KINGS
ENGINEERING COLLEGE
An Autonomous Institution
Affiliated to Anna University, Chennai

Department of Computer Science And Engineering

Regulation 2024

III Year – V Semester

CS243501-Distributed Computing

CS243501	DISTRIBUTED COMPUTING	L	T	P	C
		3	0	0	3

COURSE OBJECTIVES:

- To introduce the computation and communication models of distributed systems.
- To illustrate the issues of synchronization and collection of information in distributed
- systems
- To describe distributed mutual exclusion and distributed deadlock detection techniques
- To elucidate agreement protocols and fault tolerance mechanisms in distributed systems
- To learn the characteristics of peer-to-peer and distributed shared memory systems.

UNIT I INTRODUCTION 8

Introduction: Definition-Relation to Computer System Components – Motivation-Flynn‘s Taxonomy Message -Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication – Design Issues and Challenges; A Model of Distributed Computations: A Distributed Program – A Model of Distributed Executions –Global State of a Distributed System.

UNIT II MESSAGE ORDERING AND GLOBAL STATE 10

Message Ordering and Group Communication: Message Ordering Paradigms – Asynchronous Execution with Synchronous Communication– Group Communication – Causal Order – Total Order; Global State and Snapshot Recording Algorithms: Introduction – System Model and Definitions.

UNIT III DISTRIBUTED MUTEX AND DEADLOCK 10

Distributed Mutual exclusion Algorithms: Introduction – Preliminaries – Lamport‘s algorithm– RicartAgrawala‘s Algorithm — Token-Based Algorithms – Suzuki-Kasami‘s Broadcast Algorithm; Deadlock Detection in Distributed Systems: Introduction – System Model – Preliminaries – Models of Deadlocks .

UNIT IV CONSENSUS AND RECOVERY 10

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System – Check pointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Check pointing Algorithm.

UNIT V P2P & DISTRIBUTED SHARED MEMORY 7

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.

TOTAL: 45 PERIODS

COURSE OUTCOMES:

Upon the completion of this course, the student will be able to
CO1: Explain the foundations of distributed systems
CO2: Solve synchronization and state consistency problems
CO3 Use resource sharing techniques in distributed systems
CO4: Apply working model of consensus and reliability of distributed systems
CO5: Describe the features of peer-to-peer and distributed shared memory systems

TEXT BOOKS

1.Kshemkalyani Ajay D, Mukesh Singhal, ―Distributed Computing: Principles, Algorithms and Systems‖, Cambridge Press, 2011.

2.George Coulouris, Jean Dollimore and Tim Kindberg, "Distributed Systems Concepts and Design", Fifth Edition, Pearson Education, 2012.

REFERENCES

1.Pradeep L Sinha, ―Distributed Operating Systems: Concepts and Design‖, Prentice Hall of India,2007.

2.Tanenbaum A S, Van Steen M, ―Distributed Systems: Principles and Paradigms‖, Pearson Education, 2007.

3.Liu M L, ―Distributed Computing: Principles and Applications‖, Pearson Education, 2004.

4.Nancy A Lynch, ―Distributed Algorithms‖, Morgan Kaufman Publishers, 2003.

CO's-PO's&PSO'sMAPPING

CO's	PO's												PSO's		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**
1	2	2	3	3	1	1	-	-	2	1	3	3	2	1	1
2	1	3	2	1	2	-	-	-	2	2	2	2	1	3	2
3	2	2	1	3	3	-	-	-	3	2	1	1	1	2	1
4	1	2	2	3	1	-	-	-	3	3	2	1	3	1	1
5	3	3	1	2	3	-	-	-	3	3	3	1	3	2	3
AVg.	1.8	2.4	1.8	2.4	2	0.2	-	-	2.6	2.2	2.2	1.6	2	1.8	1.6

low,2-medium,3-high,'-"-no correlation

1

INTRODUCTION TO DISTRIBUTED SYSTEMS

1. 1 FUNDAMENTALS OF DISTRIBTED COMPUTING

The process of computation was started from working on a single processor. This uni-processor computing can be termed as **centralized computing**. As the demand for the increased processing capability grew high, multiprocessor systems came to existence. The advent of multiprocessor systems, led to the development of distributed systems with high degree of scalability and resource sharing. The modern day parallel computing is a subset of distributed computing

A distributed system is a collection of independent computers, interconnected via a network, capable of collaborating on a task. Distributed computing is computing performed in a distributed system.

Distributed computing is widely used due to advancements in machines; faster and cheaper networks. In distributed systems, the entire network will be viewed as a computer. The multiple systems connected to the network will appear as a single system to the user. Thus the distributed systems hide the complexity of the underlying architecture to the user. Distributed computing is a special version of parallel computing where the processors are in different computers and tasks are distributed to computers over a network.

The definition of distributed systems deals with two aspects that:

- **Deals with hardware:** The machines linked in a distributed system are autonomous.
- **Deals with software:** A distributed system gives an impression to the users that they are dealing with a single system.

Features of Distributed Systems:

- Communication is hidden from users

- Applications interact in uniform and consistent way
- High degree of scalability
- A distributed system is functionally equivalent to the systems of which it is composed.
- Resource sharing is possible in distributed systems.
- Distributed systems act as fault tolerant systems
- Enhanced performance
- No common clock
- Geographical isolation
- Autonomous
- Heterogeneous

Issues in distributed systems

- Concurrency
- Distributed system function in a heterogeneous environment. So adaptability is a major issue.
- Latency
- Memory considerations: The distributed systems work on both local and shared memory.
- Synchronization issues
- Applications must need to adapt gracefully without affecting other parts of the systems in case of failures.
- Since they are widespread, security is a major issue.
- Limits imposed on scalability
- They are less transparent.
- Knowledge about the dynamic network topology is a must.

QOS parameters

The distributed systems must offer the following QOS:

- Performance
- Reliability
- Availability
- Security

Differences between centralized and distributed systems

Centralized Systems	Distributed Systems
In Centralized Systems, several jobs are done on a particular central processing unit(CPU)	In Distributed Systems, jobs are distributed among several processors. The Processor are interconnected by a computer network
They have shared memory and shared variables.	They have no global state (i.e.) no shared memory and no shared variables.
Clocking is present.	No global clock.

1.1.1 Relation to Computer Components

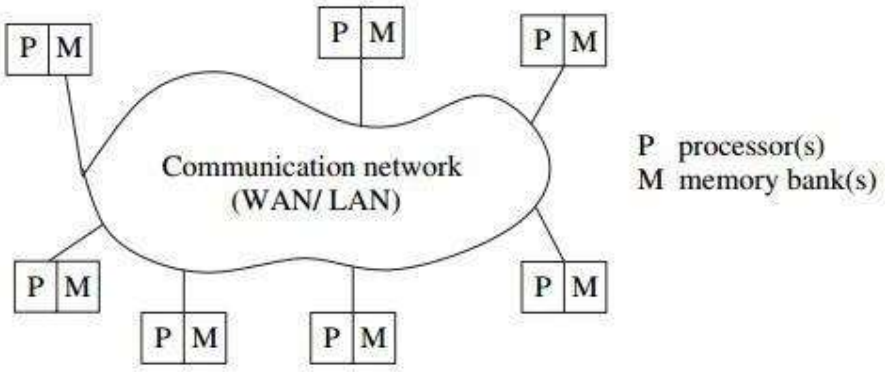


Fig 1.1: Example of a Distributed System

As shown in Fig 1.1, the processors are connected together by interconnecting network. Each system connected to the distributed networks hosts distributed software which is a middleware technology. This drives the Distributed System (DS) at the same time preserves the heterogeneity of the DS. The term **computation or run** in a distributed system is the execution of processes to achieve a common goal.

The interaction of the layers of the network with the operating system and middleware is shown in Fig 1.2. The middleware contains important library functions for facilitating the operations of DS.

Examples of middleware: Object Management Group's (OMG), Common Object Request Broker Architecture (CORBA) [36], Remote Procedure Call (RPC), Message Passing Interface (MPI).

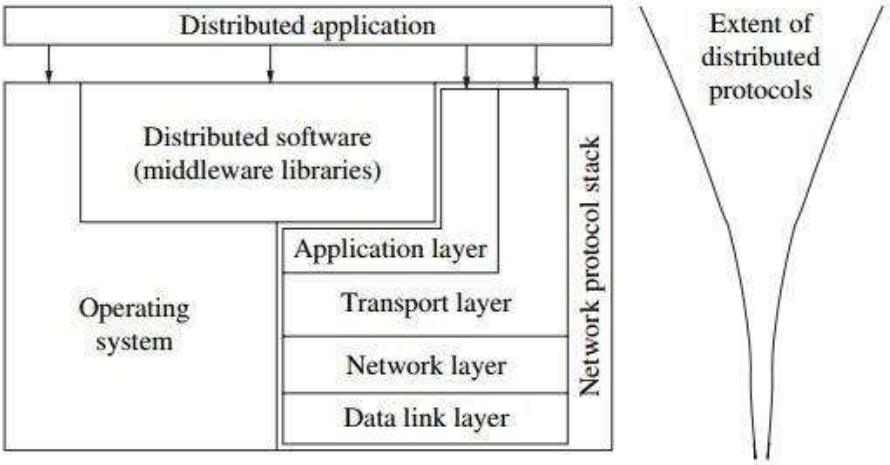


Fig 1.2: Interaction of layers of network

1.1.2 Motivation

The following are the key points that acts as a driving force behind DS:

- **Inherently distributed computations**: DS can process the computations at geographically remote locations.
- **Resource sharing:** The hardware, databases, special libraries cab be shared between systems without owning a dedicated copy or a replica. This is cost effective and reliable.
- **Access to geographically remote data and resources:** As mentioned previously, computations may happen at remote locations. Resources such as centralized servers can also be accessed from distant locations.
- **Enhanced reliability**:DS provides enhanced reliability, since they run on multiple copies of resources. The distribution of resources at distant locations makes them less susceptible for faults. The term reliability comprises of:
  1. **Availability:**the resource/ service provided by the resourceshould be accessible at all times
  2. **Integrity:** the value/state of the resource should be correct and consistent.
  3. **Fault-Tolerance:**the ability to recover from system failures
- **Increased performance/cost ratio:** The resource sharing and remote access features of DS naturally increase the performance / cost ratio.
- **Scalable:** The number of systems operating in a distributed environment can be increased as the demand increases.

1.2 RELATION TO PARALLEL SYSTEMS

Parallel Processing Systems divide the program into multiple segments and process them simultaneously.

The main objective of parallel systems is to improve the processing speed. They aresometimes known as **multiprocessor or multi computers or tightly coupled systems.** They refer to simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.

1.2.1 Types of multi-processor systems

There are three types of shared memory multiprocessors:

i) Uniform Memory Access (UMA)

ii) Non Uniform Memory Access (NUMA)

iii) Cache Only Memory Access (COMA)

i) Uniform Memory Access (UMA)

- Here, all the processors share the physical memory in a centralized manner with equal access time to all the memory words.
- Each processor may have a private cache memory. Same rule is followed for peripheral devices.
- When all the processors have equal access to all the peripheral devices, the system is called a **symmetric multiprocessor.**
- When only one or a few processors can access the peripheral devices, the system is called an **asymmetric multiprocessor.**
- When a CPU wants to access a memory location, it checks if the bus is free, then it sends the request to the memory interface module and waits for the requested data to be available on the bus.
- Multicore processors are small UMA multiprocessor systems, where the first shared cache is actually the communication channel.
- Shared memory can quickly become a bottleneck for system performances, since all processors must synchronize on the single bus and memory access.

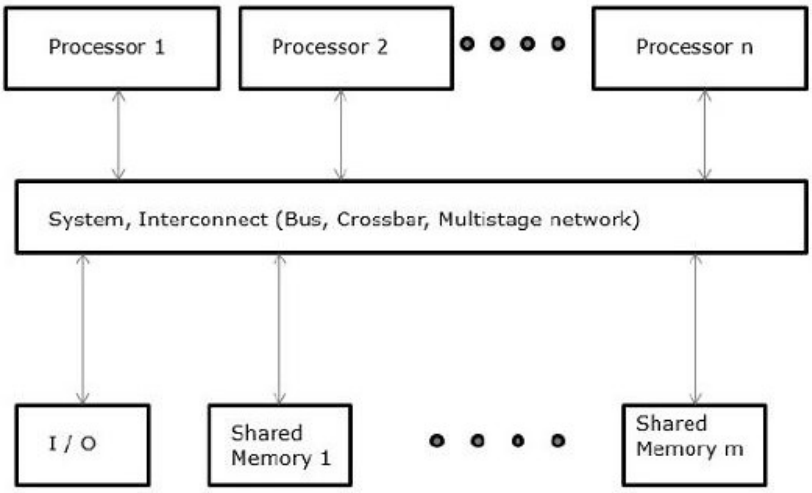


Fig 1.3: Uniform memory access model

ii) Non-uniform Memory Access (NUMA)

- In NUMA multiprocessor model, the access time varies with the location of the memory word.
- Here, the shared memory is physically distributed among all the processors, called local memories.
- The collection of all local memories forms a global address space which can be accessed by all the processors.
- NUMA systems also share CPUs and the address space, but each processor has a local memory, visible to all other processors.
- In NUMA systems access to local memory blocks is quicker than access to remote memory blocks.
- Programs written for UMA systems run with no change in NUMA ones, possibly with different performances because of slower access times to remote memory blocks.
- Single bus UMA systems are limited in the number of processors, and costly hardware is necessary to connect more processors.
- Current technology prevents building UMA systems with more than 256 processors.
- To build larger processors, a compromise is mandatory: not all memory blocks can have the same access time with respect to each CPU.

- Since all NUMA systems have a single logical address space shared by all CPUs, while physical memory is distributed among processors, there are two types of memories: local and remote memory.

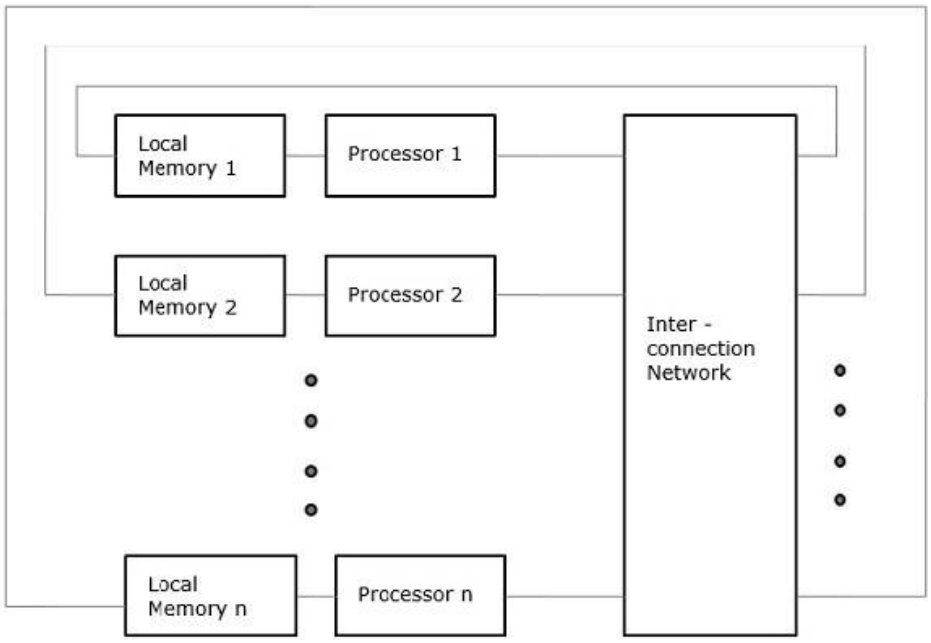


Fig 1.4:Non-uniform Memory Access model

- There are two types of NUMA systems: Non-Caching NUMA (NC-NUMA) Cache-Coherent NUMA (CC-NUMA).

Non-Caching NUMA (NC-NUMA):

- In a NC-NUMA system, processors have no local cache. Each memory access is managed with a modified MMU, which controls if the request is for a local or for a remote block; in the latter case, the request is forwarded to the node containing the requested data.
- Obviously, programs using remote data will run much slower than what they would, if the data were stored in the local memory. In NC-NUMA systems there is no cache coherency problem, because there is no caching at all: each memory item is in a single location.
- Remote memory access is however very inefficient. For this reason, NC-NUMA systems can resort to special software that relocates memory pages from one block to another, just to maximise performances.

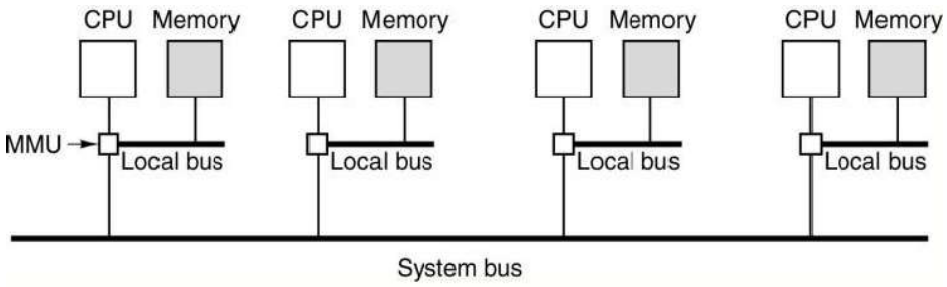


Fig 1.5:Non-Caching NUMA

- **Cache-Coherent NUMA (CC-NUMA):**
  - Caching can alleviate the problem due to remote data access, but brings the cache coherency issue.
  - A method to enforce coherency is obviously bus snooping, but this techniques gets too expensive beyond a certain number of CPUs, and it is much too difficult to implement in systems that do not rely on bus-based interconnections.
  - The common approach in CC-NUMA systems with many CPUs to enforce cache coherency is the directory-based protocol.
  - The basic idea is to associate each node in the system with a directory for its RAM blocks: a database stating in which cache is located a block, and what is its state.
  - When a block of memory is addressed, the directory in the node where the block is located is queried, to know if the block is in any cache and, if so, if it has been changed respect to the copy in RAM.
  - Since a directory is queried at each access by an instruction to the corresponding memory block, it must be implemented with very quick hardware, as an instance with an associative cache, or at least with static RAM.

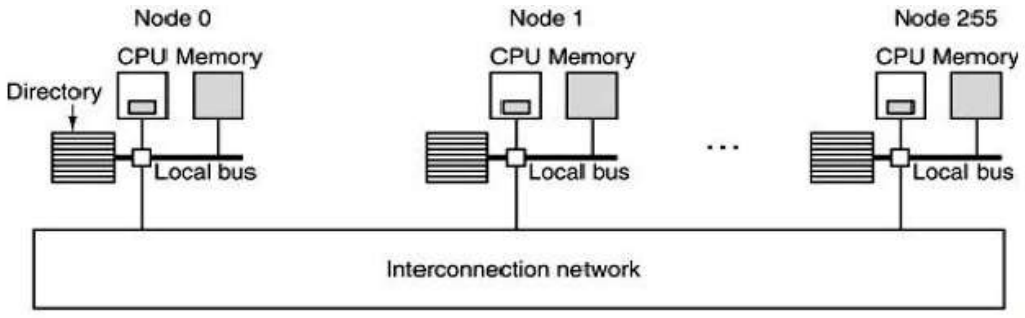


Fig 1.6:Cache-Coherent NUMA

iii) Cache Only Memory Access (COMA)

- The COMA model is a special case of the NUMA model. Here, all the distributed main memories are converted to cache memories.
- In a monoprocessor architecture and in shared memory architectures each block and each line are located in a single, precise position of the logical address space, and have therefore an address called home address.
- When a processor accesses a data item, its logical address is translated into the physical address, and the content of the memory location containing the data is copied into the cache of the processor, where it can be read and/or modified.
- In the last case, the copy in RAM will be eventually overwritten with the updated copy present in the cache of the processor that modified it.
- This property turns the relationship between processors and memory into a critical one, both in UMA and in NUMA systems:
- In NUMA systems, distributed memory can generate a high number of messages to move data from one CPU to another, and to maintain coherency in home address values. Remote memory references are much slower than local memory ones.
- In CC-NUMA systems, this effect is partially hidden by the caches.
- In UMA systems, centralized memory causes a bottleneck, and limit its the interconnection between CPU and memory, and its scalability.
- In COMA, there is no longer a home address, and the entire physical address space is considered a huge, single cache.
- Data can migrate within the whole system, from a memory bank to another, according to the request of a specific CPU, that requires that data.

1.2.2 Topologies

The important issue in the design of multiprocessor systems is how to cope with the problem of an adequate design of the interconnection network in order to achieve the desired performance at low cost. The choice of the interconnection network may affect several characteristics of the system such as node complexity, scalability and cost etc. The interconnection network form the topology to access the memory. It may be a bus or a multistage switch with a symmetric and regular design. The multi-stage networks can be constructed recursively, and the interconnection pattern between any two stages can be expressed using an iterative or a recursive generating function.

Omega network

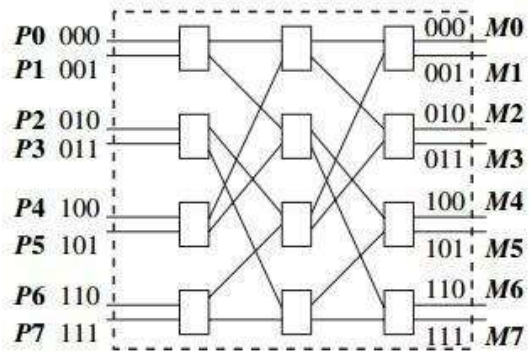


Fig 1.7: 3-stage Omega network

A multistage omega network formed by a 2x2 switch is shown in fig 1.7. The 2× 2 switch allows data on either of the two input wires. Only one data unit can be sent on an output wire at a single step. To avoid collision of data, many buffering techniques have been proposed.

The Omega network connecting n processors with n memory units has n/2logn switching elements of size 2 × 2 arranged in log n stages. Between each pair of adjacent stages of the Omega network, a link exists between output i of a stage and the input j to the next stage according to the following perfect shuffle pattern which is a left-rotation operation on the binary representation of i to get j. The generation function is given as:

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

The routing function from input line i to output line j considers only j and the stage number s, where s ∈ [0, log n – 1]. In a stage s switch, if the s + 1th 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.

Butterfly network

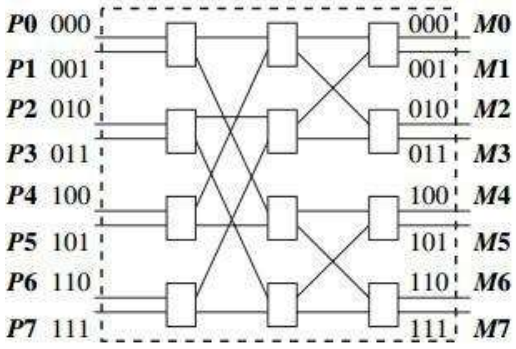


Fig 1.8: Butterfly network

A butterfly network links multiple computers into a high-speed network. For a butterfly network with n processor nodes, there need to be n (log n + 1) switching nodes. The generation of the interconnection pattern between a pair of adjacent stages depends not only on n but also on the stage numbers.In a stage (s) switch, if the s + 1th MSB of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

Torus or 2D Mesh Topology

A k X k mesh will contain k^2processor with maximum path length as 2*(k/2 - 1). Every unit in the torus topology is identified using a unique label, with dimensions distinguished as bit positions.

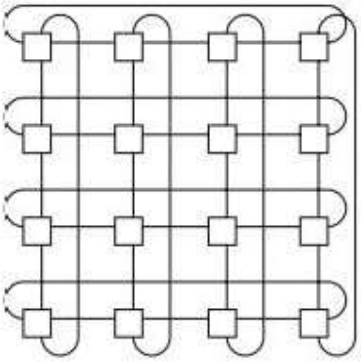


Fig 1.9: 2-D Mesh

Hypercube

The path between any two nodes in 4-D hypercube is found by Hamming distance. Routing is done in hop to hop fashion with each adjacent node differing by one bit label. This topology has good congestion control and fault tolerant mechanism.

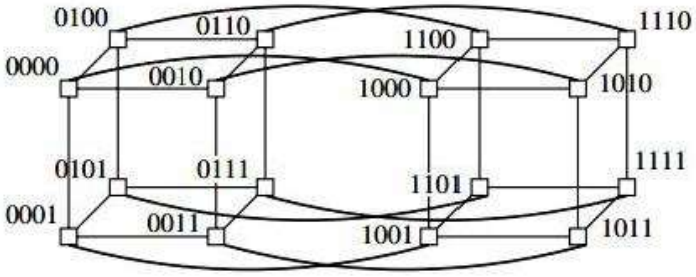


Fig 1.10: 4-D Hypercube

Array Processors

> ***They are a class of processors that executes one instruction at a time in an array or table of data at the same time rather than on single data elements on a common clock.***

They are also known as vector processors. An array processor implement the instruction set where each instruction is executed on all data items associated and then move on the other instruction. Array elements are incapable of operating autonomously, and must be driven by the control unit.

1.2.3 Flynn's Taxonomy

> ***Flynn's taxonomy is a specific classification of parallel computer architectures that are based on the number of concurrent instruction (single or multiple) and data streams (single or multiple) available in the architecture.***

Flynn's taxonomy based on the number of instruction streams and data streams are the following:

- (SISD) single instruction, single data
- (MISD) multiple instruction, single data
- (SIMD) single instruction, multiple data
- (MIMD) multiple instruction, multiple data

SISD (Single Instruction, Single Data stream)

- Single Instruction, Single Data (SISD) refers to an Instruction Set Architecture in which a single processor (one CPU) executes exactly one instruction stream at a time.
- It also fetches or stores one item of data at a time to operate on data stored in a single memory unit.
- Most of the CPU design is based on the von Neumann architecture and the follow SISD.
- The SISD model is a non-pipelined architecture with general-purpose registers, Program Counter (PC), the Instruction Register (IR), Memory Address Registers (MAR) and Memory Data Registers (MDR).

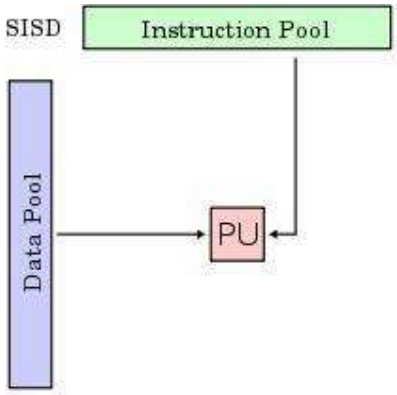


Fig 1.11: Single Instruction, Single Data Stream

SIMD (Single Instruction, Multiple Data streams)

- Single Instruction, Multiple Data (SIMD) is an Instruction Set Architecture that have a single control unit (CU) and more than one processing unit (PU) that operates like a von Neumann machine by executing a single instruction stream over PUs, handled through the CU.
- The CU generates the control signals for all of the PUs and by which executes the same operation on different data streams.
- The SIMD architecture is capable of achieving data level parallelism.

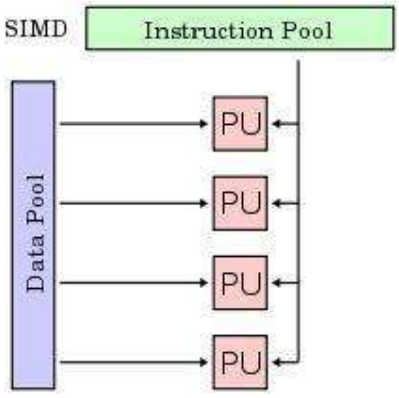


Fig 1.12:Single Instruction, Multiple Data streams

MISD (Multiple Instruction, Single Data stream)

- Multiple Instruction, Single Data (MISD) is an Instruction Set Architecture for parallel computing where many functional units perform different operations by executing different instructions on the same data set.

- This type of architecture is common mainly in the fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors.

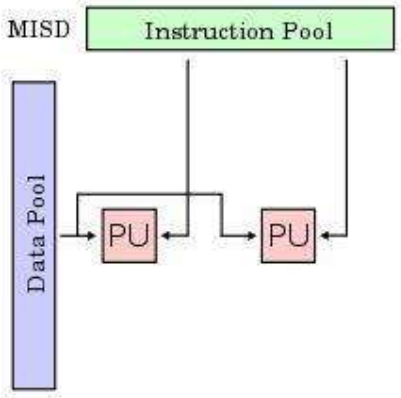


Fig 1.13:Multiple Instruction, Single Data stream

MIMD (Multiple Instruction, Multiple Data streams)

- Multiple Instruction stream, Multiple Data stream (MIMD) is an Instruction Set Architecture for parallel computing that is typical of the computers with multiprocessors.
- Using the MIMD, each processor in a multiprocessor system can execute asynchronously different set of the instructions independently on the different set of data units.
- The MIMD based computer systems can used the shared memory in a memory pool or work using distributed memory across heterogeneous network computers in a distributed environment.
- The MIMD architectures is primarily used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modelling, communication switches etc.

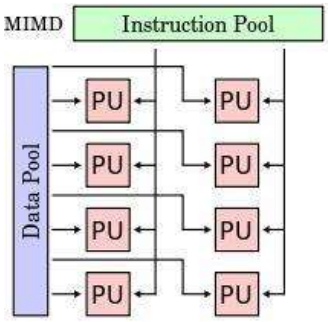


Fig 1.14:Multiple Instruction, Multiple Data streams

	Single	Multiple
Single	**SISD** Von Neumann Single computer	**MISD** May be pipelined computers
Multiple	**SIMD** Vector processors Fine grained data Parallel computers	**MIMD** Multi computers Multiprocessors

Fig 1.15: Comparison of Flynn's taxonomy

1.2.4 Coupling, parallelism, concurrency, and granularity

Coupling

- A multiprocessor system has the capability of executing Multiple Instruction, Multiple Data (MIMD) programs with multiples CPU along with memory and IO channels.
- Concurrency is a major design challenge in these systems. The term **coupling** is associated with the configuration and design of processors in a multiprocessor system.

> ***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.***

- The multiprocessor systems are classified into two types based on coupling:

  1. Loosely coupled systems

  2. Tightly coupled systems

Tightly Coupled systems:

- Tightly coupled multiprocessor systems contain multiple CPUs that are connected at the bus level with both local as well as central shared memory.
- Tightly coupled systems perform better, due to faster access to memory and intercommunication and are physically smaller and use less power. They are economically costlier.

- Tightly coupled multiprocessors with UMA shared memory may be either switch-based (e.g., NYU Ultracomputer, RP3) or bus-based (e.g., Sequent, Encore).
- Some examples of tightly coupled multiprocessors with NUMA shared memory or that communicate by message passing are the SGI Origin 2000

Loosely Coupled systems:

- Loosely coupled multiprocessors consist of distributed memory where each processor has its own memory and IO channels.
- The processors communicate with each other via message passing or interconnection switching.
- Each processor may also run a different operating system and have its own bus control logic.
- Loosely coupled systems are less costly than tightly coupled systems, but are physically bigger and have a low performance compared to tightly coupled systems.
- The individual nodes in a loosely coupled system can be easily replaced and are usually inexpensive.
- They require more power and are more robust and can resist failures.
- The extra hardware required to provide communication between the individual processors makes them complex and less portable.
- Loosely coupled multicomputers without shared memory are physically co-located. These may be bus-based (e.g., NOW connected by a LAN or Myrinet card) or using a more general communication network.
- These processors neither share memory nor have a common clock.
- Loosely coupled multicomputers without shared memory and without common clock and that are physically remote, are termed as distributed systems.

Parallelism on specific system

- It is the use of multiple processing elements simultaneously for solving any problem.
- Problems are broken down into instructions and are solved concurrently as each resource which has been applied to work is working at the same time.
- This is a measure of the relative speedup of a specific program, on a givenmachine. The speedup depends on the number of processors and the mapping.
- It is expressed as the ratio of the time T(1) witha 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 processors are executing CPU instructions productively, as opposed to waiting forcommunication operations.

Concurrency

- Concurrent programming refer to techniques for decomposing a task into subtasks that can execute in parallel and managing the risks that arise when the program executes more than one task at the same time.
- The parallelism or concurrency in a parallel or 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

> ***Granularity or grain size is a measure of the amount of work or computation that is performed by that task.***

- Granularity is also the communication overhead between multiple processors or processing elements.
- In this case, granularity as the ratio of computation time to communication time, wherein, the computation time is the time required to perform the computation of a task and communication time is the time required to exchange data between processors.
- Parallelism can be classified into three categories based on work distribution among the parallel tasks:
    1. **Fine-grained:** Partitioning the application into small amounts of work done leading to a low computation to communication ratio.
    2. **Coarse-grained parallelism:**This has high computation to communication ratio.
    3. **Medium-grained:** Here the task size and communication time greater than fine-grained parallelism and lower than coarse-grained parallelism.
- Programs with fine-grained parallelism are best suited for tightly coupled systems.

Classes of OS of Multiprocessing systems:

- **Network Operating Systems:** The operating system running on loosely coupled processors which are themselves running loosely coupled software

- **Distributed Operating systems:** The OS of the system running on loosely coupled processors, which arerunning tightly coupled software.
- **Multiprocessor Operating Systems:** The OS will run on tightly coupled processors, which arethemselves running tightly coupled software.

1.3 MESSAGE-PASSING SYSTEMS VERSUS SHARED MEMORY SYSTEMS

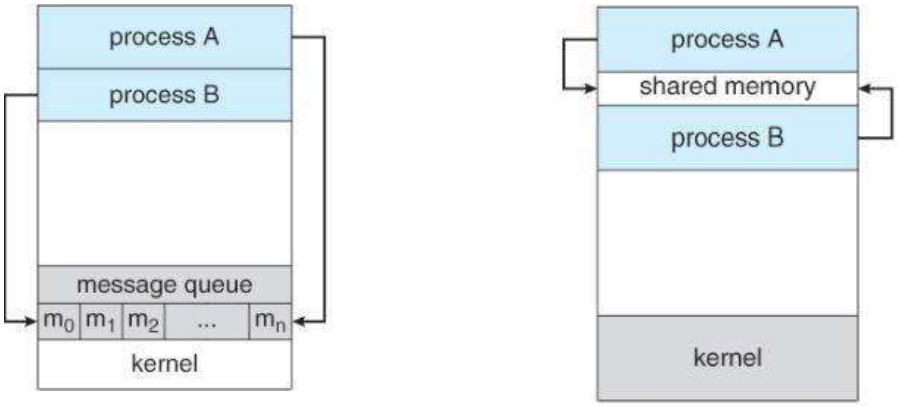
The communications between the tasks in multiprocessor systems take place through two main modes:

Message passing systems:

- This allows multiple processes to read and write data to the message queue without being connected to each other.
- Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

Shared memory systems:

- The shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other.
- Communication among processors takes place through shared data variables, and control variables for synchronization among the processors.
- Semaphores and monitors are common synchronization mechanisms on shared memory systems.
- When shared memory model is implemented in a distributed environment, it is termed as **distributed shared memory.**



a) Message Passing Model **b) Shared Memory Model**

Fig 1.16: Inter-process communication models

Differences between message passing and shared memory models

Message Passing	Distributed Shared Memory
Services Offered: Variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process.	The processes share variables directly, so no marshalling and unmarshalling. Shared variables can be named, stored and accessed in DSM.
Processes can communicate with other processes. They can be protected from one another by having private address spaces.	Here, a process does not have private address space. So one process can alter the execution of other.
This technique can be used in heterogeneous computers.	This cannot be used to heterogeneous computers.
Synchronization between processes is through message passing primitives.	Synchronization is through locks and semaphores.
Processes communicating via message passing must execute at the same time.	Processes communicating through DSM may execute with non-overlapping lifetimes.
Efficiency: All remote data accesses are explicit and therefore the programmer is always aware of whether a particular operation is in-process or involves the expense of communication.	Any particular read or update may or may not involve communication by the underlying runtime support.

