DISTRIBUTED SYSTEMS
UNIT I – INTRODUCTION

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

Distributed System

Definition

- A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.
  A distributed system is the one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages each having its own memory and operating system.

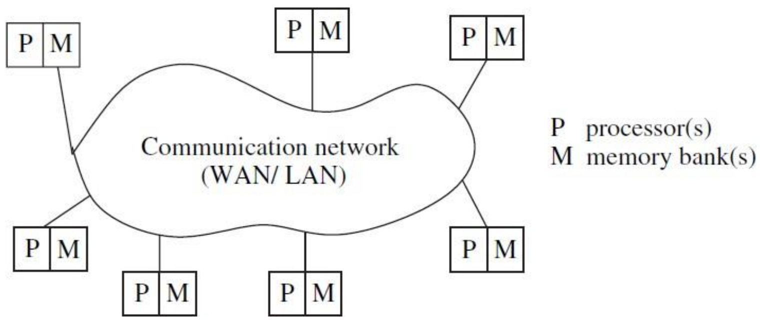
Characteristics of distributed system:

- It is a collection of autonomous processors communicating over a communication network have the following features:
- **No common physical clock.**
- **No shared memory.**
  - hence it requires message-passing for communication and implies the absence of common physical clock.
  - distributed system provides the abstraction of common address space via distributed shared memory abstraction.
- **Geographical separation**
  - The geographically wider apart processors are the representative of a distributed system i.e., it may be in wide-area network (WAN) or the network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN
  - NOW configuration is the low-cost high-speed off-the-shelf processors. Example: Google search engine.
- **Autonomy and heterogeneity**
  - The processors are "loosely coupled" having different speeds and each runs different operating system but cooperate with one another by offering services for solving a problem jointly.

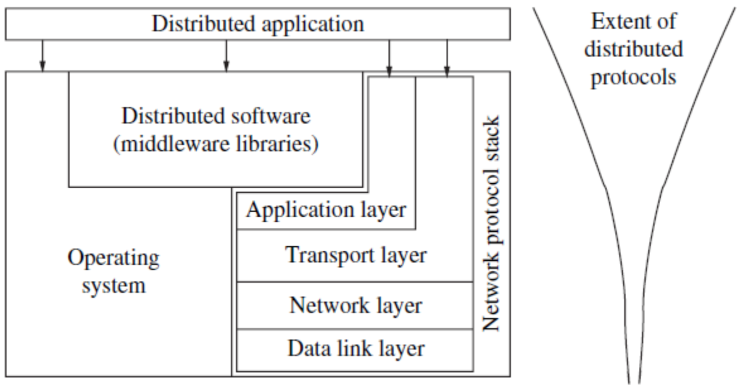
Relation to computer system components

- As shown in the Figure 1.1, In distributed system each computer has a memory-processing unit and are connected by a communication network.
- Figure 1.2 shows the relationships of software components that run on computers use the local operating system and network protocol stack for functioning.
- A distributed software is also termed as middleware.
- A distributed execution is the execution of processes across the distributed system to collaboratively achieve a common goal which is also termed a computation or a run.

- A distributed system follows a layered architecture that reduces the complexity of the system design.
- Middleware hides the heterogeneity transparently at the platform level.



(Fig.1.1 A distributed system connects processors by a communication network)



(Fig.1.2 Interaction of the software components at each processor)

- It is assumed that the middleware layer does not contain the application layer functions like http, mail, ftp, and telnet.
- User program code includes the code to invoke libraries of themiddleware layer to support the reliable and ordered multicasting.
- There are several standards such as
- Object Management Group's (OMG) common object request broker architecture (CORBA) ,
- RPC software
    - sends a message across the network to invoke the remote procedure.
    - waits for a reply,
    - after which the procedure call completes from the program perspective that invoked it.
- Some of the commercial versions of middleware often in use are CORBA, DCOM (distributed component object model), Java, and RMI (remote method invocation), message-passing interface (MPI).

Motivation

The motivation of using a distributed system is because of thefollowing requirements:

1. **Inherently distributed computations**

   Applications like money transfer in banking requires the computationthat is inherently distributed.

2. **Resource sharing**

   Resources like peripherals, databases, data (variable/files) cannot be fully replicated at all the sites. Further, they can't be placed at a single site as it leads to the bottleneck problem. Hence, such resources are distributed across the system.

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

   As the data may be too large and sensitive it cannot be replicated. Example, payroll data. Hence stored at a central server(like super computers) which can be queried using remote login. Advances in mobile devices and the wireless technology have proven the importance of distributed protocols andmiddleware.

4. **Enhanced reliability**

   A distributed system has provided increased reliability by the replicating resources and executions in geographically distributed systems which does not crash/malfunction at the same time under normal circumstances.

Reliability is defined in the aspect of

- availability, i.e., the resource should be accessible at all times;
- integrity, i.e., the value/state of the resource must be correct, in theface of concurrent access from multiple processors,
- fault-tolerance, i.e., the ability to recover from system failures.

5. **Increased performance/cost ratio**

   By resource sharing and accessing remote data will increase the performance/cost ratio. The distribution of the tasks across various computers provides a better performance/cost ratio, for example in NOW configuration.

A distributed system offers the following advantages:

6. **Scalability**

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

7. **Modularity and incremental expandability**

   Heterogeneous processors can be easily added without affecting the performance, as processors runs the same middleware algorithms. Similarly, existing processors can be easily replaced by other processors.

Relation to parallel multiprocessor/multicomputer systems

Characteristics of parallel systems

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

1. **multiprocessor system**
   - It is a parallel system in which the multiple processors have direct access to shared memory which forms a common address space. The architecture is shown in Figure 1.3(a). Such processors usually do not have a common clock and has uniform memory access architecture (UMA -waiting time, access any memory location from any processor issame).
   - The processors that are in close physical proximity are connected by an interconnection network. Interprocess communication across processors

is done through
  - read and write operations on the shared memory.
  - Message-passing primitives using MPI
- All processors run the same operating system, the hardware and software that are very tightly coupled.

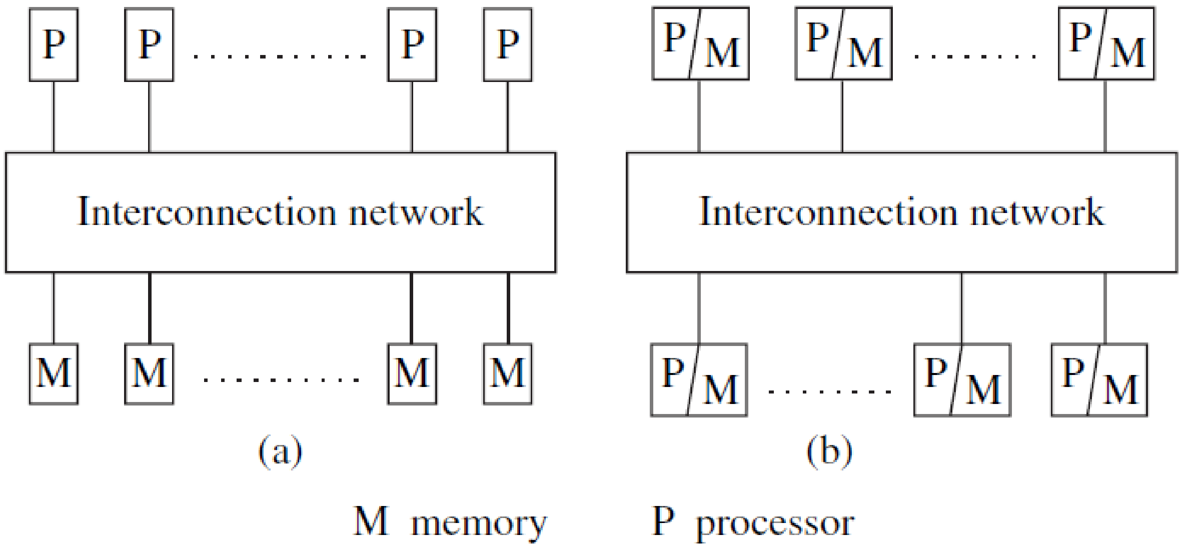
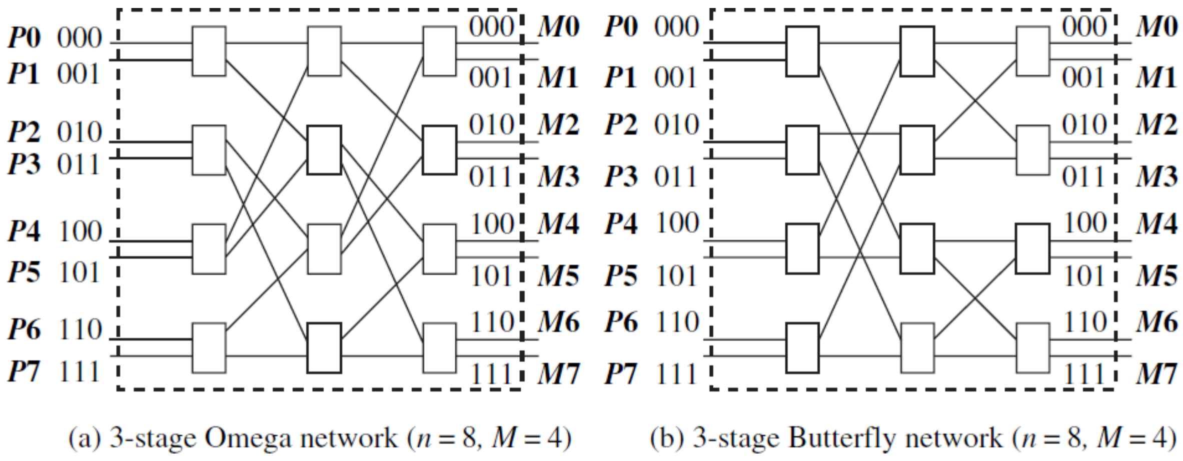


Figure 1.3 Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.



P0 000
P1 001
P2 010
P3 011
P4 100
P5 101
P6 110
P7 111
M0
M1
M2
M3
M4
M5
M6
M7
(a) 3-stage Omega network (n = 8, M = 4)
(b) 3-stage Butterfly network (n = 8, M = 4)

(Figure 1.4 Interconnection networks for shared memory multiprocessor systems. (a) Omega network [4] for n = 8 processors P0–P7 and memory banks M0–M7. (b) Butterflynetwork [10] for n = 8 processors P0–P7 and memory banks M0–M7.)

- The processors are usually of the same type, and are housed withinthe same box/container with a shared memory.
- The interconnection network to access the memory is using a bus,that provide greater efficiency, it is usually a multistage switch with a symmetric and regular design.
- Figure 1.4 shows two popular interconnection networks –
  - the Omega network and
  - the Butterfly network
- It is a multi-stage network formed of 2×2 switching elements. Each 2×2 switch allows data on either of the two input wires to be switched to the

upper or the lower output wire.

- In a single step, only one data unit can be sent on an output wire.
- So, if data from both the input wires are to be routed to the same output wire in a single step, collision happens.
- Buffering or more elaborate interconnection designs used to overcome collisions.
- Each 2×2 switch is represented as a rectangle in the figure.
- An n-input and n-output network uses log n stages and log n bits for addressing.
- Routing in the 2×2 switch at stage k uses only the kth bit, and is done at clock speed.
- 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 interconnection function

- The Omega network, connects n processors to n memory units has n/2 log2n switching elements of size 2×2 arranged in log2n stages.
- Between each pair of adjacent stages, a link exists between output i of a stage and input j to the next stage follows the perfect shuffle pattern.
- The iterative generation function is as follows:

$$j = \begin{cases} 2i, & \text{for } 0 \le i \le n/2 - 1, \\ 2i + 1 - n, & \text{for } n/2 \le i \le n - 1. \end{cases} \qquad (1.1)$$

- With respect to the Omega network in Figure 1.4(a), n = 8. Hence, for any stage, for the outputs i, where $0 \le i \le 3$, the output i is connected to input 2i of the next stage. For $4 \le i \le 7$, the output i of any stage is connected to input 2i+1−n of the next stage.

