n do
(1e)      while choosing[count] do no-op;
(1f)      while timestamp[count] ≠ 0 and (timestamp[count], count) < (timestamp[i], i) do
(1g)           no-op.
(2) P_i executes the critical section (CS) after the entry section
(3) P_i executes the following exit section after the CS:
(3a) timestamp[i] ⟵ 0.
(4) P_i executes the remainder section after the exit section
until false;
```

***Mutual exclusion***

- Role of line (1e)? Wait for others' timestamp choice to stabilize ...
- Role of line (1f)? Wait for higher priority (lex. lower timestamp) process to enter CS

***Bounded waiting: $P_i$ can be overtaken by other processes at most once (each)***

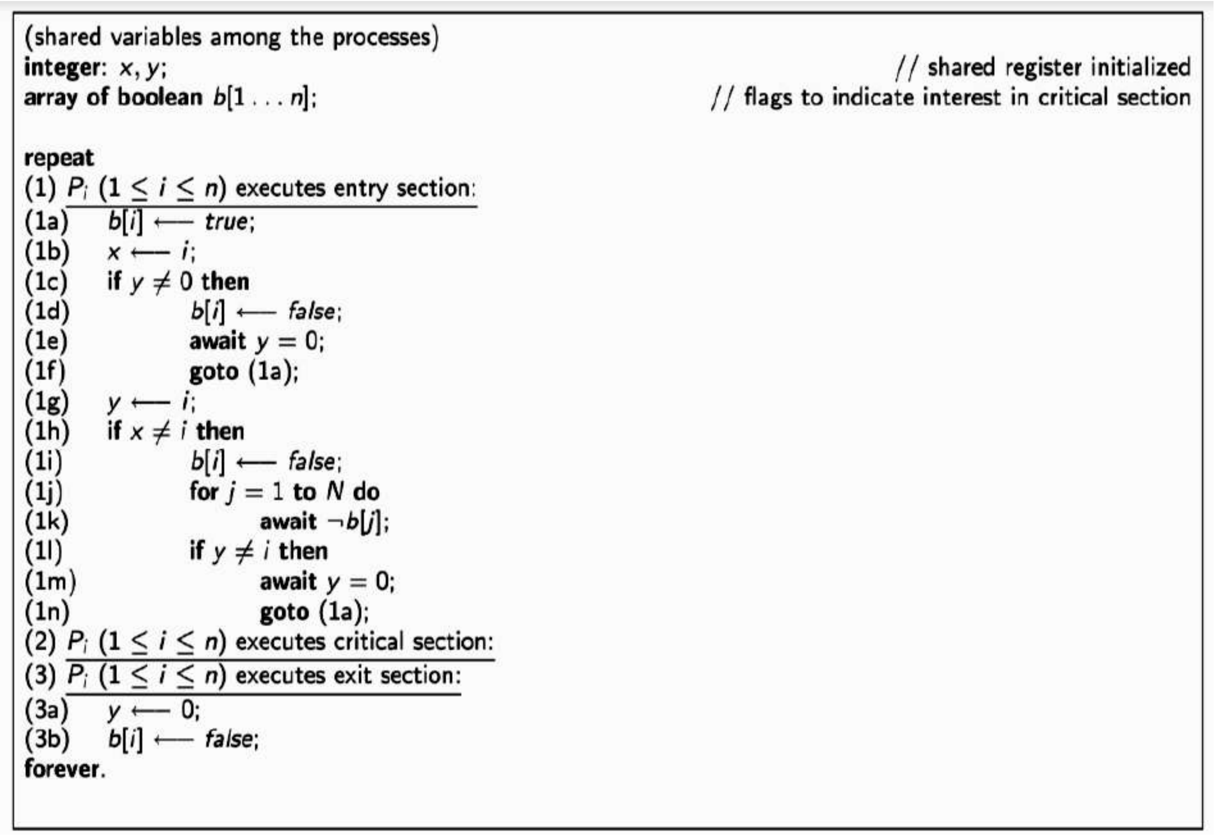
***Progress: lexicographic order is a total order; process with lowest timestamp in lines (1d)-(1g) enters CS***

**Space complexity: lower bound of *n* registers Time complexity: (*n*) time for Bakery algorithm**

## 5.8.2 Lamport's WRWR mechanism and fast mutual exclusion

Lamport's fast mutex algorithm takes $O(1)$ time in the absence of contention. However it compromises on bounded waitin̄g. Usēs $W(x)$¯- $R(y)$ - $W(y)$- $R(x)$ sequence necessary and sufficient to check for contention, and safely enter CS