1.3.1 Emulating message-passing on a shared memory system (MP → SM)

The shared memory system can be made to act as message passing system. The shared address space can be partitioned into disjoint parts, one part being assigned to each processor.

Send and receive operations care implemented by writing to and reading from the destination/sender processor's address space. The read and write operations are synchronized.

Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes.

1.3.2 Emulating shared memory on a message-passing system (SM → MP)

This is also implemented through read and write operations. Each shared location can be modeled as a separate process. Write to a shared location is emulated by sending an update message tothe corresponding owner process and read operation to a shared location is emulatedby sending a query message to the owner process.

This emulation is expensive as the processes has to gain access to other process memory location. The latencies involvedin read and write operations may be high even when using shared memoryemulation because the read and write operations are implemented by using network-wide communication.

1.4 PRIMITIVES FOR DISTRIBUTED COMMUNICATION

1.4.1 Blocking / Non blocking / Synchronous / Asynchronous

Message send and message receive communication primitives are done through Send() and Receive(), respectively. A Send primitive has two parameters: the destination, and the buffer in the user space that holds the data to be sent. The Receive primitive also has two parameters: the source from which the data is to be received and the user buffer into which the data is to be received.

There are two ways of sending data when the Send primitive is called:

- **Buffered:** 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. 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.
- **Unbuffered**: The data gets copied directly from the user buffer onto thenetwork.

Blocking primitives

- The primitive commands wait for the message to be delivered. The execution of the processes is blocked.
- The sending process must wait after a senduntil an acknowledgement is made by the receiver.
- The receiving process must wait for the expected message from the sending process
- The receipt is determined by polling common buffer orinterrupt
- This is a form of synchronization or synchronous communication.
- A primitive is blocking if control returns to the invoking process after the processing for the primitive completes.

Non Blocking primitives

- If send is nonblocking, it returns control to the caller immediately, before the message is sent.
- The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- This is a form of asynchronous communication.
- A primitive is non-blocking if control returnsback to the invoking process immediately after invocation, even thoughthe operation has not completed.
- For a non-blocking Send, control returnsto the process even before the data is copied out of the user buffer.
- For anon-blocking Receive, control returns to the process even before the datamay have arrived from the sender.

Synchronous

- A Send or a Receive primitive is synchronous if both the Send() and Receive() handshake with each other.
- The processing for the Send primitive completes only after the invoking processor learns that the other corresponding Receive primitive has also been invoked andthat the receive operation has been completed.
- The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.

Asynchronous

- 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.
- It does not make sense to define asynchronous Receiveprimitives.
- Implementing non -blocking operations are tricky.
- For non-blocking primitives, a return parameter on the primitive call returns a system-generated **handle** which can be later used to check the status of completion of the call. The process can check for the completion:

➢ checking if the handle has been flagged or posted

➢ issue a Wait with a list of handles as parameters: usually blocks until one of the parameter handles is posted.

The send and receive primitives can be implemented in four modes:

- Blocking synchronous
- Non- blocking synchronous
- Blocking asynchronous
- Non- blocking asynchronous

Four modes of send operation

Blocking synchronous Send:

- The data gets copied from the user buffer to the kernel buffer and is then sent over the network.
- After the data is copied to the receiver's system buffer and a Receive call has been issued, an acknowledgement back to the sender causes control to return to the process that invoked the Send operation and completes the Send.

Non-blocking synchronous Send:

- Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated.
- A parameter in the non-blocking call also gets set with the handle of a location that the user process can later check for the completion of the synchronous send operation.
- The location gets posted after an acknowledgement returns from the receiver.
- The user process can keep checking for the completion of the non-blocking synchronous Send by testing the returned handle, or it can invoke the blocking Wait operation on the returned handle

Blocking asynchronous Send:

- The user process that invokes the Send is blocked until the data is copied from the user's buffer to the kernel buffer.

Non-blocking asynchronous Send:

- The user process that invokes the Send is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated.
- Control returns to the user process as soon as this transfer is initiated, and a parameter in the non-blocking call also gets set with the handle of a location that the user process can check later using the Wait operation for the completion of the asynchronous Send.

- The asynchronous Send completes when the data has been copied out of the user's buffer. The checking for the completion may be necessary if the user wants to reuse the buffer from which the data was sent.

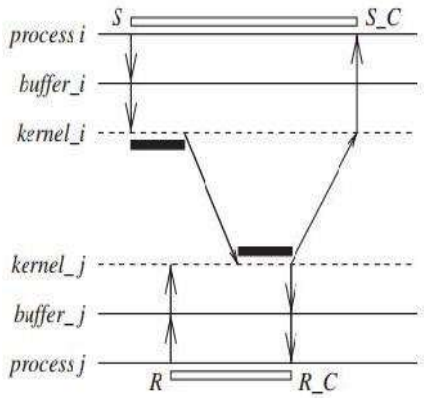
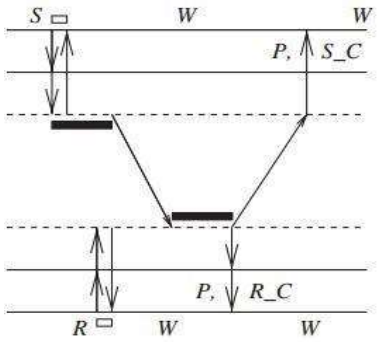


Fig 1.17 a) Blocking synchronous send and blocking receive



Fug 1.17 b) Non-blocking synchronous send and blocking receive

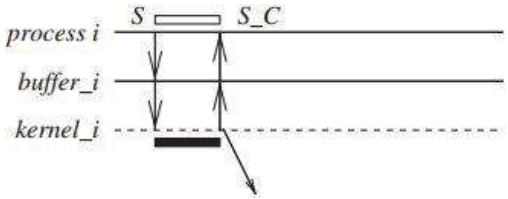


Fig 1.17 c) Blocking asynchronous send

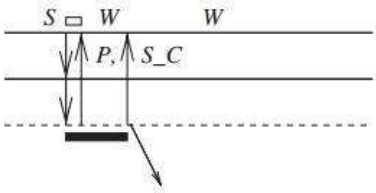


Fig 1.17 d) Non-blocking asynchronous send

Modes of receive operation

Blocking Receive:

- The Receive call blocks until the data expected arrives and is written in the specified user buffer. Then control is returned to the user process.

Non-blocking Receive:

- The Receive call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking Receive operation.
- This location gets posted by the kernel after the expected data arrives and is copied to the user-specified buffer. The user process can check for the completion of the non-blocking Receive by invoking the Wait operation on the returned handle.

1.4.2 Processor Synchrony

> ***Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.***

Since distributed systems do not follow a common clock, this abstraction is implemented using some form of barrier synchronization 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.

1.4.3 Libraries and standards

- **Message Passing Interface (MPI):** This is a standardized and portable message-passing system to function on a wide variety of parallel computers. MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

  The primary goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.

- **Parallel Virtual Machine (PVM):** It is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor.

- **Remote Procedure Call (RPC):** The Remote Procedure Call (RPC) is a common model of request reply protocol. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. RPC makes the client/server model of computing more powerful and easier to program.

- **Remote Method Invocation (RMI):** RMI (Remote Method Invocation) is a way that a programmer can write object-oriented programming in which objects on different computers can interact in a distributed network. It is a set of protocols being developed by Sun's JavaSoft division that enables Java objects to communicate remotely with other Java objects.

- **Remote Procedure Call (RPC):** It is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. RPC is a powerful technique for constructing distributed, client-server based applications. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. RPC makes the client/server model of computing more powerful and easier to program.

Differences between RMI and RPC

RMI	RPC
RMI uses an object oriented paradigm where the user needs to know the object and the method of the object he needs to invoke.	RPC is not object oriented and does not deal with objects. Rather, it calls specific subroutines that are already established
With RPC looks like a local call. RPC handles the complexities involved with passing the call from the local to the remote computer.	RMI handles the complexities of passing along the invocation from the local to the remote computer. But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called.

The commonalities between RMI and RPC are as follows:

- ✓ They both support programming with interfaces.
- ✓ They are constructed on top of request-reply protocols.
- ✓ They both offer a similar level of transparency.

- **Common Object Request Broker Architecture (CORBA):** CORBA describes a messaging mechanism by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects. The data representation is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages.

1.5 SYNCHRONOUS VS ASYNCHRONOUS EXECUTIONS

The execution of process in distributed systems may be synchronous or asynchronous.

Asynchronous Execution:

A communication among processes is considered asynchronous, when every communicating process can have a different observation of the order of the messages being exchanged. In an asynchronous execution:

- there is no processor synchrony and there is no bound on the drift rate of processor clocks
- message delays are finite but unbounded
- no upper bound on the time taken by a process

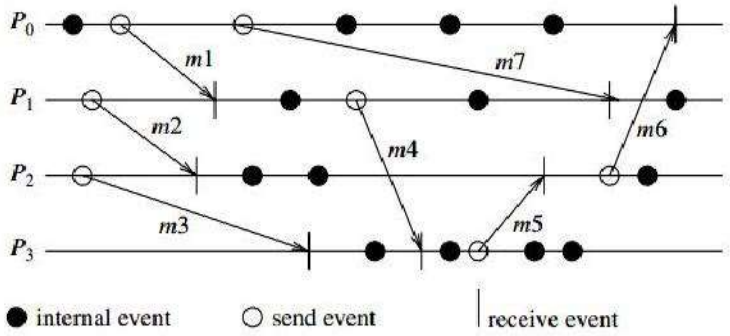


Fig 1.18: Asynchronous execution in message passing system

Synchronous Execution:

A communication among processes is considered synchronous when every process observes the same order of messages within the system. In the same manner, the execution is considered synchronous, when every individual process in the system observes the same total order of all the processes which happen within it. In an synchronous execution:

- processors are synchronized and the clock drift rate between any two processors is bounded
- message delivery times are such that theyoccur in one logical step or round
- upper boundon the time taken by a process to execute a step.

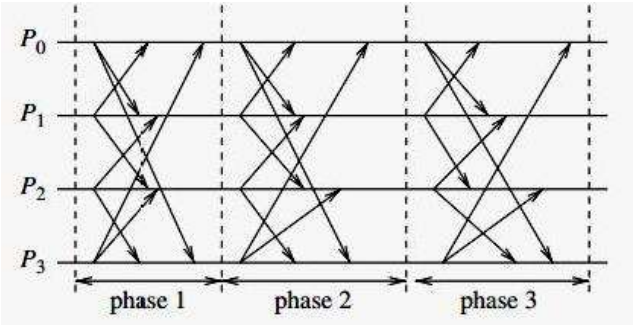


Fig 1.19: Synchronous execution

Emulating an asynchronous system by a synchronous system (A → S)

An asynchronous program can be emulated on a synchronous system fairly trivially as the synchronous system is a special case of an asynchronous system – all communication finishes within the same round in which it is initiated.

Emulating a synchronous system by an asynchronous system (S → A)

A synchronous program can be emulated on an asynchronous system using a tool called synchronizer.

Emulation for a fault free system

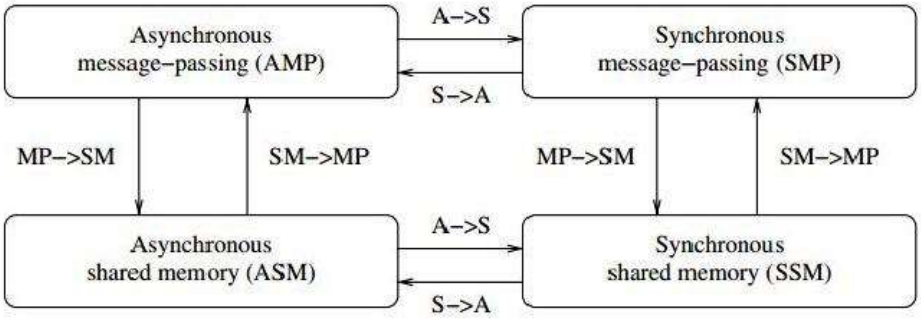


Fig 1.20: Emulations in a failure free message passing system

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. 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 in failure-free systems.

1.6 DESIGN ISSUES AND CHALLENGES IN DISTRIBUTED SYSTEMS

The design of distributed systems has numerous challenges. They can be categorized into:

- Issues related to system and operating systems design
- Issues related to algorithm design
- Issues arising due to emerging technologies

The above three classes are not mutually exclusive.

1.6.1 Issues related to system and operating systems design

The following are some of the common challenges to be addressed in designing a distributed system from system perspective:

- **Communication:** This task involves designing suitable communication mechanisms among the various processes in the networks.

  Examples: RPC, RMI

- **Processes:**The main challenges involved are:process and thread management at both client and server environments, migration of code between systems, design of software and mobile agents.

- **Naming:** Devising easy to use and robust schemes for names, identifiers,and addresses is essential for locating resources and processes in a transparent and scalable manner. The remote and highly varied geographical locations make this task difficult.

- **Synchronization:** Mutual exclusion, leader election, deploying physical clocks, global state recording are some synchronization mechanisms.
- **Data storage and access Schemes:** Designing file systems for easy and efficient data storage with implicit accessing mechanism is very much essential for distributed operation
- **Consistency and replication:** The notion of Distributed systems goes hand in hand with replication of data, to provide high degree of scalability. The replicas should be handed with care since data consistency is prime issue.
- **Fault tolerance:** This requires maintenance of fail proof links, nodes, and processes. Some of the common fault tolerant techniques are resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization.
- **Security:** Cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management are some of the security measure that is imposed on distributed systems.
- **Applications Programming Interface (API) and transparency:** The user friendliness and ease of use is very important to make the distributed services to be used by wide community. Transparency, which is hiding inner implementation policy from users, is of the following types:
  - **Access transparency:** hides differences in data representation
  - **Location transparency:** hides differences in locations y providing uniform access to data located at remote locations.
  - **Migration transparency:** allows relocating resources without changing names.
  - **Replication transparency:** Makes the user unaware whether he is working on original or replicated data.
  - **Concurrency transparency:** Masks the concurrent use of shared resources for the user.
  - **Failure transparency**: system being reliable and fault-tolerant.
- **Scalability and modularity:** The algorithms, data and servicesmust be as distributed as possible. Various techniques such as replication,caching and cache management, and asynchronous processing help toachieve scalability.

1.6.2 Algorithmic challenges in distributed computing

- **Designing useful execution models and frameworks**
  - The interleaving model, partial order model, input/output automata model and the Temporal Logic of Actions (TLA) are some examples of models that provide different degrees of infrastructure.

- **Dynamic distributed graph algorithms and distributed routingalgorithms**
  - The distributed system is generally modeled as a distributed graph.
  - Hence graphal gorithms are the base for large number of higher level communication, data dissemination, object location, and object search functions.
  - These algorithms must have the capacity to deal with highly dynamic graph characteristics. They are expected to function like routing algorithms.
  - The performance of these algorithms has direct impact on user-perceived latency, data traffic and load in the network.
- **Time and global state in a distributed system**
  - The geographically remote resources demands the synchronization based on logical time.
  - Logical time is relative and eliminates the overheads of providing physical time for applications.Logical time can

    (i) capture logic and inter-process dependencies

    (ii) track the relative progress at each process
  - Maintaining the global state of the system across space involves the role of time dimension for consistency. This can be done with extra effort in a coordinated manner.
  - Deriving appropriate measures of concurrency also involves the timedimension, as the execution and communication speed of threads may vary a lot.
- **Synchronization/coordination mechanisms**
  - Synchronization is essential for the distributed processes to facilitate concurrent execution without affecting other processes.
  - The synchronization mechanisms also involveresource management and concurrency managementmechanisms.
  - Some techniques for providing synchronization are:
  - **Physical clock synchronization:** Physical clocks usually diverge in theirvalues due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.
  - **Leader election:** All the processes need to agree on which process willplay the role of a distinguished process or a leader process. A leader is necessary even for many distributed algorithms because there is often some asymmetry.

- **Mutual exclusion:** Access to the critical resource(s) has to be coordinated.
- **Deadlock detection and resolution:** This is done to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.
- **Termination detection**: cooperation among the processesto detect the specific global state of quiescence.
- **Garbage collection:** Detecting garbage requirescoordination among the processes.

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

- A group is a collection of processes that share a common context and collaborate on a common task within an application domain. Group management protocols are needed for group communication wherein processes can join and leave groups dynamically, or fail.
- The concurrent execution of remote processes may sometimes violate the semantics and order of the distributed program. Hence, a formal specification of the semantics ofordered delivery need to be formulated, and then implemented.

➢ **Monitoring distributed events and predicates**

- Predicates defined on program variables that are local to different processesare used for specifying conditions on the global system state.
- On-line algorithms for monitoring such predicates are hence important.
- An important paradigm for monitoring distributed events is thatof event streaming, wherein streams of relevant events reported from different processes are examined collectively to detect predicates.
- The specification of such predicates uses physical or logical time relationships.

➢ **Distributed program design and verification tools**

- Methodically designed and verifiably correct programs can greatly reduce theoverhead of software design, debugging, and engineering.Designing these is a big challenge.

➢ **Debugging distributed programs**

- Debugging distributed programs is much harder because of the concurrency and replications. Adequate debugging mechanisms and tools are need of the hour.

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

- Fast access to data and other resources is important in distributed systems.

- Managing replicas and their updates faces concurrency problems.
- Placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.

➢ **World Wide Web design – caching, searching, scheduling**

- WWWis a commonly known distributed system.
- The issues of object replication and caching, prefetching of objects have to be done on WWW also.
- Object search and navigationon the web are important functions in the operation of the web.

➢ **Distributed shared memory abstraction**

- A shared memory is easier to implement since it does not involve managing the communication tasks.
- The communication is done by the middleware by message passing.
- The overhead of shared memory is to be dealt by the middleware technology.
- Some of the methodologies that does the task of communication in shared memory distributed systems are:

▪ **Wait-free algorithms:** The ability of a process to complete its execution irrespective of the actions of other processes is wait free algorithm. They control the access to shared resources in the shared memory abstraction. They are expensive.

▪ **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 at any given time. In a distributed system, shared variables or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion.

▪ **Register constructions:**Architectures must be designed in such a way that, registers allows concurrent access without any restrictions on the concurrency permitted.

➢ **Reliable and fault-tolerant distributed systems**

The following are some of the fault tolerant strategies:

▪ **Consensus algorithms:** Consensus algorithms allow correctly functioning processes to reach agreement among themselves in spite of the existence of malicious processes. The goal of the malicious processesis to prevent the correctly

functioning processes from reaching agreement.The malicious processes operate by sending messages with misleading information, to confuse the correctly functioning processes.

- **Replication and replica management:** The Triple Modular Redundancy (TMR) technique is used in software and hardware implementation. TMR is a fault-tolerant form of N-modular redundancy, in which three systems perform a process and that result is processed by a majority-voting system to produce a single output.

- **Voting and quorum systems**: Providing redundancy in the active or passive components in the systemand then performing voting based on some quorum criterion is a classicalway of dealing with fault-tolerance. Designing efficient algorithms for thispurpose is the challenge.

- **Distributed databases and distributed commit:** The distributed databases should also follow atomicity, consistency, isolation and durability (ACID) properties.

- **Self-stabilizing systems:** All system executions have associated good (or legal) states and bad (or illegal) states; during correct functioning, the system makes transitions among the good states. A self-stabilizing algorithm guarantee to take the system to a good state even if a bad state were toarise due to some error. Self-stabilizing algorithms require some in-builtredundancy to track additional variables of the state and do extra work.

- **Checkpointing and recovery algorithms:** Checkpointing is periodically recording the current state on secondary storage so that, in case of a failure. The entire computation is not lost but can be recovered from oneof the recently taken checkpoints. Checkpointing in a distributed environment is difficult because if the checkpoints at the different processes arenot coordinated, the local checkpoints may become useless because theyare inconsistent with the checkpoints at other processes.

- **Failure detectors:** The asynchronous distributed do not have a bound on the message transmission time. This makes the message passing very difficult, since the receiver do not know the waiting time. Failure detectors probabilistically suspect another process as havingfailed and then converge on a determination of the up/down status of thesuspected process.

➢ **Load balancing**

The objective of load balancing is to gain higher throughput, and reduce the userperceived latency. Load balancing may be necessary because of a variety of factors such as high network traffic or high request rate causing the networkconnection to be a bottleneck, or high computational load. The following are some forms of load balancing:

- **Data migration:** The ability to move data around in the system, based on the access pattern of the users

- **Computation migration**: The ability to relocate processes in order to perform a redistribution of the workload.

- **Distributed scheduling:** This achieves a better turnaround time for theusers by using idle processing power in the system more efficiently.

➢ **Real-time scheduling**

Real-time scheduling becomes more challenging when a global view of the system state is absent with more frequent on-line or dynamic changes. The message propagation delays which are network-dependent are hard to control or predict. This is an hindrance to meet the QoS requirements of the network.

➢ **Performance**

User perceived latency in distributed systems must be reduced. The common issues in performance:

- **Metrics:** Appropriate metrics must be defined for measuring the performance of theoretical distributed algorithms and its implementation.

- **Measurement methods/tools:** The distributed system is a complexentity appropriate methodology and tools must be developed for measuring the performance metrics.

1.6.3 Applications of distributed computing and newer challenges

The deployment environment of distributed systems ranges from mobile systems to cloud storage. All the environments have their own challenges:

➢ **Mobile systems**

- Mobile systems which use wireless communication in shared broadcast mediumhave issues related to physical layer such as transmission range, power, battery power consumption, interfacing with wired internet, signal processing and interference.

- The issues pertaining to other higher layers includerouting, location management, channel allocation, localization and position estimation, and mobility management.

- Apart from the above mentioned common challenges, the architectural differences of the mobile network demands varied treatment. The two architectures are:

    - **Base-station approach (cellular approach):** The geographical region is divided into hexagonal physical locations called cells. The powerful base station transmits signals to all other nodes in itsrange

- **Ad-hoc network approach:**This is an infrastructure-less approach which do not have any base station to transmit signals. Instead all the responsibility is distributed among the mobile nodes.

- It is evident that both the approaches work in different environment with different principles of communication. Designing a distributed system to cater the varied need is a great challenge.

➢ **Sensor networks**

- A sensor is a processor with an electro-mechanical interface that is capable ofsensing physical parameters.

- They are low cost equipment with limited computational power and battery life. They are designed to handle streaming data and route it to external computer network and processes.

- They are susceptible to faults and have to reconfigure themselves.

- These features introduces a whole new set of challenges, such as position estimation and time estimation when designing a distributed system.

➢ **Ubiquitous or pervasive computing**

- In Ubiquitous systems the processors are embedded in the environment to perform application functions in the background.

- **Examples:** Intelligent devices, smart homes etc.

- They are distributed systems with recent advancements operating in wireless environments through actuator mechanisms.

- They can be self-organizing and network-centric with limited resources.

➢ **Peer-to-peer computing**

- Peer-to-peer (P2P) computing is computing over an application layernetwork where all interactions among the processors are at a same level.

- This is a form of symmetric computation against the client sever paradigm.

- They are self-organizing with or without regular structure to the network.

- Some of the key challenges include: object storage mechanisms, efficient object lookup, and retrieval in a scalable manner; dynamic reconfiguration with nodes as well as objects joining and leaving the network randomly; replication strategies to expedite object search; tradeoffs between object size latency and table sizes; anonymity, privacy, and security.

- **Publish-subscribe, content distribution, and multimedia**
  - The users in present day require only the information of interest.
  - In a dynamic environment where the information constantly fluctuates there is great demand for
    (i) **Publish:**an efficient mechanism for distributing this information
    (ii) **Subscribe:** an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information
    (iii) An efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter.
  - Content distribution refers to a mechanism that categorizes the information based on parameters.
  - The publish subscribe and content distribution overlap each other.
  - Multimedia data introduces special issue because of its large size.
- **Distributed agents**
  - Agents are software processes or sometimes robots that move around the systemto do specific tasks for which they are programmed.
  - Agents collect and process information and can exchange such information with other agents.
  - Challenges in distributed agent systems include coordination mechanisms among the agents, controlling the mobility of the agents,their software design and interfaces.
- **Distributed data mining**
  - Data mining algorithms process large amount of data to detect patterns and trends in the data, to mine or extract useful information.
  - The mining can be done by applying database and artificial intelligence techniques to a data repository.
- **Grid computing**
  - Grid computing is deployed to manage resources. For instance, idle CPU cycles of machines connected to the network will be available to others.
  - The challenges includes: scheduling jobs, framework for implementing quality of service, real-time guarantees, security.

- **Security in distributed systems**

  - The challenges of security in a distributed setting include: confidentiality, authentication and availability. This can be addressed using efficient and scalable solutions.

1.7 A MODEL OF DISTRIBUTED COMPUTATIONS: DISTRIBUTED PROGRAM

- A distributed program is composed of a set of asynchronous processesthat communicate by message passing over the communication network. Each process may run on different processor.

- The processes do not share a global memory and communicate solely by passing messages. These processes do not share a global clock that is instantaneously accessible tothese processes.

- Process execution and message transfer are asynchronous – a process may execute an action spontaneously and a process sending a message does not wait for the delivery of the message to be complete.

- The global state of a distributed computation is composed of the states of the processes and the communication channels. The state of a process ischaracterized by the state of its local memory and depends upon the context.

- The state of a channel is characterized by the set of messages in transit in the channel.

1.8 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: internal events, message send events, and message receive events.

- The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state.

- 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.

- The execution of process pi produces a sequence of events e1, e2, e3, …, and it is denoted by H_i: $H_i = (h_i \rightarrow_i)$. Here $h_i$are states produced by p_i and $\rightarrow$are the casual dependencies among events p_i.

- $\rightarrow_{msg}$indicates the dependency that exists due to message passing between two events.

$$send(m) \rightarrow_{msg} rec(m)$$

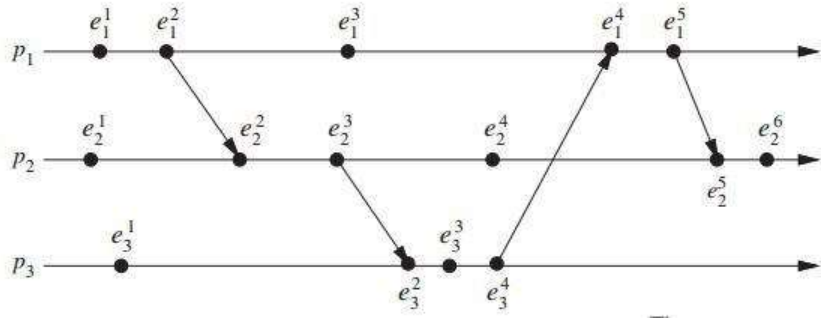


Fig 1.21: Space time distribution of distributed systems

- 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 messageand the state of the channel on which the message is received.

1.8.1 Casual Precedence Relations

Causal message ordering is a partial ordering of messages in a distributed computing environment. It is the delivery of messages to a process in the order in which they were transmitted to that process.

> ***It places a restriction on communication between processes by requiring that if the transmission of message m_i to process p_k necessarily preceded the transmission of message m_j to the same process, then the delivery of these messages to that process must be ordered such that m_i is delivered before m_j.***

Happen Before Relation

The partial ordering obtained by generalizing the relationship between two process is called as happened-before relation or causal ordering or potential causal ordering. This term was coined by Lamport. Happens-before defines a partial order of events in a distributed system. Some events can't be placed in the order. If say A $\rightarrow$B if A happens before B. A$\rightarrow$B is defined using the following rules:

- **Local ordering:**A and B occur on same process and A occurs before B.
- **Messages:** send(m) $\rightarrow$ receive(m) for any message m

- **Transitivity:** e → e'' if e → e' and e' → e''

- Ordering can be based on two situations:

  1. If two events occur in same process then they occurred in the order observed.

  2. During message passing, the event of sending message occurred before the event of receiving it.

  Lamports ordering is happen before relation denoted by →

- a→b, if a and b are events in the same process and a occurred before b.

- a→b, if a is the vent of sending a message m in a process and b is the event of the same message m being received by another process.

- If a→b and b→c, then a→c. Lamports law follow transitivity property.

  When all the above conditions are satisfied, then it can be concluded that a→b is casually related. Consider two events c and d; c→d and d→c is false (i.e) they are not casually related, then c and d are said to be concurrent events denoted as c||d.

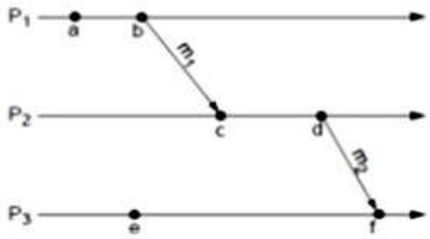


Fig 1.22: Communication between processes

Fig 1.22 shows the communication of messages m1 and m2 between three processes p1, p2 and p3. a, b, c, d, e and f are events. It can be inferred from the diagram that, a→b; c→d; e→f; b->c; d→f; a→d; a→f; b→d; b→f. Also a||e and c||e.

1.8.2 Logical vs physical concurrency

Physical as well as logical concurrency is two events that creates confusion in distributed systems.

Physical concurrency: Several program units from the same program that execute simultaneously.

Logical concurrency: Multiple processors providing actual concurrency. The actual execution of programs is taking place in interleaved fashion on a single processor.

Differences between logical and physical concurrency

Logical concurrency	Physical concurrency
Several units of the same program execute simultaneously on same processor, giving an illusion to the programmer that they are executing on multiple processors.	Several program units of the same program execute at the same time on different processors.
They are implemented through interleaving.	They are implemented as uni-processor with I/O channels, multiple CPUs, network of uni or multi CPU machines.

1.9 MODELS OF COMMUNICATION NETWORK

The three main types of communication models in distributed systems are:

- **FIFO (first-in, first-out):** each channel acts as a FIFO message queue.
- **Non-FIFO (N-FIFO):** a channel acts like a set in which a sender process adds messages and receiver removes messages in random order.
- **Causal Ordering (CO):** It follows Lamport's law.

The relation between the three models is given by CO $\subset$ FIFO $\subset$ N-FIFO.

1.20 GLOBAL STATE

Distributed Snapshot represents a state in which the distributed system might have been in. A snapshot of the system is a single configuration of the system.

A distributed snapshot should reflect a consistent state. A global state is consistent if it could have been observed by an external observer. For a successful Global State, all states must be consistent:

- If we have recorded that a process P has received a message from a process Q, then we should have also recorded that process Q had actually send that message.
- Otherwise, a snapshot will contain the recording of messages that have been received but never sent.
- The reverse condition (Q has sent a message that P has not received) is allowed.

The notion of a global state can be graphically represented by what is called a **cut**. A cut represents the last event that has been recorded for each process.

The history of each process if given by:

$$\text{history}(p_i) = \langle e_i^0, e_i^1, e_i^2 \ldots \rangle$$

Each event either is an internal action of the process. We denote by s_i^k the state of process p_i immediately before the k^{th} event occurs. The state s_i in the global state S corresponding to the cut C is that of p_i immediately after the last event processed by p_i in the cut – ei^{ci} . The set of events ei^{ci} is called the frontier of the cut.

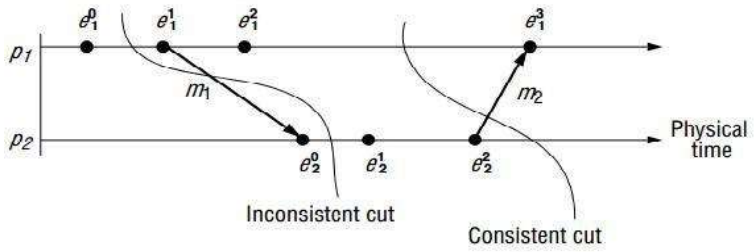


Fig 1.23: Types of cuts

Consistent states: The states should not violate causality. Such states are called consistent global states and are meaningful global states.

Inconsistent global states: They are not meaningful in the sense that a distributed system can never be in an inconsistent state.

1.10 CUTS OF A DISTRIBUTED COMPUTATION

A cut is a set of cut events, one per node, each of which captures the state of the node on which it occurs.

Cut is pictorially a line slices the space–time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE. The PAST contains allthe 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.

Consistent cut: 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.

Inconsistent cut: A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST.

1.11 PAST AND FUTURE CONES OF AN EVENT

In a distributed computation, an event $e_j$could have been affected only by all events e_i, such that $e_i \rightarrow e_j$ and all the information available at $e_i$could be made accessible at e_j. In other

word e_i and e_j should have a causal relationship. Let Past(e_j) denote all events in the past of e_j in any computation.

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

The term max(past(e_i)) denotes the latest event of process p_i that has affected e_j. This will always be a message sent event.

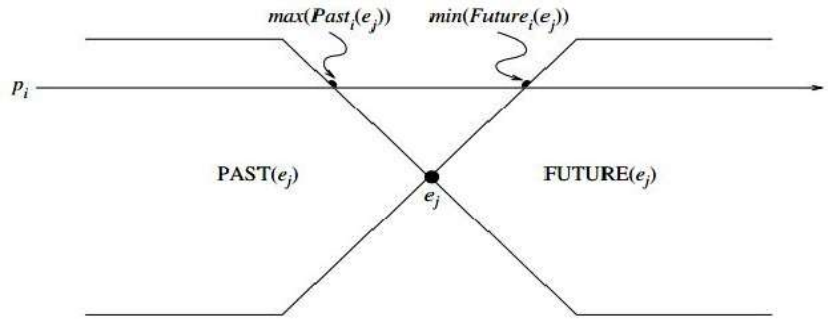


Fig 1.24: Past and future cones of event

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.

The future of an event e_j denoted by Future(e_j) contains all the events e_i that are casually affected by e_j.

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

Future$_i$(e_i) is the set of those events of Future (e_j) are the process p_i and min(Future$_i$(e_j)) as the first event on process p_i that is affected by e_j. All events at a process p_i that occurred after Max(Past(e_j)) but before min(Future$_i$(e_j)) are concurrent with e_j.

1.12 MODELS OF PROCESS COMMUNICATIONS

There are two basic models of process communications

- **Synchronous:** the sender process blocks until the message has been received by the receiver process. The sender process resumes executiononly after it learns that the receiver process has accepted the message. The sender and the receiver processes must synchronize to exchange a message.
- **Asynchronous**: It is non- blocking communication where the sender and the receiver do not synchronize to exchange 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. A buffer overflow may occur if a process sends a large number of messages in a burst to another process, thus causing a message burst.

Asynchronous communication achieves high degree of parallelism and non-determinism at the cost of implementation complexity with buffers. On the other hand, synchronization is simpler with low performance. The occurrence of deadlocks and frequent blocking of events prevents it from reaching higher performance levels.

1.13 LOGICAL TIME

Logical clocks are based on capturing chronological and causal relationships of processes and ordering events based on these relationships.