Omega routing function

- The routing function from input line i to output line j considers only j and the stage number s, where s ∈ [0, log2n−1].
- In a stage s switch, if the s+1th MSB (most significant bit) of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

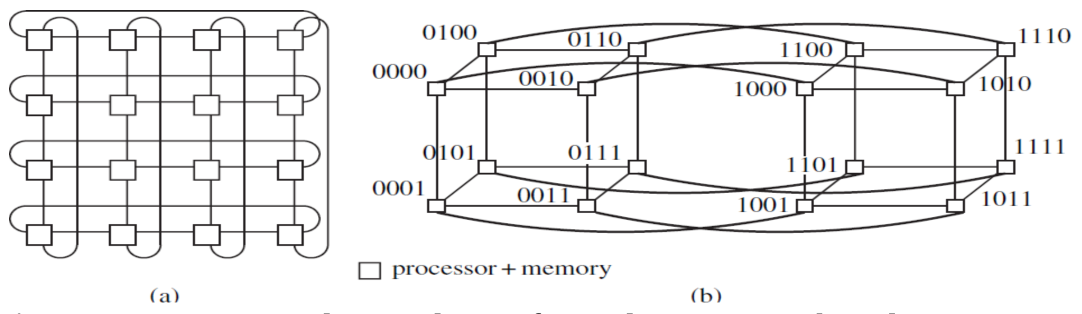
Butterfly interconnection function

- The generation of the interconnection pattern between a pair ofadjacent stages depends not only on n but also on the stage number s. The recursive expression is as follows.
- Let there be M =n/2 switches per stage, and let a switch be denotedby the tuple <x,s>, where x ∈ [0,M−1] and stage s ∈[0,log2n−1].
- The two outgoing edges from any switch <x, s> are as follows. Thereis an edge from switch <x,s> to switch <y,s+1> if
  (i) x = y or
  (ii) x XOR y has exactly one 1 bit, which is in the _s+1_th MSB.
- For stage s, apply the rule above for M/2^{s} switches.
- Whether the two incoming connections go to the upper or lower inputport is not important because of the routing function, given below.

Butterfly routing function

- 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. **Note:-**
- In Butterfly and Omega networks, the paths from the different inputs to any one output form a spanning tree. This implies that collisions will occur when data is destined to the same output line.
- Advantage: Data can be combined at the switches if the application semantics are known.

2. **multicomputer parallel system**
    - It is a parallel system in which the multiple processors do not have direct access to shared memory.
    - The memory of multiple processors may or may not form a common address space and do not have a common clock as shown in Figure 1.3(b).
    - The processors that are in close physical proximity are very tightly coupled (homogenous hardware and software), and connected by an interconnection network.
    - The processors communicate either via a common address space or via message-passing.
    - A multicomputer system that has a common address space usually corresponds to a non-uniform memory access (NUMA – access various shared memory locations from different processors varies) architecture.
    - Examples: NYU Ultracomputer, Sequent shared memory machines, the CM* Connection machine
    - Processors configured in regular and symmetrical topologies such as an array or mesh, ring, torus, cube, and hypercube (message-passing machines).
    - Figure 1.5(a) shows a wrap-around 4×4 mesh. For a $k\times k$ mesh which will contain k^2 processors, the maximum path length between any two processors is $2(k/2-1)$. Routing can be done along the Manhattan grid.
    - Figure 1.5(b) shows a four-dimensional hypercube. A k-dimensional hypercube has 2^k processor-and-memory units.
    - Each such unit is a node in the hypercube and has a unique k-bit label. The processors are labelled such that the shortest path between any two processors is the Hamming distance between the processor labels. This is clearly bounded by k.



(Figure 1.5 Some popular topologies for multicomputer shared-memory machines. (a) Wrap-around 2D-mesh, also known as torus. (b)

Hypercube of dimension 4.)

- The hypercube and its variant topologies have very interesting mathematical properties with implications for routing and fault- tolerance.

3. Array processors

- It belong to a class of parallel computers that are physically co- located, are very tightly coupled, and have a common system clockbut may not share memory and communicate by passing data using messages.
- Array processors perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing belong to this category. These applications usually involve a large number of iterations on the data.

Flynn's taxonomy

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

Single instruction stream, single data stream (SISD)

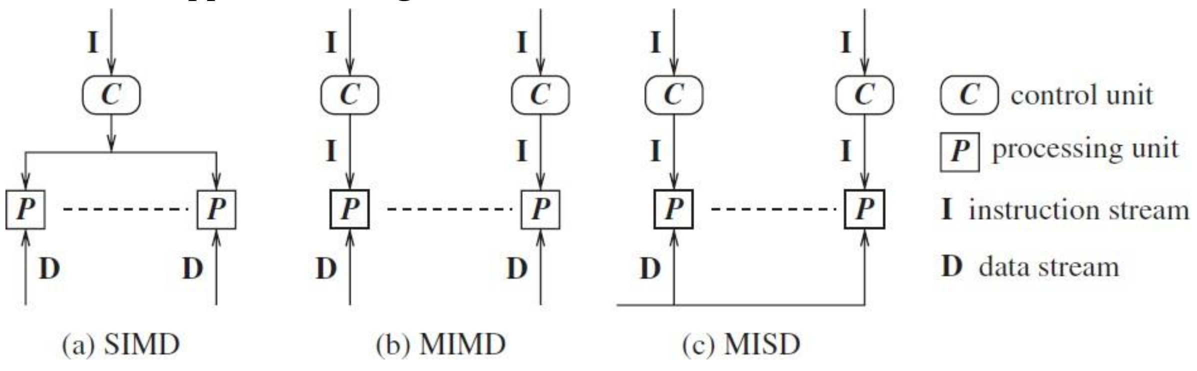
This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

Single instruction stream, multiple data stream (SIMD)

- In this mode processing by multiple homogenous processors executes in lock-step on different data items.
- Applications that involve operations on large arrays and matrices like scientific applications can exploit systems with SIMD mode of operation as the data sets can be partitioned easily.
- Example : parallel computers like Illiac-IV, MPP, CM2 were SIMD machines.

Multiple instruction stream, single data stream (MISD)

- This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications, e.g., visualization.



(Figure 1.6 Flynn's taxonomy of SIMD, MIMD, and MISD architectures for multiprocessor/multicomputer systems.)

Multiple instruction stream, multiple data stream (MIMD)

- In this mode, various processors execute different code on different data. It is used in distributed systems as well as parallel systems.
- There is no common clock among the system processors.
- Examples: Sun Ultra servers, multicomputer PCs, and IBM SP machines.
- It allows much flexibility in partitioning code and data to be processed among the processors.

Coupling, parallelism, concurrency, and granularity

Coupling

- The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules.
- When the degree of coupling is high (low), the modules are said to be tightly (loosely) coupled.
- SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.

Parallelism or speedup of a program on a specific system

- This is a measure of the relative speedup of a specific program, on a given machine.
- The speedup depends on the number of processors and the mapping of the code to the processors.
- It is expressed as the ratio of the time T(1) with a single processor, to the time T(n) with n processors.

Parallelism within a parallel/distributed program

- This is an aggregate measure of the percentage of time that all the processors are executing CPU instructions productively.

Concurrency of a program

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

Granularity of a program

- The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity.
- If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions and wait to get synchronized with the other processors.
- Programs with fine-grained parallelism are best suited for tightly coupled systems like SIMD and MISD architectures.
- The latency delays for the frequent communication over the WAN would significantly degrade the overall throughput.
- Hence loosely coupled multicomputers, programs with a coarse-grained communication/message-passing granularity will incur less overhead.

various classes of multiprocessor/multicomputer operating systems:

- The operating system running on loosely coupled processors (i.e., heterogenous distant processors) running loosely coupled software (i.e., heterogenous), is classified as a network operating system.
- The operating system running on loosely coupled processors, running tightly coupled

software (i.e., middleware software), is classified as a distributed operating system.

- The operating system running on tightly coupled processors, running tightly coupled software, is classified as a multiprocessor operating system.

Message-passing systems versus shared memory systems

- In Shared memory systems there is a (common) shared address space throughout the system.
- Communication among processors takes place via shared data variables, and control variables for synchronization (Semaphores and monitors) among the processors.
- If a shared memory is distributed then it is called distributed shared memory.
- All multicomputer (NUMA as well as message-passing) systems that do not have a shared address space hence communicate by message passing.

Emulating message-passing on a shared memory system (MP $\rightarrow$SM**)**

- The shared address space is partitioned into disjoint parts, one part being assigned to each processor.
- “Send” and “receive” operations are implemented for writing to and reading from the destination/sender processor’s address space, respectively.
- Specifically, a separate location is reserved as **mailbox** (assumed to have unbounded in size) for each ordered pair of processes.
- A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox.
- The write and read operations are controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received.

Emulating shared memory on a message-passing system (SM $\rightarrow$MP**)**

- This involves use of “send” and “receive” operations for “write” and “read” operations.
- Each shared location can be modeled as a separate process;
- “write” to a shared location is emulated by sending an update message to the corresponding owner process and a “read” by sending a query message.
- As accessing another processor’s memory requires send and receive operations, this emulation is expensive.
- In a MIMD message-passing multicomputer system, each “processor” may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory.
- Between two computers, the communication by message passing are more suited for wide-area distributed systems.

Primitives for distributed communication

Blocking/non-blocking, synchronous/asynchronous primitives

- Message send and receive communication primitives are denoted Send() and Receive(), respectively.
- A Send primitive has at least two parameters the destination, and the buffer in the user space, containing the data to be sent.
- a Receive primitive has at least two parameters – the source of the data, and the user buffer into which the data is to be received.
- There are two ways of sending data while Send is invoked – the buffered option and the unbuffered option.
- The buffered option - copies the data from user buffer to kernel buffer. The data later gets copied from kernel buffer onto the network.
- The unbuffered option - the data gets copied directly from user buffer onto the

network.

- For Receive, buffered option is required as the data has already arrived when the primitive is invoked, and needs a storage place in the kernel.
- The following are some definitions of blocking/non-blocking and synchronous/ asynchronous primitives:
- **Synchronous primitives** 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 other corresponding Receive primitive has also been completed. The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.
- **Asynchronous primitives** A Send primitive is said to be asynchronous if control returns back to the invoking process after the data item to be sent is copied out of the user-specified buffer.
- **Blocking primitives** A primitive is blocking if control returns to the invoking process after the processing completes.
- **Non-blocking primitives**

➢ A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even the operation has not completed.

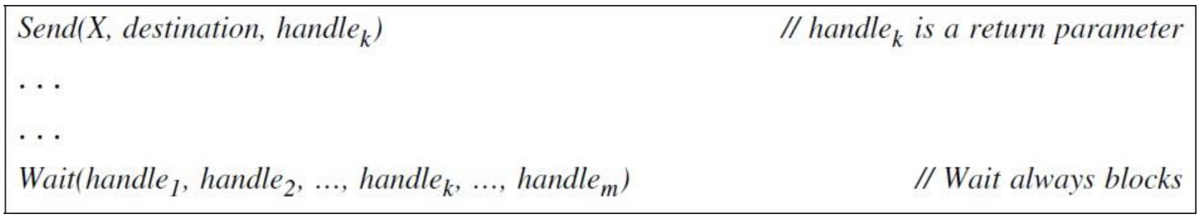
➢ For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer. For a non-blocking Receive, control returns to the process even before the data may have arrived from the sender.