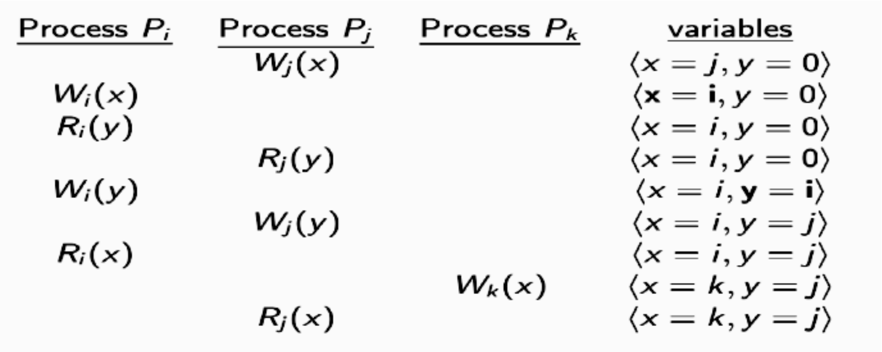
### Lamport's Fast Mutual Exclusion Algorithm



```
(shared variables among the processes)
integer: $x, y$;                                  // shared register initialized
array of boolean $b[1 \ldots n]$;                 // flags to indicate interest in critical section

repeat
(1) $P_i$ $(1 \leq i \leq n)$ executes entry section:
(1a)    $b[i] \longleftarrow true$;
(1b)    $x \longleftarrow i$;
(1c)    if $y \neq 0$ then
(1d)            $b[i] \longleftarrow false$;
(1e)            await $y = 0$;
(1f)            goto (1a);
(1g)    $y \longleftarrow i$;
(1h)    if $x \neq i$ then
(1i)            $b[i] \longleftarrow false$;
(1j)            for $j = 1$ to $N$ do
(1k)                    await $\neg b[j]$;
(1l)            if $y \neq i$ then
(1m)                    await $y = 0$;
(1n)                    goto (1a);
(2) $P_i$ $(1 \leq i \leq n)$ executes critical section:
(3) $P_i$ $(1 \leq i \leq n)$ executes exit section:
(3a)    $y \longleftarrow 0$;
(3b)    $b[i] \longleftarrow false$;
forever.
```