Precise physical clocking is not possible in distributed systems. The asynchronous distributed systems spans logical clock for coordinating the events.Three types of logical clock are maintained in distributed systems:

- Scalar clock
- Vector clock
- Matrix clock

Differences between physical and logical clock

Physical Clock	**Logical Clock**
A physical clock is a physical procedure combined with a strategy for measuring that procedure to record the progression of time.	A logical clock is a component for catching sequential and causal connections in a dispersed framework.
The physical clocks are based on cyclic processes such as a celestial rotation.	A logical clock allows global ordering onevents from different processes.

1.13.1 A Framework for a system of logical clocks

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 $<$. This relation is usually called the happened before or causal precedence.

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).

$C : H \mapsto T$ such that

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

This monotonicity property is called the **clock consistency condition.**When T and C satisfy the following condition,

$$e_{i,.} \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$$

Then the system of clocks is **strongly consistent.**

1.13.2 Implementing logical clocks

The two major issues in implanting logical clocks are:

- **Data structures:** representation of each process
- **Protocols:** rules for updating the data structures to ensure consistent conditions.

Data structures:

Each process p_i maintains data structures with the given capabilities:

- A local logical clock (lc_i), that helps process p_i measure its own progress.
- A logical global clock (gc_i), that is a representation of process p_i's local view of the logical global time. It allows this process to assign consistent timestamps to its local events.

Protocol:

The protocol ensures that a process's logical clock, and thus its view of theglobal time, is managed consistently with the following rules:

Rule 1: Decides the updates of the logical clock by a process. It controls send, receive and other operations.

Rule 2: Decides how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

The rules are mentioned in the next section.

1.13.3 Scalar Time

Scalar time is designed by Lamport to synchronize all the events in distributed systems. A Lamport logical clock is an incrementing counter maintained in each process. This logical clock has meaning only in relation to messages moving between processes. When a

process receives a message, it resynchronizes its logical clock with that sender maintaining causal relationship.

The Lamport's algorithm is governed using the following rules:

- The algorithm of Lamport Timestamps can be captured in a few rules:
- All the process counters start with value 0.
- A process increments its counter for each event (internal event, message sending, message receiving) in that process.
- When a process sends a message, it includes its (incremented) counter value with the message.
- On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.

If C_i is the local clock for process P_i then,

- if a and b are two successive events in P_i, then $C_i(b) = C_i(a) + d1$, where $d1 > 0$
- if a is the sending of message m by P_i, then m is assigned timestamp $t_m = C_i(a)$
- if b is the receipt of m by P_j, then $C_j(b) = \max\{C_j(b), t_m + d2\}$, where $d2 > 0$

Rules of Lamport's clock

Rule 1: $C_i(b) = C_i(a) + d1$, where $d1 > 0$
Rule 2: The following actions are implemented when p_i receives a message m with timestamp C_m:
a) $C_i = max(C_i, C_m)$
b) Execute Rule 1
c) deliver the message

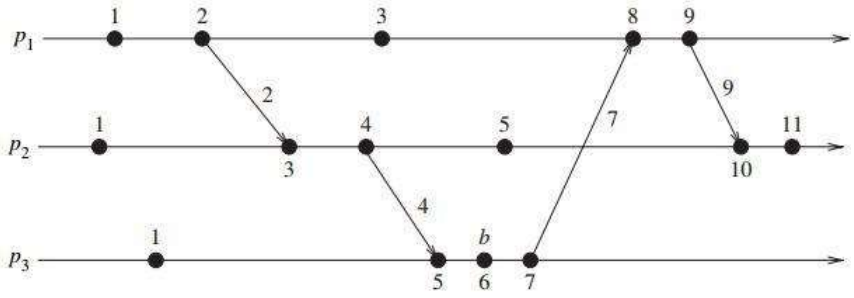


Fig 1.25: Evolution of scalar time

Basic properties of scalar time:

1. **Consistency property:** Scalar clock always satisfies monotonicity. A monotonic clock only increments its timestamp and never jump.Hence it is consistent.

$$C(e_i) < C(e_j)$$

2. **Total Reordering:**Scalar clocks order the events in distributed systems.But all the events do not follow a common identical timestamp. Hence a tie breaking mechanism is essential to order the events. The tie breaking is done through:

   - Linearly order process identifiers.
   - Process with low identifier value will be given higher priority.

   The term (t, i) indicates timestamp of an event, where t is its time of occurrence and i is the identity of the process where it occurred.

The total order relation ($\prec$) over two events x and y with timestamp (h, i) and (k, j) is given by:

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

A total order is generally used to ensure liveness properties in distributed algorithms.

3. **Event Counting**

   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. This is called **height** of the event e. h-1 events have been produced sequentially before the event e regardless of the processes that produced these events.

4. **No strong consistency**

   The 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.

1.13.4 Vector Time

The ordering from Lamport's clocks is not enough to guarantee that if two events precede one another in the ordering relation they are also causally related. Vector Clocks use a vector counter instead of an integer counter. The vector clock of a system with N processes is a vector of N counters, one counter per process. Vector counters have to follow the following update rules:

- Initially, all counters are zero.

- Each time a process experiences an event, it increments its own counter in the vector by one.
- Each time a process sends a message, it includes a copy of its own (incremented) vector in the message.
- Each time a process receives a message, it increments its own counter in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector counter and the value in the vector in the received message.

The time domain is represented by a set of n-dimensional non-negative integer vectors in vector time.

Rules of Vector Time

Rule 1: Before executing an event, process pi updates its local logical time as follows:

$$\upsilon t_i[i] := \upsilon t_i[i] + d \ (d > 0)$$

Rule 2: 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

$$1 \le k \le n : \upsilon t_i[k] := \max(\upsilon t_i[k], \upsilon t[k])$$

2. execute R1

3. deliver the message m

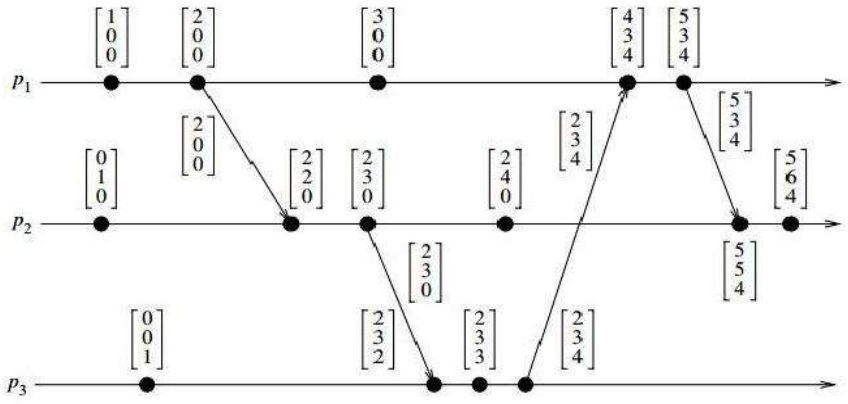


Fig 1.26: Evolution of vector scale

Basic properties of vector time

1. Isomorphism:

- "→" induces a partial order on the set of events that are produced by a distributed execution.
- If events x and y are timestamped as vh and vk then,

$$x \rightarrow y \Leftrightarrow \upsilon h < \upsilon k$$

$$x \parallel y \Leftrightarrow \upsilon h \parallel \upsilon k$$

- There is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.
- If the process at which an event occurred is known, the test to compare two timestamps can be simplified as:

$$x \rightarrow y \Leftrightarrow \upsilon h[i] \leq \upsilon k[i]$$

$$x \parallel y \Leftrightarrow \upsilon h[i] > \upsilon k[i] \wedge \upsilon h[j] < \upsilon k[j]$$

2. 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.

3. Event counting

If an event e has timestamp vh, vh[j] denotes the number of events executed by process p_j that causally precede e.

Vector clock ordering relation

$$t = t` \Leftrightarrow \forall i\ t[i] = t`[i]$$

$$t \neq t` \Leftrightarrow \exists i\ t[i] \neq t`[i]$$

$$t \leq t` \Leftrightarrow \forall i\ t[i] \leq t`[i]$$

$$t < t` \Leftrightarrow (t \leq t` \text{ and } t \neq t`)$$

$$t \,||\, t` \Leftrightarrow \text{not}(t < t` \text{ and } t` < t)$$

t[i]- timestamp of process i.

Implementation of vector clock

If the number of processes in a distributed computation is large, then vector clocks will require piggybacking of huge amount of information in messages for the purpose of disseminating time progress and updating clocks. There are two implementation techniques:

- **Singhal–Kshemkalyani's differential technique**:
  - This approach improves the message passing mechanism by only sending updates to the vector clock that have occurred since the last message sent from Process(i) → Process(j).
  - This drastically reduces the message size being sent, but does require $O(n^2)$ storage. This is due to each node now needing to remember, for each other process, the state of the vector at the last message sent.
  - This also requires FIFO message passing between processes, as it relies upon the guarantee of knowing what the last message sent is, and if messages arrive out of order this would not be possible.
  - If entries $i_1, i_2, \ldots, i_n$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_n$ respectively, since the last message sentto pj, then process pi piggybacks a compressed timestamp of the form.

    $$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n_1}, v_{n_1})\}$$

  - When p_j receives this message, it updates its vector clock as follows:

    $$vt_i[i_k] = max(vt_i[i_k], v_k) \text{ for } k = 1, 2, \ldots, n_1.$$

  - This cuts down the message size, communication bandwidth and buffer (to store messages) requirements.
  - The storage overhead is resolved by maintaining two vectors by process p_i :
    - LSi[1.....n] (' Last Sent '):

      $LS_i[j]$ indicates the value of $\upsilon\, t_i[i]$ when process p_i last send a message to process p_j.

    - LUi[1.....n] (' Last Update'):

      $LU_i[j]$ indicates the value of $\upsilon\, t_i[i]$ when process p_i last update the entry $\upsilon\, t_i[j]$.

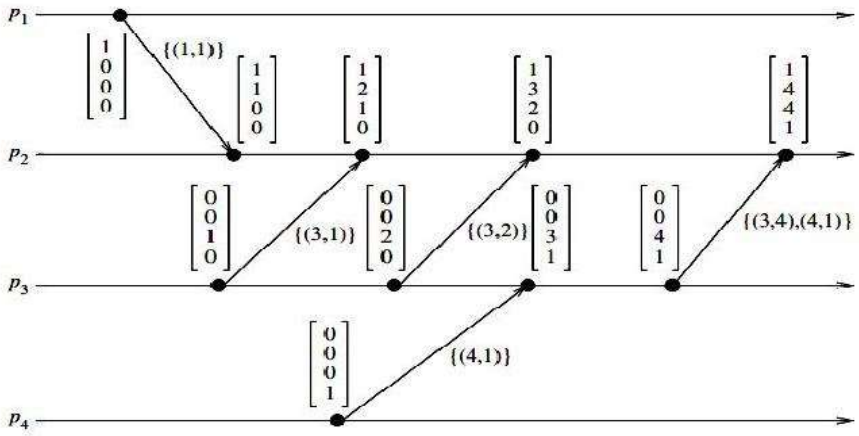


Fig 1.27: Vector clocks progress in Singhal–Kshemkalyani technique

- **Fowler-Zwaenepoel direct-dependency technique:**
  - This technique further reduces the message size by only sending the single clock value of the sending process with a message.
  - However, this means processes cannot know their transitive dependencies when looking at the causality of events.
  - In order to gain a full view of all dependencies that lead to a specific event, an offline search must be made across processes.
  - Each process p_i maintains a dependency vector D_i. Initially,

    $D_i[\ j] = 0$ for $j = 1,\ldots\ldots, n$

  - i is updated as follows:
    1. Whenever an event occurs at p_i such that,

       $$D_i[i] := D_i[i] + 1$$

    2. When a process pi sends a message to process p_j, it piggybacks the updatedvalue of $D_i[i]$ in the message.
    3. When p_i receives a message from p_j with piggybacked value d, $p_i$updates its dependency vector as follows: $D_i[j] := \max\{D_i[j], d\}$.

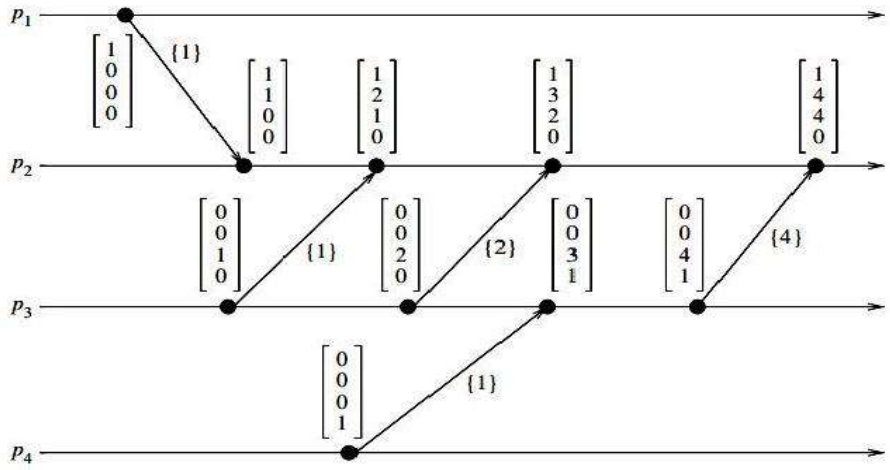


Fig 1.28: Vector clock progress in Fowler–Zwaenepoel technique

- Process p_4 sends a message to process p_3, it piggybacks a scalar that indicates the direct dependency of p_3 on p_4 because of this message.
- Process p_3 sends a message to process p_2 piggybacking a scalar to indicate the direct dependency of p_2 on p_3 because of this message.
- Process p_2 is in fact indirectly dependent on process p4 since process p_3 is dependent on process p_4. However, process p_2 is never informed about its indirect dependency on p_4.

This technique results in considerable saving in the cost; only one scalar is piggybacked on every message.

1.14 PHYSICAL CLOCK SYNCHRONIZATION: NEWTWORK TIME PROTOCOL (NTP)

Centralized systems do not need clock synchronization, as they work under a common clock. But the distributed systems do not follow common clock: each system functions based on its own internal clock and its own notion of time.The time in distributed systems is measured in 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.

Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time.

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**. This degree of synchronization additionally enables to coordinate and schedule actions between multiple computers connected to a common network.

Basic terminologies:

If C_a and C_b are two different clocks, then:

- **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 clockand the real time. The offset of the clock C_a is given by $C_a(t)- t$. Theoffset 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 clockand the perfect clock. The skew of a clock C_a relative to clock C_b at timet is $C_a'(t)- C_b'(t)$.
- **Drift (rate):** The drift of clock C_a the second derivative of the clockvalue with respect to time. The drift is calculated as:

$$C_a''(t) \in (t_b'')$$

Clocking Inaccuracies

Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time). Due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if:

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

ρ - maximum skew rate.

Offset delay estimation

A time service for the Internet - synchronizes clients to UTC Reliability from redundant paths, scalable, authenticates time sources Architecture. The design of NTP involves a hierarchical tree of time servers with 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 isthe synchronization subnet which has the clients.

Clock offset and delay estimation

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 very common practice of performing several trials and chooses the trial with the minimum delay.

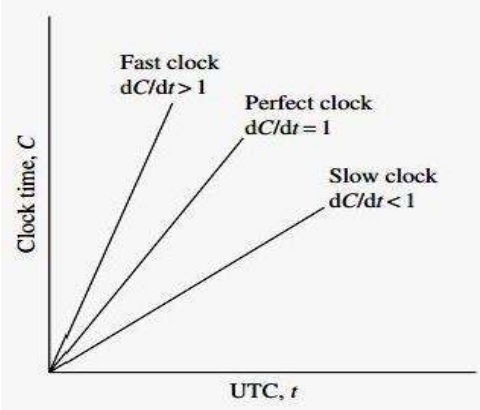


Fig 1.29: Behavior of clocks

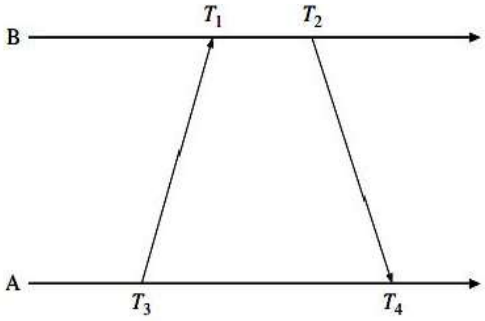


Fig 1.30 a) Offset and delay estimation between processes from same server

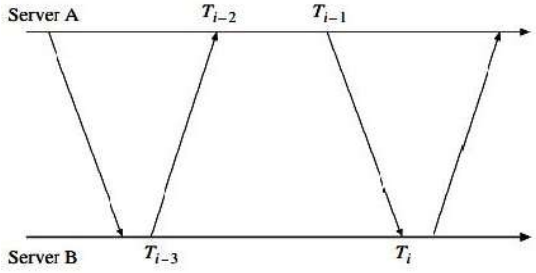


Fig 1.30 b) Offset and delay estimation between processes from different servers

Let T1, T2, T3, T4 be the values of the four most recent timestamps. The clocks A and B are stable and running at the same speed. Let a = T1 − T3 and b = T2 − T4. If the network delay difference from A to B and from B to A, called **differential delay**, is small, the clock offset θ and roundtrip delayδ of B relative to A at time T4 are approximately given by the following:

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

Each NTP message includes the latest three timestamps T1, T2, and T3, while T4 is determined upon arrival.

2

MESSAGE ORDERING & SNAPSHOT

2.1 MESSAGE ORDERING AND GROUP COMMUNICATION

As the distributed systems are a network of systems at various physical locations, the coordination between them should always be preserved. The message ordering means the order of delivering the messages to the intended recipients. The common message order schemes are First in First out (FIFO), non FIFO, causal order and synchronous order. In case of group communication with multicasting, the causal and total ordering scheme is followed. It is also essential to define the behaviour of the system in case of failures. The following are the notations that are widely used in this chapter:

- Distributed systems are denoted by a graph (N, L).
- The set of events are represented by event set $\{E, \prec\}$
- Message is denoted as m^i: send and receive events as s^i and r^i respectively.
- Send (M) and receive (M) indicates the message M send and received.
- a~b denotes a and b occurs at the same process
- The send receive pairs $T=\{(s, r)\ \varepsilon E_i \times E_j$ corresponds to $r\}$

2.1.1 Message Ordering Paradigms

The message orderings are

(i) Non-FIFO

(ii) FIFO

(iii) Causal order

(iv) Synchronous order

There is always a trade-off between concurrency and ease of use and implementation.

Asynchronous Executions

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

- There cannot be any causal relationship between events in asynchronous execution.
- The messages can be delivered in any order even in non FIFO.
- Though there is a physical link that delivers the messages senton it in FIFO order due to the physical properties of the medium, a logicallink may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link.

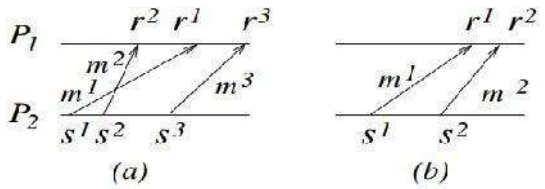


Fig 2.1: a) FIFO executions b) non FIFO executions

FIFO executions

A FIFO execution is an A-execution in which, for all

$(s, r)$and $(s`, r`) T \in$ s $(s`$ and s $r`$ and s $s`\prec$ $) \Rightarrow r`\prec$

- The logical link is non-FIFO.
- FIFO logical channels can be realistically assumed when designing distributed algorithms since most of the transport layer protocols follow connection oriented service.
- A FIFO logical channel can be created over a non-FIFO channel by using a separate numbering scheme to sequence the messages on each logical channel.
- The sender assigns and appends a <sequence_num, connection_id> tuple to each message.
- The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence.

Causally Ordered (CO) executions

CO execution is an A-execution in which, for all,

$(s, r)$and $(s`, r`) T \in$ r $(r`$ and s $s`\prec$ $) \Rightarrow r`\prec$

- Two send events s and s' are related by causality ordering (not physical time ordering), then a causally ordered execution requires that 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.
- Causal order is used in applications that update shared data, distributed shared memory, or fair resource allocation.
- 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 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.**

> ***If send(m^1) $\prec$ send(m^2) then for each common destination d of messages m^1 and m^2, deliverd(m^1) $\prec$deliverd(m^2) must be satisfied.***

- No message overtaken by a chain of messages between the same (sender, receiver) pair.

Other properties of causal ordering

1. **Message Order (MO):** A MO execution is an A-execution in which, for all

   $(s,r)$and $(s`,r`)T \not\in s \quad s` \prec \quad \Rightarrow r` \neg (r \prec \ )$.

2. **Empty Interval Execution:** An execution $(E \prec)$ is an empty-interval (EI) execution if for each pair of events $(s, r) \in T$, the open interval set

   $\{x \in E \mid s \prec x \prec r\}$

   in the partial order is empty.

3. An execution $(E, \prec)$ is CO if and only if for each pair of events $(s, r) \in T$ and each event $e \in E$,

   - weak common past:

     $e \prec r \Rightarrow \neg(s \prec e)$

   - weak common future:

     $s \prec e \Rightarrow \neg(e \prec r)$

Synchronous Execution

- When all the communication between pairs of processes uses synchronous send and receives primitives, the resulting order is the synchronous order.

- The synchronous communication always involves a handshake between the receiver and the sender, the handshake events may appear to be occurring instantaneously and atomically.
- The instantaneous communication property of synchronous executions requires a modified definition of the causality relation because for each (s, r) ∈ T, the send event is not causally ordered before the receive event.
- The two events are viewed as being atomic and simultaneous, and neither event precedes the other.

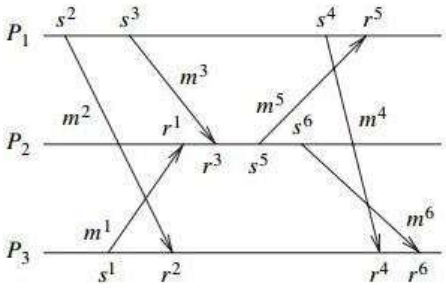


Fig 2.2 a) Execution in an asynchronous system

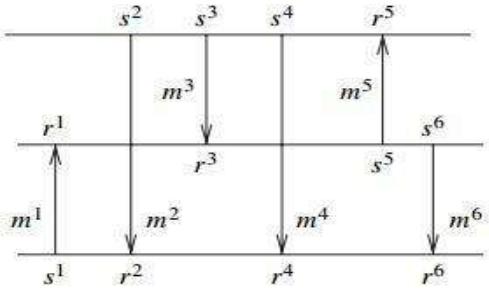


Fig 2.2 b) Equivalent synchronous communication

Causality 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 << y.

S2: If (s, r ∈ T, then) for all x ∈ E, [(x<< s ⟸⟹ x<<r) and (s<< x ⟸⟹ r<< x)].

S3: If x<<y and y<<z, then x<<z.

Synchronous execution: A synchronous execution or S-execution is an execution (E, <<) for which the causality relation << is a partial order.

Timestamping a synchronous execution: An execution (E, ≺) is synchronous if and only if there exists a mapping from E to T (scalar timestamps) such that

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

2.1.2 Asynchronous execution with synchronous communication

When all the communication between pairs of processes is by using synchronous send and receive primitives, the resulting order is synchronous order. The algorithms run on asynchronous systems will not work in synchronous system and vice versa is also true.

Realizable Synchronous Communication (RSC)

A-execution can be realized under synchronous communication is called a realizable with synchronous communication (RSC).

- An execution can be modeled to give a total order that extends the partial order (E, $\prec$).
- In an A-execution, the messages can be made to appear instantaneous if there exist a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event in this linear extension.

Non-separated linear extension is an extension of (E, $\prec$) is a linear extension of (E, $\prec$) such that for each pair (s, r) $\in$ T, the interval { x $\in$ E s $\prec$ x $\prec$ r } is empty.

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

- In the non-separated linear extension, if the adjacent send event and its corresponding receive event are viewed atomically, then that pair of events shares a common past and a common future with each other.

Crown

Let E be an execution. A crown of size k in E is a sequence $<(s^i, r^i), i \in \{0,\ldots, k-1\}>$ of pairs of corresponding send and receive events such that: s0 $\prec$ r1, s1 $\prec$ r2, sk−2 $\prec$ rk−1, sk−1 $\prec$ r0.

The crown is $<(s^1, r^1) (s^2, r^2)>$ as we have s1 $\prec$ r2 and s2 $\prec$ r1. Cyclic dependencies may exist in a crown. The crown criterion states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.

Timestamp criterion for RSC execution

An execution (E, $\prec$) is RSC if and only if 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 x E) \ T, a $\prec$ b $\Rightarrow$ T(a) <T(b)

2.1.3 Hierarchy of ordering paradigms

The orders of executions are:

- Synchronous order (SYNC)
- Causal order (CO)
- FIFO order (FIFO)
- Non FIFO order (non-FIFO)
  - For an A-execution, A is RSC if and only if A is an S-execution.
  - RSC ⊂ CO ⊂ FIFO ⊂ A.
  - The above hierarchy implies that some executions belonging to a class X will not belong to any of the classes included in X. The degree of concurrency is most in A and least in SYNC.
  - A program using synchronous communication is easiest to develop and verify.
  - A program using non-FIFO communication, resulting in an A execution, is hardest to design and verify.

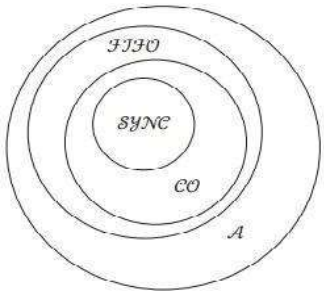


Fig 2.3: Hierarchy of execution classes

2.1.4 Simulations

- The events in the RSC execution are scheduled as per some non-separated linear extension, and adjacent (s, r) events in this linear extension are executed sequentially in the synchronous system.
- The partial order of the asynchronous execution remains unchanged.
- If an A-execution is not RSC, then there is no way to schedule the events to make them RSC, without actually altering the partial order of the given A-execution.

– However, the following indirect strategy that does not alter the partial order can be used.

– Each channel $C_{i,j}$ is modeled by a control process $P_{i,j}$ that simulates the channel buffer.

– An asynchronous communication from i to j becomes a synchronous communication from i to $P_{i,j}$ followed by a synchronous communication from $P_{i,j}$ to j.

– This enables the decoupling of the sender from the receiver, a feature that is essential in asynchronous systems.

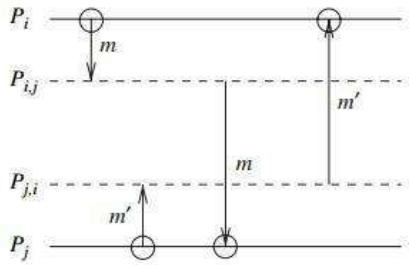


Fig 2.4: Modeling channels as processes to simulate an execution using asynchronous primitives on synchronous system

Synchronous programs on asynchronous systems

– A (valid) S-execution can be trivially realized on an asynchronous system by scheduling the messages in the order in which they appear in the S-execution.

– The partial order of the S-execution remains unchanged but the communication occurs on an asynchronous system that uses asynchronous communication primitives.

– Once a message send event is scheduled, the middleware layer waits for acknowledgment; after the ack is received, the synchronous send primitive completes.

2.2 SYNCHRONOUS PROGRAM ORDER ON AN ASYNCHRONOUS SYSTEM

Non deterministic programs

The partial ordering of messages in the distributed systems makes the repeated runs of the same program will produce the same partial order, thus preserving deterministic nature. But sometimes the distributed systems exhibit non determinism:

- A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.

- Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order.
- If i sends to j, and j sends to i concurrently using blocking synchronous calls, there results a deadlock.
- There is no semantic dependency between the send and the immediately following receive at each of the processes. If the receive call at one of the processes can be scheduled before the send call, then there is no deadlock.

2.2.1 Rendezvous

Rendezvous systems are a form of synchronous communication among an arbitrary number of asynchronous processes. All the processes involved meet with each other, i.e., communicate synchronously with each other at one time. Two types of rendezvous systems are possible:

- Binary rendezvous: When two processes agree to synchronize.
- Multi-way rendezvous: When more than two processes agree to synchronize.

Features of binary rendezvous:

- 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 receives commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

2.2.2 Binary rendezvous algorithm

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.
- Schedule in a deadlock-free manner (i.e., crown-free).
- Schedule to satisfy the progress property in addition to the safety property.

Steps in Bagrodia algorithm

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 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 message (M) types used are: M, ack(M), request(M), and permission(M). 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 control messages. The messages request(M), ack(M), and permission(M) use M's unique tag; the message M is not included in these messages.

(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 execute 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 synchronizaation (from a lowre priority processes) received during the blocking period are queued.

(2) P_i wants to execute SEND(M) to a higher priority process P_j:

(2a) Pi seeks permission from Pj by executing sned(request(M))

// to avoid deadlock in which cyclically blocked processes queue messages.

(2b) while Pi is waiting for permission, it remains unblocked.

(i) If a message M' arrives from a higher priority process P_k, Pi accepts M' by scheduling a RECEIVER(M') event and then executes send(ack(M')) to P_k.

(ii) If a request(M') arrives from a lower priority process P_k, P_i executes send(premission(M')) to Pk and blocks waiting for the message M'. when M' arrives, the RECEIVER (M') event is executed.

(2c) when the permission (M) arrives Pi knows partner P_j is synchronized and P_i executes send(M). The SEND(M) now completes.

(3) request(M) arrival at Pi from a higher priority proces P_j:

At the time a request(M) is processed by pi proces pi executes send(permission(M)) to Pj 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, proess P_i executed RECEIVE(M) (which is assumed to be always enabled) and then send(ack(M)) to P_j.

(5) Processing when Pi is unblocked:

When Pi is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per rule 3 or 4).

Fig 2.5: Bagrodia Algorithm

2.4 GROUP COMMUNICATION

Group communication is done by broadcasting of messages. A message broadcast is the sending of a message to all members in the distributed system. The communication may be

- **Multicast:** A message is sent to a certain subset or a group.
- **Unicasting:** A point-to-point message communication.

The network layer protocol cannot provide the following functionalities:

- 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.

The multicast algorithms can be open or closed group.

Differences between closed and open group algorithms:

Closed group algorithms	**Open group algorithms**
If sender is also one of the receiver in the multicast algorithm, then it is closed group algorithm.	If sender is not a part of the communication group, then it is open group algorithm.
They are specific and easy to implement.	They are more general, difficult to design and expensive.
It does not support large systems where client processes have short life.	It can support large systems.

2.5 CAUSAL ORDER (CO)

In the context of group communication, there are two modes of communication: causal order and total order. Given a system with FIFO channels, causal order needs to be explicitly enforced by a protocol. 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. 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

- Each message M should carry a log of all other messages sent causally before M's send event, and sent to the same destination dest(M).
- The Raynal–Schiper–Toueg algorithm canonical algorithm is a representative of several algorithms that reduces the size of the local space and message space overhead by various techniques.
- This log can then be examined to ensure whether it is safe to deliver a message.
- All algorithms aim to reduce this log overhead, and the space and time overhead of maintaining the log information at the processes.
- To distribute this log information, broadcast and multicast communication is used.
- The hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features:
  - 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

An optimal CO algorithm stores in local message logs and propagates onmessages, information of the form d is a destination of M about a messageM 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.
- **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:

- not in the causal future of $Deliver_d(M_1, a)$
- not in the causal future of $e_{k,c}$ where d $\in M_{k,c}$Dests and there is no other message sent causally between M_i,a and $M_{k,c}$ to the same destination d.

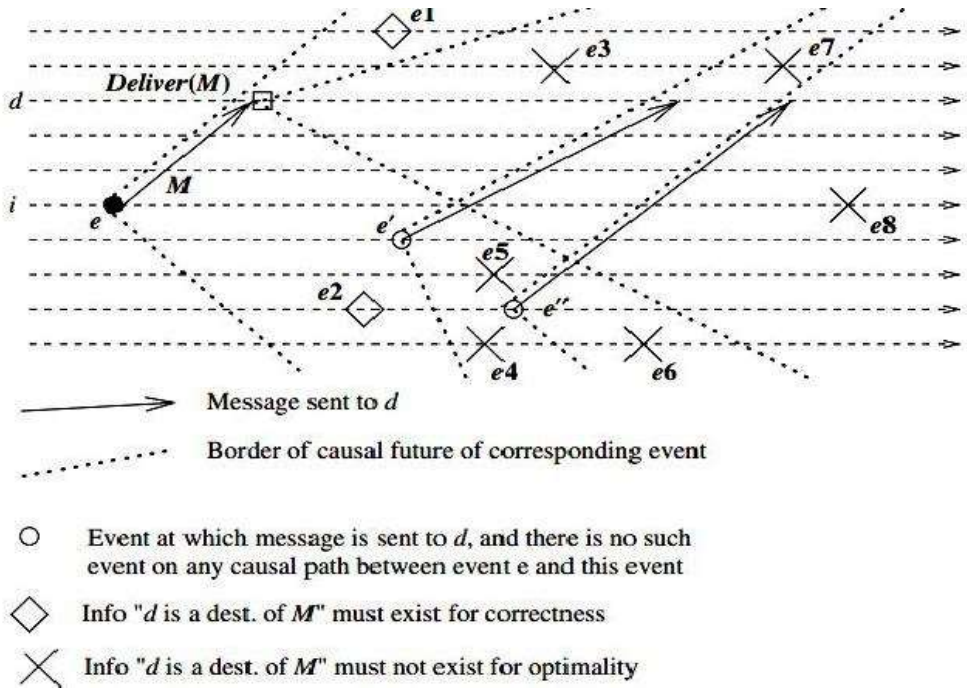


Fig 2.6: Conditions for causal ordering

Information about messages:

(i) not known to be delivered
(ii) not guaranteed to be delivered in CO, is explicitly tracked by the algorithm using (source, timestamp, destination) information.

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. The algorithm for the send and receive operations is given in Fig. 2.7 a) and b). Procedure SND is executed atomically. Procedure RCV is executed atomically except for a possible interruptionin line 2a where a non-blocking wait is required to meet the Delivery Condition.

(1) SND: j sends a message M to Dests

(1a) $clock_j \leftarrow clock_j + 1$;

(1b) for all d $\in$ M Dests do:

OM $\leftarrow$ LOG_j ; // OM denotes $O_{Mj,clockj}$

for all 0 $\in$ O_M, modify o.Dests as follows:

if d $\notin$ o.Dests then o.Dests $\leftarrow$ (o.Dests \ M.Dests);

if d $\in$ o.Dests then o.Dests $\leftarrow$ (o.Dests \ M.Dests) $\cup$ {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 = $\varnothing \wedge \left(\exists\, o`_{s,t`}, \in O_M \mid t < t`\right)$ then $O_M \leftarrow O_M \setminus \{o_{s,t}\}$;

// do not propagate older entries for which Dests field is $\varnothing$

Send (j, $clock_j$, M, Dests, O_M) to d;

(1c) for all $\ell \in LOG_j$ fo ℓ .Dests $\leftarrow$ ℓ .Dests / Dests;

// Do not store information about indirect dependencies that are guaranteed

// to be transitively satisfied when dependencies that are guaranteed

Execute PURGE_NULL_ENTRIES(LOG_j); //purge ℓ .Dests = $\varnothing$

(1d) $LOG_j \leftarrow LOG_j \cup \{(j, clock_j, Dests)\}$

Fig 2.7 a) Send algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

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

(2a) // Delivery condition: ensure that message sent causally before M are delivered

For all $o_{m,t} \in O_M$ do

If j $\in$ $o_{m,t}$. Dests wait until $t_m \leq SR_j[m]$

(2b) Deliver M, $SR_j[k] \leftarrow t_k$;

(2c) $O_M \leftarrow \{(k, t_k, Dests)\} \cup O_M$;

For all $o_{m,t} \in O_M$ do $o_{m,t}$. Dests $\leftarrow$ $o_{m,t}$. Dests \ {j};

// delete the now redundant dependency of message represented by o_{m,t_m} sent to j

(2d) // Merge O_M and LOG_j by eliminating all redundant entries.