➢ For non-blocking primitives, a return parameter of the 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 of the call in two ways.

1. keep checking (in loop or periodically), if the handle has been flagged or posted.
2. issue a Wait with a list of handles as parameters which will block until posted.

- The code for a non-blocking Send would look as shown in Figure 1.7.



Send$(X, destination, handle_k)$ // $handle_k$ *is a return parameter*

. . .

. . .

Wait$(handle_1, handle_2, \ldots, handle_k, \ldots, handle_m)$ // *Wait always blocks*

Figure 1.7 A non-blocking send primitive. When the Wait call returns, at least one of its parameters is posted

- If the processing of the primitive has not completed, the Wait blocks and waits for a signal to wake it up.
- When the processing for the primitive completes, the communication subsystem software sets the value of handle$_k$ and wakes up (signals) any process with a Wait call blocked on this handle$_k$. This is called posting the completion of the operation.

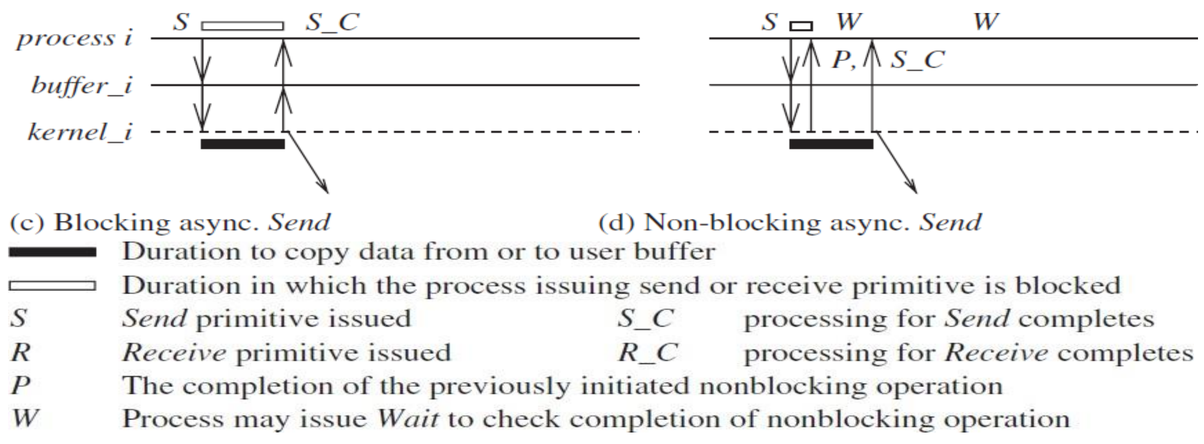
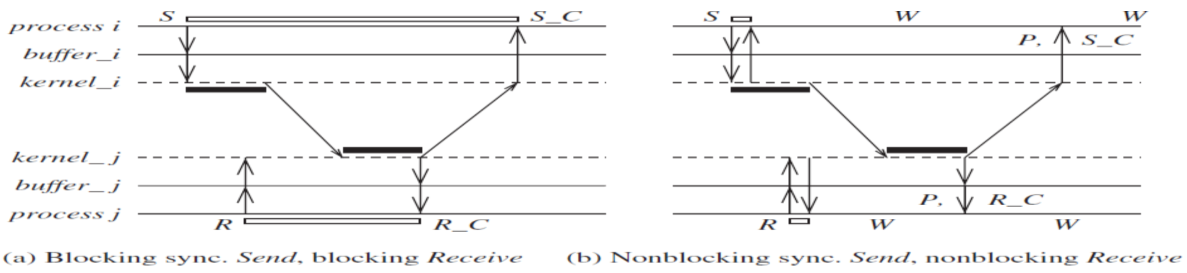

process i
buffer_i
kernel_i
S
S_C
W
P, S_C
(c) Blocking async. Send
(d) Non-blocking async. Send
Duration to copy data from or to user buffer
Duration in which the process issuing send or receive primitive is blocked
S Send primitive issued
S_C processing for Send completes
R Receive primitive issued
R_C processing for Receive completes
P The completion of the previously initiated nonblocking operation
W Process may issue Wait to check completion of nonblocking operation

Figure 1.8 Blocking/non-blocking and synchronous/asynchronous primitives. Process Pi is sending and process Pj is receiving. (a) Blocking synchronous Send and blocking (synchronous) Receive. (b) Non-blocking synchronous Send and nonblocking (synchronous) Receive. (c) Blocking asynchronous Send. (d) Non-blocking asynchronous Send.


process i
buffer_i
kernel_i
kernel_j
buffer_j
process j
S
S_C
R
R_C
W
P, S_C
P, R_C
(a) Blocking sync. Send, blocking Receive
(b) Nonblocking sync. Send, nonblocking Receive

- Here, three time lines are shown for each process: (1) for the process execution, (2) for the user buffer from/to which data is sent/received, and (3) for the kernel/communication subsystem.

- **Blocking synchronous Send** : The data gets copied from user buffer to 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 it copies the data from user buffer to kernel buffer is initiated.
  - A parameter in non-blocking call gets set with the handle of a location that a user process can check later for the completion of synchronous send operation.
- **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. For the unbuffered option, until the data is copied from the user's buffer to the network.
- **non-blocking asynchronous Send**
  - Send is blocked until the transfer of data from the user's buffer to the kernel buffer is initiated. For the unbuffered option, it is blocked until data gets transferred from user's buffer to network is initiated.
  - Control returns to the user process as soon as this transfer is initiated, and a parameter in non-blocking call gets set with the handle to check later using Wait operation for the completion of the asynchronous Send operation.

- **Blocking Receive**
  It 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**
  – It 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.
  – The user process can check for the completion of the non-blocking Receive by invoking the Wait operation on the returned handle.
- The non-blocking asynchronous Send is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the Send.
- The non-blocking synchronous Send also avoids the large delays for handshaking, particularly when the receiver has not yet issued the Receive call.
- The non-blocking Receive is useful when a large data item is being received and/or when the sender has not yet issued the Send call, because it allows the process to perform other instructions in parallel with the completion of the Receive.

Processor synchrony

- Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.
- As this synchrony difficult in distributed system, a large granularity of code, is termed as a step, the processors are synchronized.
- This synchronization ensures 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.

Libraries and standards

- There exists a wide range of primitives for message-passing.

1. Many commercial software products(banking, payroll applications) use proprietary primitive libraries supplied with software vendors (i.e., IBM CICS software).
2. The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community
3. Commercial software is often written using remote procedure calls (RPC) mechanism in which procedures that resides across the network are invoked transparently to the user, in the same manner that a local procedure is invoked.
4. socket primitives or socket-like transport layer primitives are invoked to call the procedure remotely.
5. There exist many implementations of RPC like Sun RPC, and distributed computing environment (DCE) RPC.
6. “Messaging” and “streaming” are two other mechanisms for communication.
7. For object based software, libraries for remote method invocation (RMI) and remote object invocation (ROI) is used.
8. CORBA (common object request broker architecture) and DCOM (distributed component object model) are two other standardized architectures with their own set of primitives.

Synchronous versus asynchronous executions

- An asynchronous execution is an execution in which (i) there is no processor synchrony and there is no bound on the drift rate of processor clocks, (ii) message delays (transmission + propagation times) are finite but unbounded, and (iii) there is no upper bound on the time taken by a process to execute a step.

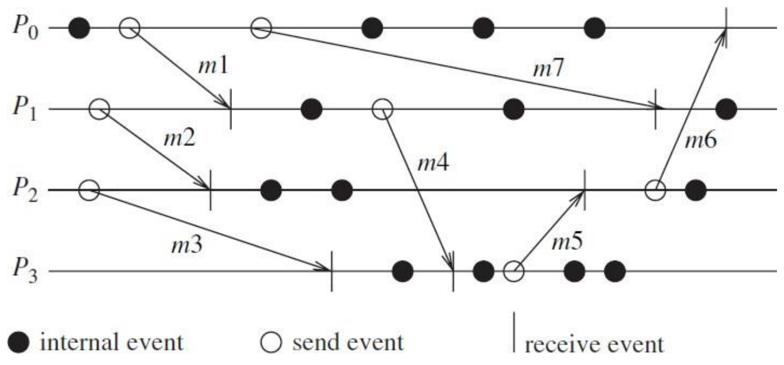


Figure 1.9 An example timing diagram of an asynchronous execution in a message-passing system.

- A synchronous execution is an execution in which (i) processors are synchronized and the clock drift rate between any two processors is bounded, (ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round, and (iii) there is a known upper bound on the time taken by a process to execute a step.

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

Ideally, many programs want the processes to execute a series of instructions in rounds (also termed as steps or phases) asynchronously, with the requirement that after each round/step/phase, all the processes should be synchronized and all messages sent should be delivered.

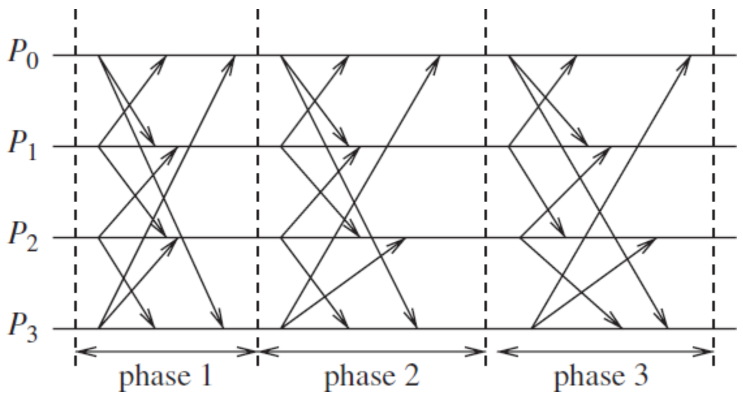


Figure 1.10 An example of a synchronous execution in a message-passing system. All the messages sent in a round are received within that same round.

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

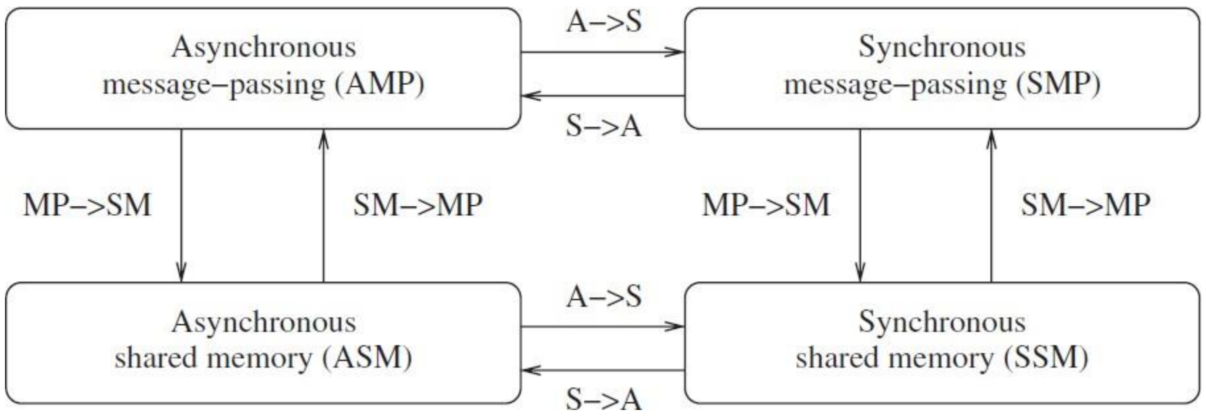
An asynchronous program can be emulated on a synchronous system as a special case of an asynchronous system – all communication finishes within the same round in which it is initiated.

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

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

Emulations

Using the emulations shown, any class can be emulated by any other. If system A can be emulated by system B, denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A. Likewise, if a problem is solvable in A, it is also solvable in B.
Hence, all four classes are equivalent in terms of "computability" i.e., what can and cannot be computed – in failure-free systems.