## Shared Memory: Fast Mutual Exclusion Algorithm

Need for a boolean vector of size n: For Pi, there needs to be a trace of its identity and that it had written to the mutex variables. Other processes need to know who (and when) leaves the CS. Hence need for a boolean array b[1..n].



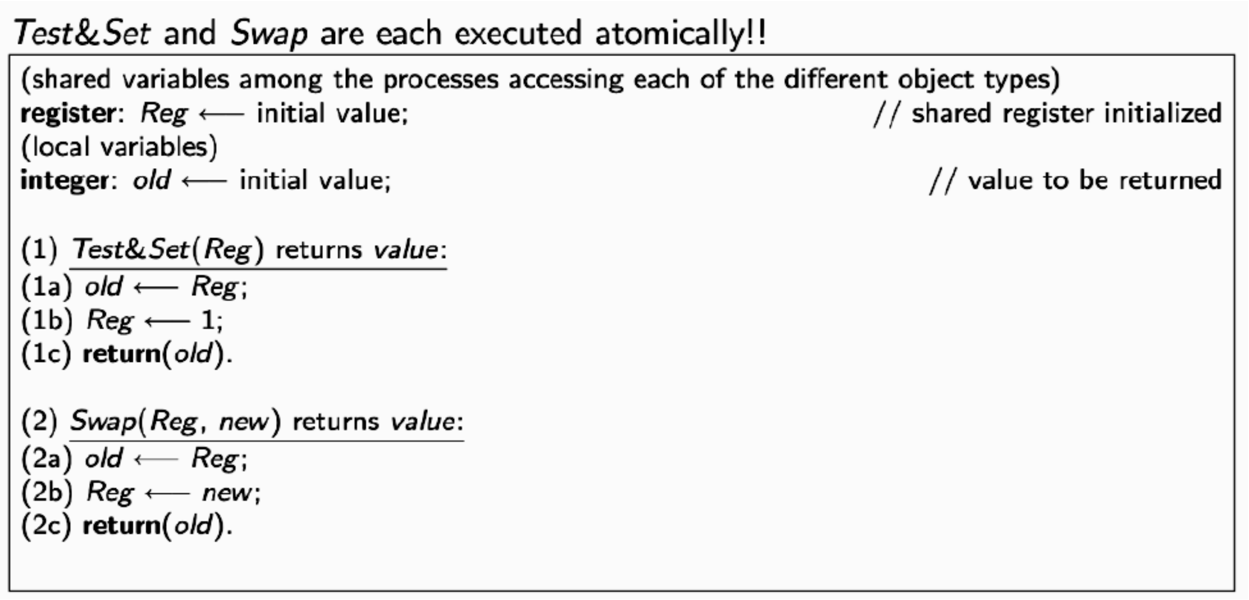
| Process $P_i$ | Process $P_j$ | Process $P_k$ | variables |
|---|---|---|---|
| | $W_j(x)$ | | $\langle x = j, y = 0 \rangle$ |
| $W_i(x)$ | | | $\langle \mathbf{x} = \mathbf{i}, y = 0 \rangle$ |
| $R_i(y)$ | | | $\langle x = i, y = 0 \rangle$ |
| | $R_j(y)$ | | $\langle x = i, y = 0 \rangle$ |
| $W_i(y)$ | | | $\langle x = i, \mathbf{y} = \mathbf{i} \rangle$ |
| | $W_j(y)$ | | $\langle x = i, y = j \rangle$ |
| $R_i(x)$ | | | $\langle x = i, y = j \rangle$ |
| | | $W_k(x)$ | $\langle x = k, y = j \rangle$ |
| | $R_j(x)$ | | $\langle x = k, y = j \rangle$ |

Examine all possible race conditions in algorithm code to analyze the algorithm.

## 5.8.3 Hardware Support for Mutual Exclusion

Hardware support can allow for special instructions that perform two or more operations atomically.
Test&Set and Swap are each executed atomically!!

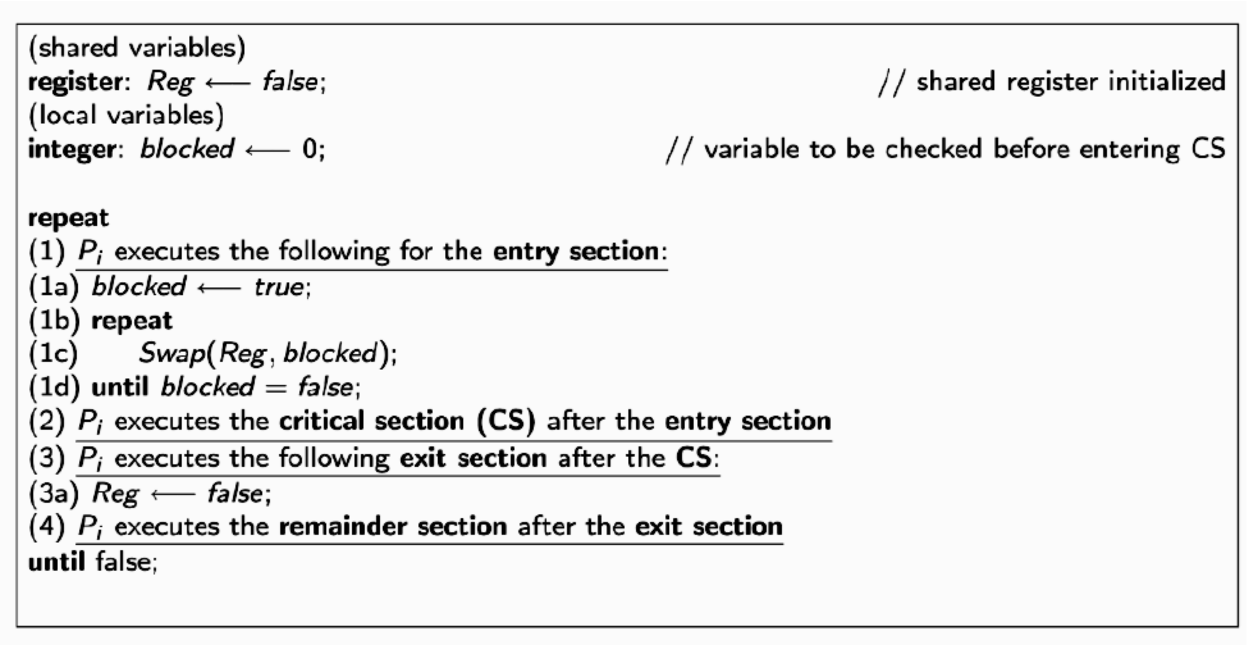
## Definitions of synchronization operations Test&Set and Swap.



*Test&Set* and *Swap* are each executed atomically!!