// Implicitly track "already delivered" & guaranteed to be delivered in CO'' // messages.

for all $o_{m,t_m} \in O_M$ and $\ell_{s,t`} \in LOG_j$ such that s = m do

if $t < t` \wedge \ell_{s,t`} \notin LOG_j$ then mark o_{m,t_m};

// $\ell_{s,t}$ had been deleted or never inserted, as $\ell_{s,t}$.Dests = $\emptyset$ in the causal past

if $t` < t \wedge o_{m,t_m} \notin O_M$ then mark $\ell_{s,t`}$;

// $o_{m,t_m} \notin O_M$ because $\ell_{s,t`}$ 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 $\ell_{s,t`} \in LOG_j$ and $o_{m,t_m} \in O_M$, such that s = m $\wedge$ t` = t do

$\ell_{s,t`}$.Dests $\leftarrow$ $\ell_{s,t`}$.Dests $\cap$ o_{m,t_m}. 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 $\ell_{s,t`}$

$LOG_j \leftarrow LOG_j \cup O_M$; // merge non-redundant information of O_M into LOG_j

(2e) PURGE_NILL_ENTRIES(LOG_j). //Purge older entries ℓ for which ℓ.Dests= $\emptyset$

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

Fig 2.7 b) Receive algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

The data structures maintained are sorted row–major and then column–major:

1. Explicit tracking:

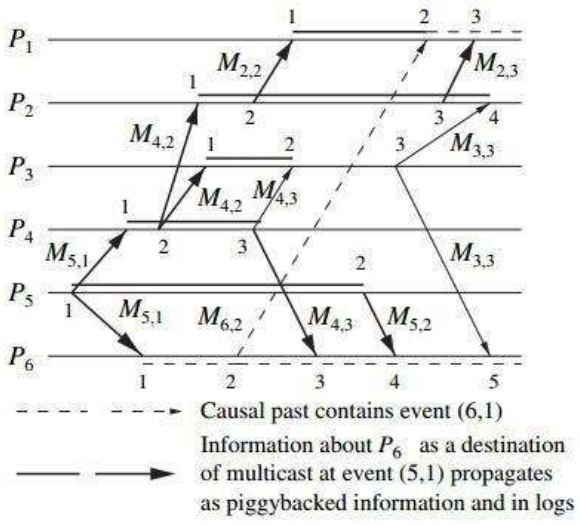
- Tracking of (source, timestamp, destination) information for messages (i) not known to be delivered and (ii) not guaranteed tobe delivered in CO, is done explicitly using

the I.Dests field of entries in local logs at nodes and co. Dests field of entries in messages.

- Sets $l_{i,a}$Destsand $o_{i,a.}$ Dests contain explicit information of destinations to which $M_{i,a}$ is not guaranteed to be delivered in CO and is not known to be delivered.
- The information about d $\in M_{i,a}$ .Dests is propagated up to the earliest events on all causal paths from (i, a) at which it is known that $M_{i,a}$ is delivered to d or is guaranteed to be delivered to d in CO.

2. Implicit tracking:

- Tracking of messages that are either (i) already delivered, or (ii) guaranteed to be delivered in CO, is performed implicitly.
- The information about messages (i) already delivered or (ii) guaranteed tobe delivered in CO is deleted and not propagated because it is redundant as far as enforcing CO is concerned.
- It is useful in determining what information that is being carried in other messages and is being storedin logs at other nodes has become redundant and thus can be purged.
- These mantics are implicitly stored and propagated. This information aboutmessages that are (i) already delivered or (ii) guaranteed to be delivered in CO is tracked without explicitly storing it.
- The algorithm derives it from the existing explicit information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, by examining only $o_{i,a}$ Dests or $l_{i,a}$ Dests, which is a part of the explicit information.



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 $P_{2,6}$	$\{\}$

Fig 2.8: Illustration of propagation constraints

Multicasts $M_{5,1}$ and $M_{4,1}$

Message $M_{5,1}$ sent to processes P4 and P6 contains the piggybacked information $M_{5,1}.Dest = \{P4, P6\}$. Additionally, at the send event (5, 1), the information $M_{5,1}.Dests = \{P4,P6\}$ is also inserted in the local log Log_5. When $M_{5,1}$ is delivered to P6, the (new) piggybacked information $P4 \in M_{5,1}.Dests$ is stored in Log_6 as $M_{5,1}.Dests = \{P4\}$ information about $P6 \in M_{5,1}.Dests$ which was needed for routing, must not be stored in Log_6 because of constraint I. In the same way when $M_{5,1}$ is delivered to process P4 at event (4, 1), only the new piggybacked information $P6 \in M_{5,1}.Dests$ is inserted in Log_4 as $M_{5,1}.Dests = P6$which is later propagated during multicast $M_{4,2}$.

Multicast $M_{4,3}$

At event (4, 3), the information $P6 \in M_{5,1}.Dests$ in Log4 is propagated onmulticast $M_{4,3}$ only to process P6 to ensure causal delivery using the Delivery Condition. The piggybacked information on message $M_{4,3}$ sent to process P3must not contain this information because of constraint II. As long as any future message sent to P6 is delivered in causal order w.r.t. $M_{4,3}$ sent to P6, it will also be delivered in causal order w.r.t. $M_{5,1}$. And as $M_{5,1}$ is already delivered to P4, the information $M_{5,1}Dests = \emptyset$ is piggybacked on $M_{4,3}$ sent to P 3. Similarly, the information $P6 \in M_{5,1}Dests$ must be deleted from Log4 as it will no longer be needed, because of constraint II. $M_{5,1}Dests = \emptyset$ is stored in Log4 to remember that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to all its destinations.

Learning implicit information at P2 and P3

When message $M_{4,2}$ is received by processes P2 and P3, they insert the (new) piggybacked information in their local logs, as information $M_{5,1}.Dests = P6$. They both continue to store this in Log2 and Log3 and propagate this information on multicasts until they learn at events (2, 4) and (3, 2) on receipt of messages $M_{3,3}$ and $M_{4,3}$, respectively, that any future message is expected to be delivered in causal order to process P6, w.r.t. $M_{5,1}$sent toP6. Hence by constraint II, this information must be deleted from Log2 andLog3. The flow of events is given by;

- When $M_{4,3}$ with piggybacked information $M_{5,1}Dests = \emptyset$ is received byP3at (3, 2), this is inferred to be valid current implicit information about multicast $M_{5,1}$because the log Log3 already contains explicit information$P6 \in M_{5,1}.Dests$ about that multicast. Therefore, the explicit informationin Log3 is inferred to be old and must be deleted to achieve optimality. $M_{5,1}$ Dests is set to $\emptyset$ in Log3.

- The logic by which P2 learns this implicit knowledge on the arrival of $M_{3,3}$ is identical.

Processing at P6

When message $M_{5,1}$ is delivered to P6, only $M_{5,1}.Dests = P4$ is added to Log6. Further, P6 propagates only $M_{5,1}.Dests = P4$ on message $M_{6,2}$, and this conveys the current implicit information $M_{5,1}$ has been delivered to P6 by its very absence in the explicit information.

- When the information P6 ∈ $M_{5,1}$ Dests arrives on $M_{4,3}$, piggybacked as $M_{5,1}$.Dests = P6 it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log6 (constraint I) – further, the presence of $M_{5,1}$.Dests = P4 in Log6 implies the implicit information that $M_{5,1}$ has already been delivered to P6. Also, the absence of P4 in $M_{5,1}$.Dests in the explicit piggybacked information implies the implicit information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to P4, and, therefore, $M_{5,1}$. Dests is set to ∅ in Log6.

- When the information P6 ∈ $M_{5,1}$.Dests arrives on $M_{5,2}$ piggybacked as $M_{5,1}$. Dests = {P4, P6} it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log6 because Log6 contains $M_{5,1}$.Dests = ∅, which gives the implicit information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to both P4 and P6.

Processing at P1

- When $M_{2,2}$ arrives carrying piggybacked information $M_{5,1}$.Dests = P6 this (new) information is inserted in Log1.

- When $M_{6,2}$ arrives with piggybacked information $M_{5,1}$.Dests ={P4}, P1learns implicit information $M_{5,1}$has been delivered to P6 by the very absence of explicit information P6 ∈ $M_{5,1}$.Dests in the piggybacked information, and hence marks information P6 ∈ $M_{5,1}$Dests for deletion from Log1. Simultaneously, $M_{5,1}$Dests = P6 in Log1 implies the implicit information that $M_{5,1}$has been delivered or is guaranteed to be delivered in causal order to P4.Thus, P1 also learns that the explicit piggybacked information $M_{5,1}$.Dests = P4 is outdated. $M_{5,1}$.Dests in Log1 is set to ∅.

- The information "P6 ∈$M_{5,1}$.Dests piggybacked on $M_{2,3}$,which arrives at P 1, is inferred to be outdated usingthe implicit knowledge derived from $M_{5,1}$.Dest= ∅" in Log1.

2.6 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 for total ordering

Each process sends the message it wants to broadcast to a centralized process, which relays all the messages it receives to every other process over FIFO channels.

(1) When process Pi wants to multicasts a message M to group G:

(1a) send M(i, G) to central coordinator

(2) When M(i, G) arrives from P_i at the central coordinator

(2a) send M(i, G) to all members of the group G.

(3) When M(i, G) arrives at p_j from the central coordinator

(3a) deliver M(i, G) to the application.

Complexity: Each message transmission takes two message hops and exactly n messages in a system of n processes.

Drawbacks: A centralized algorithm has a single point of failure and congestion, and is not an elegant solution.

Three phase distributed algorithm

Three phases can be seen in both sender and receiver side.

Sender side

Phase 1

- In the first phase, a process multicasts the message M with a locally unique tag and the local timestamp to the group members.

Phase 2

- 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 is non-blocking.