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

Figure 1.11 Emulations among the principal system classes in a failure-free system

Design issues and challenges

- The important design issues and challenges is categorized as
  - related to systems design and operating systems design, or
  - component related to algorithm design, or
  - emerging from recent technology

Distributed systems challenges from a system perspective

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

- **Communication** This task involves designing appropriate mechanisms for communication among the processes in the network. Example: remote procedure call (RPC), remote object invocation (ROI), etc.
- **Processes** Some of the issues involved are: management of processes and threads at clients/servers; code migration; and the 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. Naming in mobile systems provides additional challenges.
- **Synchronization** Mutual exclusion is an example of synchronization, other forms of synchronization are leader election and synchronizing physical clocks.
- **Data storage and access** Schemes for data storage, and implicitly for accessing the data in a fast and scalable manner across the network are important for efficiency.
- **Consistency and replication** To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable. This leads to issues of managing the replicas, and dealing with consistency among the replicas/caches in a distributed setting.
- **Fault tolerance** to maintaining correct and efficient operation in spite of any failures of links, nodes, and processes. Process resilience, reliable communication, distributed commit, and check pointing and recovery are some of the fault-tolerance mechanisms.
- **Security** Distributed systems security involves various aspects of cryptography, secure channels, access control, authorization, and secure group management.
- **Applications Programming Interface (API) and transparency** The API for

communication and other services for the ease of use and wider adoption of distributed systems services by non-technical users.

- Transparency deals with hiding the implementation policies from user, and is classified as follows:
  - Access transparency: hides differences in data representation on different systems and provides uniform operations to access system resources.
  - Location transparency: makes the locations of resources transparent to the users.
  - Migration transparency: allows relocating resources without changing names.
  - Concurrency transparency deals with masking concurrent use of shared resources for user.
  - Failure transparency: refers to the system being reliable and fault-tolerant.
- **Scalability and modularity** The algorithms, data (objects), and services must be as distributed as possible. Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

Algorithmic challenges in distributed computing

- The key algorithmic challenges in distributed computing is as summarized below:
- **Designing useful execution models and frameworks**
  - The interleaving model and partial order model are two widely adopted models of distributed system executions are useful for operational reasoning and the design of distributed algorithms.
  - The input/output automata model and the TLA (temporal logic of actions) are two other examples of models that provide different degrees of infrastructure for proving the correctness of distributed programs.
- **Dynamic distributed graph algorithms and distributed routing algorithms**
- The distributed system is modeled as a distributed graph.
- the graph algorithms form the building blocks for a large number of higher level communication, data dissemination, object location, and object search functions.
- The algorithms need to deal with dynamically changing graph characteristics, such as varying link loads, user-perceived latency, congestion in the network in a routing algorithm. Hence, the design of efficient distributed graph algorithms is important.
- **Time and global state in a distributed system**
- The challenges pertain to providing accurate physical time, and to providing a variant of time, called logical time.
- Logical time is relative time, and can (i) capture the logic and inter-process dependencies within the distributed program, and also (ii) track the relative progress at each process.
- Observing the global state of the system (across space) also involves the time dimension for consistent observation.

Synchronization/coordination mechanisms

- The processes must be allowed to execute concurrently, except when they need to synchronize to exchange information, i.e., communicate about shared data.
- Synchronization is essential for the distributed processes to overcome the limited observation of the system state from the viewpoint of any one process.
- The synchronization mechanisms is viewed as resource and concurrency management mechanisms to control the behavior of the processes.

- some examples of problems requiring synchronization:
  - **Physical clock synchronization** Physical clocks ususally diverge in their values 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 will play the role of a distinguished process – called a leader process which is necessary for many distributed algorithms to initiate some action like a broadcast or collecting the state of the system.
  - **Mutual exclusion** This is clearly a synchronization problem because access to the critical resource(s) has to be coordinated.
  - **Deadlock detection and resolution**
    - Deadlock detection needs coordination to avoid duplicate work, and
    - deadlock resolution needs coordination to avoid unnecessary aborts of processes.
  - **Termination detection** This requires cooperation among the processes to detect the specific global state of quiescence.
  - **Garbage collection** Garbage refers to objects that are no longer in use and that are not pointed to by any other process. Detecting garbage requires coordination among the processes.

Group communication, multicast, and ordered message delivery

- A group is a collection of processes on an application domain.
- Specific algorithms need to be designed to enable efficient group communication and group management wherein processes can join and leave groups dynamically, or even fail.
- When multiple processes send messages concurrently, different recipients may receive the messages in different orders that violates the semantics of distributed program. Hence, formal specifications for ordered delivery need to be formulated.

Monitoring distributed events and predicates

- Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state, and are useful for applications like debugging, sensing the environment, and in industrial process control hence for monitoring such predicates are important.
- An important paradigm for monitoring distributed events is that of event streaming.

Distributed program design and verification tools

- Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering.
- Designing mechanisms to achieve these design and verification goals is a challenge.

Debugging distributed programs

- debugging distributed programs is much harder because of the concurrency in actions and uncertainty due to the large number of possible executions defined by the interleaved concurrent actions.
- Adequate debugging mechanisms and tools need to be designed to meet this challenge.

Data replication, consistency models, and caching

- Fast access to data and other resources requires replication in the distributed system.
- Managing such replicas in the face of updates introduces consistency problems among the replicas and cached copies.
- Additionally, placement of the replicas in the systems is also a challenge.

World Wide Web design – caching, searching, scheduling

- The Web is an example of widespread distributed system with direct interface to the end user where the operations are read-intensive on most objects.
- The issues of object replication and caching has to be considered.
- Example: Prefetching can be used for subscribing of Content Distribution Servers.
- Minimizing response time to minimize user perceived latencies is an important challenge.
- Object search and navigation on the web are resource-intensive. Designing mechanisms to do this efficiently and accurately is a great challenge.

Distributed shared memory abstraction

- A shared memory abstraction deals only with read and write operations, and no message communication primitives.
- But the middleware layer abstraction has to be implemented using message-passing.
- Hence, in terms of overheads, the shared memory abstraction is not less expensive.
- **Wait-free algorithms**
  - Wait-freedom is defined as the ability of a process to complete its execution irrespective of the actions of other processes.
  - While wait-free algorithms are highly desirable and expensive hence it is a a challenge.
- **Mutual exclusion**
  - the Bakery algorithm and semaphores are used for mutual exclusion in a multiprocessing (uniprocessor or multiprocessor) shared memory setting.
- **Register constructions**
  - emerging technologies like biocomputing and quantum computing alter the present foundations of computer “hardware” design assumptions of memory access of current systems that are exclusively based on semiconductor technology and the von Neumann architecture.
  - The study of register constructions deals with the design of registers from scratch, with very weak assumptions on the accesses allowed to a register.
  - This forms a foundation for future architectures that allow concurrent access even to primitive units of memory (independent of technology) without any restrictions on the concurrency.
- **Consistency models**
  - For multiple copies of a variable/object, varying degrees of consistency among the replicas can be allowed.
  - a strict definition of consistency in a uniprocessor system would be expensive to implement in terms of high latency, high message overhead, and low concurrency.
  - But still meaningful models of consistency are desirable.

Reliable and fault-tolerant distributed systems

- A reliable and fault-tolerant environment has multiple requirements and aspects and addressed using various strategies:
- ➢ **Consensus algorithms**
  - It relies on message passing, and the recipients take actions based on the contents of the received messages.
  - It allows correct functioning of processes to reach agreement among themselves in spite of the existence of some malicious (adversarial) processes whose identities are not known to the correctly functioning processes.
- ➢ **Replication and replica management**
  - Replication ie., having backup servers is a classical method of providing fault-

tolerance.
    - The triple modular redundancy (TMR) technique is used in software as well as hardware installations.
- **Voting and quorum systems** Providing redundancy in the active (e.g., processes) or passive (e.g., hardware resources) components in the system and then performing voting based on some quorum criterion is a classical way of dealing with fault-tolerance. Designing efficient algorithms for this purpose is the challenge.
- **Distributed databases and distributed commit**
    - For distributed databases, the ACID properties of the transaction (atomicity, consistency, isolation, durability) need to be preserved in distributed setting.
    - The "transaction commit" protocols is a fairly mature area that can be applied for guarantees on message delivery in group communication in the presence of failures.
- **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.
    - Faults, internal or external to the program and system, may cause a bad state to arise in the execution.
    - A self-stabilizing algorithm is any algorithm that is guaranteed to eventually take the system to a good state even if a bad state were to arise due to some error.
    - Designing efficient self-stabilizing algorithms is a challenge.

➢ **Checkpointing and recovery algorithms**
- Checkpointing involves 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 one of the recently taken checkpoints.
- Checkpointing in a distributed environment is difficult because if the checkpoints at the different processes are not coordinated.

➢ **Failure detectors**
- In asynchronous distributed systems there is no bound on time for message transmission.
- Hence, it is impossible to distinguish a sent-but-not-yet-arrived message from a message that was never sent ie., alive or failed.
- Failure detectors represent a class of algorithms that probabilistically suspect another process as having failed and then converge on a determination of the up/down status of the suspected process.

➢ **Load balancing**
- The goal of load balancing is to gain higher throughput, and reduce the user perceived latency.
- Need: high network traffic or high request rate causing the network connection to be a bottleneck, or high computational load and to service incoming client requests with the least turnaround time.
- 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 the users by using idle processing power in the system more efficiently.

- **Real-time scheduling**
    - o Real-time scheduling is important for mission-critical applications, to accomplish the task execution on schedule.
    - o The problem becomes more challenging in a distributed system where a global view of the system state is absent.
    - o message propagation delays are hard to control or predict, which makes meeting real-time guarantees that are inherently dependent on communication among the processes harder.
- **Performance**
    - o Although high throughput is not the primary goal of using a distributed system, achieving good performance is important.
    - o In large distributed systems, network latency and access to shared resources can lead to large delays which must be minimized.

Applications of distributed computing and newer challenges

Mobile systems

- Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium.
- characteristics and issues are
    - – communication : range and power of transmission,
    - – engineering : battery power conservation, interfacing with wired Internet, signal processing and interference.
    - – computer science Perspective: routing, location management, channel allocation, localization and position estimation, and the overall management of mobility.
- There are two architectures for a mobile network.
    1. base-station approach are cellular approach,
    - – where a cell is the geographical region within range of a static and powerful base transmission station is associated with base station.
    - – All mobile processes in that cell communicate via the base station.
    2. ad-hoc network approach
    - – where there is no base station
    - – All responsibility for communication is distributed among the mobile nodes,
    - – Mobile nodes participate in routing by forwarding packets of other pairs of communicating nodes.
    - – Hence complex and poses many challenges.
- **Sensor networks**
    - – A sensor is a processor with an electro-mechanical interface i.e., capable of sensing physical parameters like temperature, velocity, pressure, humidity, and chemicals.
    - – Recent developments in cost-effective hardware technology made very large low-cost sensors.
    - – In event streaming, the streaming data reported from a sensor network differs from the streaming data reported by “computer processes”. This limits the nature of information about the reported event in a sensor network.
    - – Sensors have to self-configure to form an ad-hoc network, that creates a new set of challenges like position estimation and time estimation.

Ubiquitous or pervasive computing

- It is a class of computing where the processors embedded in the environment and perform computing appear anytime and everywhere like in sci-fi movies.

- The intelligent home, and the smart workplace are some example of ubiquitous environments currently under intense research and development.
- It is an distributed systems with wireless communication, sensor and actuator mechanisms. They are self-organizing, network-centric and resource constrained.
- It has small processors operating collectively in a dynamic network. The processors is connected to networks and processes the resources for collating data.