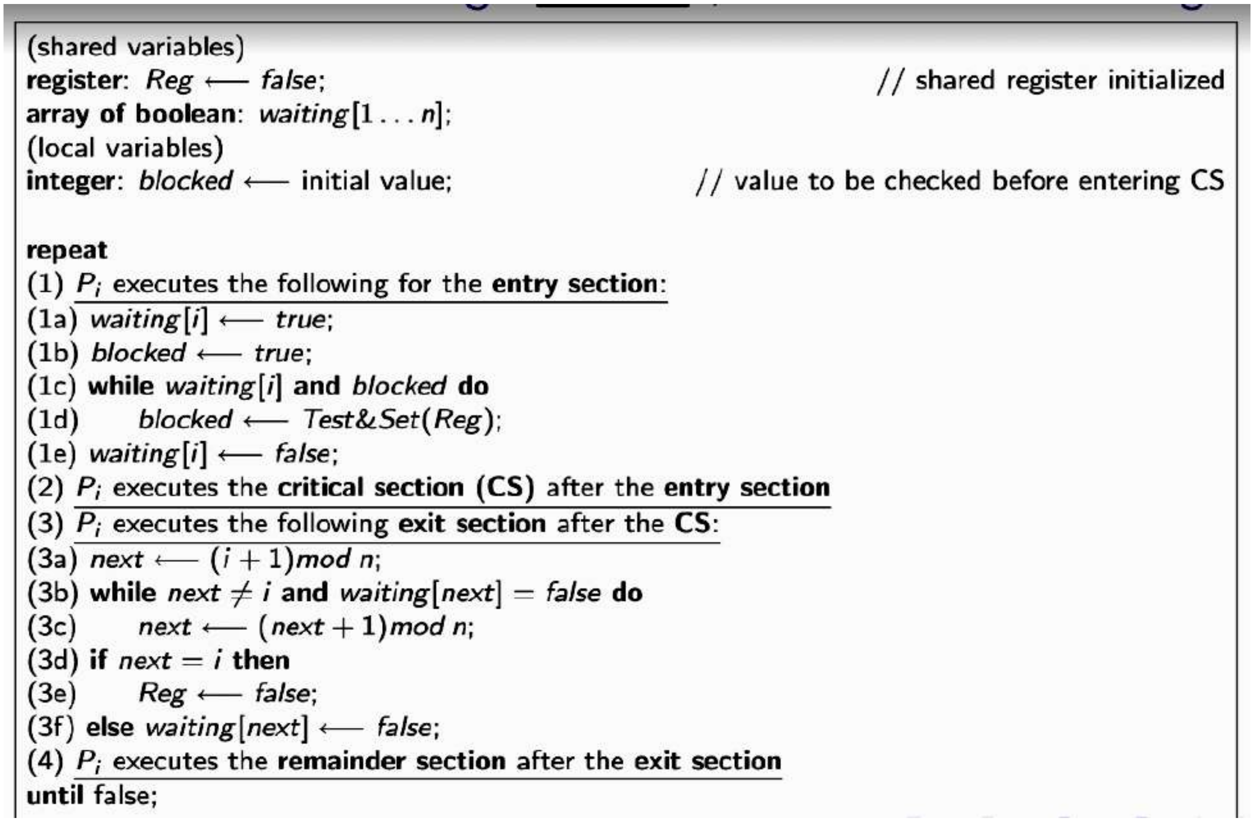
```
(shared variables among the processes accessing each of the different object types)
register: $Reg \longleftarrow$ initial value;                    // shared register initialized
(local variables)
integer: $old \longleftarrow$ initial value;                     // value to be returned

(1) Test&Set(Reg) returns value:
(1a) $old \longleftarrow Reg$;
(1b) $Reg \longleftarrow 1$;
(1c) return(old).