Phase 3

- The process multicasts the final timestamp to the group.

```
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 P_i

FINAL_TS(i, tag, ts) // Phase 3 message sent by P_i, with final timestamp

(1) When process Pi 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)

Fig 2.9: Sender side of three phase distributed algorithm

Receiver Side

Phase 1

- The receiver receives the message with a tentative timestamp. It updates the variable priority that tracks the highest proposed timestamp, 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. In the queue, the entry is marked as undeliverable.

Phase 2

- The receiver sends the revised timestamp back to the sender. The receiver then waits in a non-blocking manner for the final timestamp.

Phase 3

- The final timestamp is received from the multicaster. The corresponding message entry in temp_Q is identified using the tag, and is marked as deliverable after the revised timestamp is overwritten by the final timestamp.

- The queue is then resorted using the timestamp field of the entries as the key. 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.

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

2.7 GLOBAL STATE AND SNAPSHOT RECORDING ALGORITHMS

- A distributed computing system consists of processes that do not share a common memory and communicate asynchronously with each other by message passing.

- Each component of has a local state. The state of the process is the local memory and a history of its activity.

- The state of a channel is characterized by the set of messages sent along the channel lessthe messages received along the channel. The global state of a distributed system is a collection of the local states of its components.

- If shared memory were available, an up-to-date state of the entire systemwould be available to the processes sharing the memory.

- The absence of shared memory necessitates ways of getting a coherent and complete view ofthe system based on the local states of individual processes.

- A meaningful global snapshot can be obtained if the components of the distributed systemrecord their local states at the same time.

- This would be possible if thelocal clocks at processes were perfectly synchronized or if there were aglobal system clock that could be instantaneously read by the processes.

- If processes read time froma single common clock, various indeterminate transmission delays during the read operation will cause the processes to identify various physical instants as the same time.

2.7.1 System Model

- The system consists of a collection of n processes, p1, p2,…, pn that are connected by channels.

- Let C_{ij} denote the channel from process p_i to process p_j.
- Processes and channels have states associated with them.
- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc., and may be highly dependent on the local context of the distributed application.
- The state of channel C_{ij}, denoted by SC_{ij}, is given by the set of messages in transit in the channel.
- The events that may happen are: internal event, send (send (m_{ij})) and receive (rec(m_{ij})) events.
- The occurrences of events cause changes in the processstate.
- A **channel** is a distributed entity and its state depends on the local states of the processes on which it is incident.

  Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$
- The transit function records the state of the channel C_{ij}.
- 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.

2.7.2 A consistent global state

The global state of a distributed system is a collection of the local states of the processes and the channels. The global state is given by:

$$GS = \{ \cup_i LS_i, \cup_{i,j} SC_{ij}\}$$

The two conditions for global state are:

$$C1 : send(m_{ij}) \in LS_i \Leftrightarrow m_{ij} \in SC_{ij} \vee rec(m_{ij}) \in LS_j$$

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

Condition 1 preserves **law of conservation of messages.** Condition C2 states that in the collected global state, for every effect, its cause must be present.

> ***Law of conservation of messages: Every messagem_{ij} that is recorded as sent in the local state of a process p_i must be captured in the state of the channel C_{ij} or in the collected local state of the receiver process p_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 statesare meaningful global states and inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistentstate.

2.7.3 Interpretation of cuts

- Cuts in a space–time diagram provide a powerful graphical aid in representing and reasoning about the global states of a computation. A cut 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 has been sent in the PAST of that cut. Such a cut is known as a consistent cut.
- In a consistent snapshot, all the recorded local states of processes are concurrent; that is, the recorded local state of no process casually affects the recorded local state of any other process.

2.7.4 Issues in recording global state

The non-availability of global clock in distributed system, raises the following issues:

Issue 1:

How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?

Answer:

- 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).

Issue 2:

How to determine the instant when a process takes its snapshot?

This answer

Answer:

A process pj must record its snapshot before processing a message m_{ij} that was sent by process pi after recording its snapshot.

2.8 SNAPSHOT ALGORITHMS FOR FIFO CHANNELS

Each distributed application has number of processes running on different physical servers. These processes communicate with each other through messaging channels.

A snapshot captures the local states of each process along with the state of each communication channel.

Snapshots are required to:

- Checkpointing
- Collecting garbage
- Detecting deadlocks
- Debugging

2.8.1 Chandy–Lamport algorithm

- The algorithm will record a global snapshot for each process channel.
- The Chandy-Lamport algorithm uses a control message, called a marker.
- Aftera site has recorded its snapshot, it sends a marker along all of its outgoing channels before sending out any more messages.
- Since channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded inthe snapshot.
- This addresses issue I1. The role of markers in a FIFO systemis to act as delimiters for the messages in the channels so that the channelstate recorded by the process at the receiving end of the channel satisfies thecondition C2.

Marker sending rule for process p_i

(1) Process pi records its state.

(2) For each outgoing channel C on which a marker has not been sent p_i sends a marker along C befor pi sends further message along C.

marker receiving rule for process p_j

on receiving a marker along channel C:

if pj has not recorded its state then

Record the state of C as the empty set

Execute the "marker sending rule"

else

Record the state of C as the set of messages
received along C after pj,s state was recorded
and before pj received the marker along C

Fig 2.10: Chandy–Lamport algorithm

Initiating a snapshot

- Process P_i initiates the snapshot
- P_i records its own state and prepares a special marker message.
- Send the marker message to all other processes.
- Start recording all incoming messages from channels C_{ij} for j not equal to i.

Propagating a snapshot

- For all processes $P_j$consider a message on channel C_{kj}.
- If marker message is seen for the first time:
  - $P_j$records own sate and marks C_{kj} as empty
  - Send the marker message to all other processes.
  - Record all incoming messages from channels C_{lj} for 1 not equal to j or k.
  - Else add all messages from inbound channels.

Terminating a snapshot

- All processes have received a marker.
- All process have received a marker on all the N-1 incoming channels.
- A central server can gather the partial state to build a global snapshot.

Correctness of the algorithm

- Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process's snapshot.
- A process stops recording the state of an incoming channel when a marker is received on that channel.
- Due to FIFO property of channels, itfollows that no message sent after the marker on that channel is recorded inthe 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 thediameter of the network.

2.8.2 Properties of the recorded global state

The recorded global state may not correspond to any of the global states that occurred during the computation. Consider two possible executions of the snapshot algorithm

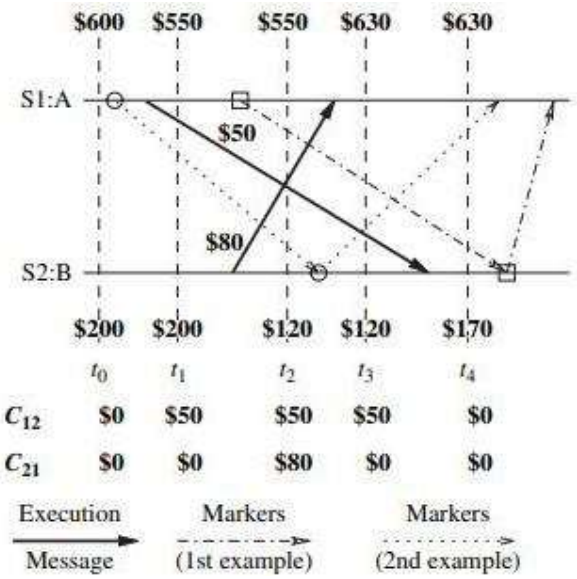


Figure 2.11: Timing diagram of two possible executions of the banking example.

1. (Markers shown using dashed-and-dotted arrows.) Let site S1 initiate the algorithm just after t1. Site S1 records its local state (account A = $550) and sends a marker to site S2. The marker is received by site S2 after t4. 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 C12 = $0 C21 = $80

2. (Markers shown using dotted arrows.) Let site S1 initiate the algorithm just after t0 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 t2 and t3. 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.

3

DISTRIBUTED MUTEX & DEADLOCK

In distributed systems, a process may request resources in any order, which may not be known a priori, and a process can request a resource while holding others. If the allocation sequence of process resources is not controlled in such environments, deadlocks can occur.

Deadlocks can be dealt with using any one of the following three strategies: deadlock prevention, deadlock avoidance, and deadlock detection.

i) **Deadlock prevention** is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-empting a process that holds the needed resource.

ii) In the **deadlock avoidance** approach to distributed systems, a resource is granted to a process if the resulting global system is safe. Deadlock detection requires an examination of the status of the process–resources interaction for the presence of a deadlock condition.

iii) To resolve the deadlock, we have to abort a deadlocked process.

3.1 DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

- Mutual exclusion is a concurrency control property which is introduced to prevent race conditions.
- It is the requirement that a process cannot access a shared resource while another concurrent process is currently present or executing the same resource.

Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time.

- Message passing is the sole means for implementing distributed mutual exclusion.

- The decision as to which process is allowed access to the CS next is arrived at by message passing, in which each process learns about the state of all other processes in some consistent way.
- There are three basic approaches for implementing distributed mutual exclusion:

1. Token-based approach:

- A unique token is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence number to order requests for the critical section.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique.
- Eg: Suzuki-Kasami's Broadcast Algorithm

2. Non-token-based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme.
- Eg: Lamport's algorithm, Ricart–Agrawala algorithm

3. Quorum-based approach:

- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a quorum.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion.
- Eg: Maekawa's Algorithm

3.1.1 Preliminaries

- The system consists of N sites, S_1, S_2, S_3, …, S_N.
- Assume that a single process is running on each site.
- The process at site S_i is denoted by p_i. All these processes communicate asynchronously over an underlying communication network.
- A process wishing to enter the CS requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the CS.
- While waiting the process is not allowed to make further requests to enter the CS.
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS.
- In the requesting the CS state, the site is blocked and cannot make further requests for the CS.
- In the idle state, the site is executing outside the CS.
- In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such state is referred to as 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.
- N denotes the number of processes or sites involved in invoking the critical section, T denotes the average message delay, and E denotes the average critical section execution time.

3.1.2 Requirements of mutual exclusion algorithms

- **Safety property:**

  The safety property states that at any instant, only one process can execute the critical section. This is an essential property of a mutual exclusion algorithm.

- **Liveness property:**

  This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages that will never arrive. In addition, a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS. That is, every requesting site should get an opportunity to execute the CS in finite time.

- **Fairness:**

  Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS. In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system.

3.1.3 Performance metrics

- **Message complexity:** This is the number of messages that are 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:** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out. Thus, response time does not include the time a request waits at a site before its request messages have been sent out.
- **System throughput:** This is the rate at which the system executes requests for the CS. If SD is the synchronization delay and E is the average critical section execution time.

$$\text{System throughput} = \frac{1}{(SD+E)}$$

3.2 LAMPORT'S ALGORITHM

- Lamport's Distributed Mutual Exclusion Algorithm is a permission based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems.
- In permission based timestamp is used to order critical section requests and to resolve any conflict between requests.
- In Lamport's Algorithm critical section requests are executed in the increasing order of timestamps i.e a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp.
- Three type of messages (REQUEST, REPLY and RELEASE) are used and communication channels are assumed to follow FIFO order.
- A site send a REQUEST message to all other site to get their permission to enter critical section.
- A site send a REPLY message to requesting site to give its permission to enter the critical section.
- A site send a RELEASE message to all other site upon exiting the critical section.
- Every site Si, keeps a queue to store critical section requests ordered by their timestamps.
- request_queuei denotes the queue of site Si.

- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

Requesting the critical section

- When a site Si wants to enter the CS, it broadcasts a REQUEST(tsi, i) message to all other sites and places the request on request_queuei. ((tsi, i) denotes the timestamp of the request)
- When a site Sj receives the REQUEST(ts_i, i) message from site S_i, it places side S_i`s request on $request\_queue_j$ and return a timestamped REPLY message to S_i.

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

L1: Si has received a message with timestamp larger then (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 broadcast a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE messge from site S_i, it removes S_i`s request from its request queue.

To enter Critical section:

- When a site S_i wants to enter the critical section, it sends a request message Request(ts_i, i) to all other sites and places the request on $request\_queue_i$. Here, T_{si} denotes the timestamp of Site S_i.
- When a site S_j receives the request message REQUEST(ts_i, i) from site Si, it returns a timestamped REPLY message to site Si and places the request of site Si on $request\_queue_j$

To execute the critical section:

- A site S_i can enter the critical section if it has received the message with timestamp larger than (ts_i, i) from all other sites and its own request is at the top of $request\_queue_i$.

When a site S_i exits the critical section, it removes its own request from the top of its request queue and sends a timestamped RELEASE message to all other sites. When a site S_j receives the timestamped RELEASE message from site S_i, it removes the request of S_{ia} from its request queue.

Message Complexity:

Lamport's Algorithm requires invocation of 3(N – 1) messages per critical section execution. These 3(N – 1) messages involves

- (N – 1) request messages
- (N – 1) reply messages
- (N – 1) release messages

Drawbacks of Lamport's Algorithm:

- **Unreliable approach**: failure of any one of the processes will halt the progress of entire system.
- **High message complexity:** Algorithm requires 3(N-1) messages per critical section invocation.

Performance:

Synchronization delay is equal to maximum message transmission time. It requires 3(N – 1) messages per CS execution. Algorithm can be optimized to 2(N – 1) messages by omitting the REPLY message in some situations.

3.3 RICART–AGRAWALA ALGORITHM

- Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala.
- This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm.
- It follows permission based approach to ensure mutual exclusion.
- Two type of messages (REQUEST and REPLY) are used and communication channels are assumed to follow FIFO order.
- A site send a REQUEST message to all other site to get their permission to enter critical section.

- A site send a REPLY message to other site to give its permission to enter the critical section.
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests.
- Smaller timestamp gets high priority over larger timestamp.
- The execution of critical section request is always in the order of their timestamp.

Requesting the critical section

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

(b) When site Sj receives a REQUEST message from site Si, it sends a REPLY message to site Si if site Sj is neither requesting nor executing the CS, or if the site Sj 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 Sj set $RD_j[i] := 1$.

Executing the critical section

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

Releasing the critical section

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

To enter Critical section:

- When a site S_i wants to enter the critical section, it send a timestamped REQUEST message to all other sites.
- When a site S_j receives a REQUEST message from site S_i, It sends a REPLY message to site S_i if and only if Site S_j is neither requesting nor currently executing the critical section.
- In case Site S_j is requesting, the timestamp of Site S_i's request is smaller than its own request.
- Otherwise the request is deferred by site S_j.

To execute the critical section:

Site S_i enters the critical section if it has received the REPLY message from all other sites.

Upon exiting site S_i sends REPLY message to all the deferred requests.

Message Complexity:

Ricart–Agrawala algorithm requires invocation of 2(N – 1) messages per critical section execution. These 2(N – 1) messages involve:

- (N – 1) request messages
- (N – 1) reply messages

Drawbacks of Ricart–Agrawala algorithm:

- **Unreliable approach:** failure of any one of node in the system can halt the progress of the system. In this situation, the process will starve forever. The problem of failure of node can be solved by detecting failure after some timeout.

Performance:

Synchronization delay is equal to maximum message transmission time It requires 2(N – 1) messages per Critical section execution.

3.4 MAEKAWA's ALGORITHM

- Maekawa's Algorithm is quorum based approach to ensure mutual exclusion in distributed systems.
- In permission based algorithms like Lamport's Algorithm, Ricart-Agrawala Algorithm etc. a site request permission from every other site but in quorum based approach, a site does not request permission from every other site but from a subset of sites which is called quorum.
- Three type of messages (REQUEST, REPLY and RELEASE) are used.
- A site send a REQUEST message to all other site in its request set or quorum to get their permission to enter critical section.
- A site send a REPLY message to requesting site to give its permission to enter the critical section.
- A site send a RELEASE message to all other site in its request set or quorum upon exiting the critical section.

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 Sj receives the REQUEST(i) message, it sends a REPLY(j) message to Si 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 execute the CS only after it has received a REPLY message from every site in R_1.

Releasing the critical section:

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

(e) When a site S_j receives a RELEASE(i) message from site Si, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. It 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.

The following are the conditions for Maekawa's algorithm:

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

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

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

M4 Any site S_j is contained in K number of R_is, $1 \leq i, j \leq N$

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

To enter Critical section:

- When a site S_i wants to enter the critical section, it sends a request message REQUEST(i) to all other sites in the request set R_i.
- When a site S_j receives the request message REQUEST(i) from site S_i, it returns a REPLY message to site S_i if it has not sent a REPLY message to the site from the time it received the last RELEASE message. Otherwise, it queues up the request.

To execute the critical section:

- A site S_i can enter the critical section if it has received the REPLY message from all the site in request set R_i

- When a site S_i exits the critical section, it sends RELEASE(i) message to all other sites in request set R_i
- When a site S_j receives the RELEASE(i) message from site S_i, it send REPLY message to the next site waiting in the queue and deletes that entry from the queue
- In case queue is empty, site S_j update its status to show that it has not sent any REPLY message since the receipt of the last RELEASE message.

Message Complexity:

Maekawa's Algorithm requires invocation of $3\sqrt{N}$ messages per critical section execution as the size of a request set is $\sqrt{N}$. These $3\sqrt{N}$ messages involves.

- $\sqrt{N}$ request messages
- $\sqrt{N}$ reply messages
- $\sqrt{N}$ release messages

Drawbacks of Maekawa's Algorithm:

This algorithm is deadlock prone because a site is exclusively locked by other sites and requests are not prioritized by their timestamp.

Performance:

Synchronization delay is equal to twice the message propagation delay time. It requires $3\sqrt{n}$ messages per critical section execution.

3.5 SUZUKI–KASAMI's BROADCAST ALGORITHM

- Suzuki–Kasami algorithm is a token-based algorithm for achieving mutual exclusion in distributed systems.
- This is modification of Ricart–Agrawala algorithm, a permission based (Non-token based) algorithm which uses REQUEST and REPLY messages to ensure mutual exclusion.
- In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token.
- Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.

Requesting the critical section:

(a) If requesting site Si does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sides. ("sn" is the update value of $RN_i[i]$.)

(b) When a site Sj 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 Si if $RN_j[i] = LN[i] + 1$.

Executing the critical section:

(c) Site Si execute the CS after it has received the token.

Releasing the critical section: Having finished the execution of the CS, site Si takes the following actions:

(d) It sets LN[i] element of the token array equal to RN[i].

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

(f) If the token queue is nonempty after the above update, S_i deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

To enter Critical section:

- When a site S_i wants to enter the critical section and it does not have the token then it increments its sequence number $RN_i[i]$ and sends a request message REQUEST(i, s_n) to all other sites in order to request the token.
- Here s_n is update value of $RN_i[i]$
- When a site S_j receives the request message REQUEST(i, s_n) from site S_i, it sets $RN_j[i]$ to maximum of $RN_j[i]$ and s_ni.e$RN_j[i] = max(RN_j[i], s_n)$.

After updating $RN_j[i]$, Site S_j sends the token to site S_i if it has token and $RN_j[i] = LN[i] + 1$

To execute the critical section:

- Site S_i executes the critical section if it has acquired the token.

To release the critical section:

After finishing the execution Site Si exits the critical section and does following:

- sets $LN[i] = RN_i[i]$ to indicate that its critical section request $RN_i[i]$ has been executed

- For every site S_j, whose ID is not prsent in the token queue Q, it appends its ID to Q if $RN_j[j] = LN[j] + 1$ to indicate that site S_j has an outstanding request.
- After above updation, if the Queue Q is non-empty, it pops a site ID from the Q and sends the token to site indicated by popped ID.
- If the queue Q is empty, it keeps the token

Message Complexity:

The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution. This N messages involves

- (N – 1) request messages
- 1 reply message

Drawbacks of Suzuki–Kasami Algorithm:

- Non-symmetric Algorithm: A site retains the token even if it does not have requested for critical section.

Performance:

Synchronization delay is 0 and no message is needed if the site holds the idle token at the time of its request. In case site does not holds the idle token, the maximum synchronization delay is equal to maximum message transmission time and a maximum of N message is required per critical section invocation.

3.6 DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

Deadlock can neither be prevented nor avoided in distributed system as the system is so vast that it is impossible to do so. Therefore, only deadlock detection can be implemented. The techniques of deadlock detection in the distributed system require the following:

- **Progress:**The method should be able to detect all the deadlocks in the system.
- **Safety:** The method should not detect false of phantom deadlocks.

There are three approaches to detect deadlocks in distributed systems.

Centralized approach:

- Here there is only one responsible resource to detect deadlock.

- The advantage of this approach is that it is simple and easy to implement, while the drawbacks include excessive workload at one node, single point failure which in turns makes the system less reliable.

Distributed approach:

- In the distributed approach different nodes work together to detect deadlocks. No single point failure as workload is equally divided among all nodes.
- The speed of deadlock detection also increases.

Hierarchical approach:

- This approach is the most advantageous approach.
- It is the combination of both centralized and distributed approaches of deadlock detection in a distributed system.
- In this approach, some selected nodes or cluster of nodes are responsible for deadlock detection and these selected nodes are controlled by a single node.

Wait for graph

This is used for deadlock deduction. A graph is drawn based on the request and acquirement of the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

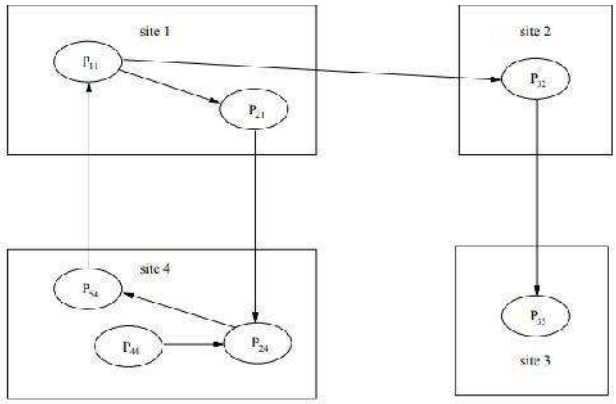


Fig 3.1: Wait for graph

3.6.1 Deadlock Handling Strategies

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. There are three strategies for handling deadlocks:

- **Deadlock prevention:**
  - This is 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.
- **Deadlock avoidance:**
  - A resource is granted to a process if the resulting global system state is safe. This is impractical in distributed systems.
- **Deadlock detection:**
  - This 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.

3.6.2 Issues in deadlock Detection

Deadlock handling faces two major issues

1. Detection of existing deadlocks
2. Resolutionof detected deadlocks

Deadlock Detection

- Detection of deadlocks involves addressing two issues namely maintenance of the WFG and searching of the WFG for the presence of cycles or **knots**.
- In distributed systems, a cycle or knot may involve several sites, the search for cycles greatly depends upon how the WFG of the system is represented across the system.
- Depending upon the way WFG information is maintained and the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems.

Correctness criteria

A deadlock detection algorithm must satisfy the following two conditions:

1. **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.

2. **Safety -No false deadlocks:**

   The algorithm should not report deadlocks which do not exist. This is also called as 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.
- The deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph.
- When a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system.
- If this information is not cleaned in a timely manner, it may result in detection of phantom deadlocks.

3.7 MODELS OF DEADLOCKS

The models of deadlocks are explained based on their hierarchy. The diagrams illustrate the working of the deadlock models. P_a, P_b, P_c, $P_d$are passive processes that had already acquired the resources. $P_e$is active process that is requesting the resource.

3.7.1 Single Resource Model

- A process can have at most one outstanding request for only one unit of a resource.
- 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.

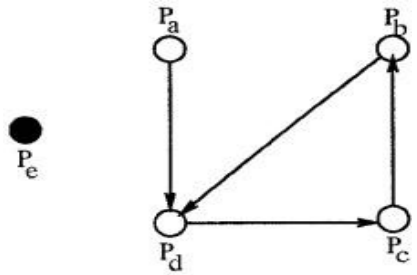


Fig 3.2: Deadlock in single resource model

3.7.2 AND Model

- In the AND model, a passive process becomes active (i.e., its activation condition is fulfilled) only after a message from each process in its dependent set has arrived.
- In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The requested resources may exist at different locations.
- 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.
- Each node of the WFG in such a model is called an AND node.
- In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked.

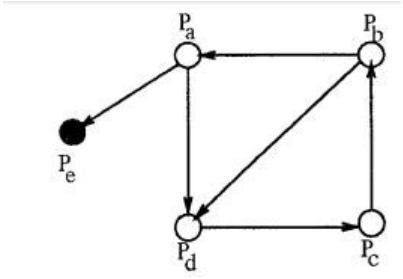


Fig 3.3: Deadlock in AND model

3.7.3 OR Model

- In the OR model, a passive process becomes active only after a message from any process in its dependent set has arrived.
- This models classical nondeterministic choices of receive statements.
- A process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- The requested resources may exist at different locations.
- If all requests in the WFG are OR requests, then the nodes are called OR nodes.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- In the OR model, the presence of a knot indicates a deadlock.

Deadlock in OR model: a process Pi is blocked if it has a pending OR request to be satisfied.

- With every blocked process, there is an associated set of processes called **dependent set.**
- A process shall move from an idle to an active state on receiving agrant message from any of the processes in its dependent set.
- A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set.
- A set of processes S is deadlocked if all the processes in S are permanently blocked.
- In short, a process is deadlocked or permanently blocked, if the following conditions are met:
    1. Each of the process is the set S is blocked.
    2. The dependent set for each process in S is a subset of S.
    3. No grant message is in transit between any two processes in set S.
- A blocked process P is the set S becomes active only after receiving a grant message from a process in its dependent set, which is a subset of S.

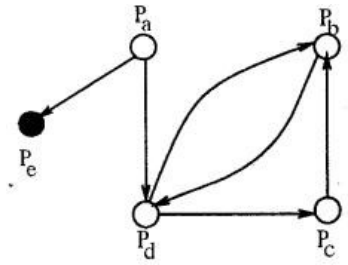


Fig 3.4: OR Model

3.7.4 $\binom{p}{q}$ Model (p out of q model)

- This is a variation of AND-OR model.
- This allows a request to obtain any k available resources from a pool of n resources. Both the models are the same in expressive power.
- This favours more compact formation of a request.

- Every request in this model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as $\binom{p}{q}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

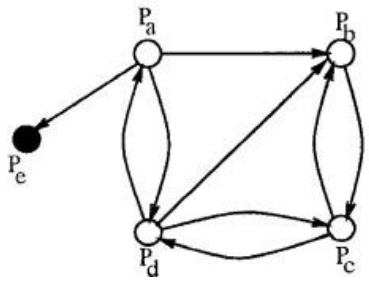


Fig 3.5: p out of q Model

3.7.5 Unrestricted model

- No assumptions are made regarding the underlying structure of resource requests.
- In this model, only one assumption that the deadlock is stable is made and hence it is the most general model.
- This way of looking at the deadlock problem helps in separation of concerns: concerns about properties of the problem are separated from underlying distributed systems computations. Hence, these algorithms can be used to detect other stable properties as they deal with this general model.
- These algorithms are of more theoretical value for distributed systems since no further assumptions are made about the underlying distributed systems computations which leads to a great deal of overhead.

3.8 KNAPP'S CLASSIFICATION OF DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

The four classes of distributed deadlock detection algorithm are:

1. Path-pushing
2. Edge-chasing
3. Diffusion computation
4. Global state detection

3.8.1 Path Pushing algorithms

- In path pushing algorithm, the distributed deadlock detection are detected by maintaining an explicit global wait for graph.
- The basic idea is to build a global WFG (Wait For Graph) for each site of the distributed system.
- At each site whenever deadlock computation is performed, it sends its local WFG to all the neighbouring 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 WFGhas led to the term path-pushing algorithms.

Examples: Menasce-Muntz ,Gligor and Shattuck, Ho and Ramamoorthy, Obermarck

3.8.2 Edge Chasing Algorithms

- 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.

Examples: Chandy et al., Choudhary et al., Kshemkalyani–Singhal, Sinha–Natarajan algorithms.

3.8.3 Diffusing Computation 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.
  Examples: Chandy–Misra–Haas algorithm for one OR model, Chandy–Herman algorithm

3.8.4 Global state detection-based algorithms

Global state detection based deadlock detection algorithms exploit the following facts:

1. A consistent snapshot of a distributed system can be obtained without freezing the underlying computation.
2. 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

3.9 MITCHELL AND MERRITT'S ALGORITHM FOR THE SINGLE-RESOURCE MODEL

- This deadlock detection algorithm assumes a single resource model.
- This detects the local and global deadlocks each process has assumed two different labels namely private and public each label is accountant the process id guarantees only one process will detect a deadlock.
- Probes are sent in the opposite direction to the edges of the WFG.
- When a probe initiated by a process comes back to it, the process declares deadlock.

Features:

1. Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock. This algorithm can be improvised by including priorities, and the lowest priority process in a cycle detects deadlock and aborts.
2. In this algorithm, a process that is detected in deadlock is aborted spontaneously, even though under this assumption phantom deadlocks cannot be excluded. It can be shown, however, that only genuine deadlocks will be detected in the absence of spontaneous aborts.

Each node of the WFG has two local variables, called labels:

1. a private label, which is unique to the node at all times, though it is not constant.
2. 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 sof the system.

- The algorithm is defined by the four state transitions as shown in Fig.3.10, where z = inc(u, v), and inc(u, v) yields aunique label greater than both u and v labels that are notshown do not change.
- The transitions in the defined by the algorithm are block, activate , transmit and detect.
- **Block** creates an edge in the WFG.
- Two messages are needed, one resource request and onemessage back to the blocked process to inform it of thepublic label of the process it is waiting for.
- **Activate** denotes that a process has acquired the resourcefrom the process it was waiting for.
- **Transmit** propagates larger labels in the opposite directionof the edges by sending a probe message.
- **Detect** means that the probe with the private label of some process has returned to it, indicating a deadlock.
- This algorithm can easily be extended to include priorities, so that whenever a deadlock occurs, the lowest priority process gets aborted.
- This priority based algorithm has two phases.

  1. The first phase is almost identical to the algorithm.

2. The second phase the smallest priority is propagated around the circle. The propagation stops when one process recognizes the propagated priority as its own.

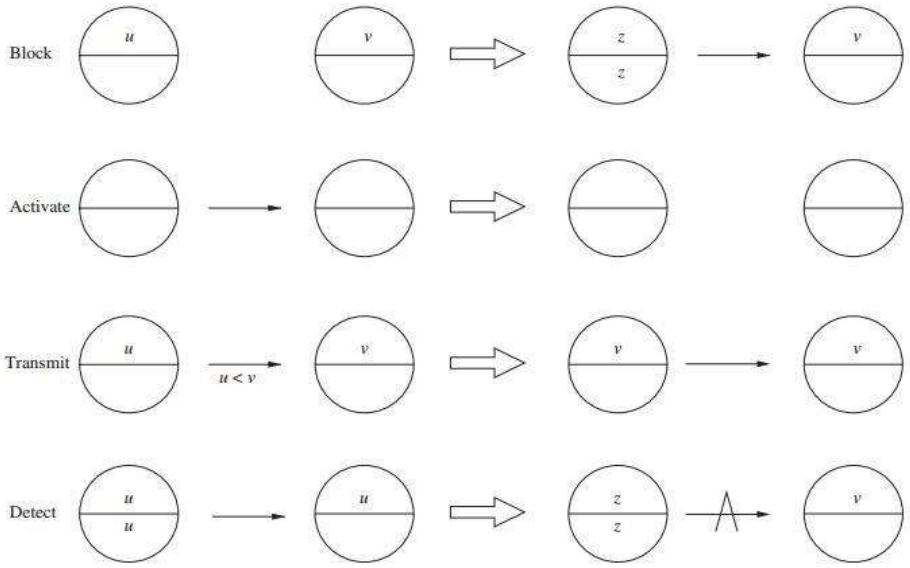


Fig 3.6: Four possible state transitions

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.

3.10 CHANDY–MISRA–HAAS ALGORITHM FOR THE AND MODEL

- This is considered an edge-chasing, probe-based algorithm.
- It is also considered one of the best deadlock detection algorithms for distributed systems.
- If a process makes a request for a resource which fails or times out, the process generates a probe message and sends it to each of the processes holding one or more of its requested resources.
- This 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 Pi andit is being sent by the home site of process Pj to the home site of process Pk.
- Each probe message contains the following information:
  - the id of the process that is blocked (the one that initiates the probe message);

- the id of the process is sending this particular version of the probe message;
- the id of the process that should receive this probe message.

- 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.
- When a process receives a probe message,it checks to see if it is also waiting for resources
- If not, it is currently using the needed resource and will eventually finish and release the resource.
- If it is waiting for resources, it passes on the probe message to all processes it knows to be holding resources it has itself requested.
- The process first modifies the probe message, changing the sender and receiver ids.
- If a process receives a probe message that it recognizes as having initiated,it knows there is a cycle in the system and thus, deadlock.

Data structures

Each process Pi maintains a boolean array, dependenti, where dependent(j) is true only if Pi knows that Pj is dependent on it. Initially, dependenti (j) is false for all i and j.

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 Pk, 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 followin 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 Pm, and

(b`) P_m is waiting on P_n, and

(c`) Pm and Pn are on different sites,

send a probe (i, m, n) to the home site of P_n

end.

Performance analysis

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

Advantages:

- It is easy to implement.
- Each probe message is of fixed length.
- There is very little computation.
- There is very little overhead.
- There is no need to construct a graph, nor to pass graph information to other sites.
- This algorithm does not find false (phantom) deadlock.
- There is no need for special data structures.

3.11 CHANDY–MISRA–HAAS ALGORITHM FOR THE OR MODEL

- 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)
  - 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 Pk receives a query(i, j, k) message, it takes the following actions:
  1. If this is the first query message received by Pk for the deadlock detection initiated by Pi, 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.
  2. If this is not the engaging query, then Pk returns a reply message to it immediately provided Pk has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.
     - Process Pk 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 Pi.
     - When a blocked process Pk 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 has sent out for this engaging query.
     - The initiator process detects a deadlock when it has received reply messages to all the query messages it has sent out.

Intitate 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

$num_k(i) := num_k(i) - 1$;

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.

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 edge

4

RECOVERY & CONSENSUS

4.1 CHECKPOINTING AND ROLLBACK RECOVERY

- Rollback and checkpointing are used to ensure consistency in distributed systems.
- The following are the prominent functions of 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

The saved state is called a checkpoint, and the procedure of restarting from a previously checkpointed state is called rollback recovery.

- The rollback recovery of distributed systems is complicated due to inter-process dependencies during failure-free operation.
- Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called a rollback propagation.
- In rollback propagation the dependencies may force some of the processes that did not fail to rollback. This phenomenon is called **domino effect.**
- **Independent checkpointing: I**f each participating process takes its checkpoints independently, then the system is susceptible to the domino effect. This approach is called independent or uncoordinated checkpointing.
- **Coordinated checkpointing:** It is desirable to avoid the domino effect and therefore several techniques have been developed to prevent it. One such

technique is coordinated checkpointing where processes coordinate their checkpoints to form a system-wide consistent state.

- **Communication induced checkpointing**: In case of a process failure, the system state can be restored to such a consistent set of checkpoints, preventing the rollback propagation. 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 recovery

- **Log based rollback** recovery combines checkpointing with logging of nondeterministic events.
- This 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.

> ***Rollback recovery protocol is a process in which a system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure.***

- 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.
- When a system crash occurs, user must consult the log.
- In principle, that demands searching the entire log to determine this information. There are two major difficulties with this approach:
  - The search process is time-consuming.
  - Most of the transactions need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.

4.1.1 System Model

- A distributed system consists of a fixed number of processes, P_1, P_2, ..., P_n that communicate with each other only through send and receive messages.

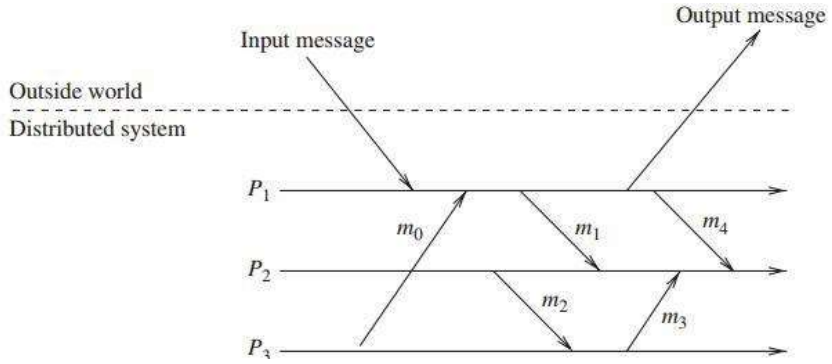


Fig 4.1: Example of communications in a distributed system

- Rollback-recovery protocols always keep track of information about the internal interactions among processes and also the external interactions with the outside world.

4.1.2 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.

- All processes save their local states at certain instants of time.
- A local check point is a snapshot of the state of the process at a given instance.
- Depending upon the checkpointing method used, a process may keep several local checkpoints or just a single checkpoint at any time.
- The assumptions made in local checkpointing are:
  - A process stores all local checkpoints on the stable storage
  - A process is able to roll back to any of its existing local checkpoints
- $C_{i,k}$ denotes the k^{th} local checkpoint at process P_i. $C_{i,0}$ indicates that a process P takes a checkpoint $C_{i,0}$ before it starts execution.

4.1.3 Consistent States

A consistent global state is one that may occur during a failure-free execution of a distributed computation.

- 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.
- 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 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 **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 point that is received by another process before taking its checkpoint.
- 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.

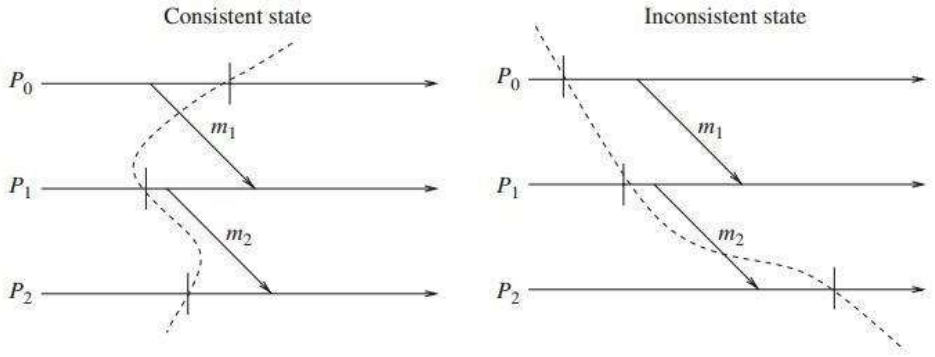


Fig 4.2: Consistent and Inconsistent states

- The primary objective 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 enough to restore the reconstructed state be the 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.

4.1.4 Communication with outside world

- Communication with outside world is mainly to receive input data or deliver the outcome of a computation.
- The Outside World Process (OWP) is a defined as a special process that interacts with the rest of the distributed system through message passing.
- Before sending output to the OWP, the system must ensure that the state from which the output is sent will be recovered despite any future failure. This is commonly called the **output commit problem.**
- The most common approach in interacting with outside world is to save each input message on the stable storage before allowing the application program to process it.
- Also, the input messages that a system receives from the OWP may not be reproducible during recovery, because it may not be possible for the outside world to regenerate them.
- Outside world interaction is indicated using the Symbol "||". This specifies that an interaction with the outside world to deliver the outcome of a computation is to be done.

4.1.5 Types of Messages

- A process failure and recovery involve exchange and storing of messages that were sent and received before the failure in abnormal states.
- This is to facilitate the rollback of processes for recovery. This operation involves sending and receiving of several types of messages.

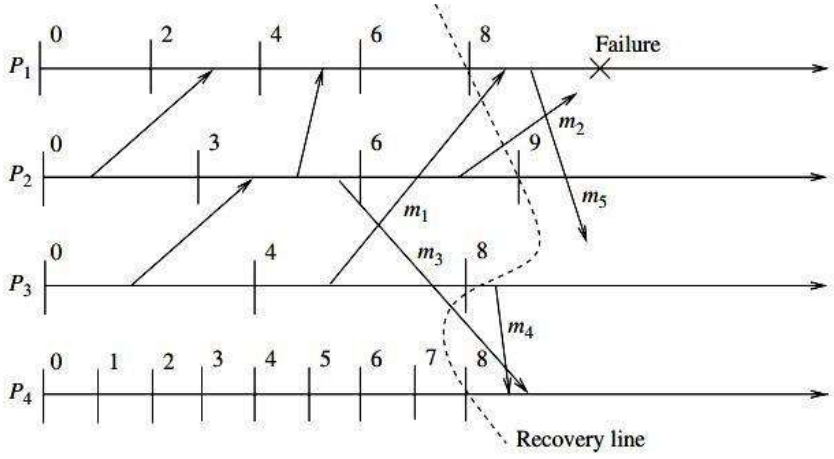


Fig 4.3: Types of messages

In Transit Messages

- These are the messages that have been sent but not yet received.
- These messages do not cause any inconsistency. However, depending on whether the system model assumes reliable communication channels, rollback-recovery protocols may have to guarantee the delivery of in-transit messages when failures occur.
- For reliable communication channels, a consistent state must include in-transit messages because they will always be delivered to their destinations in any legal execution of the system.
- If a system model assumes lossy communication channels, then in-transit messages can be omitted from system state.
- The global state $\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$ in Fig 4.3 shows that message m1 has been sent but not yet received.

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.
- The message m1 in Fig 4.3 is a lost message.

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.
- Messages m2 and m5 are delayed messages.

Orphan messages

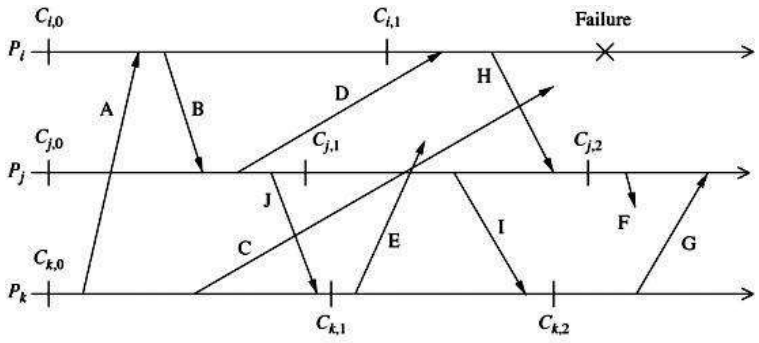
- Messages with receive recorded but message send not recorded are called orphan messages.