Peer-to-peer computing

- Peer-to-peer (P2P) computing represents computing over an application layer network wherein all interactions among the processors are at a "peer" level.
- all processors are equal and play a symmetric role in computation.
- P2P networks are typically self-organizing, and may or may not have a regular structure to the network.
- No central directories for name resolution and object lookup are allowed.
- key challenges include:
  - object storage mechanisms, efficient object lookup, and retrieval; dynamic reconfiguration as the nodes and objects join and leave the network randomly; anonymity, privacy, and security.

Publish-subscribe, content distribution, and multimedia

- There exists large amount of information, hence there is a greater need to receive and access only information of interest. Such information can be specified using filters.
- In a dynamic environment the information constantly fluctuates (stock prices), there needs to be an efficient mechanism for :
  (i) distributing this information (publish),
  (ii) to allow end users to indicate interest in receiving specific kinds of information (subscribe), and
  (iii) aggregating large volumes of information and filtering based on user's subscription.
- Content distribution refers to multimedia data, where the multimedia data is very large and information-intensive, requires compression, and often requires special synchronization during storage and playback.

Distributed agents

- Agents are software processes or robots that move around the system to do a specific task for which they are specially programmed.
- Agents collect and process information, and can exchange such information with other agents. It cooperate like an ant colony.
- Challenges: coordination mechanisms among the agents, controlling the mobility of the agents, and their software design and interfaces.

Distributed data mining

- Data mining algorithms examine large amounts of data to detect patterns and trends in the data, to mine or extract useful information.
- Example: examining the purchasing patterns of customers to enhance the marketing.
- The mining is done by applying database and artificial intelligence techniques to a data repository.
- In many situations, data is distributed and cannot be collected in a single repository like banking applications where the data is private and sensitive, or in atmospheric weather prediction where the data sets are far too massive to collect and process at a single repository in real-time.
- In such cases, efficient distributed data mining algorithms are required.

Grid computing

- It is the technology that utilizes the idle CPU cycles of machines connected to the network and make it available to others.
- Challenges: scheduling jobs in a distributed environment, to implementing quality of service and real-time guarantees, and security.

Security in distributed systems

Challenges are:

- confidentiality – only authorized processes can access information.
- Authentication – whether the information is from the correct source, identity
- Availability – maintaining allowed access to services despite malicious actions.
- Goal: meet these challenges with efficient and scalable solutions.

- For peer-to-peer, grid, and pervasive computing, these challenges are difficult because of the resource-constrained environment, a broadcast medium, the lack of structure and in the network.

A model of distributed Computations

- A distributed system consists of a set of processors that are connected by a communication network.

A distributed program

- A distributed program is composed of a set of n asynchronous processes p1, p2,..,pi,…,pn that communicate by message passing over the communication network.
- The communication delay is finite and unpredictable. The processes do not share a global memory and communicate by passing messages.
- Let Cij denote the channel from process pi to process pj and
- Let mij denote a message sent by pi to pj.
- Process execution and message transfer are asynchronous – ie., a process does not wait for the delivery of the message to be complete after sending it.
- The global state of a distributed computation is composed of the states of the processes and the communication channels.
- The state of a process is the state of its local memory and depends upon the context.
- The state of a channel is the set of messages in transit on the channel.

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,
    - message receive events.
- Let e_i^x denote the xth event at process p_i.
- For a message m, let send(m) & rec(m) denote send and receive events, respectively.
- 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 or receives and
    - the state of the channel on which the message is sent.
- An internal event only affects the process at which it occurs.
- The events at a process are linearly ordered by their order of occurrence.

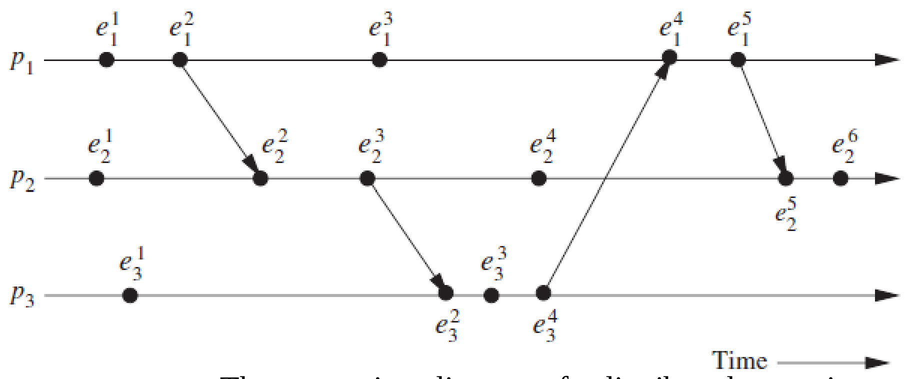
  The execution of process p_i produces a sequence of events $e_i^1, e_i^2, \ldots, e_i^x, e_i^{x+1}, \ldots$ and is denoted by $\mathcal{H}_i$:

$$\mathcal{H}_i = (h_i, \rightarrow_i),$$

where h_i is the set of events produced by p_i and binary relation $\rightarrow_i$ defines a linear order on these events. Relation $\rightarrow_i$ expresses causal dependencies among the events of p_i.

- For every message m that is exchanged between two processes, have Send(m)→msg rec(m)
- Relation →msg defines causal dependencies between send and receive events.
- Fig 2.1 shows a distributed execution using space–time diagram that involves three processes.

- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.



The space–time diagram of a distributed execution.

- In this figure, for process p1, the 2nd event is a message send event, the 3rd event is an internal event, and the 4th event is a message receive event.

Causal precedence relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let H =Uihi denote the set of events executed in a distributed computation.
- a binary relation on the set H is denoted as →, that expresses causal dependencies between events in the distributed execution.

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

- the relation → is Lamport's "happens before" relation
- For any two events ei and ej, if ei→ej , then event ej is directly or transitively

$e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$

dependent on event ei; in the figure 2.1,

- Note that relation → denotes flow of information in a distributed computation i.e., all information available at ei is accessible at ej
- For any two events ei and ej

$e_i \not\rightarrow e_j$

- denotes the fact that event ej does not directly or transitively dependent on event ei.
- For example in the figure 2.1:

$e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$

- Note the following rules:
  - for any two events e_i and e_j, $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$
  - for any two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.

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

Logical vs. physical concurrency

- Physical concurrency if and only if the events occur at the same instant in physical time.
- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other. For example, in Figure 2.1, events in the set $\{e3^1, e4^2, e3^3\}$ are logically concurrent, but they occurred at different instants in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not they does not change the outcome of the computation.

Models of communication networks

- models of the service provided by communication networks are:

1. In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
2. 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.
3. The "causal ordering" model is based on Lamport's "happens before" relation. A system that supports the causal ordering model satisfies the following property:
   CO: For any two messages m_{ij} and m_{kj}, if
   $send(m_{ij}) \rightarrow send(m_{kj})$ then
   $rec(m_{ij}) \rightarrow rec(m_{kj})$

- Causally ordered delivery of messages implies FIFO message delivery.
- Note that CO $\subset$ FIFO $\subset$ Non-FIFO.
- Causal ordering model is useful in developing distributed algorithms. Example:replicated database systems, every process that updates a replica must receives updates in the same order to maintain database consistency.

Global state of a distributed system

- The global state of a distributed system is a collection of the local states of its processes and the messages in the communication channels.
- The occurrence of events changes the states of respective processes and channels, thus causing transitions in global system state.
- LS_i^0 denotes the initial state of process p_i.
- LS_i^x is a result of the execution of all the events executed by process p_i till e_i^x.
- Let $send(m) \leq LS_i^x$ denote the fact that $\exists y{:}1\leq y\leq x :: e_i^y = send(m)$ and $rec(m) \not\leq LS_i^x$ denote the fact that $\forall y{:}1\leq y\leq x :: e_i^y \neq rec(m)$.
- Let $SCij^{x,y}$ denote the state of a channel Cij defined as follows:
- $SCij^{x,y} = \{mij \mid send(mij) \leq LSi^x \wedge rec(mij) \not\leq LSj^y\}$.
- Thus, channel state $SCij^{x,y}$ denotes all messages that pi sent up to event e_i^x and whichprocess pj had not received until event e_j^y.

Global state

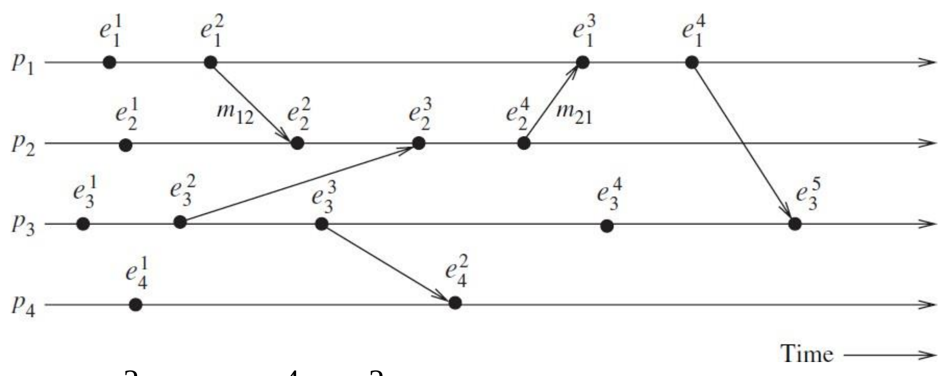
- The global state GS of a distributed system is a collection of the local states of the processes and the channels is defined as
  $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$
- For a global snapshot, the states of all the components of the distributed system must be recorded at the same instant. This is possible if the local clocks at processes were perfectly synchronized by the processes.
- Basic idea is that a message cannot be received if it was not sent i.e., the state should not violate causality. Such states are called consistent global states.
- Inconsistent global states are not meaningful in a distributed system.

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

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

global state GS consisting of local states

- {LS1^1, LS$_2^3$, LS3^3 , LS4^2} is inconsistent



- {LS1^2, LS ^{4}, LS3^4, LS4^2} is consistent;

(The space–time diagram of a distributed execution)

- A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{i,k} SC_{ik}^{y_j,z_k}\}$ is *transitless* iff

$$\forall i, \forall j \; : 1 \leq i, j \; \leq n \; :: SC_{ij}^{y_i,z_j} = \phi.$$

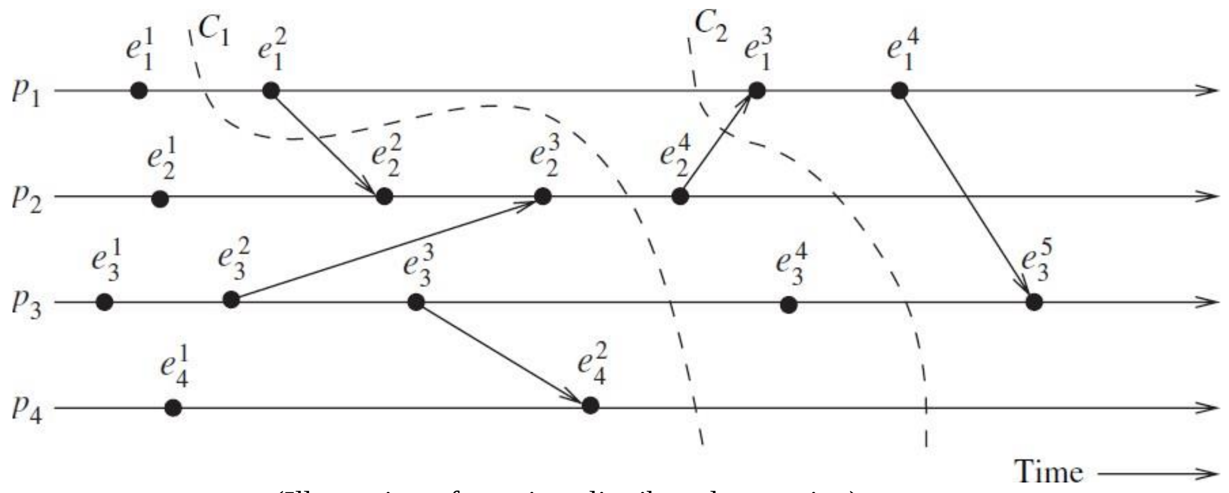
- A global state is strongly consistent iff it is transitless as well as consistent.
  - {LS1^2, LS2^3,LS3^4, LS4^2} is strongly consistent.