(2) Swap(Reg, new) returns value:
(2a) $old \longleftarrow Reg$;
(2b) $Reg \longleftarrow new$;
(2c) return(old).
```

## Mutual Exclusion using Swap



```
(shared variables)
register: $Reg \longleftarrow false$;                            // shared register initialized
(local variables)
integer: $blocked \longleftarrow 0$;              // variable to be checked before entering CS

repeat
(1) $P_i$ executes the following for the entry section:
(1a) $blocked \longleftarrow true$;
(1b) repeat
(1c)       Swap(Reg, blocked);
(1d) until $blocked = false$;
(2) $P_i$ executes the critical section (CS) after the entry section
(3) $P_i$ executes the following exit section after the CS:
(3a) $Reg \longleftarrow false$;
(4) $P_i$ executes the remainder section after the exit section
until false;
```

## Mutual Exclusion using *Test&Set*, with Bounded Waiting



```
(shared variables)
register: $Reg \longleftarrow false$;                      // shared register initialized
array of boolean: $waiting[1 \ldots n]$;
(local variables)
integer: $blocked \longleftarrow$ initial value;         // value to be checked before entering CS

repeat
(1) $P_i$ executes the following for the entry section:
(1a) $waiting[i] \longleftarrow true$;
(1b) $blocked \longleftarrow true$;
(1c) while $waiting[i]$ and $blocked$ do
(1d)      $blocked \longleftarrow Test\&Set(Reg)$;
(1e) $waiting[i] \longleftarrow false$;
(2) $P_i$ executes the critical section (CS) after the entry section
(3) $P_i$ executes the following exit section after the CS:
(3a) $next \longleftarrow (i+1) mod\ n$;
(3b) while $next \neq i$ and $waiting[next] = false$ do
(3c)      $next \longleftarrow (next+1) mod\ n$;
(3d) if $next = i$ then
(3e)      $Reg \longleftarrow false$;
(3f) else $waiting[next] \longleftarrow false$;
(4) $P_i$ executes the remainder section after the exit section
until false;
```

***Code shown is for process Pi, 1 ≤ i ≤ n.***

# PART A

## UNIT 4

1. Define rollback recovery.

Rollback recovery treats a distributed system application as a collection of processes that communicate over a network. It achieves fault tolerance by periodically saving the state of a process during the failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work.

2. What is domino effect

The program may have to roll back past more than one local checkpoint to achieve a consistent global state. In the worst case, the program has to roll all the way back to the beginning. This extended roll back is called the ``*domino effect*".

3. What are the Techniques that avoid domino effect?

— Coordinated checkpointing rollback recovery

processes coordinate their checkpoints to form a system-wide consistent state

— Communication-induced checkpointing rollback recovery

forces each process to take checkpoints based on information piggybacked on the application

— Log-based rollback recovery

combines checkpointing with logging of non-deterministic events
relies on piecewise deterministic (PWD) assumption

4. Describe local check pointing?

- In distributed systems, all processes save their local states at certain instants of time. This saved state is known as a local checkpoint.
- A local checkpoint is a snapshot of the state of the process at a given instance and the event of recording the state of a process is called local checkpointing.
- The contents of a checkpoint depend upon the application context and the checkpointing method being used.

5. What is meant by “outside world process (OWP).”?

- A distributed system often interacts with the outside worldto receive input data or deliver the outcome of a computation
- Outside World Process (OWP)
  a special process that interacts with the rest of the system through message passing

6. What is a global state and Consistent global state of a distributed system

Global state:

a collection of the individual states of all participating processes and the states of the communication channels

Consistent global state

a global state that may occur during a failure-free execution of distribution of distributed computation if a process‟s state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message

7. What is global checkpoint and consistent global checkpoint?

A global checkpoint

a set of local checkpoints, one from each process

A consistent global checkpoint

a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before taking its checkpoint

8. Formulate the different types of messages.

- In-transit message
  - messages that have been sent but not yet received
- Lost messages
  - messages whose „send‟ is done but „receive‟ is undone due to rollback
- Delayed messages
  - messages whose „receive‟ is not recorded because the receiving process was either down or the message arrived after rollback
- Orphan messages
  - messages with „receive‟ recorded but message „send‟ not recorded
  - do not arise if processes roll back to a consistent global state
- Duplicate messages
  - arise due to message logging and replaying during process recovery