- A rollback might have undone the send of such messages, leaving the receive event intact at the receiving process.
- Orphan messages do not arise if processes roll back to a consistent global state.

Duplicate messages

- Duplicate messages arise due to message logging and replaying during process recovery.
- In fig 4.3, message m4 was sent and received before the rollback.
- Due to the rollback of process P4 to $\mathbf{C_{4,8}}$ and process P3 to $\mathbf{C_{3,8}}$, both send and receipt of message m4 are undone.
- When process P3 restarts from $\mathbf{C_{3,8}}$, it will resend message m4. Therefore, P4 should not replay message m4 from its log. If P4 replays message m4, then message m 4 is called a duplicate message.

4.2 ISSUES IN RECOVERY FROM FAILURES

During the recovery process, it is essential to not only restore the system to a consistentstate, but also to handle the messages appropriately that is left in an abnormal state due to the failure and recovery.



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 : $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

Fig 4.4: Illustration of issues in failure recovery

- Assume that the process P_i fails and all the contents of the volatile memory of P_i are lost and, after P_i has recovered from the failure, the system needs to be restored to a consistent global state from where the processes can resume their execution.
- Process P_i's state is restored to a valid state by rolling it back to its most recent checkpoint $C_{i,1}$.
- To restore the system to a consistent state, the process P_j rolls back to checkpoint $C_{j,1}$ because the rollback of process P_i to checkpoint $C_{i,1}$ created an orphan message H.

- P_j does not roll back to checkpoint $C_{j,1}$ but to checkpoint $C_{j,1}$ because rolling back to checkpoint $C_{j,2}$ does not eliminate the orphan message H.
- Even this resulting state is not a consistent global state, as an orphan message I is created due to the roll back of process P_j to checkpoint $C_{j,1.}$
- To eliminate this orphan message, process $\mathbf{P_k}$ rolls back to checkpoint $\mathbf{Ck_{,1}}$.
- The restored global state $\{\mathbf{C_{i,1}}, \mathbf{C_{j,1}}, \mathbf{C_{k,1}}\}$ is a consistent state as it is free from orphan messages.
- The system state has been restored to a consistent state but there are several messages left in an erroneous state which must be handled correctly.
- Messages A, B, D, G, H, I, and J had been received at the points indicated in the figure and messages C, E, and F were in transit when the failure occurred.
- Restoration of system state to checkpoints $\{ \mathbf{C_{i,1}}, \mathbf{C_{j,1}}, \mathbf{C_{k,1}} \}$ automatically handles messages A, B, and J because the send and receive events of messages A, B, and J have been recorded, and both the events for G, H, and I have been completely undone.
- These messages cause no problem and we call messages A, B, and J normal messages and messages G, H, and I vanished messages.
- Messages C, D, E, and F are potentially problematic.
- Message C is in transit during the failure and it is a delayed message.
- The delayed message C has several possibilities: C might arrive at process Pi before it recovers, it might arrive while $\mathbf{P_i}$ is recovering, or it might arrive after $\mathbf{P_i}$ has completed recovery.
- Each of these cases must be dealt with correctly.
- Message D is a lost message since the send event for D is recorded in the restored state for process $\mathbf{P_j}$, but the receive event has been undone at process $\mathbf{P_i}$.
- Process $\mathbf{P_j}$ will not resend D without an additional mechanism, since the send D at $\mathbf{P_j}$ occurred before the checkpoint and the communication system successfully delivered D.
- Messages E and F are delayed orphan messages and raises a serious problem of all the messages.
- When messages E and F arriveat their respective destinations, they must be discarded since their sendevents have been undone.

- Processes, after resuming execution from theircheckpoints, will generate both of these messages, and recovery techniques must be able to distinguish between messages like C and those likeE and F.
- Lost messages like D can be handled by having processes keep a messagelog of all the sent messages.
- So when a process restores to a checkpoint,it replays the messages from its log to handle the lost message problem.
- However, message logging and message replaying during recovery can resultin duplicate messages.
- Process $\mathbf{P_k}$,which has already received message J, will receive it again, thereby causinginconsistency in the system state.
- Overlapping failures further complicate the recovery process. A process$\mathbf{P_j}$ that begins rollback/recovery in response to the failure of a process $\mathbf{P_i}$ can itself fail and develop amnesia with respect process $\mathbf{P_i}$'s failure.
- $\mathbf{P_j}$ can act in a fashion that exhibits ignorance of process $\mathbf{P_i}$'s failure.

4.3 CHECKPOINT-BASED RECOVERY

In check point based recovery, the state of each process and the communication channel is checkpointed frequently so that when a failure occurs, the system can be restored to a globally consistent set of checkpoints.

- This scheme does not depend on the PWD assumption, and hence it does not need to detect, log, or replay non-deterministic events.
- These protocols are less restrictive and simpler to implement than log-based rollback recovery.
- The drawback here is, it does not guarantee that pre-failure execution can be deterministically regenerated after a rollback.
- This property makes its unsuitable for applications that involve interactions with the outside world.
- The three types of rollback-recovery techniques are:
    1. Uncoordinated checkpointing
    2. Coordinated checkpointing
    3. Communication-induced checkpointing

4.3.1 Uncoordinated Checkpointing

- Here, each process has autonomy in deciding when to take checkpoints.
- This eliminates the synchronization overhead as there is no need for coordination between processes and it allows processes to take checkpoints when it is most convenient or efficient.
- The major advantage of this method: lower runtime overhead during normal execution
- The limitations are:
  - 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
- As each process takes checkpoints independently, it is necessary to determine a consistent global checkpoint to rollback to, when a failure occurs.
- To accomplish this, the processes record the dependencies among their checkpoints caused by message exchange during failure free operation.
- When a failure occurs, the recovering process initiates rollback by broadcasting 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.

4.3.2 Coordinated checkpointing

- In coordinated checkpointing, processes orchestrate their checkpointing activities 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.
- This 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.
- 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.
- There are two types of Coordinated checkpoints:
- **Blocking Checkpointing:** After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete
- The disadvantages are 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.

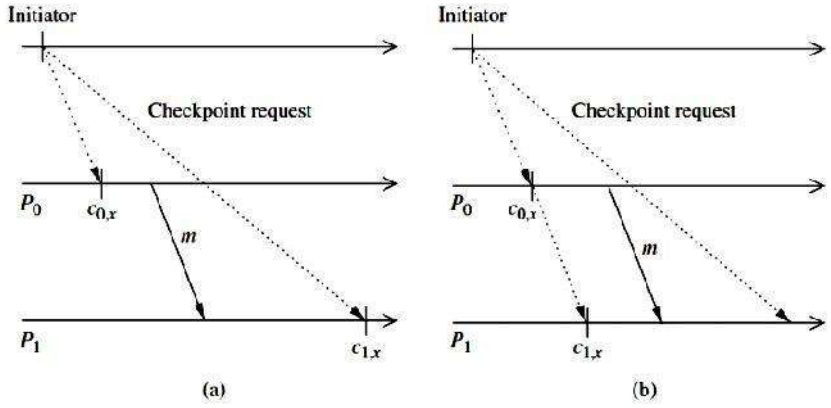


Fig 4.5:a) Non blocking coordinated checkpoint b) Inconsistency in checkpoint

- Coordinated checkpointing requires all processes to participate in every checkpoint.
- This requirement generates valid concerns about its scalability. Hence it is desirable to reduce the number of processes involved in a coordinated checkpointing session.
- This can be done since only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints.

4.3.3 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.
- The algorithm consists of two phases.
  - **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.
  - **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.
- **Z-dependency** is that there does not exist a non-blocking algorithm that will allow a minimum number of processes to take their checkpoints.

4.3.4 Communication-induced checkpointing

- Communication-induced checkpointing avoids the domino effect, while allowing processes to take some of their checkpoints independently.
- Processes may be forced to take additional 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.

- In communication-induced checkpointing, processes take two types of checkpoints: autonomous and forced checkpoints.
- The checkpoints that a process takes independently are called **local checkpoints**.
- The process is forced to take are called **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: model-based heckpointing and index-based checkpointing.

Model based checkpointing

- This prevents patterns of communications and checkpoints that may result in inconsistent states among the existing checkpoints.
- A process detects the inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns.
- A forced checkpoint prevents the undesirable patterns from occurring.
- No control messages are exchanged among the processes during the normal operation.
- All information necessary to execute the protocol is piggybacked on application messages.
- The decision to take a forced checkpoint is done locally using the information available.
- There are several domino-effect-free checkpoint and communication models.
- The **MRS (mark, send, and receive)** model avoids the domino effect by ensuring that within every checkpoint interval all message receiving events precede all message-sending events.

- This model can be maintained by taking an additional checkpoint before every message-receiving event that is not separated from its previous message-sending event by a checkpoint.
- Another method is by taking a checkpoint immediately after every message-sending event.

Index-based checkpointing

- This 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.

4.4 LOG BASED RECOVERY

A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.

Deterministic and non-deterministic events

- In log-based rollback, a process execution canbe modeled as a sequence of deterministic state intervals, each starting withthe execution of a non-deterministic event.
- A non-deterministic event can bethe receipt of a message from another process or an event internal to theprocess.
- In Fig 4.6, the execution of process P0 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 m0, m3, andm7, respectively.
- Send event of message m2 is uniquely determined by theinitial state of P0 and by the receipt of message m0, and is therefore not anon-deterministic event.
- Log-based rollback recovery assumes that all non-deterministic events canbe 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.
- Each process also takes checkpoints to reduce the extent of rollback during recovery. After a failure occurs, the failed processes recover by using the

checkpoints and logged determinants to replay the corresponding non-deterministic events precisely as they occurred during the pre-failure execution.

- The execution within each deterministic interval depends only on the sequence of non-deterministic events that preceded the interval's beginning, the pre-failure execution of a failed process can be reconstructed during recovery up to the first non-deterministic event whose determinant is not logged.

No orphans consistency conditions

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.

$$\forall(e):\ \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$$

This property is called the always-no-orphans condition.

No-orphans condition: It states that if any surviving process depends on an event e, then either event e is logged on the stable storage, or the process has a copy of the determinant of event e.

- Log-based rollback-recovery protocols guarantee that upon recovery of all failed processes, the system does not contain any orphan process.
- A process whose state depends on a non-deterministic event that cannot be reproduced during recovery.
- Log-based rollback-recovery protocols are of three types: pessimistic logging, optimistic logging, and causal logging protocols.
- They differ in their failure-free performance overhead, latency of output commit, simplicity of recovery and garbage collection, and the potential for rolling back surviving processes.

4.4.1 Pessimistic logging

Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation.

- Pessimistic protocols log to the stable storage the determinant of each non-deterministic event before the event affects the computation.
- Pessimistic protocols implement as synchronous logging, which is a stronger than the always-no-orphans condition.

$$\forall e : \neg \text{Stable}(e) \Rightarrow \text{Depend}(e) = 0$$

- The processes must take periodic checkpoints to minimize the amount of work that has to be repeated during recovery.
- When a process fails, the process is restarted from the most recent checkpoint and the logged determinants are used to recreate the prefailure execution.
- In a pessimistic logging system, the observable state of each process is always recoverable.
- But there is performance penalty incurred by synchronous logging which may lead to high performance overhead.
- Implementations of pessimistic logging must use special techniques to reduce the effects of synchronous logging on the performance.
- This overhead can be lowered using special hardware.
- Magnetic disk devices and a special bus to guarantee atomic logging of all messages exchanged in the system can mitigate this overhead.

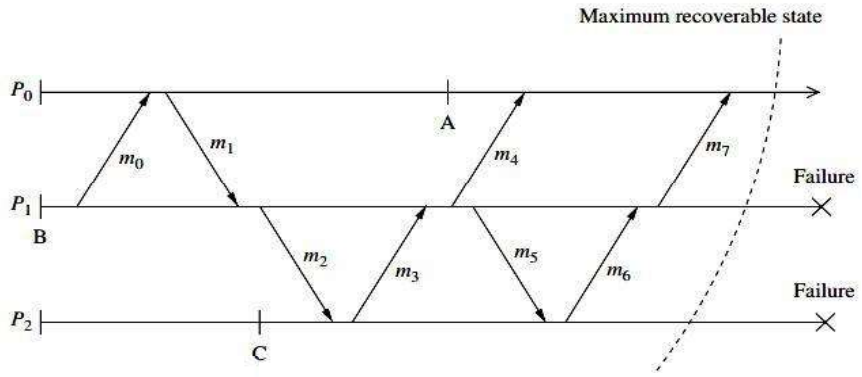


Fig 4.6: Pessimistic logging

- Suppose processes P and P 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

- Once the recovery is complete, both processes will be consistent with the state of P0 that includes the receipt of message m from P.

4.4.2Optimistic logging

- In these protocols, processes log determinants asynchronously to the stable storage.
- Optimistically assume that logging will be complete before a failure occurs.
- This do not implement the always-no-orphans condition.
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution.
- Optimistic logging protocols require a non-trivial garbage collection 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.
- The overheads in optimistic logging are 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.
- 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.

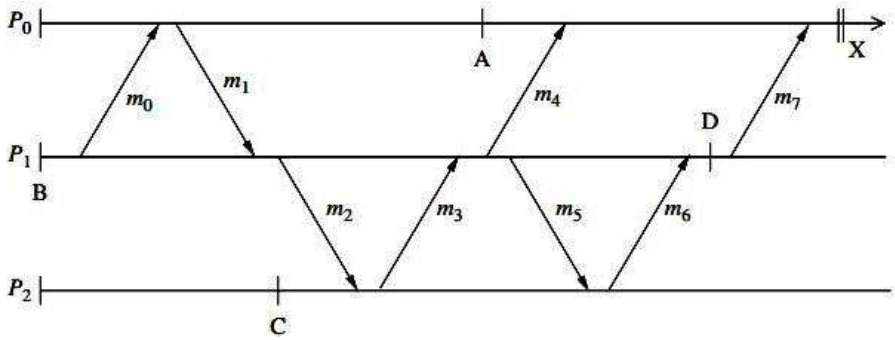


Fig 4.7: Optimistic logging

- Upon a failure, the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan.
- Optimistic logging protocols require a non-trivial garbage collection scheme.

- The pessimistic protocols track only the most recent checkpoint of each process, whereas optimistic protocols may need to keep multiple checkpoints for each process.
- Suppose process P2 fails before the determinant for m5 is logged to the stable storage. Process P1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message m6.
- The rollback of P1 further forces P0 to roll back to undo the effects of receiving message m7.

4.4.3 Casual Logging

- This 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.
- Make sure that the always-no-orphans property holds.
- Each process maintains information about all the events that have causally affected its state.
- Causal logging protocols make sure that the always-no-orphans property holds by ensuring that the determinant of each non-deterministic event that causally precedes the state of a process is either stable or it is available locally to that process.
- Messages m5 and m6 are likely to be lost on the failures of P1 and P2 at the indicated instants.
- Process P0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's happened - before relation.
- These events consist of the delivery of messages m0, m1, m2, m3, and m4.
- The determinant of each of these non-deterministic events is either logged on the stable storage or is available in the volatile log of process P0.
- The process P0 will be able to guide the recovery of P1 and P2 since it knows the order in which P1 should replay messages m1 and m3 to reach the state from which P1 sent message m4.
- P0 has the order in which P2 should replay message m2 to be consistent with both P0 and P1.

- The content of these messages is obtained from the sender log of P0 or regenerated deterministically during the recovery of P1 and P2.
- The information about messages m5 and m6 is lost due to failures. These messages may be resent after recovery possibly in a different order.
- Each process maintains information about all the events that have causally affected its state.
- This information protects it from the failures of other processes and also allows the process to make its state recoverable by simply logging the information available locally.
- Thus, a process does not need to run a multi-host protocol to commit output. It can commit output independently.

4.5 COORDINATED CHECKPOINTING ALGORITHM (KOO-TOUEG)

- Uncoordinated checkpointing may lead to domino effect or to livelock .
- Two basic approaches to checkpoint coordination:
  1. The Koo-Toueg algorithm: process to initiates the system-wide checkpointing process
  2. Staggering checkpoints: An algorithm which staggers checkpoints in time; Staggering checkpoints can help avoid near-simultaneous heavy loading of the disk system
- The communication is induced by checkpointing procedures.
- But the uncoordinated checkpointing algorithms can deal the isolated failures.

> ***A coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and avoids domino effect and livelock problems during the recovery.***

- This algorithm includes 2 parts: the checkpointing algorithm and the recovery algorithm.

4.5.1 Checkpointing algorithm

- The following are the assumptions made in checkpointing algorithm:
  - FIFO channel
  - end-to-end protocols
  - communication failures do not partition the network

- single process initiation
- no process failures during the execution of the algorithm

- The algorithm facilitates two kinds of checkpoints:
  - **Permanent:** local checkpoint, part of a consistent global checkpoint
  - **Tentative:** temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully
- The algorithm is implemented in two phases:

  Phase I: Initiation

  - This process takes a tentative checkpoint and requests all other processes to take tentative checkpoints.
  - Every process cannot send messages after taking tentative checkpoint.
  - All processes will finally have the single same decision: do or discard.
  - A process says no to a request if it fails to take a tentative checkpoint, which could be due to several reasons, depending upon the underlying application.
  - If Pi learns that all the processes have successfully taken tentative checkpoints, Pi decides that all tentative checkpoints should be made permanent; otherwise, Pi decides that all the tentative checkpoints should be discarded.

  Phase II:

  - All processes will receive the final decision from initiating process and act accordingly.
  - Pi informs all the processes of the decision it reached at the end of the first phase.
  - A process, on receiving the message from Pi, will act accordingly.
  - Either all or none of the processes advance the checkpoint by taking permanent checkpoints.
  - The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the underlying computation until it is informed of Pi's decision.

Correctness of the algorithm

- Either all or none of the processes take permanent checkpoint

- No process sends message after taking permanent checkpoint. The optimization may be not all of the processes need to take checkpoints (if not change since the last checkpoint).

4.5.2 Rollback recovery algorithm

- This 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.
- This is implemented in two phases:
  - The initiating process send a message to all other processes and ask for the preferences – restarting to the previous checkpoints. All need to agree about either do or not.
  - The initiating process send the final decision to all processes, all the processes act accordingly after receiving the final decision.

Phase I:

- An initiating process Pi 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.
- If Pi learns that all processes are willing to restart from their previous checkpoints, Pi decides that all processes should roll back to their previous checkpoints.
- Otherwise, Pi aborts the rollback attempt and it may attempt a recovery at a later time.

Phase II:

- Pi propagates its decision to all the processes.
- On receiving Pi'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 Pi's decision.

Correctness

All processes restart from an appropriate state because, if they decide to restart, they resume execution from a consistent state.

Optimization:

This may not to recover all, since some of the processes did not change anything.

4.6 ALGORITHM FOR ASYNCHRONOUS CHECKPOINTING AND RECOVERY (JUANG-VENKATESAN)

- This algorithm helps in recovery in asynchronous checkpointing.
- The following are the assumptions made:
  - communication channels are reliable
  - delivery messages in FIFO order
  - infinite buffers
  - message transmission delay is arbitrary but finite
- The underlying computation or application is event-driven: When process P is at states, receives message m, it processes the message; moves to state s' and send messages out. So the triplet (s, m, msgs_sent) represents the state of P.
- To facilitate recovery after a process failure and restore the system to a consistent state, two types of log storage are maintained:
  - **Volatile log:** It takes short time to access but lost if processor crash. The contents of volatile log are moved to stable log periodically.
  - **Stable log:** longer time to access but remained if crashed.

4.6.1 Asynchronous checkpointing

- After executing an event, a processor records a triplet (s, m, msg_sent) in its volatile storage.
  - s:state of the processor before the event
  - m: message
  - msgs_sent: set of messages that were sent by the processor during the event.
- A local checkpoint at a processor consists of the record of an event occurring at the processor and it is taken without any synchronization with other processors.
- Periodically, a processor independently saves the contents of the volatile log in the stable storage and clears the volatile log.
- This operation is equivalent to taking a local checkpoint.

4.6.2 Recovery Algorithm

The data structures followed in the algorithm are:

$$RCVD_{i \leftarrow j}(CkPt_i)$$

This represents the number of messages received by processor pi from processor pj, from the beginning of the computation until the checkpoint $CkPt_i$.

$$SENT_{i \to j}(CkPt_i)$$

This represents the number of messages sent by processor pi to processor pj, from the beginning of the computation until the checkpoint $CkPt_i$.

- The main idea of the algorithm is to find a set of consistent checkpoints, from the set of checkpoints.
- This is done based on the number of messages sent and received.
- Recovery may involve multiple iterations of roll backs by processors.
- Whenever a processor rolls back, it is necessary for all other processors to find out if any message sent by the rolled back processor has become an orphan message.
- The orphan messages are identified by comparing the number of messages sent to and received from neighboring processors.
- When a processor restarts after a failure, it broadcasts a ROLLBACK message that it has failed.
- The recovery algorithm at a processor is initiated when it restarts after a failure or when it learns of a failure at another processor.
- Because of the broadcast of ROLLBACK messages, the recovery algorithm is initiated at all processors.

Procedure RollBack_Recovery: processor pi executes the following:

STEP (a)

if processor pi is recovering after a failure **then**

$C_k\ Pt_i$:= latest event logged in the stable storage

else

$C_k\ Pt_i$:= latest event that look place in p_i {The latest event at pi 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 $SENT_{i \to j}(C_k\ Pt_i)$

send a ROLLBACK(i, $SENT_{i\rightarrow j}$ (C_k Pt_i)) message to p_j

end for

for every ROLLBACK(j,c) message received from a neighbor j **do**

if RCVD $_{i\rightarrow j}$ (C_k Pt_i) > c {Implies the presence of orphan message}

then

find the latest event e such that RCVD $_{i\rightarrow j}$ (e) = c {Such an event e may be in the volatile storage or stable storage}

C_k Pt_i := e

end if

end for

end for {for k}

Fig 4.8: Algorithm for Asynchronous Checkpointing and Recovery (Juang- Venkatesan)

- The rollback starts at the failed processor and slowly diffuses into the entire system through ROLLBACK messages.
- During the kth iteration (k != 1), a processor pi does the following:

  (i) based on the state $CkPt_i$ it was rolled back in the (k − 1)th iteration, it computes $SENT_{i\rightarrow j}$ ($CkPt_i$) for each neighbor pj and sends this value in a ROLLBACK message to that neighbor

  (ii) p_i waits for and processes ROLLBACK messages that it receives from its neighbors in kth iteration and determines a new recovery point $CkPt_i$ for p_i based on information in these messages.

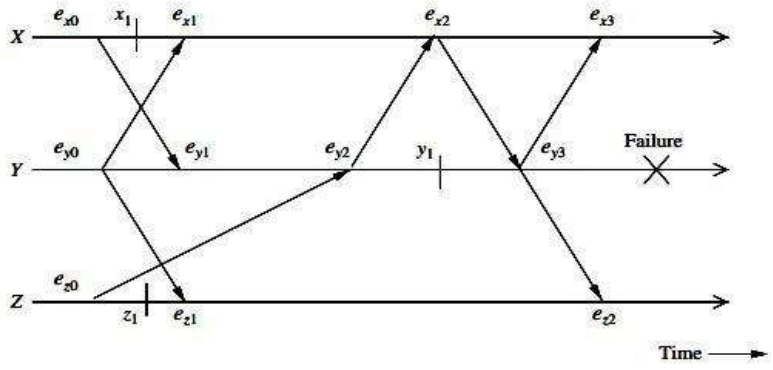


Fig 4.9: Asynchronous Checkpointing And Recovery

- At the end of each iteration, at least one processor will rollback to its final recovery point, unless the current recovery points are already consistent.

4.7 CONSENSUS AND AGREEMENT

A consensus algorithm is a process that achieves agreement on a single data value among distributed processes or systems.

- Consensus algorithms necessarily assume that some processes and systems will be unavailable and that some communications will be lost.
- Hence these algorithms must be fault-tolerant.

Examples of consensus algorithm:

- Deciding whether to commit a distributed transaction to a database.
- Designating node as a leader for some distributed task.
- Synchronizing state machine replicas and ensuring consistency among them.

Assumptions in Consensus algorithms

- **Failure models:**
  - Some of the processes may be faulty in distributed systems.
  - A faulty process can behave in any manner allowed by the failure model assumed.
  - Some of the well known failure models includes fail-stop, send omission and receive omission, and Byzantine failures.
  - **Fail stop model:** a process may crash in the middle of a step, which could be the execution of a local operation or processing of a message for a send or receive event. it may send a message to only a subset of the destination set before crashing.
  - **Byzantine failure model:** a process may behave arbitrarily.
  - The choice of the failure model determines the feasibility and complexity of solving consensus.
- **Synchronous/asynchronous communication:**
  - If a failure-prone process chooses to send a message to process but fails, then intended process cannot detect the non-arrival of the message.
  - This is because scenario is indistinguishable from the scenario in which the message takes a very long time in transit. This is a major hurdle in asynchronous system.

- In a synchronous system, a unsent message scenario can be identified by the intended recipient, at the end of the round.
- The intended recipient can deal with the non-arrival of the expected message by assuming the arrival of a message containing some default data, and then proceeding with the next round of the algorithm.

- **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.
  - When multiple messages are expected from the same sender in a single round, a scheduling algorithm is employed that sends these messages in sub-rounds, so that each message sent within the round can be uniquely identified.

- **Channel reliability:**
  - The channels are reliable, and only the processes may fail.

- Authenticated vs. non-authenticated messages:
  - With unauthenticated messages, 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,
    (ii) it can also tamper with the contents of a received message before relaying it.
  - When a process receives a message, it has no way to verify it sauthenticity. This is known as **un authenticated message or oral message or an unsigned message.**
  - Using authentication via techniques such as digital signatures, it is easier to solve the agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering.
  - Thus, faulty processes can inflict less damage.

- **Agreement variable:**
  - The agreement variable may be boolean or multivalued, and need not be an integer.
  - This simplifying assumption does not affect the results for other data types, but helps in the abstraction while presenting the algorithms.

4.7.1 Byzantine General problem

- The Byzantine Generals' Problem (BGP) is a classic problem faced by any distributed computer system network.
- Imagine that the grand Eastern Roman empire aka Byzantine empire has decided to capture a city.
- There is fierce resistance from within the city.
- The Byzantine army has completely encircled the city.
- The army has many divisions and each division has a general.
- The generals communicate between each as well as between all lieutenants within their division only through messengers.
- All the generals or commanders have to agree upon one of the two plans of action.
- Exact time to attack all at once or if faced by fierce resistance then the time to retreat all at once. The army cannot hold on forever.
- If the attack or retreat is without full strength then it means only one thing— Unacceptable brutal defeat.
- If all generals and/or messengers were trustworthy then it is a very simple solution.
- However, some of the messengers and even a few generals/commanders are traitors. They are spies or even enemy soldiers.
- There is a very high chance that they will not follow orders or pass on the incorrect message. The level of trust in the army is very less.
- Consider just a case of 1 commander and 2 Lieutenants and just 2 types of messages- 'Attack' and 'Retreat'.

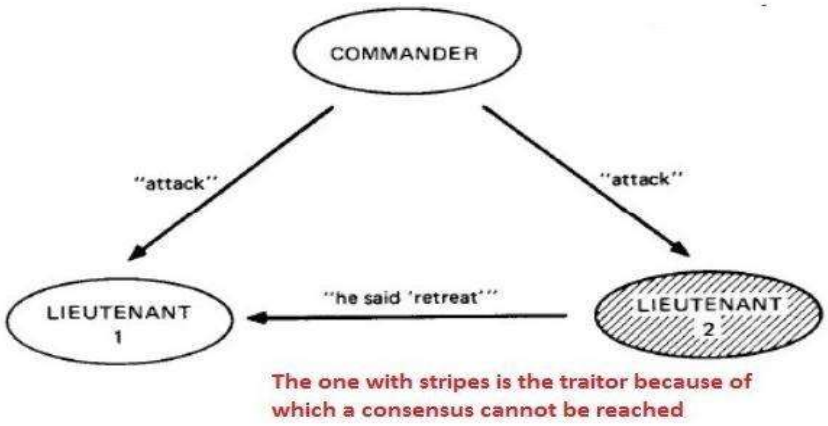


Fig 4.10: BGP algorithm

- In Fig 4.10, the Lieutenant 2 is a traitor who purposely changes the message that is to be passed to Lieutenant 1.
- Now Lieutenant 1 has received 2 messages and does not know which one to follow. Assuming Lieutenant 1 follows the Commander because of strict hierarchy in the army.
- Still, 1/3rd of the army is weaker by force as Lieutenant 2 is a traitor and this creates a lot of confusion.
- However what if the Commander is a traitor (as explained in Fig 4.11). Now 2/3rd of the total army has followed the incorrect order and failure is certain.

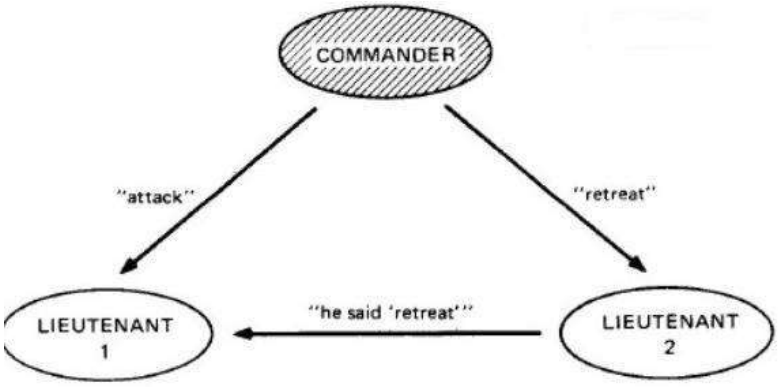


Fig 4.11: BGP algorithm

- After adding 1 more Lieutenant and 1 more type of message (Let's say the 3rd message is 'Not sure'), the complexity of finding a consensus between all the Lieutenants and the Commander is increased.
- Now imagine the exponential increase when there are hundreds of Lieutenants.

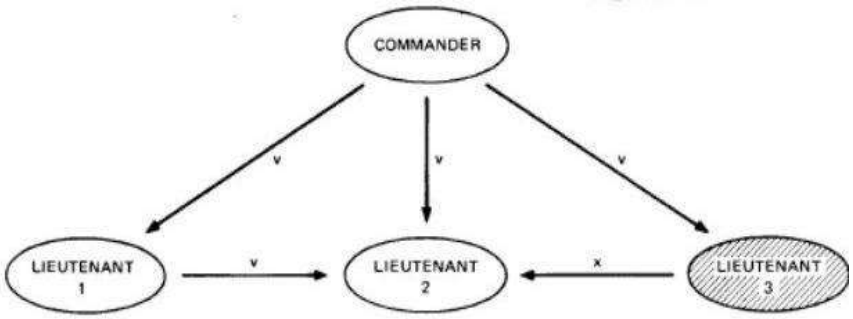


Fig 4.12: Adding one more lieutenant

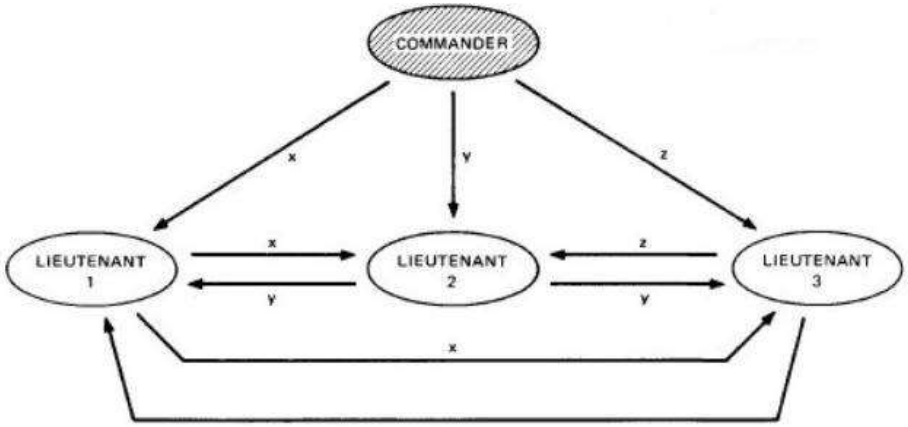


Fig 4.13: 1 commandant, 3 lieutenant and 3 types of messages

- This is BGP. It is applicable to every distributed network. All participants or nodes ('Lieutenant') are exactly of equal hierarchy. If agreement is reachable, then protocols to reach it need to bedevised.
- All participating nodes have to agree upon every message that is transmitted between the nodes.
- If a group of nodes is corrupt or the message that they transmit is corrupt then still the network as a whole should not be affected by it and should resist this 'Attack'.
- The network in its entirety has to agree upon every message transmitted in the network. This agreement is called as **consensus.**

The Byzantine agreement problem requires a designated source process, with an initial value, to reach agreement with the other processes about its initial value, subject to:

- **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.

There are two other versions of the Byzantine agreement problem:

- Consensus problem
- Interactive consistency problem.

- A correct process is a process that does not exhibit a Byzantine behaviour.
- A process is Byzantine if, during its execution, one of the following faults occurs:
  - **Crash:** The process stops executing statements of its program and halts.
  - **Corruption**: The process changes arbitrarily the value of a local variable with respect to its program specification. This fault could be propagated to other processes by including incorrect values in the content of a message sent by the process.
  - **Omission:** The process omits to execute a statement of its program. If a process omits to execute an assignment, this could lead to a corruption fault.
  - **Duplication:** The process executes more than one time a statement of its program. If a process executes an assignment more than one time, this could lead to a corruption fault.
  - **Misevaluation:** The process misevaluates an expression included in its program. This fault is different from a corruption fault: misevaluating an expression does not imply the update of the variables involved in the expression and, in some cases the result of an evaluation is not as-signed to a variable.

4.7.2 Consensus Problem

All the process has an initial value and all the correct processes must agree on single value. This is consensus problem.

Consensus is a fundamental paradigm for fault-tolerant asynchronous distributed systems. Each process proposes a value to the others. All correct processes have to agree (Termination) on the same value (Agreement) which must be one of the initially proposed values (Validity).

The requirements of the consensus problem are:

- **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.

4.7.3 Interactive Consistency Problem

All the process has an initial value, and all the correct processes must agree upon a set of values, with one value for each process. This is interactive consistency problem.

The formal specifications are:

- **Agreement:** All non-faulty processes must agree on the same array of values A [v1, …,vn].
- **Validity:** If process i is non-faulty and its initial value is vi, then all nonfaulty processes agree on vi as the ith element of the array A. If process jis 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.

The difference between the agreement problem and the consensus problem is that, in the agreement problem, a single process has the initial value, whereas in the consensus problem, all processes have an initial value.

4.8 RESULTS OF CONSENSUS PROBLEM

Some important facts to remember are:

- Consensus is not solvable in asynchronous systems even if one process can fail by crashing.
- Consensus is attainable for no failure case.
- In a synchronous system, common knowledge of the consensus value is also attainable.
- In asynchronous case, concurrent common knowledge of the consensus value is attainable.

- The results are tabulated below. f indicates the number of processes that can fail and n indicates the total number of processes.

S.No	Failure Mode	Synchronous System	Asynchronous System
1.	No failure	Agreement is attainable. Common knowledge is also attainable.	Agreement is attainable. Concurrent common knowledge is also attainable.
2.	Crash failure	Agreement is attainable. f < n process Ω(f+1) rounds	Agreement is not attainable.
3.	Byzantine (malicious) failure	Agreement is attainable. f<= floor((n-1)/3) Byzantine process Ω(f+1) rounds	Agreement is not attainable.

Solvable variants of agreement problem

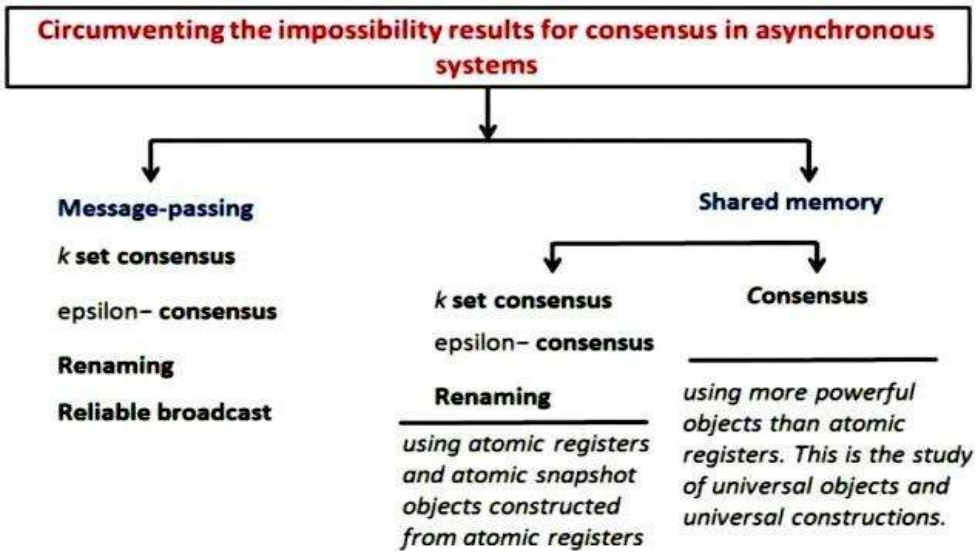


Fig 4.14: Circumventing the impossibility results

A synchronous message passing system and a shared memory system can be used solve the consensus problem. The following are the weaker consensus problem in asynchronous system:

- **Terminating reliable broadcast:** A correct process will always get a message even if the sender crashes while sending. If the sender crashes while sending the message, the message may be even null, but still it has to be delivered to the correct process.

- **K-set consensus:** It is solvable as long as the number of crashes is less than the parameter k, which indicates the non-faulty processes that agree on different values, as long as the size of the set of values agreed upon is bounded by k.

- **Approximate agreement:** The consensus value is from multi valued domain. The agreed upon values by the non-faulty processes be within ε of each other.

- **Renaming problem:** It requires the processes to agree on necessarily distinct values.

- **Reliable broadcast**: A weaker version of reliable terminating broadcast (RTB), is the one in which the terminating condition is dropped and is solvable under crash failures.

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 at most k
ϵ-agreement	Crash failures, $n \geq 5f + 1$ (MP)	Values agreed upon are within ϵ of each other
Renaming	Up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash failures, $f \leq n - 1$ (SM)	Select a unique name from a set of names

Fig 4.15: Solvable variants of agreement problem in asynchronous system

4.9 AGREEMENT IN A FAILURE-FREE SYSTEM

- In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a decision, and distributing this decision in the system.

- A distributed mechanism would have each process broadcast its values to others, and each process computes the same function on the values received.

- The decision can be reached by using an application specific function.

- Algorithms to collect the initial values and then distribute the decision may be based on the token circulation on a logical ring, or the three-phase tree-based broadcast converge cast: broadcast, or direct communication with all nodes.

- In a synchronous system, this can be done simply in a constant number of rounds.
- Further, common knowledge of the decision value can be obtained using an additional round.
- In an asynchronous system, consensus can similarly be reached in a constant number of message hops.
- Further, concurrent common knowledge of the consensus value can also be attained.

4.10 AGREEMENT IN (MESSAGE-PASSING) SYNCHRONOUS SYSTEMS WITH FAILURES

4.10.1 Consensus algorithm for crash failures (synchronous system)

- Consensus algorithm for crash failures message passing synchronous system.
- The consensus algorithm for n processes where up to f processes where $f < n$ may fail in a fail stop failure model.
- Here the consensus variable x is integer value; each process has initial value xi. If up to f failures are to be tolerated than algorithm has f+1 rounds, in each round a process i sense the value of its variable xi to all other processes if that value has not been sent before.
- So, of all the values received within that round and its own value xi at that start of the round the process takes minimum and updates xi occur f + 1 rounds the local value xi guaranteed to be the consensus value.
- In one process is faulty, among three processes then f = 1. So the agreement requires f + 1 that is equal to two rounds.
- If it is faulty let us say it will send 0 to 1 process and 1 to another process i, j and k. Now, on receiving one on receiving 0 it will broadcast 0 over here and this particular process on receiving 1 it will broadcast 1 over here.
- So, this will complete one round in this one round and this particular process on receiving 1 it will send 1 over here and this on the receiving 0 it will send 0 over here.

(global constants)

integer: f; // maximum number of crash failures tolerated

(local variables)

Integer: x ← local value;

(1) Process P_i ($1 \leq i \leq n$) execute 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) $y_i \leftarrow$ value (if any) received from process j in this round;

(1e) $x \leftarrow \min_{\forall j}(x, y_j)$;

(1f) **output** x as the consensus vaule.

Fig 4.16: Consensus with up to f fail-stop processes in a system of n processes, n > f

- The agreement condition is satisfied because in the f+ 1 rounds, there must be at least one round in which no process failed.
- In this round, say round r, all the processes that have not failed so far succeed in broadcasting their values, and all these processes take the minimum of the values broadcast and received in that round.
- Thus, the local values at the end of the round are the same, say x^i_r for all non-failed processes.
- In further rounds, only this value may be sent by each process at most once, and no process i will update its value x^i_r.
- The validity condition is satisfied because processes do not send fictitious values in this failure model.
- For all i, if the initial value is identical, then the only value sent by any process is the value that has been agreed upon as per the agreement condition.
- The termination condition is seen to be satisfied.

Complexity

- The complexity of this particular algorithm is it requires f + 1 rounds where f < n and the number of messages is $O(n^2)$ in each round and each message has one integers hence the total number of messages is $O((f+1)\cdot n^2)$ is the total number of rounds and in each round n^2 messages are required.

Lower bound on the number of rounds

- At least f + 1 rounds are required, where f < n.
- In the worst-case scenario, one process may fail in each round; with f + 1 rounds, there is at least one round in which no process fails.

- In that guaranteed failure-free round, all messages broadcast can be delivered reliably, and all processes that have not failed can compute the common function of the received values to reach an agreement value.

4.10.2 Consensus algorithms for Byzantine failures (synchronous system)

Upper bound on Byzantine processes

- In a system of n processes, the Byzantine agreement problem can be solved in a synchronous system only if the number of Byzantine processes f is such that

$$f \leq \left\lceil \frac{\lceil n-1 \rceil}{3} \right\rceil$$

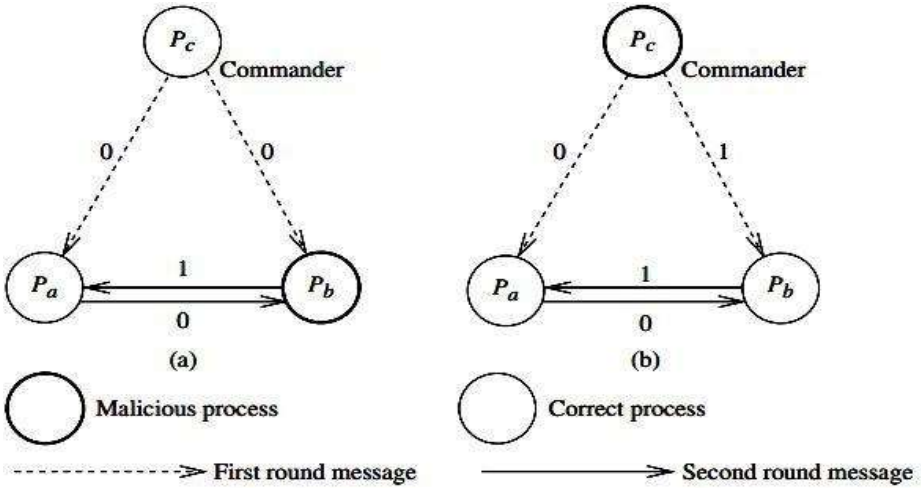


Fig 4.17: Impossibility of achieving Byzantine agreement with n = 3 processes and f = 1 malicious process

- The condition where f < (n – 1) / 2 is violated over here; that means, if f = 1 and n = 2 this particular assumption is violated
- (n – 1) / 2 is not 1 in that case, but we are assuming 1 so obviously, as per the previous condition agreement byzantine agreement is not possible.
- Here P 0 is faulty is non faulty and here P 0 is faulty so that means P 0 is the source, the source is faulty here in this case and source is non faulty in the other case.
- So, source is non faulty, but some other process is faulty let us say that P 2 is faulty. P 1 will send because it is non faulty same values to P 1 and P 2 and as far as the P 2s concerned it will send a different value because it is a faulty.
- Agreement is possible when f = 1 and the total number of processor is 4.

- So, agreement we can see how it is possible we can see about the commander P c.
- So, this is the source it will send the message 0 since it is faulty. It will send 0 to P d 0 to P b, but 1 to pa in the first column. So, P a after receiving this one it will send one to both the neighbors, similarly P b after receiving 0 it will send 0 since it is not faulty.
- Similarity P d will send after receiving 0 at both the ends.
- If we take these values which will be received here it is 1 and basically it is 0 and this is also 0.
- So, the majority is basically 0 here in this case here also if you see the values 10 and 0. The majority is 0 and here also majority is 0.
- In this particular case even if the source is faulty, it will reach to an agreement, reach an agreement and that value will be agreed upon value or agreement variable will be equal to 0.

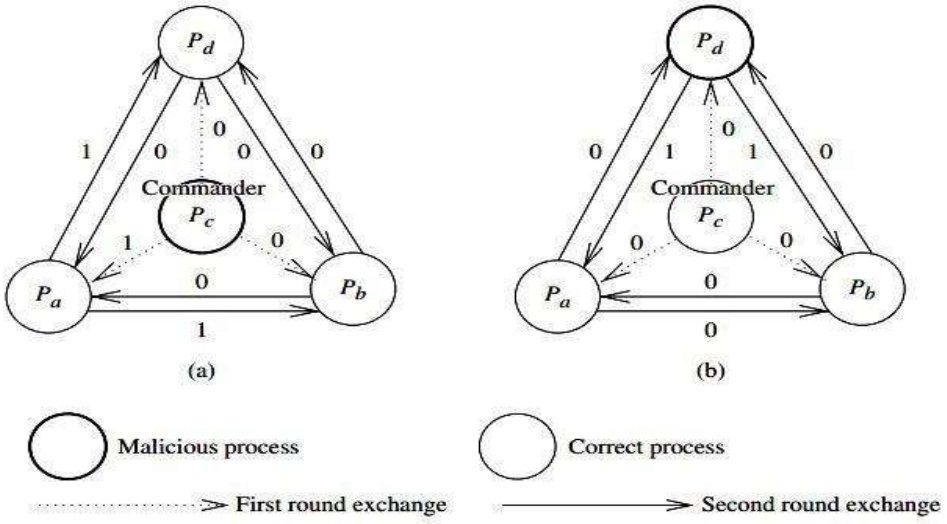


Fig 4.18: Achieving Byzantine agreement when n = 4 processes and f = 1

Malicious process

Byzantine agreement tree algorithm: exponential (synchronous system)- recursive formulation

- Consider the Fig 4.18. Commander Pc sends its value to the other three lieutenants.
- 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

  (ii) from the other two lieutenants in the second round.

- The majority gives a correct estimate of the commander's value.
- All three lieutenants take the majority of (1, 0, 0) which is "0," the agreement value.
- P_d is malicious. Lieutenants Pa and Pb agree on "0," the value of the commander.

(variable)

boolean : $\upsilon \leftarrow$ intial value;

integer: f $\leftarrow$ maximum number of malicious processes, $\leq [(n-1)/3]$;

(message type)

OM(υ, Dests, List, faulty) where

υ is a boolean,

Dests is a set of destination process i.d.s to which the message is sent.

List is a list of proess i.d.s 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), whre f > 0:

(1) The algorithm is initiated by the commander, who sends his source value υ to all other processes using a OM(υ, N, <i>, f) message. The commander return his own value υ and terminates.

(2) [**Recursion unfolding**:] For each message of the form OM(υ_j, Dests, List, f`) received in this round from some proess j the process i uses the value υ_j it receives from the source j and using that the 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 indicate oral_Msg(f` - 1), wherein it sends

OM(υj, Dests – {i}, concat<i>, L), (f`-1))

To destination not in concat<i>, L)

in the next round.

(3) [**Recursion folding:**] For each message of the form OM(υ_j, Dests, List, f`) received in step 2, each process i has computed the agreement vlaue υ_k for each k not in List and k ≠ 1, corresponding to the value received from pk after travesing 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 $majority_{k\neq1}$(υ_j, υ_k) as the agreement vlaue and return it to the next higher level in the recursive invocation.

Oral_Msg(o):

(1) [**Recursion unfolding:**] Process acts as a source and sends its value to each other process.

(2) [**Recursion folding:**] Each process uses the vlaue it receives from the other sources, and uses that value as the agreement value. If no value is received a default value is assumed.

Fig 4.19: Byzantine generals algorithm – exponential number of unsigned messages, n > 3f. Recursive formulation.

Lamport-Shostak-Pease Algorithm

This is also known as Oral Message algorithm of f faulty process (OM(f)) and n is the total number of processes (n>=3f+1). The algorithm is recursively defined as:

1. Source process sends its value to each other process.
2. Each process uses the value it receives from the source.

- The algorithm is recursive and the base of the recursion that is OM(0) says that the source process sends its values to each other process. Now each process uses its value, value it receives from the source if no value is received the default 0 is assumed.
- Each message has the following parameters:
    i) a consensus estimate value (v)