Cuts of a distributed computation

- In the space–time diagram of a distributed computation, a zigzag line joining one arbitrary point on each process line is termed as **cut** in the computation.
- The PAST(C) contains all the events to the left of the cut C and the FUTURE(C) contains all the events to the right of the cut C.
- Every cut corresponds to a global state .

definition 2.1 *If* $e_i^{Max\_PAST_i(C)}$ *denotes the latest event at process* p_i *that is in the* **PAST** *of a cut C, then the global state represented by the cut is* $\{\bigcup_i LS_i^{Max\_PAST_i(C)}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ where $SC_{jk}^{y_j,z_k} = \{m \mid send(m) \in \mathbf{PAST(C)} \wedge rec(m) \in \mathbf{FUTURE}(C)\}$.

- A consistent global state corresponds to a cut in which every message received in the PAST of the cut is sent in the PAST of that cut. Such a cut is known as a **consistent cut**.
- All messages that cross the cut from the PAST to FUTURE are in transit of consistent global state.
- A cut is **inconsistent** if a message crosses the cut from the FUTURE to PAST.



(Illustration of cuts in a distributed execution)

- Cuts in a space–time diagram is a powerful graphical aid to represent and reason about global states of a computation.

Past and future cones of an event

- Let Past(e_j) denote all events in the past of e_j in a computation (H,→).
  Then,Past (e_j) = { e_i |∀ e_i ∈ H, e_i → e_j }
- $Past_i(e_j)$ be the set of all those events of Past(e_j) that are on process p_i.
- Max_Past(e_j) = $\cup_{\forall i}\{max(Past_i(e_j))$. Max_$Past_i(e_j)$ consists of the latest event at everyprocess that affected event e_j called as the surface of the past cone of e_j.
- max(Pasti(ej)) is always a message send event.
- The future of an event e_j ,Future(e_j) contains all the events e_i that are causally affectedby e_j.
- In a computation (H, →), Future(e_j) is defined as:Future(e_j) = {e_i | ∀ e_i∈ H, e_j → e_i}
- $Future_i(e_j)$ as the set of events of Future(e_j) on process p_i and
- min(Futurei(ej)) as the first event on process pi that is affected by ej . It is always a message receive event.
- Min_Past(e_j) is $\cup_{\forall i}\{min\_Future_i(e_j))\}$, consists of first event at every process i.e., causally affected by event e_j is called the surface of the future cone of e_j.
- all events at a process p_i that occurred after max_$Past_i(e_j)$) but before min_$Future_i(e_j)$are concurrent with e_j .
- Therefore, all and only those events of computation H that belong to the set "H − Past(ej) − Future(ej)" are concurrent with event ej.

Models of process communications

- There are two models of process communications synchronous and asynchronous.
- **Synchronous communication model:**
  - It is a blocking type where the sender process blocks until the message received by the receiver process.
  - the sender and receiver processes must synchronize to exchange a message.
  - Synchronous communication is simple to handle and implement.
  - The frequent blocking lead to poor performance and deadlocks.
- **Asychronous Communication Model:**
  - It is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
  - After sent a message, the sender process buffers the message and is delivered to the receiver process when it is ready to accept the message.
  - A buffer overflow may occur if sender sends a large number of messages.
  - It provides higher parallelism as the sender executes while the message is in transit to the receiver.
  - Implementation of asynchronous communication requires complex buffer management.
  - Due to higher degree of parallelism and non-determinism, it is difficult to design, verify, and implement.

A framework for a system of logical clocks

Definition

- A system of logical clocks consists of a time domain T and a logical clock C.
- Elements of T form a partially ordered set over a relation < called as happened before

or causal precedence.

- The logical clock C is a function that maps an event e to the time domain T, denoted as C(e) and called the timestamp of e, and is defined as follows:

  $$C: H \mapsto T$$

- such that the following monotonicity property is satisfied then it is callled the clock consistency condition.

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

- When T and C satisfy the following condition, the system of clocks is said to be strongly consistent.

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

Implementing logical clocks

- Implementation of logical clocks requires addressing two issues:
- data structures local to every process to represent logical time and
- a protocol to update the data structures to ensure the consistency condition.
- Each process p_i maintains data structures with the following two capabilities:
  - A local logical clock, lc_i, that helps process p_i to measure its own progress.
  - A logical global clock, gc_i, represents the process p_i's local view of logical global time. It allows this process to assign consistent timestamps to its local events.
  - The protocol ensures that a process's logical clock, and thus its view of global time, is managed consistently.
  - The protocol consists of the following two rules:
  - **R1** This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).
  - **R2** This rule governs 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 it is used by the process to update its view of global time.

Scalar time

- **3.3.1 Definition**
- The scalar time representation was proposed by Lamport to totally order events in a distributed system. Time domain is represented as the set of non-negative integers.
- The logical local clock of a process p_i and its local view of global time are squashedinto one integer variable C_i.
- Rules **R1** and **R2** used to update the clocks is as follows:

  R1 : Before executing an event (send, receive, or internal), process pi executes:

  $C_i := C_i + d$ (d > 0)

- d can have a different value, may be application-dependent. Here d is kept at 1.

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

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

- Figure 3.1 shows the evolution of scalar time with d=1.

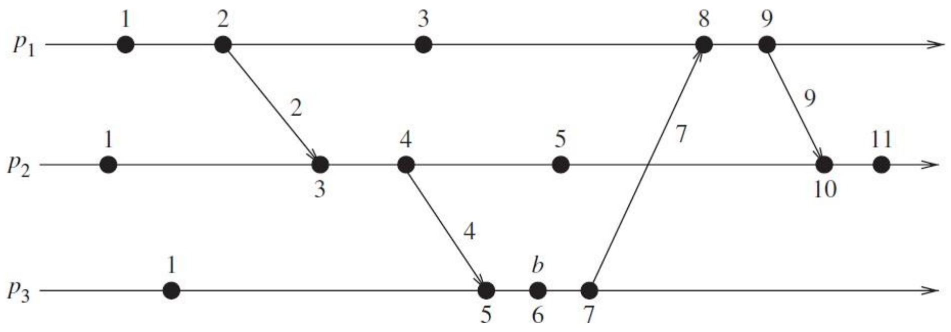


Figure 3.1 Evolution of scalar time

Basic properties

Consistency property

- scalar clocks satisfy the monotonicity and consistency property. i.e., for two events e_i and e_j,

$$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j).$$

Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- Problem in totally ordering events: Two or more events at different processes may have an identical timestamp. i.e., for two events e_1 and e_2, $C(e_1) = C(e_2) \Rightarrow e_1 || e_2$.
- In Figure 3.1, 3rd event of process P_1 and 2nd event of process P_2 have same scalar timestamp. Thus, a tie-breaking mechanism is needed to order such events.
- A tie among events with identical scalar timestamp is broken on the basis of their process identifiers. The lower the process identifier then it is higher in priority.
- The timestamp of an event is a tuple (t, i) where t - time of occurrence and i - identity of the process where it occurred. The total order relation $\prec$ on two events x and y with timestamps (h,i) and (k,j) is:

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

Event counting

- If the increment value of d is 1 then, if the event e has a timestamp h, then h−1 represents minimum number of events that happened before producing the event e;
- In the figure 3.1,five events precede event b on the longest causal path ending at b.

No strong consistency

- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j,

$$C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j .$$

- In Figure 3.1, the 3rd event of process P_1 has smaller scalar timestamp than 3rd eventof process P_2.

Vector time

Definition

- The system of vector clocks was developed Fidge, Mattern, and Schmuck.
- Here, the time domain is represented by a set of n-dimensional non-negative integer vectors.
- Each process p_i maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of $p_i$that specifies the progress at process.
- $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time.
- If $vt_i[j] = x$, then process p_i knowledge on process p_j till progressed x.
- The entire vector vt_i constitutes p_i's view of global logical time and is used to timestamp events.

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

  R1:
- Before executing an event, process pi updates its local logical time as follows:

  $vt_i[i] = vt_i[i] + d \ (d>0)$

  R2:
- Each message m is piggybacked with the vector clock vt of the sender process at sending time.
- On receipt of message (m,vt), process pi executes:

  1. update its global logical time as follows:

     $1 \leq k \leq n : vt_i[k] := \max(vt_i[k],$
     $vt[k])$
  2. execute **R1**;
  3. deliver the message m.
- The timestamp associated with an event is the value of vector clock of its process when the event is executed.
- The vector clocks progress with the increment value d = 1. Initially, it is [0, 0, 0, .. , 0].
- The following relations are defined to compare two vector timestamps, vh and vk:

  $vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$
  $vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$
  $vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$
  $vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)$

Basic properties
Isomorphism

- The relation "→" denotes partial order on the set of events in a distributed execution.
- If events are timestamped using vector clocks, then
- If two events x and y have timestamps vh and vk, respectively,

  then$x \rightarrow y \Leftrightarrow vh < vk$
  $x \parallel y \Leftrightarrow vh \parallel vk$
- Thus, there is an isomorphism between the set of partially ordered events and their vector timestamps.
- Hence, to compare two timestamps consider the events x and y occurred at processes pi and pj are assigned timestamps vh and vk, respectively, then

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

  Strong consistency
- The system of vector clocks is strongly consistent;
- Hence, by examining the vector timestamp of two events, it can be determined that whether the events are causally related.

Event counting

- If d is always 1 in rule **R1**, then the ith component of vector clock at process pi, vti[i], denotes the number of events that have occurred at pi until that instant.
- so, if an event e has timestamp vh, vh[j] denotes the number of events executed by process pj that causally precede e.
- $\Sigma vh[j]-1$ represents the total number of events that causally precede e in the distributed computation.

Applications

- As vector time tracks causal dependencies exactly, it's applications are as follows:
- distributed debugging,
- Implementations of causal ordering communication in distributed shared memory.
- Establishment of global breakpoints to determine consistency of checkpoints in recovery.

Linear Extension

- A linear extension of a partial order $(E,\prec)$ is a linear ordering of E i.e., consistent with partial order, if two events are ordered in the partial order, they are also ordered in the linear order. It is viewed as projecting all the events from the different processes on a single time axis.

Dimension

- The dimension of a partial order is the minimum number of linear extensions whose intersection gives exactly the partial order.

Physical clock synchronization: NTP

Motivation

- In centralized systems:
    - there is no need for clock synchronization because, there is only a single clock. A process gets the time by issuing a system call to the kernel.
    - When another process after that get the time, it will get a higher time value. Thus, there is a clear ordering of events and no ambiguity about events occurances.
- In distributed systems:
    - there is no global clock or common memory.
    - Each processor has its own internal clock and its own notion of time drift apart by several seconds per day, accumulating significant errors over time.
- For most applications and algorithms that runs in a distributed system requires:

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

- Example applications that need synchronization are: secure systems, fault diagnosis and recovery, scheduled operations, database systems.
- Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time.
- Due to different clocks rates, the clocks at various sites may diverge with time.
- To correct this periodically a clock synchronization is performed. Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).
- Clocks that are not synchronized with each other will adhere to physical time termed as physical clocks.

Definitions and terminology

Let Ca and Cb be any two clocks.

1. **Time** The time of a clock in a machine p is given by the function Cp(t), where Cp(t) = t for a perfect clock.