9. Compare Coordinated with uncoordinated checkpointing.

If each process takes its checkpoints independently, then the system can not avoid the domino effect

- this scheme is called independent or uncoordinated checkpointing
- Coordinated checkpointing rollback recovery

processes coordinate their checkpoints to form a system-wide consistent state

10. Disadvantages of uncoordinated checkpointing

Domino effect during a recovery

Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints

Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm

Not suitable for application with frequent output commits

11. Compare blocking with non blocking checkpointing.

• Blocking Checkpointing

— After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete

— Disadvantages

the computation is blocked during the checkpointing

- Non-blocking Checkpointing

— The processes need not stop their execution while taking checkpoints

— A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

12. What is pessimistic logging protocol.

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation

13. What is Z-dependency.

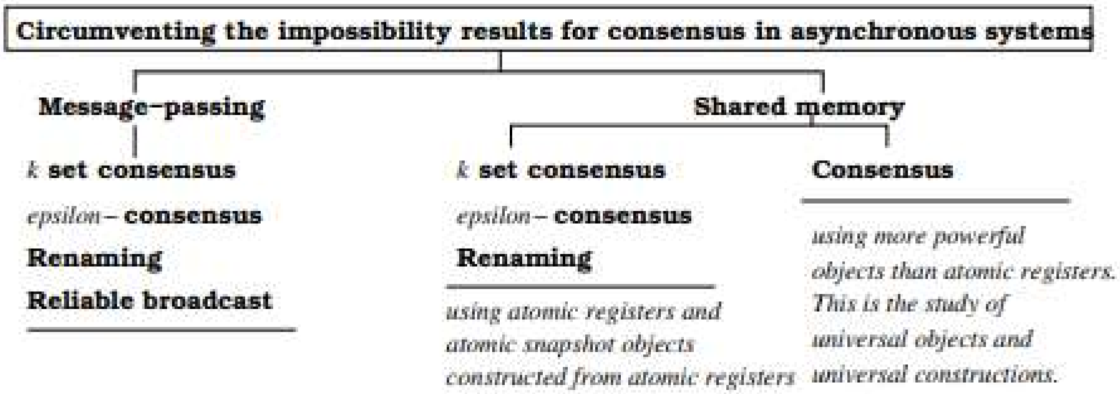
Z-dependency is defined as follows: if a process $P_p$ sends a message to process $P_q$ during its ith checkpoint interval and process $P_q$ receives the message during its jth checkpoint interval, then $P_q$ Z-depends on $P_p$ during $P_p$'s ith checkpoint interval and $P_q$ 's jth checkpoint interval, denoted by $P_p \rightarrow^{i}_{j} P_q$ . If $P_p \rightarrow^{i}_{j} P_q$ and $P_q \rightarrow^{j}_{k} P_r$ , then $P_r$ transitively Z-depends depends on $P_p$ during $P_r$ 's kth checkpoint interval and $P_p$'s ith checkpoint interval, and this is denoted as $P_p \ {}^{*}\!\!\rightarrow^{i}_{k} P_r$ .

14. Two types of communication-induced checkpointing
model-based checkpointing and index-based checkpointing.

Model-based checkpointing
Model-based checkpointing prevents patterns of communications and check-points that could result in inconsistent states among the existing checkpoints.
Index-based checkpointing
Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.

15.What is causal logging
Combines the advantages of both pessimistic and optimistic logging at the expense of a more complex recovery protocol. Like optimistic logging, it does not require synchronous access to the stable storage except during output commit. Like pessimistic logging, it allows each process to commit output independently and never creates orphans, thus isolating processes from the effects of failures at other processes

**16.** Solvable Variants of the Consensus Problem in Async Systems


Circumventing the impossibility results for consensus in asynchronous systems
Message–passing
Shared memory
k set consensus
epsilon – consensus
Renaming
Reliable broadcast
k set consensus
epsilon – consensus
Renaming
using atomic registers and atomic snapshot objects constructed from atomic registers
Consensus
using more powerful objects than atomic registers. This is the study of universal objects and universal constructions.