    ii) a set of destinations (Dests)
    iii) a list of nodes traversed by the message, from most recent to least recent (List)
    iv) the number of Byzantine processes that the algorithm still needs to tolerate (faulty).
- The commander invokes the algorithm with parameter faulty set to f, the maximum number of malicious processes to be tolerated.
- The algorithm uses f + 1 synchronous rounds. Each message (having this parameter faulty = k) received by a process invokes several other instances of the algorithm with parameter faulty = k − 1.

- The terminating case of the recursion is when the parameter faulty is 0.
- As the recursion folds, each process progressively computes the majority function over the values it used as a source for that level of invocation in the unfolding, and the values it has just computed as consensus values using the majority function for the lower level of invocations.
- There are an exponential number of messages O (n^f) used by this algorithm

Round number	A message has already visited	Aims to tolerate these many failures	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	$n-f-1$	$(n-1)(n-2)\ldots(n-f-1)$

Fig 4.20: Relationship between number of rounds and messages

- As multiple messages are received in any one round from each of the other processes, they can be distinguished using the List, or by using a scheduling algorithm within each round.
- Each process maintains a tree of boolean variables. The tree data structure at a non-initiator is used as follows:
  - There are f + 1 levels from level 0 through level f.
  - Level 0 has one root node, v_{init} , after round 1.
  - Level h, $0 < h \leq f$ has $1(n-2)(n-3)\ldots(n-h)(n-(h+1))$ nodes after round h + 1. Each node at level (h – 1) has (n –(h + 1)) child nodes.
  - Node v^L_k denotes the command received from the node head(L) by node k which forwards it to node i. The command was relayed to head(L) by head(tail(L)), which received it from head(tail(tail(L))) , and so on. The very last element of L is the commander, denoted $P_{init.}$
  - In the f + 1 rounds of the algorithm (lines 2a–2e of the iterative version),each level k, $0 \leq k \leq f$, of the tree is successively filled to remember the values received at the end of round k + 1, and with which the processs ends the multiple instances of the OM message with the fourth parameter

- as f- (k + 1) for round k + 2.
- For each message that arrives in a round, a process sets $vhead^{head(L}{}_{tail(L)})$.It then removes itself from Dests, prepends itself to L, decrements faulty, and forwards the value v to the updated Dests.
- Once the entire tree is filled from root to leaves, the actions in the folding of the recursion are simulated in lines 2f–2h of the iterative version, proceeding from the leaves up to the root of the tree. These actions are crucial – they entail taking the majority of the values at each level of the tree. The final value of the root is the agreement value, which will be the same at all processes.

(variables)

boolean : $\upsilon \leftarrow$ intial value;

integer: f $\leftarrow$ maximum number of malicious processes, $\leq [(n-1)/3]$;

tree of boolean:

- Level 0 root is υ^{L}_{init} where L = <>;
- Level h($f \geq h \geq 0$) nodes: for each υ^{L}_{j} at level h – 1 = sizeof(L), its n – 2 – sizeof(L) descendants at level h are $\upsilon^{concat(j),L}_{k}$, $\forall$ k such that $k \neq j, i$ and k is not a member of list L.

(message type)

OM(υ, Dests, List, faulty), where the parameter are as in the recursive formulation

(1) Initiator (i.e commander) initiates the oral Byzantine agreement:

(1a) send OM(υ, N – {i}, $<P_i>$, f) to N – {i};

(1b) return(υ).

(2) (non-indicator, i.e. lieutenant) receives the oral message (OM):

(2a) for rnd = 0 to f do

(2b) for each message OM that arrives in this round, do

(2c) receive OM(υ, Dests, $L = < P_{k_i}.....P_{k+1=faulty} >$), faulty from P_k;

// faulty + rnd = f; |Dests| + sizeof(L) = n

(2d) $\upsilon^{tail(L)}_{head(L)} \leftarrow \upsilon$; // sizeof(L) + faulty = f + 1. Fill in estimate.

(2e) send OM(υ, Dests – {i}, $< P_i, P_{k_i}.....P_{k_i+1-faulty} >$, faulty – 1)

to Dests – {i} if rnd < f;

(2f) **for** level = f – 1 **down** to 0 **do**

(2g) **for** each of the 1.(n – 2) … (n – (level+1)) nodes υ^L_x in level level, **do**

(2h) $\upsilon^L_x(x \neq i, x \notin L) = \text{majority}(\upsilon^{concat<x>,L}_y : y \notin concat<x>,L) : y \neq i)$;

Fig 4.21: Byzantine generals algorithm – exponential number of unsigned messages, n > 3f.Iterative formulation. Code for process Pi.

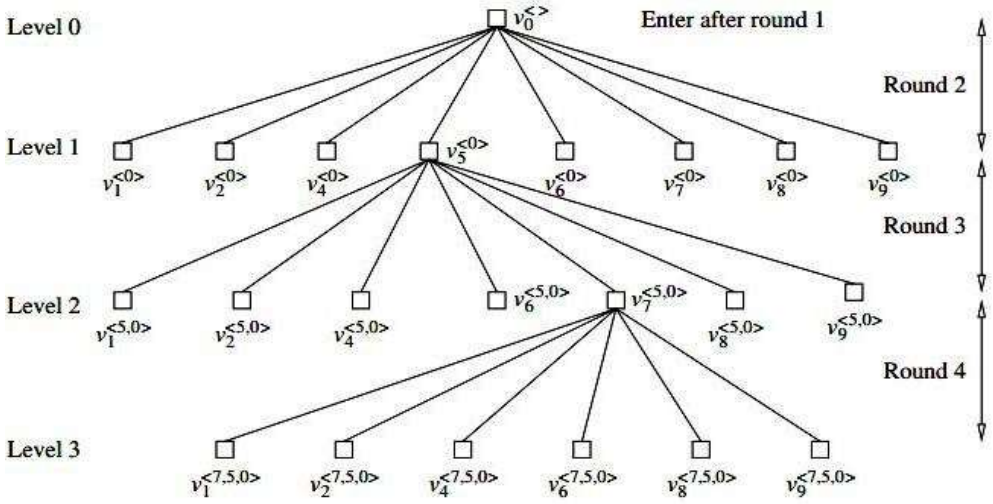


Fig 4.22: Local tree at P3for solving the Byzantine agreement, for n = 10 and f = 3. Only one branch of the tree is shown for simplicity

Correctness

The correctness of the Byzantine agreement algorithm can be observed from the following two informal inductive arguments. Here we assume that the Oral_Msg algorithm is invoked with parameter x, and that there are a total of f malicious processes. There are two cases depending on whether the commander is malicious. A malicious commander causes more chaos than an honest commander.

Phase-king algorithm for consensus: polynomial (synchronous system)

- The phase-king algorithm proposed by Berman and Garay solves the consensus problem under the same model, requiring f + 1 phases, and a polynomial number of messages but can tolerate only f < Ceil (n/4), malicious processes.

- The algorithm is so called because it operates in f + 1 phases, each with two rounds, and a unique process plays an asymmetrical role as a leader in each round.

(variables)

boolean : $\upsilon \leftarrow$ intial value;

integer: f $\leftarrow$ maximum number of malicious processes, f < [n/4];

(1) Each process executes the following f+1 phases, where f < n / 4:

(1a) **for** phase = 1 to f + 1 **do**

(1b) Execute the following round 1 action:

(1c) **broadcast** υ to all processes;

(1d) **await** value υ_j from each process P_j;

(1e) majority $\leftarrow$ value among the υ_j that occurs > n/2 times
(default value if no majority);

(1f) mult $\leftarrow$ number of times that majority occurs;

(1g) Execute the following round 2 actions:

(1h) **if** i=phase **then**

(1i) **broadcast** majority to all processes;

(1j) **receive** tiebreaker from P_{Phase} (default value if nothing is received);

(1k) **if** mult > n/2+f **then**

(1l) $\upsilon \leftarrow$ majority;

(1m) **else** $\upsilon \leftarrow$ tiebreaker;

(1n) **if** phase = f + 1 **then**

(1o) output decision value υ.

Fig 4.23: Phase-king algorithm polynomial number of unsigned messages, n > 4f. Code is for process Pi, $1 \leq i \leq n$

- In the first round of each phase, each process broadcasts its estimate of the consensus value to all other processes, and likewise awaits the values broadcast by others.
- At the end of the round, it counts the number of "1" votes and the number of "0" votes. If number is greater than n/2, then it sets its majority variable to that consensus value, and sets mult to the number of votes received for the majority value.
- If neither number is greater than n/2, which may happen when the malicious processes do not respond, and the correct processes are split among themselves, then a default value is used for the majority variable.

- In the second round (lines 1g–1o) of each phase, the phaseking initiates processing – the phase king for phase k is the process with identifier P_k, where $k \in \{1 \ldots n\}$.
- The phase king broadcasts its majority value majority, which serves the role of a tie-breaker vote for those other processes that have a value of mult of less than n/2 + f.
- Thus, when a process receives the tie-breaker from the phase king, it updates its estimate of the decision variable v to the value sent by the phase king if its own mult variable < n/2 + f.
- The reason for this is that among the votes for its own majority value, f votes could be bogus and hence it does not have a clear majority of votes (i.e., > n/2) from the non-malicious processes.
- Hence, it adopts the value of the phase king. However, if mult > n/2 + f(lines 1k–1l), then it has received a clear majority of votes from the nonmalicious processes, and hence it updates its estimate of the consensus variable v to its own majority value, irrespective of what tie-breaker value the phase king has sent in the second round.
- At the end of f + 1 phases, it is guaranteed that the estimate v of all the processes is the correct consensus value.

Correctness

The correctness reasoning is in three steps:

- Among the f + 1 phases, the phase king of some phase k is non-malicious because there are at most f malicious processes.
- As the phase king of phase k is non-malicious, all non-malicious processes can be seen to have the same estimate value v at the end of phase k.
- All non-malicious processes have the same consensus estimate x at the start of phase k+ 1 and they continue to have the same estimate at the end of phase k+ 1.

Complexity

The algorithm requires f + 1 phases with two sub-rounds in each phase, and (f + 1)[(n −1)(n +1)] message

5

P2P & DISTRIBUTED SHARED MEMORY

5.1 PEER TO PEER COMPUTING AND OVERLAY GRAPHS

Peer to peer (P2P) systems refers to the applications that take advantage of resources like storage, time cycles, content, manpower available at the end systems of the internet. In other words, the peer to peer computing architecture contains nodes that are equal participants in data sharing. All the tasks are equally divided between all the nodes. The nodes interact with each other as required as share resources. This deals with application layer organization of network overlay for flexibility of sharing resources.

The prominent feature of P2P networks is their ability to provide a large combined storage, CPU power, and other resources while imposing a low cost for scalability, and for entry into and exit from the network.

The ongoing entry and exit of various nodes, as well as dynamic insertion and deletion of objects in P2P network is called **churn.** The impact of churn should be as transparent as possible. There are two types of P2P systems: structured and unstructured.

Differences between structured and unstructured P2P

Unstructured P2P	**Structured P2P**
The construction of overlay network is highly flexible.	The construction of overlay network has low level of flexibility.
The resources are indexed locally.	The resources are distributed remotely and indexed using hash tables.
The messages can be broadcast or random walk.	The messages are unicast.
The network puts best effort content location.	The network guarantees the content location.

High overhead.	Comparatively low overhead.
This supports high failure rates.	Supports moderate failure rates.
This is suitable for small scale and highly dynamic applications.	This is suitable for large scale and relatively stable applications.

Overlay network is constructed over another network. For example, connecting internet over telephone lines is an overlay network. The topology of the overlay network is independent from the underlying network.

Characteristics of Peer to Peer Computing

- Peer to peer networks are usually formed by groups computers. These computers all store their data using individual security but also share data with all the other nodes.
- The nodes in peer to peer networks both use resources and provide resources. So, if the nodes increase, then the resource sharing capacity of the peer to peer network increases.
- The nodes in peer to peer networks act as both clients and servers. Hence, it is difficult to provide adequate security for the nodes. This can lead to denial of service attacks.
- Most modern operating systems such as Windows and Mac OS contain software to implement peer to peer networks.
- Efficient usage of resources.
- Self -organizing nature: because of scalable storage, CPU power and other resources.
- Distributed control: fast and efficient searching for data.
- Symmetric: highly scalable
- Anonymity: efficient management of churns
- Naming mechanism: selection of geographically close server.
- Security, authentication, trust: Redundancy in storage

Advantages of Peer to Peer Computing

- Each computer in the peer to peer network manages itself. The network is quite easy to set up and maintain.

- The server handles all the requests of the clients. This provision is not required in peer to peer computing and the cost of the server is saved.
- It is easy to scale the peer to peer network and add more nodes. This only increases the data sharing capacity of the system.
- None of the nodes in the peer to peer network are dependent on the others for their functioning. Hence the network is fault tolerant.
- Easy deployment and organization.

Disadvantages of Peer to Peer Computing

- It is difficult to back-up the data as it is stored in different computer systems and there is no central server.
- It is difficult to provide overall security in the peer to peer network as each system is independent and contains its own data.

5.1.1 Napster P2P system

- The developers of the original Napster launched the service as a peer-to-peer (P2P) file sharing network.
- The software application was easy to use with a free account, and it was specifically designed for sharing digital music files (in the MP3 format) across a Web-connected network.

Napster used a server-mediated, central index architecture organized around clusters of servers that store direct indices of the files in the system.

- The central server maintains a table with the:
  i) the client's address (IP) and port, and offered bandwidth
  ii) information about the files that the client can allow to share
- The basic steps of operation to search for content and to determine a node from which to download the content are the following:
  - A client connects to a meta-server that assigns a lightly loaded server from one of the close-by clusters of servers to process the client's query.
  - The client connects to the assigned server and forwards its query along with its own identity.
  - The server responds to the client with information about the users connected to it and the files they are sharing.

  - 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.

- 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

- The fundamental mechanism in P2P networks is data searching.
- This depends on the organization of data and networks.
- The search algorithms for P2P networks tend to be data-centric.
- P2P search uses the P2P overlay, a logical graph among the peers that is used for the object search and object storage and management algorithms.
- Overlays can be thought as a network built over another network.
- Above the P2P overlay is the application layer overlay, where communication between peers is point-to-point once a connection is established.
- The P2P overlay can be structured or unstructured, i.e., no particular graph structure is used.
- Structured overlays use some rigid organizational principles based on the properties of the P2P overlay graph structure, for the object storage algorithms and the object search algorithms.
- Unstructured overlays use very loose guidelines for object storage. As there is no definite structure to the overlay graph, the search mechanisms are more ad-hoc, and use some forms of flooding or random walk strategies.
- Thus, object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

5.2 DATA INDEXING AND OVERLAYS

Data stored in distributed systems are located through indexing mechanisms. There are three types of indexing:

1. **Centralized:** This indexing entails the use of one or a few central servers to store references or 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.
2. **Local:** This indexes to the objects at various peers being scattered across other peers throughout the P2P network. 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.

3. **Distributed**: This 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.

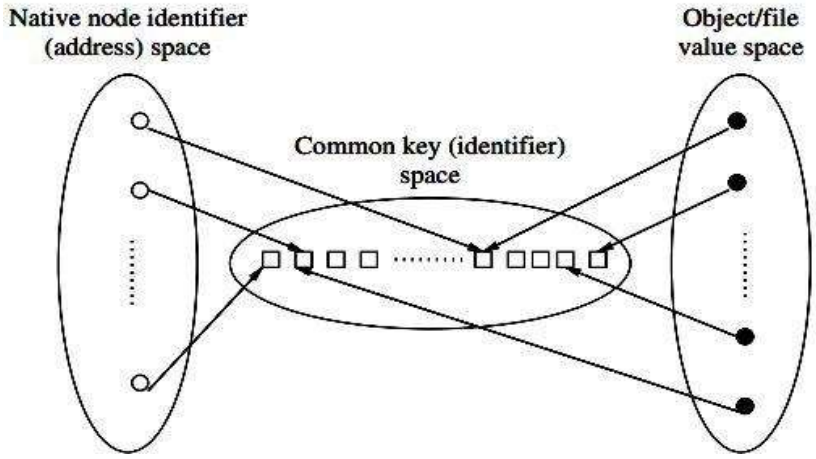


Fig 5.1: Mapping between address space and object space

Another criterion for classifying indexing mechanism are based on their semantic structure: semantic indexing and semantic free indexing.

- **Semantic indexing:** A semantic index is human readable. A semantic index mechanism supports keyword searches, range searches, and approximate searches.
- **Semantic free indexing:** This is not human readable and corresponds to the index obtained by a hash mechanism. These searches are not supported by semantic free index mechanisms.

5.2.1 Distributed Indexing

Structured Overlay

The P2P network topology has a definite structure, and the placement of files or data in this network is highly deterministic according to an algorithmic mapping.

- Deterministic mapping allows a very fast and deterministic lookup to satisfy queries for the data. These lookup systems use a hash table for the mapping.

- The hash function maps keys to values, along with the regular structure of the overlay. This facilitates fast search for the location of the file.
- An implicit characteristic of such a deterministic mapping of a file to a location is that the mapping can be based on a single characteristic of the file.
- The main drawback in this mapping is that arbitrary queries cannot be handled directly.
- Another notable limitation is the overhead occurred due to insertions and deletions of files in distributed environment.

Unstructured overlays

- This P2P network topology does not have any particular controlled structure.
- It do not have any control over where files or data is located.
- Each peer typically indexes only its local data objects, hence, it uses **local indexing.**
- Node joins and departures are easy since, the local overlay is simply adjusted.
- File placement is independent of the topology.
- But searching a file may incur high message overhead and high delays.
- The major advantage is that unstructured overlays supports complex queries because the search criteria can be arbitrary.
- The lack of fixed topology paves way for the formation of new topology.
- Some of the topologies are:

Power law random graph (PLRG): This is a random graph where the node degrees follow the power law

Normal random graph: This is a normal random graph where the nodes typically have a uniform degree.

Differences between structured and unstructured overlay networks

Structured overlay	**Unstructured overlay**
The networks are constructed over a predetermined topology.	There is no specific topology.
The connections are also predetermined.	Random and dynamic connections can be established.

The insertion and deletion of nodes imposes high overhead over the network performance.	They offer better resilience to network dynamics. Insertion and deletions of nodes is simpler.
This offers faster response time, better performance and lower diameter.	This has comparatively worse performance, node reachability, response time and no guarantee for the diameter.
They are more scalable.	They lack scalability.
They do not support arbitrary searches.	They support arbitrary searches.
Vulnerable to attacks.	Resilience to attacks.
EG: Chord	EG: Gnutella

5.3 CHORD

- Chord is a protocol proposed by Stoica that associates mapping of key/ value pairs in distributed environment using a hash table.
- A **distributed hash table** (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value) pairs are stored in a DHT, and any participating node can retrieve the value associated with a given key.
- Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption.
- This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

Properties of DHT:

- **Autonomy and decentralization:** the nodes collectively form the system without any central coordination.
- **Fault tolerance:** the system should be reliable even with nodes continuously joining, leaving, and failing.
- **Scalability:** the system should function efficiently even with thousands or millions of nodes.

CHORD

Chord uses a flat key space to associate the mapping between network nodes and data objects/files/values.

- Chord is a protocol and algorithm for a peer-to-peer distributed hash table.
- A distributed hash table stores key-value pairs by assigning keys to different computers (known as "nodes"); a node will store the values for all the keys for which it is responsible.
- Chord specifies how keys are assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key.
- 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.
- Chord supports a single operation, lookup(x), that maps a given key x to a network node.
- Chord stores a file/object/value at the node to which the file/object/value's key maps.
- The steps involved in lookup are:
    1. Map the object/file/value to its key in the common address space.
    2. Map the key to the node in its native address space using lookup. The design of lookup is the main challenge.

Basic Querying

- The Chord protocol is used to query a key from a client i.e. to find successor(k).
- The basic approach is to pass the query to a node's successor, if it cannot find the key locally.
- This will lead to a O(N) query time where N is the number of machines in the ring.
- To avoid the linear search above, Chord implements a faster search method by requiring each node to keep a **finger table** containing up to m entries, where m is the number of bits in the hash key.
- In Chord, a node's IP address is hashed to an m-bit identifier that serves as the node identifier in the common key (identifier) space.
- The file/data key is hashed to an m-bit identifier that serves as the key identifier.
- The value of m is sufficiently large so that the probability of collisions during the hash is negligible.
- The Chord overlay uses a logical ring of size 2^m.
- The identifier space is ordered on the logical ring modulo 2^m.
- A key k gets assigned to the first node such that its node identifier equals or follows the key identifier of k in the common identifier space.

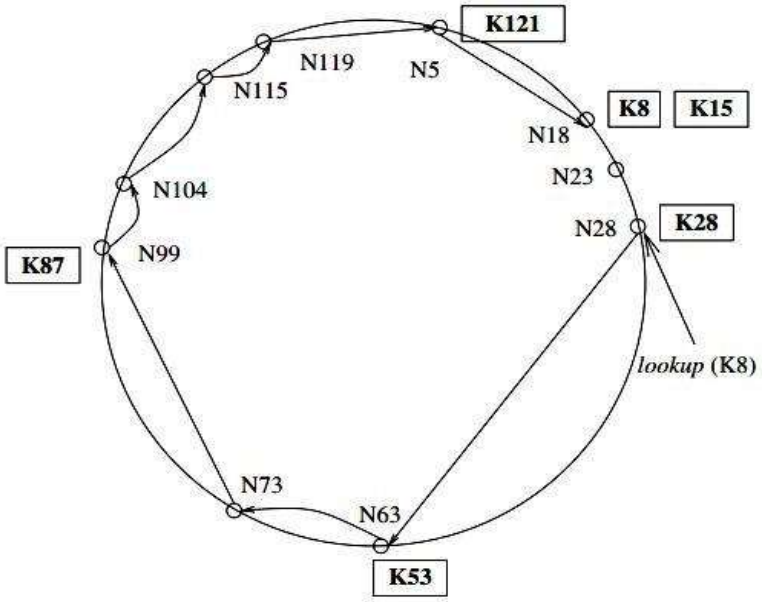


Fig 5.1: Chord ring with m=7

- The node is the successor of k denoted as succ(k).
- N denoted the nodes and K denoted the keys stored by the nodes.

  Succ(8)=18

  Succ(15)=18

  Succ(28)=28

  Succ(53)=63

  Succ(87)=99

  Succ(121)=5

Simple lookup

- 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, where n is the number of nodes in the P2P network.

(variables)

Integer: successor ← initial value;

(1) i.Locate_Successor(key), where key≠i:

(1a) if key ∈ (i, successor] then

(1b) return(successor)

(1c) else return (successor.Locate_Successor(key)).

Fig 5.2: Simple object lookup algorithm

Scalable Lookup

- A scalable look up algorithm that uses O(log n) message hops at the cost of O(m) space in the local routing tables, uses the following idea.
- Each node i maintains a routing table, called the finger table, with at most O(log n) distinct 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 the first node whose key is greater than the key of node i by at least $2x-1$ mod 2^m.
- The size of the finger table is bounded by m entries.
- Due to the logarithmic structure, the finger table has more information about nodes closer ahead of it in the Chord overlay, than about nodes further away.

(variables)

Integer: successor ← initial value;

Integer: predecessor ← initial value;

Integer finger[1…..m];

(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]).

Fig 5.3: A scalable object location algorithm

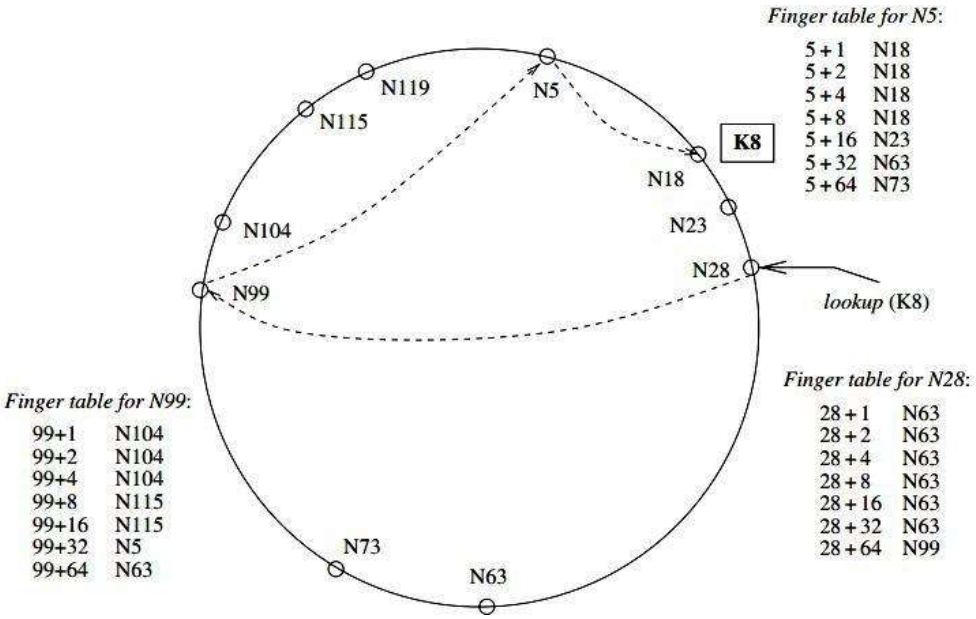


Fig 5.4: Query lookup using the logarithmically-structured finger tables

- For a query on key key at node i, if key lies between i and its successor, the key would reside at the successor and the successor's 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.

Managing Churn

The behavior of the protocol during node joins, failures and departures is discussed here.

i) Node Join

- 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.
- A recent joiner node i that has executed Join_Ring() gets integrated into the ring is shown in Fig 5.5.

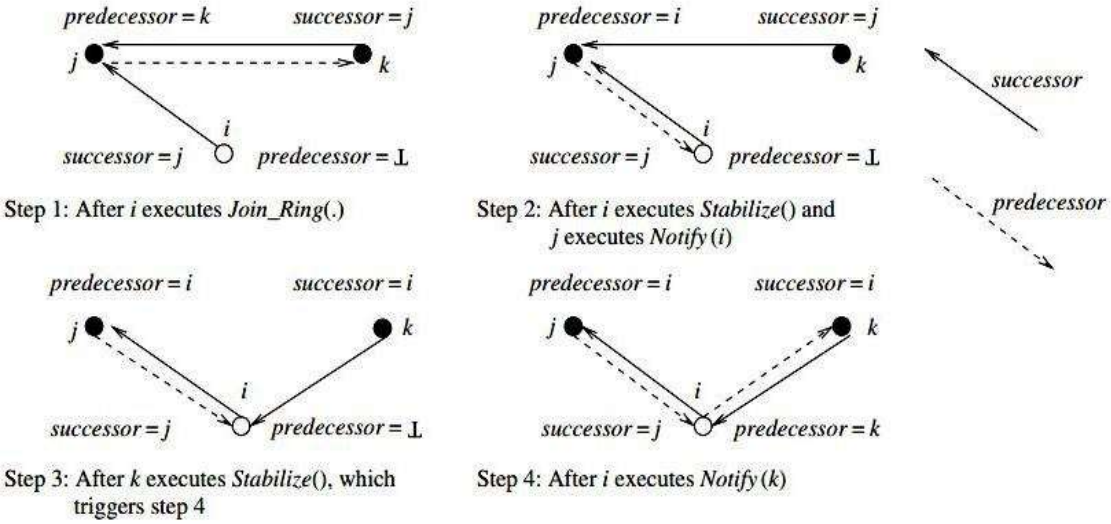


Fig 5.5: Steps in the integration of node i in the ring, where j > i > k

The following are the sequence:

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 notify 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.

- Once all the successor variables and finger tables have stabilized, a call by any node to Locate_Successor() will reflect the new joiner i.
- Until then, a call to Locate_Successor() may result in the Locate_Successor() call performing a conservative scan.
- The loop in Closest_Preceding_Node that scans the finger table will result in a search traversal using smaller hops rather than truly logarithmic hops, resulting in some
- It can be shown that for any set of concurrent join operations, at some point after the last join operation completes, all the pointers and finger tables will be correct. This ensures the correctness.
- But the transit period can have the following outcomes:

1. The finger tables used in a search are up to date and the correct successor of the key is sought in O(log n) hops.
2. The finger tables are not up to date but the successor pointers are correct. The sought key will be located but may take more steps as the full advantage of a logarithmic search space pruning cannot be used.
3. If the successor pointers are incorrect, or the key transfer to the new joiners in procedure Notify has not completed, the search may fail. This is during a transient duration, and the source has the choice of reissuing the query.

ii) Node failures and departures

- When a node j fails abruptly, its successor i on the ring will discover the failure when the successor i executes Check_Predecessor() periodically.
- Process i gets a chance to update its predecessor field when another node k causes i to execute Notify(k). But that can happen only if k's successor variable is i.
- This requires the predecessor of the failed node to recognize that its successor has failed, and get a new functioning successor.
- In fact, the successor pointers are required for object search; the predecessor variables are required only to accommodate new joiners.
- A solution such as introducing a Check_Successor() procedure analogous to Check_Predecessor procedure will not solve the problem because it does not help to identify the functional successor.

- The Chord protocol proposes that, rather than maintain a single successor, each node maintains a list of α successors, which are the node's first α successors.
- If the first successor does not respond, the node can try the next successor from the list, and so on. Only the simultaneous failure of all the α successors can then cause the protocol to fail.
- The provision for a successor list at each node provides a natural mechanism for the application to manage replicated objects.
- The replicas get placed at the node corresponding to the object key, as well as at the nodes in the successor list of that node.
- As Chord is able to update its successor list as the successor list changes, Chord can also interface with the application to let it track the locations of the replicas.
- A voluntary departure from the ring can be treated as a failure. However, a failed node causes all the data (keys) stored at that node to be lost until corrective action is taken.
- When a node departs voluntarily, it should first transfer all the keys it is responsible for to its successor.
- The departing node should also inform its successor and predecessor.
- This will enable the successor to update its predecessor to the predecessor of the departing node.
- The predecessor will also be able to update its successor list by deleting the departing node and adding the last successor of the departing node's successor list to its own successor list.

Complexity

- For a Chord network with n nodes, each node is responsible for at most $(1 + \varepsilon)/n$ keys.
- 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 NETWORKS (CAN)

A content-addressable network (CAN) is scalable indexing mechanism that maps objects to their locations in the network.

- The real motivation behind CAN is the existing networks are not scalable.
- CAN support basic hash table operations on key-value pairs (K,V): insert, search, delete
- CAN is composed of individual nodes and each node stores a chunk (zone) of the hash table
- A hash table is formed as a subset of the (K,V) pairs in the table.
- Each node stores state information about neighbor zones.
- The requests (insert, lookup, or delete) for a key are routed by intermediate nodes using a greedy routing algorithm.
- It do not need any centralized control (completely distributed).
- The small per-node state is independent of the number of nodes in the system (scalable) and also the nodes can route around failures (fault-tolerant).

Properties of CAN

i) Distributed

ii) fault-tolerant

iii) scalable

iv) independent of the naming structure

v) implementable at the application layer

vi) self-organizing and self-healing.

CAN is a logical d-dimensional Cartesian coordinate space organized as a d-torus logical topology, i.e., a virtual overlay d-dimensional mesh with wrap-around.

- A d-torus logical topology is a virtual overlay d-dimensional mesh with wrap-around.
- The entire space is partitioned dynamically among all the nodes present, so that each node i is assigned a disjoint region r(i) of the space.
- As nodes arrive, depart, or fail, the set of participating nodes, as well as the assignment of regions to nodes, change. =
- For any object v, its key k(v) is mapped using a deterministic hash function to a point p in the Cartesian coordinate space.

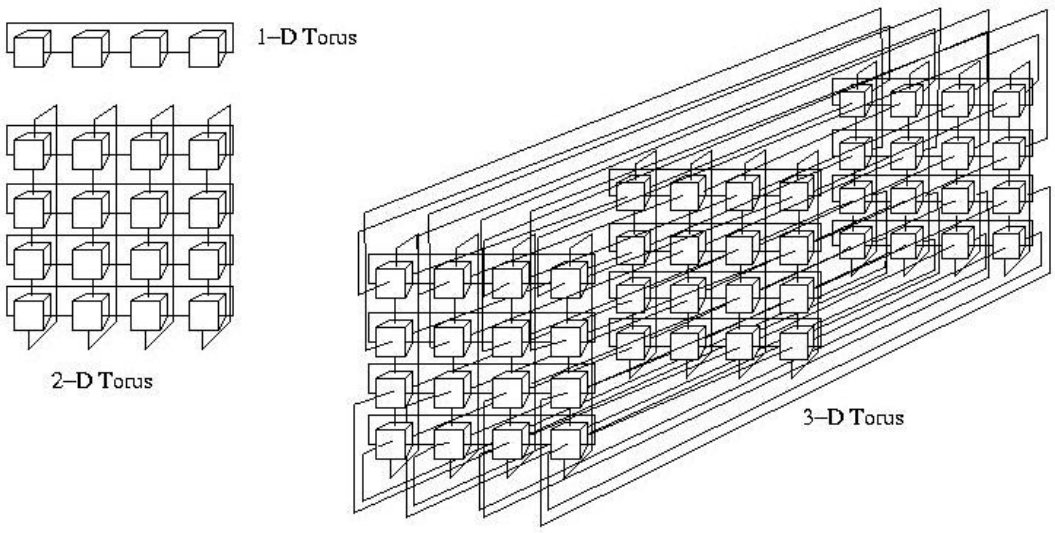


Fig 5.6: d-Torus topology

- The (k, v) pair is stored at the node that is presently assigned the region that contains the point p. This means the (k, v) pair is stored at node i if presently the point p corresponding to (k, v) lies in region (r, i).
- To retrieve object v, the same hash function is used to map its key k to the same point p.
- The node that is presently assigned the region that contains p- is accessed to retrieve v.
- The three core components ofa CAN design are the following:
  1. Setting up the CAN virtual coordinate space, and partitioning it among the nodes as they join the CAN.
  2. Routing in the virtual coordinate space to locate the node that is assigned the region containing p.
  3. Maintaining the CAN due to node departures and failures.

5.4.1 Initialization of CAN

The following are the steps in CAN initialization:

1. Each CAN is assumed to have a unique DNS name that maps to the IP address of one or a few bootstrap nodes of that CAN.

A bootstrap node is responsible for tracking a partial list of the nodes that it believes are currently participating in the CAN.

2. To join a CAN, the joiner node queries a bootstrap node via a DNS lookup, and the bootstrap node replies with the IP addresses of some randomly chosen nodes that it believes are participating in the CAN.

3. 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 the CAN routing algorithm.

4. 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, so as to decide which dimension to split along. This also helps to methodically merge regions, if necessary. The (k, v) tuples forwhich the key k now maps to the zone to be transferred to the joiner, arealso transferred to the joiner.

5. The joiner learns the IP addresses of its neighbors from old_owner(p). The neighbors are old_owner(p) and a subset of the neighbors of old_owner(p). The old_owner(p) also updates its set of neighbors. The new joiner as well as old_owner(p) inform their neighbors of the changes to the space allocation, so that they have correct information about their neighborhood and can route correctly. Each node has to send an immediate update of its assigned region, followed by periodic **HEARTBEAT** refresh messages, to all its neighbors.

   - When a node joins a CAN, only the neighboring nodes in the coordinate space are required to participate in the joining process.

   - The overhead is the order of the number of neighbors, which is O(d) and independent of n, the number of nodes in the CAN.

5.4.2 CAN Routing

- CAN routing uses the straight-line path from the source to the destination in the logical Euclidean space.

- Each node maintains a routing table that tracks its neighbor nodes in the logical coordinate space.

- In d-dimensional space, nodes x and y are neighbors if the coordinate ranges of their regions overlap in d − 1 dimensions, in one dimension.

- All the regions are convex.

- Let the region x be $[[x^1_{min}, x^1_{max}], \ldots [x^a_{min}, x^a_{max}]]$ and the region y be $[[y^1_{min}, y^1_{max}], \ldots [y^d_{min}, y^d_{max}]]$.

- X and y are neighbors if there is some dimension j such that $x^j_{max} = y^i_{min}$ and for all dimensions, $[x^i_{min}, x^i_{max}]]$ and $[y^i_{min}, y^i_{max}]]$ overlap.

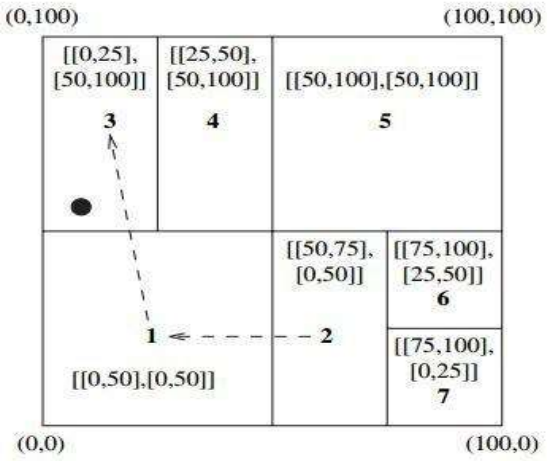


Fig 5.7: Two-dimensional CAN space

- The routing table at each node tracks the IP address and the virtual coordinate 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.
- To implement greedy routing to a destination node x, the present node routes a message to that neighbor among the neighbors $k \in$ Neighbors:

$$\underset{k \in \text{Neighbors}}{\text{argmin}} \left[\min | \vec{x} - \vec{k} | \right]$$

- Assuming equal-sized zones in d-dimensional space, the average number of neighbors for a node is O(d).
- The average path length is $(d/4)\, n^{1/d}$.
- The implication on scaling is that each node has about the same number of neighbors and needs to maintain about the same amount of state information, irrespective of the total number of nodes participating in the CAN.
- The CAN structure is superior to that of Chord.
- Unlike in Chord, there are typically many paths for any given source-destination pair.
- This greatly helps for fault-tolerance.
- Average path length in CAN scales as $O(n^{1/d})$ as opposed to log n for Chord.

5.4.3 Maintenance in CAN

- When a node voluntarily departs from CAN, it hands over its region and the associated database of (key, value) tuples to one of its neighbors.

- If the node's region can be merged with that of one of its neighbors to form a valid convex region, then such a neighbor is chosen.

- Otherwise the node's region is handed over to the neighbor whose region has the smallest volume or load – the regions are not merged and the neighbor handles both zones temporarily until a periodic background region reassignment process runs to integrate the regions and prevent further fragmentation.

- AN requires each node to periodically send a HEARTBEAT update message to each neighbor, giving its assigned region coordinates, the list of its neighbors, and their assigned region coordinates.

- When a node dies, the neighbors suspect its death and initiate a TAKEOVER protocol to decide who will take over the crashed node's region.

- Despite this TAKEOVER protocol, the (key, value) tuples in the crashed node's database remain lost until the primary sources of those tuples refresh the tuples.

- Requiring the primary sources to periodically issue such refreshes also serves the dual purpose of updating stale or dirty objects in the CAN.

TAKEOVER protocol

- When a node suspects that a neighbor has died, it starts a timer in proportion to its region's volume.

- On timeout, it sends a TAKEOVER message, with its region volume piggybacked on the message, to all the neighbors of the suspected failed node.

- When a TAKEOVER message is received, a node cancels its bid to take over the failed node's region if the received TAKEOVER message contains a smaller region volume than that of the recipient's region.

- This protocol thus helps in load balancing by choosing the neighbor whose region volume is the smallest, to take over the failed node's region. As all nodes initiate the TAKEOVER protocol, the node taking over also discovers its neighbors and vice versa.

- In the case of multiple concurrent node failures in one vicinity of the Cartesian space, a more complex protocol using an expanding ring search for the TAKEOVER messages can be used.

- A graceful departure as well as a failure can result in a neighbor holding more than one region if its region cannot be merged with that of the departed or failed node.
- To prevent the resulting fragmentation and restore the $1 \rightarrow 1$ node to region assignment, there is a background reassignment algorithm that is run periodically.
- Conceptually, consider a binary tree whose root represents the entire space. An internal node represents a region that existed earlier but is now split into regions represented by its children nodes.
- A leaf represents a currently existing region, and overloading the semantics and the notation, also the node that represents that region.
- When a leaf node x fails or departs, there are two cases:
    1. If its sibling node y is also a leaf, then the regions of x and y are merged and assigned to y. The region corresponding to the parent of x and y becomes a leaf and it is assigned to node y.
    2. If the sibling node y is not a leaf, run a depth-first search in the sub tree rooted at y until a pair of sibling leaves (say, z1 and z2) is found. Merge the regions of z1 and z2, making their parent z a leaf node, assign the merged region to node z2, and the region of x is assigned to node z1.
- A distributed version of the above depth-first centralized tree traversal can be performed by the neighbors of a departed node.
- The distributed traversal leverages the fact that when a region is split, it is done in accordance to a particular ordering on the dimensions.
- Node i performs its part of the depth first traversal as follows:
    1. Identify the highest ordered dimension dim_a that has the shortest coordinate range $[i^{dima}_{min}, i^{dima}_{max}]$. Node i's region was last halved along dimension dim_a.
    2. Identify neighbor j such that j is assigned the region that was split off from i's region in the last partition along dimension dim_a. Node j's region i's region along dimension dim_a.
    3. If j's region volume equals i's region volume, the two nodes are siblings and the regions can be combined. This is the terminating case of the depth first tree search for siblings. Node j is assigned the combined region, and node i takes over the region of the departed node x. This take over by node i is done by returning the recursive search request to the originator node, and communicating i's identity on the replies.
    4. Otherwise, j's region volume must be smaller than i's region volume. Node i forwards a recursive depth-first search request to j.

CAN Optimizations

The following are the design techniques to improve the performance of factors:

- **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:** A coordinate space is termed as a reality. The use of multiple independent realities assigns to each node a different region in each different reality. This implies that in each reality, the same node will store different (k, v) tuples belonging to the region assigned to it in that reality, and will also have a different neighbor set. The data contents (k, v) get replicated in each reality, leading to higher data availability. The multiple copies of each (k, v) tuple, one in each reality, offer a choice – the closest copy can be accessed. Routing fault tolerance improves because each reality offers a set of different paths to the same (k, v) tuple. All these contribute to more storage.
- **Delay latency:** The delay latency on each of the candidate logical links can also be used in making the routing decision.
- **Overloading coordinate regions:** Each region can be shared by multiple nodes, up to some upper limit. This reduces path length and path latency. The fault tolerance improves because a region becomes empty only if all the nodes assigned to it depart or fail concurrently. The per-hop latency decreases because a node can select the closest node from the neighboring region to forward a message towards the destination. This demands many of the aspects of the basic CAN protocol need to be reengineered to accommodate overloading of coordinate regions.
- **Multiple hash functions:** The use of multiple hash functions maps each key to different points in the coordinate space. This replicates each (k, v) pair for each hash function used. The effect is similar to that of using multiple realities.
- **Topologically sensitive overlay:** The CAN overlay has no correlation to the physical proximity or to the IP addresses of domains. Logical neighbors in the overlay may be geographically far apart, and logically distant nodes may be physical neighbors. By constructing an overlay that accounts for physical proximity in determining logical neighbors, the average query latency can be significantly reduced.

CAN Complexity

- The time overhead for a new joiner is O(d) for updating the new neighbors in the CAN, and O(d/4 log(n)) for routing to the appropriate location in the coordinate space.
- The time overhead and the overhead in terms of the number of messages for a node departure is $O(d^2)$, because the TAKEOVER protocol uses a message exchange between each pair of neighbors of the departed node.

5.5 TAPESTRY

Tapestry is a peer-to-peer overlay network which provides a distributed hash table, routing, and multicasting infrastructure for distributed applications. The Tapestry peer-to-peer system offers efficient, scalable, self-repairing, location-aware routing to nearby resources.

- Tapestry is a decentralized distributed system.
- It is an overlay network that implements simple key-based routing.
- It is a prototype of a decentralized, scalable, fault-tolerant, adaptive location and routing infrastructure
- Each node serves as both an object store and a router that applications can contact to obtain objects.
- In a Tapestry network, objects are published at nodes, and once an object has been successfully published, it is possible for any other node in the network to find the location at which that object is published.
- The difference between Chord and Tapestry is that in Tapestry the application chooses where to store data, rather than allowing the system to choose a node to store the object at.
- The application only publishes a reference to the object.
- The Tapestry P2P overlay network provides efficient scalable location independent routing to locate objects distributed across the Tapestry nodes.
- The hashed node identifiers are termed **VIDs** (Virtual ID) and the hashed object identifiers are termed as **GUIDs** (Globally Unique ID).

5.5.1 Routing and Overlays

Routing and overlay are the terms coined for looking objects and nodes in any distributed system.

- It is a middleware that takes the form of a layer which processes the route requests from the clients to the host that holds the objects.
- The objects can be placed and relocated without the information from the clients.

Functionalities of routing overlays:

- A client requests an object with GUID to the routing overlay, which routes the request to a node at which the object replica resides.

- A node that wishes to make the object available to peer-to-peer service computes the GUID for the object and announces it to the routing overlay that ensures that the object is reachable by all other clients.
- When client demands object removal, then the routing overlays must make them unavailable.
- Nodes may join or leave the service.

Routing overlays in Tapestry

- Tapestry implements Distributed Hash Table (DHT) and routes the messages to the nodes based on GUID associated with resources through prefix routing.
- **Publish (GUID)** primitive is issued by the nodes to make the network aware of its possession of resource.
- Replicated resources also use the same publish primitive with same GUID. This results in multiple routing entries for the same GUID.
- This offers an advantage that the replica of objects is close to the frequent users to avoid latency, network load, improve tolerance and host failures.

Roots and Surrogate roots

- Tapestry uses a common identifier space specified using m bit values and presently Tapestry recommends m = 160.
- Each identifier O_G in this common overlay space is mapped to a set of unique nodes that exists in the network, termed as the identifier's root set denoted O_{GR}.
- If there exists a node v such that $v_{id} = O_{GR}$, then v is the root of identifier O_G.
- If such a node does not exist, then a globally known deterministic rule is used to identify another unique node sharing the largest common prefix with O_G, that acts as the **surrogate root.**
- To access object O, the goal is to reach the root O_{GR}.
- Routing to O_{GR} is done using distributed routing tables that are constructed using prefix routing information.

Prefix Routing

Prefix routing at any node to select the next hop is done by increasing the prefix match of the next hop's VID with the destination O_{GR}.

- Let $M = 2^m$. The routing table at node vid contains $b \cdot \log_b M$ entries, organized in $\log_b M$ levels $i = 1,\ldots, \log_b M$.
- Each entry is of the form <w_{id}, IP address>.
- The following is the property of entry (b) at level i:

  Each entry denotes some neighbor node VIDs with an (i – 1) digit prefix match with v id . Further, in level i, for each digit j in the chosen base, there is an entry for which the ith digit position is j. The jth entry (counting from 0) in level i has value j for digit position i. Let an i digit prefix of vid be denoted as prefix (vid, i). Then the jth entry (counting from 0) in level i begins with an i-digit prefix prefix (vid,i– 1). j.

Routing Table

- The nodes in the router table at v_{id} are the neighbors in the overlay, and these are exactly the nodes with which v_{id} communicates.
- For each forward pointer from node v to v', there is a backward pointer from v' to v.
- There is a choice of which entry to add in the router table. The jth entry in level i can be the VID of any node whose i-digit prefix is determined; the (m – i) digit suffix can vary.
- The flexibility is useful to select a node that is close, as defined by some metric space.
- This choice also allows a more fault-tolerant strategy for routing.
- Multiple VIDs can be stored in the routing table.
- The jth 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 jth 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.

(variables)

Integer Table[1…$\log_b 2^m$, 1….b]; //routing table

(1) NEXT_HOP(i, OG = d_1 o d_2 … o d_{logbm}) 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 the 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, OG)  // locally examine next digit of
                                         //destination
(1e) else return (Table[i, d_i]).        // node Table[i, d_i] is next hop
```