2. **Frequency** Frequency is the rate at which a clock progresses. The frequency at time t of clock Ca is Ca '(t).
3. **Offset** Clock offset is the difference between the time reported by a clock and the real time. The offset of the clock Ca is given by Ca(t)−t. The offset of clock Ca relative to Cb at time t ≥ 0 is given by Ca(t)−Cb(t).
4. **Skew** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock Ca relative to clock Cb at time t is Ca'(t)−Cb'(t).
If the skew is bounded by ρ, then as per Eq.(3.1), clock values are allowed to diverge at a rate in the range of 1−ρ to 1+ρ.
5. **Drift (rate)** The drift of clock Ca is the second derivative of the clock value with respect to time, namely, Ca''(t). The drift of clock Ca relative to clock Cb at time t is is Ca''(t)−Cb''(t).

Clock inaccuracies

- Physical clocks are synchronized to an accurate real-time standard like UTC. However, due to the clock inaccuracy, a timer (clock) is said to be working within its specification if

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

where constant ρ is the maximum skew rate.

Offset delay estimation method

- The Network Time Protocol (NTP) , is widely used for clock synchronization on theInternet, uses the offset delay estimation method.
- The design of NTP involves a hierarchical tree of time servers.
- The primary server at the root synchronizes with the UTC.
- The next level contains secondary servers, which act as a backup to the primary server.
- At the lowest level is the synchronization subnet which has the clients.

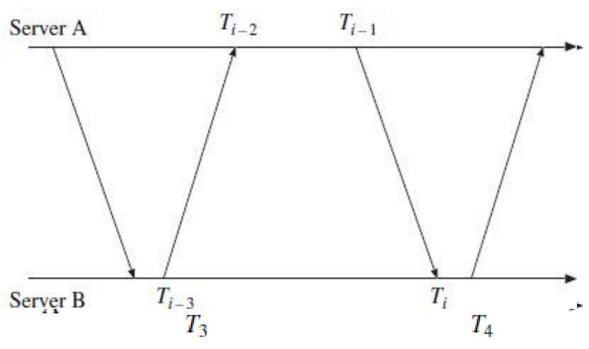
Clock offset and delay estimation

- This protocol performs several trials and chooses the trial with the minimum delay to accurately estimate the local time on the target node due to varying message or network delays between the nodes.
- Let T1,T2,T3,T4 be the values of the four most recent timestamps as shown in the figure.
- Assume that 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 delay δ of B relative to A at time T4 are approximately given by the following

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

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

Figure 3.10 Timing diagram for the two servers [15].



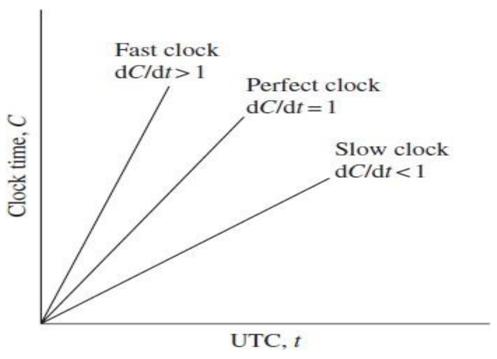
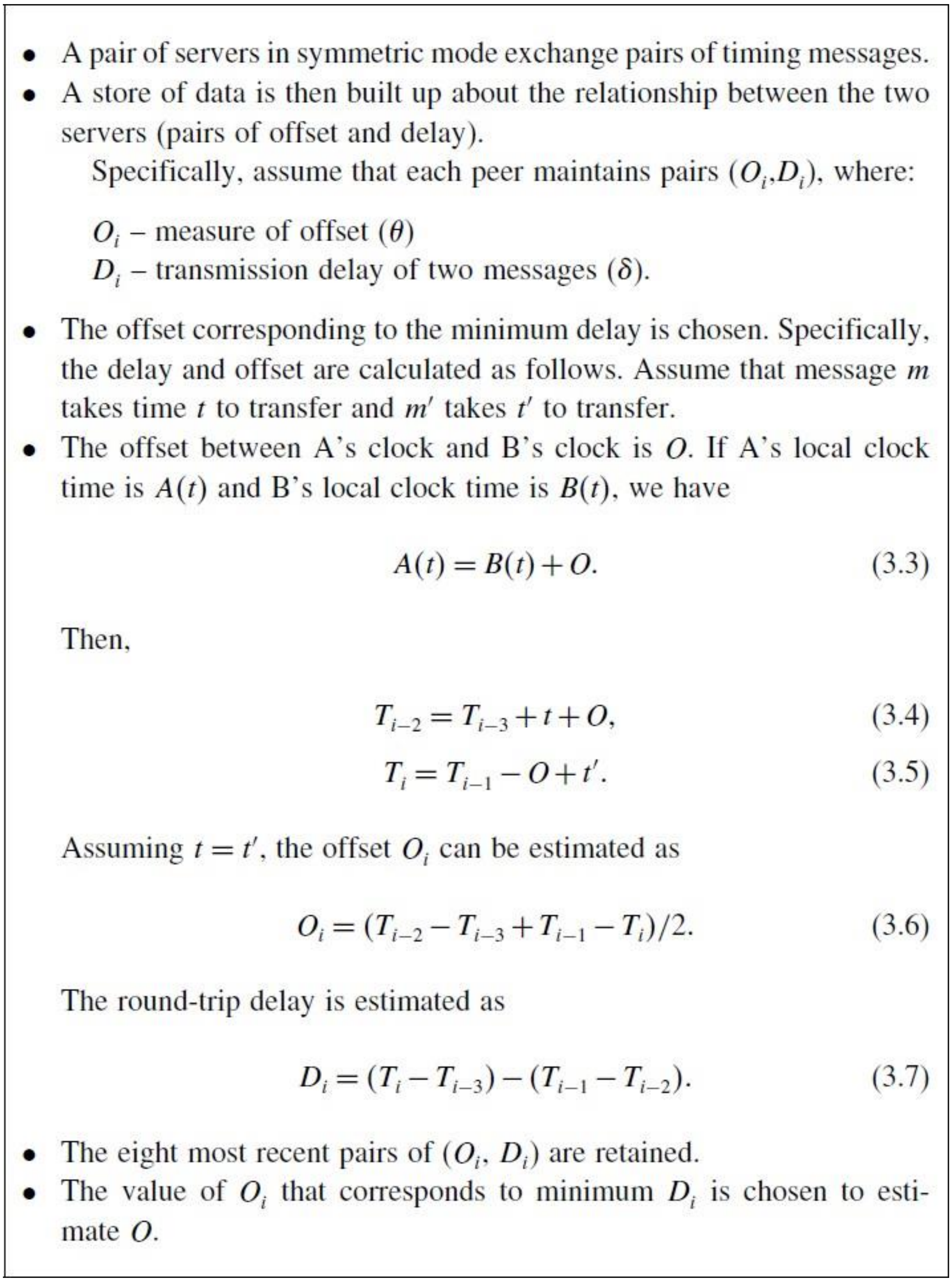


Figure :The Behaviour of fast, slow and perfect clocks with respect to UTC.



- A pair of servers in symmetric mode exchange pairs of timing messages.
- A store of data is then built up about the relationship between the two servers (pairs of offset and delay).

  Specifically, assume that each peer maintains pairs (O_i, D_i), where:

  O_i – measure of offset (θ)
  D_i – transmission delay of two messages (δ).

- The offset corresponding to the minimum delay is chosen. Specifically, the delay and offset are calculated as follows. Assume that message m takes time t to transfer and m' takes t' to transfer.
- The offset between A's clock and B's clock is O. If A's local clock time is $A(t)$ and B's local clock time is $B(t)$, we have

$$A(t) = B(t) + O. \tag{3.3}$$

Then,

$$T_{i-2} = T_{i-3} + t + O, \tag{3.4}$$

$$T_i = T_{i-1} - O + t'. \tag{3.5}$$

Assuming $t = t'$, the offset O_i can be estimated as

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2. \tag{3.6}$$

The round-trip delay is estimated as

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}). \tag{3.7}$$

- The eight most recent pairs of (O_i, D_i) are retained.
- The value of O_i that corresponds to minimum D_i is chosen to estimate O.

The network time protocol (NTP) synchronization protocol

UNIT II - MESSAGE ORDERING & SNAPSHOTS

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

Message Ordering and Group Communication

- For any two events a and b, where each can be either a send or a receive event, the notation
- a ~ b denotes that a and b occur at the same process, i.e., a ∈ E_i and b ∈ E_i for some process i. The send and receive event pair for a message called pair of corresponding events.
- For a given execution E, let the set of all send–receive event pairs be denoted as
- ↗= {(s,r) ∈ $E_i \times E_j$ | s corresponds to r}.

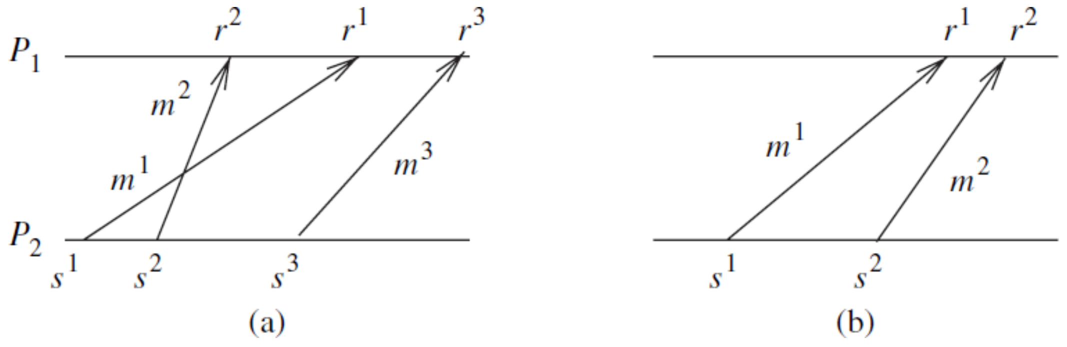
Message ordering paradigms

- Distributed program logic greatly depends on the order of delivery of messages.
- Several orderings on messages have been defined: (i) non-FIFO, (ii) FIFO, (iii) causal order, and (iv) synchronous order.

Asynchronous executions

Definition 6.1 (A-execution): An asynchronous execution (or A-execution) is an execution (E,≺) for which the causality relation is a partial order.

- On a logical link between two nodes (is formed as multiple paths may exist) in the system, if the messages are delivered in any order then it is known as non-FIFO executions. Example: IPv4.
- Each physical link delivers the messages sent on it in FIFO order due to the physical properties of the medium.



(6.1 Illustrating FIFO and non-FIFO executions. (a) An A-execution that is not a FIFO execution. (b) An A-execution that is also a FIFO execution.)

FIFO executions

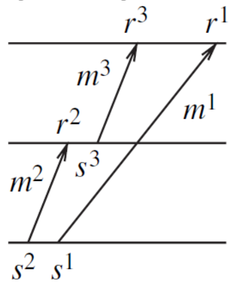
Definition 6.2 (FIFO executions) :A FIFO execution is an A-execution in which, for all (s,r) and (s′,r′) ∈ ↗ (s ∼ s′ and r ∼ r′ and s ≺s′) ⇒ r ≺r′.

- In general on any logical link, messages are delivered in the order in which they are sent.
- To implement FIFO logical channel over a non-FIFO channel, use a separate numbering scheme to sequence the messages.
- 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.
- Figure 6.1(b) illustrates an A-execution under FIFO ordering.

Causally ordered (CO) executions

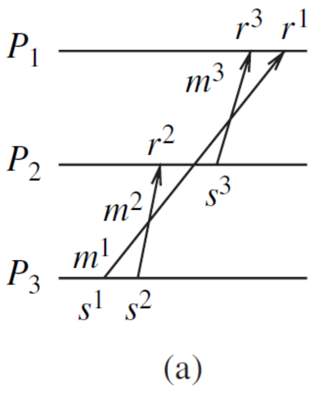
Definition 6.3 (Causal order (CO)): A CO execution is an A-execution in which,for all (s, r) and $(s', r') \in \mathcal{T}$, $(r \sim r'$ and $s \prec s') \Rightarrow r \prec r'$.