17. What is agreement variable?

Agreement variable The agreement variable may be boolean or multi-valued, and need not be an integer. When studying some of the more complex algorithms, we will use a boolean variable. This simplifying assumption does not affect the results for other data types, but helps in the abstraction while presenting the algorithms.

**18.** Problem Specifications of Consensus Problem.

Consensus Problem (all processes have an initial value)

Agreement:All non-faulty processes must agree on the same (single) value.

Validity:If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.

Termination:Each non-faulty process must eventually decide on a value.

PART B

| |
|---|
| **What** is rollback? and explain the several types of messages for rollback. (13) |
| **Examine** briefly about global states with examples. (13) |
| **Describe** the issues involved in a failure recovery with the help of a distributed computation. (13) |
| **Elaborate** the various checkpoint-based rollback-recovery techniques.(13) |
| **Describe** the pessimistic logging , optimistic logging and casual logging.(13) |
| **What** are min-process check pointing algorithms? Explain it detail.(7)<br>Examine Deterministic and non-deterministic events. (6) |
| **Summarize** the koo–toueg coordinated check pointing algorithm.(7)<br>Explain the rollback recovery algorithm. (6) |
| ). **Demonstrate** in detail about the juang–venkatesan algorithm for asynchronous check pointing and recovery.(13) |

1. **Discuss** in detail about some assumptions underlying the study of agreement algorithms. (13)

2. What is byzantine agreement problem? **Explain** the two popular flavours of the byzantine agreement problem.

3. **Develop** an overview of the results and lower bounds on solving the consensus problem under different assumptions.

4. **Explain** agreement in (message-passing) synchronous systems with failures.(13)

5. Give byzantine agreement tree algorithm and illustrate with an example. (13)

6. Analyze on phase-king algorithm for consensus.(13)

**PART C**

7. **Design** a system model of distributed system consisting of four processes and explain the interactions with the outside world.(15)

8. **Explain** with examples of consistent and inconsistent states of a distributed system.(15)

9. Consider the following simple check pointing algorithm. A process takes a local checkpoint right after sending a message. **Create** that the last checkpoint at all processes will always be
0. consistent. What are the trade-offs with this method?(15)

1. Give and **analyse** a rigorous proof of the impossibility of a min- process, non blocking check pointing algorithm.(15)

## UNIT 5

1. Desirable characteristics and performance features of P2P systems.

| Features | Performance |
|---|---|
| Self-organizing | Large combined storage, CPU power, and resources |
| Distributed control | Fast search for machines and data objects |
| Role symmetry for nodes | Scalable |
| Anonymity | Efficient management of churn |
| Naming mechanism | Selection of geographically close servers |
| Security, authentication, trust | Redundancy in storage and paths |

2. What is Centralized indexing

- Centralized indexing entails the use of one or a few central servers to store references (indexes) to the data on many peers. The DNS lookup as well as the lookup by some early P2P networks such as Napster used a central directory lookup.

3. What is distributed indexing

- Distributed indexing involves the indexes to the objects at various peers being scattered across other peers throughout the P2P network. In order to access the indexes, a structure is used in the P2P overlay to access the indexes. Distributed indexing is the most challenging of the indexing schemes, and many novel mechanisms have been proposed, most notably the *distributed hash table (DHT)*. Various DHT schemes differ in the hash mapping, search algorithms, diameter for lookup, search diameter, fault-tolerance, and resilience to churn.

4. What is local indexing

- Local indexing requires each peer to index only the local data objects and remote objects need to be searched for. This form of indexing is typically used in unstructured overlays in conjunction with flooding search or random walk search. Gnutella uses local indexing.

5. List out the advantages of unstructured overlays if certain conditions are satisfied:

- Unstructured overlays are efficient when there is some degree of data replication in the network.
- Users are satisfied with a best-effort search.
- The network is not so large as to lead to scalability problems during the search process.

6. What is Gnlutella

Gnutella uses a fully decentralized architecture [16, 17]. In Gnutella logical overlays, nodes index only their local content. The acutal overlay topology can be arbitrary as nodes join and leave randomly. A node joins the Gnutella network by forming a connection to some nodes found in standard Gnutella directory-like databases.

7. Listout message types used by Gnutella:

- *Ping* messages are used to discover hosts, and allow a new host to announce itself.