Fig 5.8: NEXT_HOP(i, O_G)

Object Publication and object searching

- The unique spanning tree used to route to vid 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.
- Each node between O and O_{GR} must maintain a pointer to O despite churn.
- If a node lies on the path from two or more servers storing replicas, that node will store a pointer to each replica, sorted in terms of a distance metric.
- This is the directory information for objects, and is maintained as a soft-state, i.e., it requires periodic updates from the server, to deal with changes and to provide fault-tolerance.
- 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 guaranteed to 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.
- Each hop towards the root reduces the choice of the selection of its next node by a factor of b; hence, the more likely by a factor of b that a query path and a publish path will meet.

- As the next hop is chosen based on the network distance metric whenever there is a choice, it is observed that the closer the client is to the server in terms of the distance metric, the more likely that their paths to the object root will meet sooner, and the faster the query will be redirected to the object.

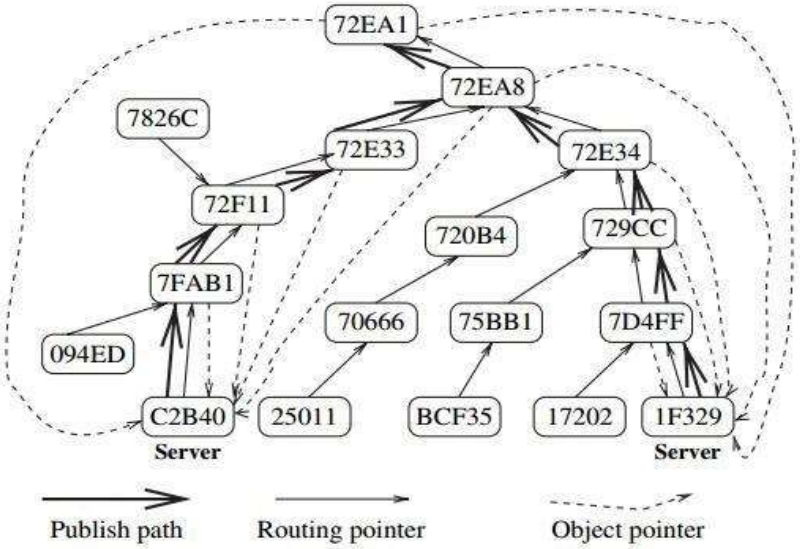


Fig 5.9: Publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40

Node Insertion

- When nodes join the network, the result should be the same as though the network and the routing tables had been initialized with the nodes as part of the network.
- The procedure for the insertion of node X should maintain the following property of Tapestry: For any node Y on the path between a publisher of object O and the root O_{GR}, node Y should have a pointer to O.

Properties for node insertion:

- Nodes that 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.

Steps in insertion

- 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 Xid). The surrogate Z identifies the length α of the longest common prefix that Zid shares with Xid.
- Node Z initiates a MULTICAST-CONVERGECAST on behalf of X by creating a logical spanning tree as follows. Acting as a root, Z contacts all the (α, j) nodes, for all $j \in \{0, 1, \ldots, b-1\}$.
- These are the nodes with prefix α followed by digitj.
- Each such (level 1) node Z1 contacts all the prefix ((Z1, $|\alpha| + 1$),j) nodes, for all $j \in \{0, 1,\ldots,b-1\}$. This continues up to level $\log_b 2^m$ and completes the MULTICAST.
- The nodes at this level are the leaves of the tree, and initiate the CONVERGECAST, which also helps to detect the termination of this phase.
- The insertion protocols are fairly complex and deal with concurrent insertions.

Node Deletion

When a node A leaves the Tapestry overlay:

1. Node A informs the nodes to which it has backpointers. It also provides them with replacement entries for each level from its routing table. This is to prevent holes in their routing tables.
2. The servers to which A has object pointers are also notified. The notified servers send object republish messages.
3. 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 are handled by using the redundancy that is built in to the routing tables and object location pointers.
- A node X detects a failure of another node A by using soft-state beacons or when a node sends a message but does not get a response.
- Node X updates its routing table entry for A with a suitable substitute node, running the nearest neighbor algorithm if necessary.
- If A's failure leaves a hole in the routing table of X, then X contacts the suggorate of A in an effort to identify a node to fill the hole.
- To repair the routing mesh, the object location pointers also have to be adjusted.

- Objects rooted at the failed node may be inaccessible until the object is republished.
- The protocols for doing so essentially have to:
  i) maintain path availability
  ii) optionally collect garbage/dangling pointers that would otherwise persist until the next soft-state refresh and timeout

Complexity

- A search for an object is expected to take $\log_b 2^m$ hops. The routing tables are optimized to identify nearest neighbor hops.
- The size of the routing table at each node is $c \cdot b \cdot \log_b 2^m$ where c is the constant that limits the size of the neighbor set that is maintained for fault-tolerance.

5.6 DISTRIBUTED SHARED MEMORY

5.6.1 Abstraction and its advantages

Distributed Shared Memory is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory. The shared memory model provides a virtual address space that is shared among all computers in a distributed system.

- It is an abstraction provided to the programmer of a distributed system.
- It gives the impression of a single memory. Programmers access the data across the network using only read and write primitives.
- 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.
- A part of each computer's memory is earmarked for shared space, and the remainder is private memory.
- To provide programmers with the illusion of a single shared address space, a memory mapping management layer is required to manage the shared virtual memory space.

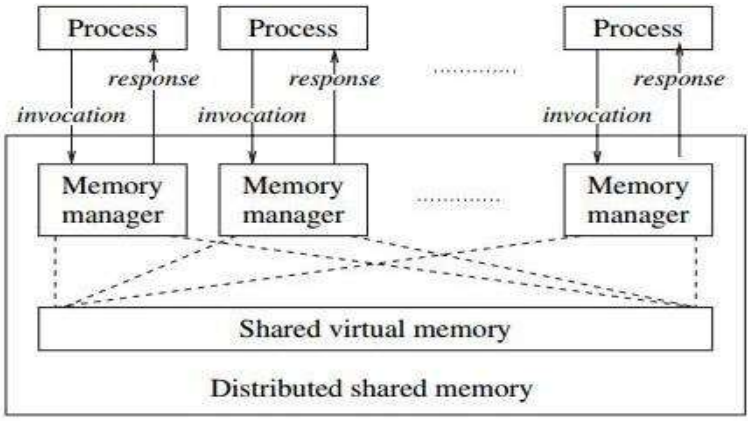


Fig 5.10: Abstract view of Distributed Shared Memory

Advantages of DSM

- Communication across the network is achieved by the read/write abstraction that simplifies the task of programmers.
- A single address space is provided, thereby providing the possibility of avoiding data movement across multiple address spaces, and simplifying passing-by-reference and passing complex data structures containing pointers.
- If a block of data needs to be moved, the system can exploit locality of reference to reduce the communication overhead.
- DSM is economical than using dedicated multiprocessor systems,because it uses simpler software interfaces and off-the-shelf hardware.
- There is no bottleneck presented by a single memory access bus.
- DSM effectively provides a large (virtual) main memory.
- DSM provides portability of programs written using DSM. This portability arises due to a common DSM programming interface, which is independent of the operating system and other low-level system characteristics.

When multiple processors wish to access the same data object, a decision about how to handle concurrent accesses needs to be made. If concurrent access is permitted by different processors to different replicas, the problem of replica consistency needs to be addressed.

Challenges in implementing replica coherency in DSM systems

1. Programmers are aware of the availability of replica consistency models and from coding their distributed applications according to the semantics of these models.

2. As DSM is implemented as asynchronous message passing, it faces the overhead of asynchronous synchronization.
3. Since the control is given to memory management, the programmers lose the ability to use their own message-passing solutions for accessing shared objects.

Issues in designing a DSM system:

- Determining the semantics to allow for concurrent access to shared objects.
- Determining the best way to implement the semantics of concurrent access to shared data either to use read or write replication.
- Selecting the locations for replication to optimize efficiency from the system's viewpoint.
- Determining the location of remote data that the application needs to access, if full replication is not used.
- Reducing communication delays and the number of messages that are involved under the covers while implementing the semantics of concurrent access to shared data.

5.7 MEMORY CONSISTENCY 8MODELS

A memory consistency model is a set of rules which specify when a written value by one thread can be read by another thread.

- These rules are essential to write a correct program**.**
- **Memory coherence** is the ability of the system to execute memory operations correctly.
- The problem of ensuring memory coherence is identifying which of the interleaving are correct, which of course requires a clear definition of correctness.
- The memory consistency model defines the set of allowable memory access orderings.
- In DSM system, the programmers write their programs keeping in mind the allowable interleaving permitted by that specific memory consistency model.
- A program written for one model may not work correctly on a DSM system that enforces a different model.
- The model can thus be viewed as a contract between the DSM system and the programmer using that system.

- The memory consistency model affects:
  - i) System implementation: hardware, OS, languages, compilers
  - ii) Programming correctness
  - iii) Performance

5.7.1 Strict consistency, atomic consistency, linearizability

- According to Von Neumann architecture/ uniprocessor machine, any Read operation to a location should return the value or variable written by the most recent Write to that location or a variable.
- The system built over the above principle is called strict or atomic consistency model.
- The features of the atomic consistency model area:
  - i) Common global time axis is implicitly available in a uniprocessor system
  - ii) The write operation is immediately visible to all processes

Atomic Consistency Model:

i) Any Read to a location is required to return the value written by the most recent Write to that location in accordance with global time reference. For non overlapping operations, with respect to the global time reference, the specification is clear. For overlapping operations the following further specifications are necessary.

ii) All operations appear to be executed atomically and sequentially.

iii) All processors see the same ordering of events, which is equivalent to the global-time occurrence of non-overlapping events.

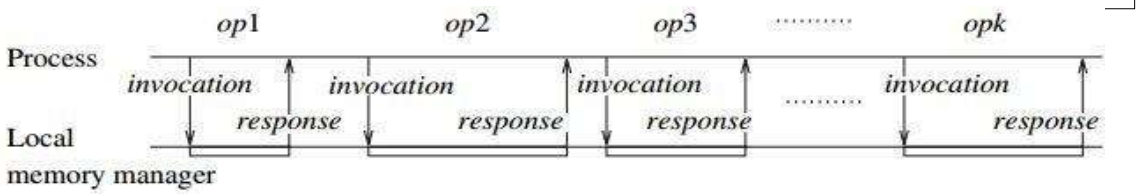


Fig 5.11: Invocations and responses in sequential system

The invocation and the response to each invocation can be viewed as being atomic events. An execution sequence in global time is viewed as a sequence Seq of such invocations and responses. The Seq must satisfy the following conditions:

- **Liveness:** Each invocation must have a corresponding response.

- **Correctness:** The projection of Seq on any processor i, denoted Seqi, must be a sequence of alternating invocations and responses if pipelining is disallowed.

A linearizable execution needs to generate an equivalent global order on the events that is a permutation of Seq, satisfying the semantics of linearizability.

Linearizable property:

A sequence Seq of invocations and responses is linearizable (LIN) if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. ***For every variable v, the projection of Seq' on v, denoted Seqv' , is such that every Read (adjacent (invoc, resp) event pair) returns the most recent Write (adjacent, (nvoc, resp) event pair) that immediately preceded it.***
2. ***If the response op1(resp) of operation op1 occurred before the invocation op2(invoc) of operation op2 in Seq, then op1 (adjacent (invoc, resp)event pair) occurs before op2 (adjacent (invoc, resp) event pair) in Seq.***

- Linearizability is a guarantee about single operations on single objects.
- It provides a real- guarantee on the behavior of a set of single operations on a single object.

Linearizability requires that each operation appears to occur atomically at some point between its invocation and completion. This point is called the linearization point.

Implementation of Linearizability

- Implementing linearizability is expensive because a global time scale needs to be simulated.
- As all processors need to agree on a common order, the implementation needs to use total order.
- For simplicity, the algorithm described here assumes full replication of each data item at all the processors.
- This demands the total ordering to be combined with a broadcast.
- The memory manager software is placed between the application above it and the total order broadcast layer below it.

(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**,

Fig 5.12: Implementing Linearizability

The algorithm in Fig 5.12 ensures total order broadcast such that all processors follow the same order:

1. For two non-overlapping operations at different processors, the response to the former operation precedes the invocation of the latter in global time.
2. For two overlapping operations, the total order ensures a common view by all processors.

5.7.2 Sequential Consistency

Sequential consistency requires that a shared memory multiprocessor appear to be a multiprogramming uniprocessor system to any program running on it.

Sequential consistency requires that:

1. All instructions are executed in order.
2. Every write operation becomes instantaneously visible throughout the system.

   The main motivation behind sequential consistency is that the atomic consistency is very difficult to implement since the it is very difficult for a system to synchronize to global clock. Sequential consistency is specified as follows:

   - 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.

Sequential Consistency:

A sequence Seq of invocation and response events is sequentially consistent if there is a permutation Seq' of adjacent pairs of corresponding (invoc, resp) events satisfying:

1. ***For every variable v, the projection of Seq' on v, denoted Seqv' , is such that every Read (adjacent, (invoc,resp) event pair) returns the most recent Write (adjacent, (invoc, resp) event pair) that immediately preceded it.***
2. ***If the response op1(resp) of operation op1 at process Pi occurred before the invocation op2(invoc) of operation op2 by process Pi in Seq, then op1 (adjacent (invoc, resp) event pair) occurs before op2 (adjacent (invoc,resp) event pair) in Seq.***

Implementation of Sequential Consistency

All processors are required to see the same global order, but global time ordering need not be preserved across processes. So it is sufficient to use total order broadcasts for the Write operations only. In the simplified algorithm, no total order broadcast is required for Read operations, because:

1. all consecutive operations by the same processor are ordered in the same order because pipelining is not used.
2. Read operations by different processors are independent of each other and need to be ordered only with respect to the Write operations in the execution.

(shared var)

int: x:

(1) When the memory manager 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 Pi receives a **total_order_broadcasts**$_j$(write, x, val) from network;

(2a) **Write** val to local replica of x;

(2b) **if** i=j **then return** acknowledgement to application.

Fig 5.13: Sequential Consistency using Local Read algorithm

Local-read algorithm

- A Read operation completes atomically, whereas a Write operation does not.

- Between the invocation of a Write by Pi (line 1b) and its acknowledgement (lines 2a, 2b), there may be multiple Write operations initiated by other processors that take effect at Pi (line 2a).
- Thus, a Write issued locally has its completion locally delayed. Such an algorithm is acceptable for Read intensive programs.

Local-write algorithm

- This does not delay acknowledgement of Writes.
- For Write intensive programs, it is desirable that a locally issued Write gets acknowledged immediately even though the total order broadcast for the Write, and the actual update for the Write may not go into effect by updating the variable at the same time.
- The algorithm achieves this at the cost of delaying a Read operation by a processor until all previously issued local Write operations by that same processor have locally gone into effect.
- The variable counter is used to track the number of Write operations that have been locally initiated but not completed at any time.
- A Read operation completes only if there are no prior locally initiated Write operations that have not written to their variables.
- Else, a Read operation is delayed until after all previously initiated Write operations have written to their local variables, which happens after the total order broadcasts associated with the Write have delivered the broadcast message locally.

(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) count ← counter + 1;

(2b) total_order_broadcast$_i$ Write(x, val)

(2c) return acknowledgement to the application.

(3) When the memory manager at P_i receives a total_order_broadcast$_j$ Write(x, val) from network:

(3a) write val to local replica of x;

(3b) if i=j then

(3c) counter ← counter – 1;

(3d) if (counter = 0 and any Reads are pending) then

(3e) perform pending responses for the Reads to the application.

Fig 5.14: Sequential Consistency using local write algorithm

5.7.3 Casual Consistency

- The causal consistency model represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not.

According to casual consistency model, only that Writes that are causally related must be seen in that same order by all processors, whereas concurrent Writes may be seen by different processors in different orders.

The causality relation is defined as follows:

- **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 operation 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.

5.7.4 Pipelined RAM (PRAM) or Processor Consistency

- In causal consistency, the concurrent writes be seen in a different order on different machines, although causally-related ones must be seen in the same order by all machines.
- PRAM consistency or Pipelined RAM states that Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

- This is a weaker form of consistency requires only that Write operations issued by the same processor are seen by all other processors in the same order that they were issued, but Write operations issued by different processors may be seen in different orders by different processors.
- The local causality relation, as defined by the local order of Write operations, needs to be seen by other processors. Hence, this form of consistency is termed processor consistency.
- An equivalent name for this consistency model is **pipelined RAM (PRAM)**, to capture the behavior that all operations issued by any processor appear to the other processors in a FIFO pipelined sequence.

5.7.5 Slow Memory

- The use of weakly consistent memories or slow memory in distributed shared memory systems to combat unacceptable network delay and to allow such systems to scale is proposed.
- Slow memory is presented as a memory that allows the effects of writes to propagate slowly through the system, eliminating the need for costly consistency maintenance protocols that limit concurrency.
- Slow memory processes a valuable locality property and supports a reduction from traditional atomic memory.Thus slow memory is as expressive as atomic memory.

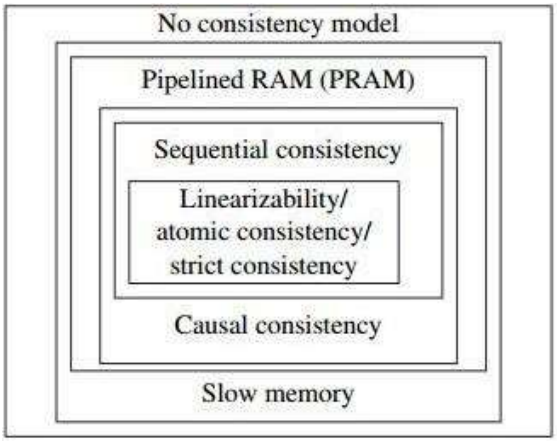


Fig 5.15: Hierarchy of memory consistency models

5.7.6 Models based on synchronization instructions

Synchronization instructions are like run-time library. The synchronization statements across the various processors must satisfy the consistency conditions; other program statements between synchronization statements may be executed by the different processors without any conditions.

i) Weak Consistency

The protocol is said to support weak consistency if:

- All accesses to synchronization variables are seen by all processes (or nodes, processors) in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
- All other accesses may be seen in different order on different processes (or nodes, processors).
- The set of both read and write operations in between different synchronization operations is the same in each process.

Drawbacks:

When a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables or about to begin reading them.

ii) Release Consistency

The drawbacks of weak consistency are overcome by:

1. Ensuring that all locally initiated Writes have been completed, i.e., propagated to all other processes.
2. Ensuring that all Writes from other machines have been locally reflected

   To differentiate the entering and leaving of CS, release consistency provides acquire and release operations.

Acquire:

- Acquire accesses are used to tell the memory system that a critical region is about to be entered.
- The actions for case 2 need to be performed to ensure that local replicas of variables are made consistent with remote ones.

Release:

- This accesses say that a critical region has just been exited.
- Hence, the actions for case 1 need to be performed to ensure that remote replicas of variables are made consistent with the local ones that have been updated.

The Acquire and Release operations can be defined to apply to a subset of the variables. The accesses themselves can be implemented either as ordinary operations on special variables or as special operations. If the semantics of a CS is not associated

with the Acquire and Release operations, then the operations effectively provide for barrier synchronization.

The barrier synchronization states that until all processes complete the previous phase, none can enter the next phase.

This is implemented through protected variables which follows the given rules:

- All previously initiated Acquire operations must complete successfully before a process can access a protected shared variable.
- All accesses to a protected shared variable must complete before a Release operation can be performed.
- The Acquire and Release operations effectively follow the PRAM consistency model.

The lazy release consistency model is relaxation of the release consistency model in which rather than propagating the updated values throughout the system as soon as a process leaves a critical region, the updated values are propagated to the rest of the system only on demand, i.e., only when they are needed.

iii) Entry Consistency

- Entry consistency requires the programmer to use Acquire and Release at the start and end of each CS, respectively.
- Entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier.
- When an Acquire is performed on a synchronization variable, only access to those ordinary shared variables that are guarded by that synchronization variable is regulated.

5.8 SHARED MEMORY MUTUAL EXCLUSION

Shared memory model is implemented in operating systems through semaphores monitors and atomically executable special purpose hardware.

5.8.1 Lamport's bakery algorithm

- Lamport proposed the classical bakery algorithm for n-process mutual exclusion in shared memory systems.
- This algorithm satisfies the **requirements of the critical section problem** namely mutual exclusion, bounded waiting, and progress.

- All process threads must take a number and wait their turn to use a shared computing resource or to enter their critical section.
- The number can be any of the global variables, and processes with the lowest number will be processed first.
- If there is a tie or similar number shared by both processes, it is managed through their process ID.
- If a process terminates before its turn, it has to start over again in the process queue.
- 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.
- Then, a unique lexicographic order is defined on the tuple (token, pid) and this dictates the order in which processes enter the critical section.

(shared vars)

boolean: choosing[1…n];

integer: timestamp[1….n];

repeat

(1) P_i executes the following for the entry section:

(1a) choosing[i] ← 1;

(1b) timestamp[i] ← $\max_{k\in[1...n]}$ (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 critial 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;

until false;

Fig 5.16: Lamport's Bakery algorithm for shared memory exclusion

Mutual exclusion

- In the entry section, a process chooses a timestamp for itself, and resets it to 0 when it leaves the exit section.
- These steps are non-atomic in the algorithm. Thus multiple processes could be choosing timestamps in overlapping durations.
- When process i reaches line 1d, it has to check the status of each other process j, to deal with the effects of any race conditions in selecting timestamps.
- In lines 1d–1f, process i serially checks the status of each other process j.
- If j is selecting a timestamp for itself, j's selection interval may have overlapped with that of i, leading to an unknown order of timestamp values.
- Process i needs to make sure that any other process j(j < i) that had begun to execute line 1b concurrently with itself and may still be executing line 1b does not assign itself the same timestamp.
- If this is not done mutual exclusion could be violated as i would enter the CS, and subsequently, j, having a lower process identifier and hence a lexicographically lower time stamp, would also enter the CS.
- The i waits for j's timestamp to stabilize, i.e., choosing [j] to be set to false.
- Once j's timestamp is stabilized, i moves from line 1e to line 1f.
- Either j is not requesting or j is requesting. Line 1f determines the relative priority between i and j.
- The process with a lexicographically lower timestamp has higher priority and enters the CS; the other process has to wait (line 1g).
- Thus mutual exclusion is satisfied by the algorithm.

Bounded Waiting

- Bounded waiting is satisfied because each other process j can overtake process i at most once after i has completed choosing its timestamp.

- The second time j chooses a timestamp, the value will necessarily be larger than i's timestamp if i has not yet entered its CS.

Progress

- Progress is guaranteed because the lexicographic order is a total order and the process with the lowest timestamp at any time in the loop is guaranteed to enter the CS.

Improvements in Lamport's Bakery Algorithm

i) Space complexity

- A lower bound of n registers, specifically, the timestamp array, has been shown for the shared memory critical section problem.

ii) Time complexity

- When the level of contention is low, the overhead of the entry section does not scale.
- This issue is addressed his concern is addressed by fast mutual exclusion with O(1).
- The limitation of this approach is that it does not guarantee bounded delay.

5.8.2 Lamport's WRWR mechanism and fast mutual exclusion

- This algorithm illustrates an important technique – the (W − R − W – R) sequence that is a necessary and sufficient sequence of operations to check for contention and to ensure safety in the entry section, by employing just two registers.
- The basic sequence of operations for W(x)–R(y)–W(y)–R(x):
  1. The first operation needs to be a Write to x. If it were a Read, then all contending processes could find the value of the variable even outside the entry section.
  2. The second operation cannot be a Write to another variable, for that could equally be combined with the first Write to a larger variable. The second operation should not be a Read of x because it follows Write of x and if there is no interleaved operation from another process, the Read does not provide any new information. So the second operation must be a Read of another variable, say y.
  3. The sequence must also contain Read(x) and Write(y) because there is no point in reading a variable that is not written to, or writing a variable that is never read.

4. The last operation in the minimal sequence of the entry section must be a Read, as it will help determine whether the process can enter CS. So the last operation should be Read(x), and the second-last operation should be the Write(y).

(shared variable among the processes)

integer: x, y; // shared register initialized

boolean b[1….n]; //flags to indicate interest in critical section

repeat

(1) $P_i(1 \le i \le n)$ executes entry section:

(1a) b[i] ← true;

(1b) x ← i;

(1c) if y ≠ 0 then

(1d) b[i] ← false;

(1e) await y=0;

(1f) goto(1a);

(1g) y ← i;

(1h) if x ≠ i then

(1i) b[i] ← false;

(1j) for j = 1 to n do

(1k) await y = 0;

(1l) if y ≠ i then

(1m) await y = 0;

(1n) goto(1a);

(2) $P_i(1 \le i \le n)$ executes entry section:

(3) $P_i(1 \le i \le n)$ executes exit section:

(3a) y ← 0;

(3b) b[i] ← false

Forever.

Fig 5.17: Lamport's fast mutual exclusion algorithm

5.8.3 Hardware Support for Mutual Exclusion

- Hardware support can allow for special instructions that perform two or moreoperations atomically.
- Two such instructions, Test&Set and Swap aredefined and implemented.
- The atomic execution of two actions, a Read and a Write operation can simplifya mutual exclusion algorithm.

(shared variables among the processes accessing each of the different object types)

register: Reg ← initial value; // shared register initialized

(local variables)

integer: old ← initial value; // value to be returned

(1) Test&Set(Reg) return value:

(1a) old ← Reg;

(1b) Reg ← 1;

(1c) return(old).

(2) Swap(Reg, new) return value:

(2a) old ← Reg;

(2b) Reg ← new;

(2c) return(old).

Fig 5.18: Definitions for Test&Set, Swap operations

(shared variables)

register: Reg ← false; // shared register initialized

(local variables)

integer: blocked ← 0 // variable to be checked before entering CS

repeat

(1) P_i executes the following for the entry section:

(1a) blocked ← true;

(1b) repeat

(1c) blocked ← 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 ← false;

(4) P_i executes the remainder section after the exit section

until false;

Fig 5.19: Code for Swap operation

(shared variable)

register: Reg ← false; // shared register initialized

boolean: waiting[1…n];

(local variables)

integer: blocked ← initial value // value to be checked before
// entering CS

repeat

(1) P_i executes the following for the entry section:

(1a) waiting[i] ← true;

(1b) blocked ← true;

(1c) repeat waiting[i] and blocked do

(1d) blocked ← Test&Set(Reg);

(1e) waiting[i] ← 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 ← (i + 1) mod n;

(3b) while next ≠ 1 and waiting [next] = false do

(3c) next ← (next + 1) mod n;

(3d) if next = i then

(3e) Reg ← false;

(3f) else waiting[j] ← false;

(4) P_i executes the remainder section after the exit section

until false;

Fig 5.20: Code for Test & Set operation