- *Pong* messages are the responses to *Ping*s. The *Pong* messages indicate the port and (IP) address of the responder, and some information about the amount of data (the number and size of files) that node can make available.
- *Query* messages. The search strategy used is flooding. *Query* messages contain a search string and the minimum download speed required of the potential responder, and are flooded in the network.
- *QueryHit* messages are sent as responses if a node receiving a *Query* detects a local match in response to a query. A *QueryHit* contains the port and address (IP), speed, the number of files

found, and related information. The path traced by a *Query* is recorded in the message, so the *QueryHit* follows the same path in reverse.

8. What is Guided versus unguided search

In unguided or blind search, there is no history of earlier searches, and hence, each search is inherently independent. In guided search, nodes store some history of past searches to aid future searches. Various mechanisms for caching hints to guide and narrow down future searches are used. In this chapter, we focus on unguided searches in the context of unstructured overlays.

9. What is random walkers.

A query is randomly forwarded by a node when it is received. Random walk greatly reduces the message overhead but it increases the search latency. Hence, k *random walkers* can be used. To terminate the k random walkers, a "checking-cum-TTL" strategy is effective.

10. Define simple key lookup algorithm.

A simple key lookup algorithm that requires each node to store only 1 entry in its routing table works as follows. Each node tracks its successor on the ring, in the variable successor; a query for key x is forwarded to the successors of nodes until it reaches the first node such that that node's identifier y is greater than the key x, modulo $2^m$

11. What is CAN.

A content-addressible network (CAN) is essentially an indexing mechanism that maps objects to their locations in the network. The CAN project originated from the observation that the bottleneck to designing a scalable P2P network is this indexing mechanism. An efficient and scalable CAN is useful not only for object location in P2P networks, but also for large-scale storage management systems and wide-area name resolution services that decouple name resolution and the naming scheme.

12. Distributed shared memory (DSM) is an abstraction provided to the programmer of a distributed system. It gives the impression of a single monolithic memory, as in traditional von Neumann architecture. Program-mers access the data across the network using only *read* and *write* primitives, as they would in a uniprocessor system.

13. What is Sequential consistency .

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.

- The operations of each individual processor appear in this sequence in the local program order.

14. Define The *causality relation* for shared memory systems .

- Local order At a processor, the serial order of the events defines the local causal order.

- Inter-process order A *Write* operation causally precedes a *Read* oper-ation issued by another processor if the *Read* returns a value written by the *Write*.
- Transitive closure The transitive closure of the above two relations defines the (global) causal order.

## PART B

| |
|---|
| Explain the structured overlays and unstructured overlays in distributed indexing. (13) |
| i) **What** is meant by napster legacy? Explain.(7)<br>Give a brief account on Indexing mechanisms. (6 |
| **Examine** the chord protocol with simple key lookup algorithm.(13) |
| **Illustrate** in detail about A scalable object location algorithm in chord.(13) |
| ***Discuss*** on managing churn in chord.(13) |
| **Describe** briefly about the following:<br>i) Content-Addressable Network (CAN) initialization (6)<br>ii) CAN routing (7). |
| **Point out** tapestry P2P overlay network and its routing with an example. (13) |
| **Discuss** the CAN maintenance and CAN optimizations. (13) |
| **State** about the consistency models: entry consistency, weak consistency, and release consistency.(13) |
| **Summarize** in detail how node insertion and node deletion are applied in tapestry. (13) |
| **i) Illustrate** the advantages and disadvantages of DSM.(6)<br>**ii)** Point out the main issues in designing a DSM system (7) |
| **Examine** how to implement linearizability (LIN) using total order broadcasts.(13) |
| **Analyse** how to implement Sequential consistency in a distributed system.(13) |
| Describe lamport's bakery algorithm lamport's WRWR mechanism and fast mutual exclusion. (13) |

## PART C

| |
|---|
| User 'A' in delhi wishes to send a file for printing to user 'B' in florida, whose system is connected to a printer; while user 'C' from tokyo wants to save a video file in the hard disk of user 'D' in london. **Analyze** and discuss the required peer-to-peer network architecture.(15) |
| **Evaluate** a formal proof to justify the correctness of algorithm that implements sequential consistency using local read operations.(15) |
| **Develop** a detailed implementation of causal consistency, and provide a correctness argument for your implementation.(15) |