- If two send events s and s′ are related by causality ordering then their corresponding receive events r and r′ must occur in the same order at all common destinations.
- Figure 6.2 shows an execution that satisfies CO. s2 and s1 are related by causality but the destinations of the corresponding messages are different. Similarly for s2 and s3.



(Fig:6.2 CO executions)

- Applications of Causal order: applications that requires update to shared data, to implement distributed shared memory, and fair resource allocation in distributed mutual exclusion.
- **Definition 6.4 (causal order (CO) for implementations)** If $send(m^1) \prec send(m^2)$ then for each common destination d of messages m^1 and m^2, $deliver_d(m^1) \prec deliver_d(m^2)$ mustbe satisfied.
- if m^1 and m^2 are sent by the same process, then property degenerates to FIFO property.
- In a FIFO execution, no message can be overtaken by another message between the same (sender, receiver) pair of processes.
- In a CO execution, no message can be overtaken by a chain of messages between the same (sender, receiver) pair of processes.
- **Definition 6.5 (Message order (MO))**: A MO execution is an A-execution in which,for all (s, r) and $(s', r') \in \mathcal{T}$, $s \prec s' \Rightarrow \neg(r' \prec r)$.
- Example: Consider any message pair, say m^1 and m^3 in Figure 6.2(a). s1 $\prec$s3 but $\neg$ (r3 $\prec$r1) is false. Hence, the execution does not satisfy MO.



(6.2 a) Not a CO execution.

- **Definition 6.6 (Empty-interval execution)** An execution $(E,\prec)$ is an empty-interval (EI) execution if for each pair of events $(s, r) \in \nearrow$ the open interval set $\{x \in E \mid s \prec x \prec r\}$ in the partial order is empty.
- **Example** Consider any message, say m^2, in Figure 6.2(b). There does not exist any event x such that $s^2 \prec x \prec r^2$. This holds for all messages in the execution. Hence, the execution is EI.

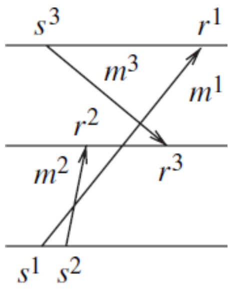


Figure:6.2(b) CO Execution

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

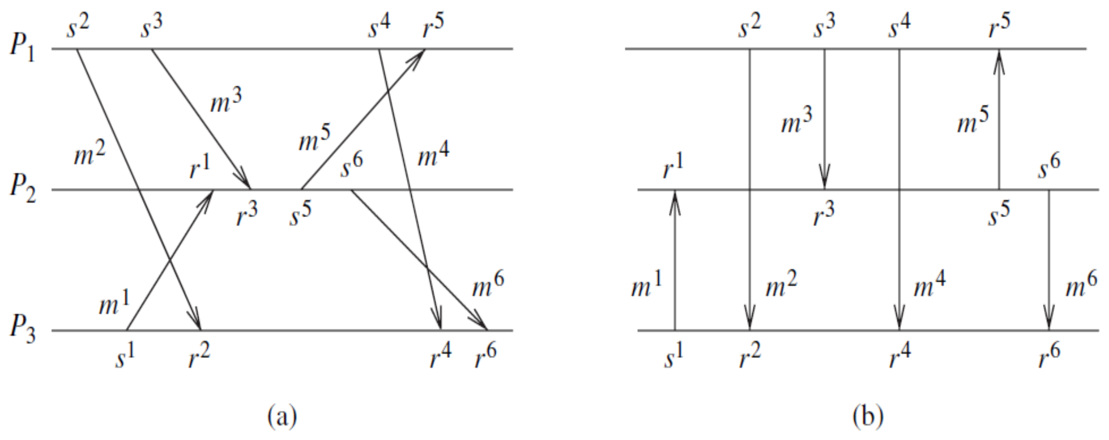
• weak common past: $e \prec r \Rightarrow \neg(s \prec e)$

• weak common future: $s \prec e \Rightarrow \neg(e \prec r)$.

Synchronous execution (SYNC)

- When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order.
- As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occuring instantaneously and atomically.
- In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines.
- The "instantaneous communication" property of synchronous executions requires that two events are viewed as being atomic and simultaneous, and neither event precedes the other.
- **Definition 6.7 (Causality in a synchronous execution)** The synchronous causality relation $\ll$ 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.
- We can now formally define a synchronous execution.



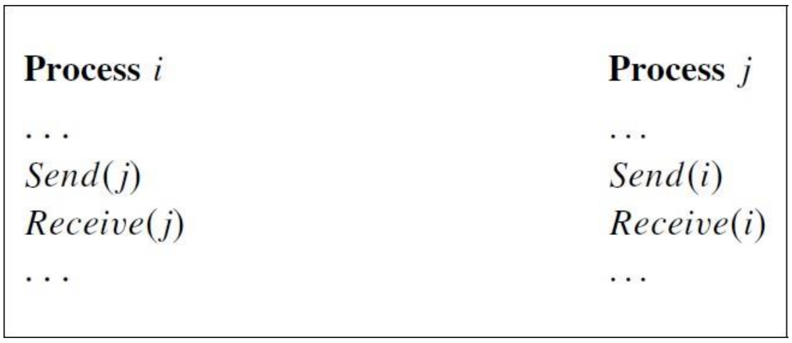
(Figure 6.3 Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b) Equivalent instantaneous communication.)

Definition (S- execution): A synchronous 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 Pi, if ei ≺ ei′ then T(ei) < T(ei′).

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.
- A distributed program that run correctly on an asynchronous system may not be executed by synchronous primitives. There is a possibility that the program may deadlock, as shown by the code in Figure 6.4.



```
Process i                        Process j
...                              ...
Send(j)                          Send(i)
Receive(j)                       Receive(i)
...                              ...
```

Figure 6.4 A communication program for an asynchronous system deadlocks when using synchronousprimitives.

- **Examples:** In Figure 6.5(a-c) using a timing diagram, will deadlock if run with synchronous primitives.

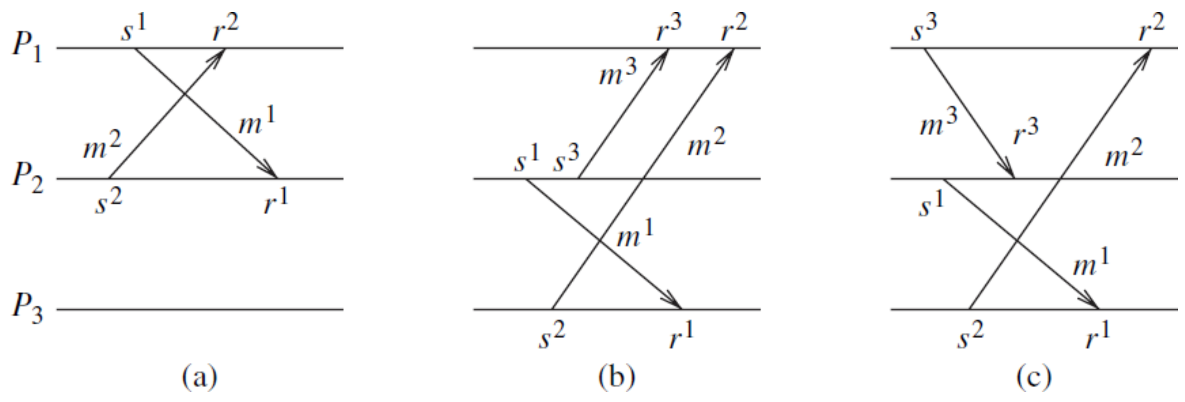
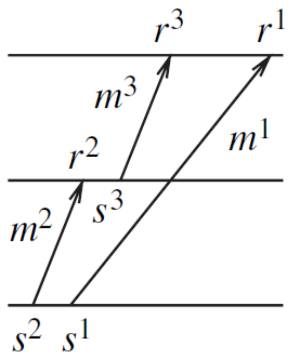


Figure 6.5 Illustrations of asynchronous executions and of crowns. (a) Crown of size 2. (b) Another crown of size 2. (c) Crown of size 3.

6.2.1 Executions realizable with synchronous communication (RSC)

- In an A-execution, the messages can be made to appear instantaneous if there exists a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event. Such an A-execution that is realized under synchronous communication is called a realizable with synchronous communication (RSC) execution.

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



(CO Executions)

Example:

- In the above figure: $<s^2, r^2, s^3, r^3, s^1, r^1>$ is a linear extension that is non separated.
  $<s^2, s^1, r^2, s^3, r^3, s^1>$ is a linear extension that is separated.

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

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

- On the set of messages T, we define an ordering $\hookrightarrow$ such that $m \hookrightarrow m'$ if and only if $s \prec r'$.

Synchronous program order on an asynchronous system

There do not exist real systems with instantaneous communication that allows for synchronous communication to be naturally realized.

Non-determinism

- This suggests that the distributed programs are deterministic, i.e., repeated runs of the same program will produce the same partial order.
- In many cases, programs are non-deterministic in the following senses
  1. A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.
  2. 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, it results in a deadlock.
- However, there is no semantic dependency between the send and immediately following receive. If the receive call at one of the processes is scheduled before the send call, then there is no deadlock.
- **Rendezvous**

Rendezvous ("meet with each other")

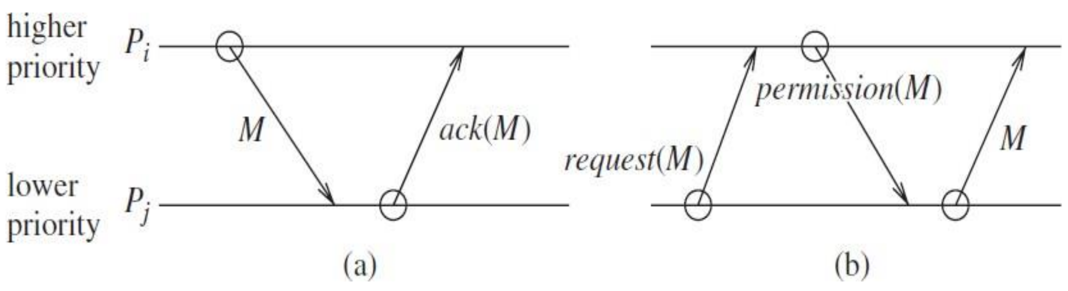
- One form of group communication is called multiway rendezvous, which is a synchronous communication among an arbitrary number of asynchronous processes.
- Rendezvous between a pair of processes at a time is called binary rendezvous as opposed to the multiway rendezvous.
- Observations about synchronous communication under binary rendezvous:
  - For the receive command, the sender must be specified eventhough the multiple recieve commands exist.
  - Send and received commands may be individually disabled or enabled.
  - Synchronous communication is implemented by scheduling messages using asynchronous communication.
- Scheduling involves pairing of matching send and receive commands that are both enabled.
- The communication events for the control messages do not alter the partial order of execution.

Algorithm for binary rendezvous

- Each process, has a set of tokens representing the current interactions that are enabled locally. If multiple interactions are enabled, a process chooses one of them and tries to "synchronize" with the partner process.
- The scheduling messages must satisfying the following constraints:
  - Schedule on-line, atomically, and in a distributed manner.
  - Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled.
  - Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.
- Additional features of a good algorithm are:
  (i) symmetry or some form of fairness, i.e., not favoring particular processes
  (ii) efficiency, i.e., using as few messages as possible
- A simple algorithm by Bagrodia, makes the following assumptions:
  1. Receive commands are forever enabled from all processes.

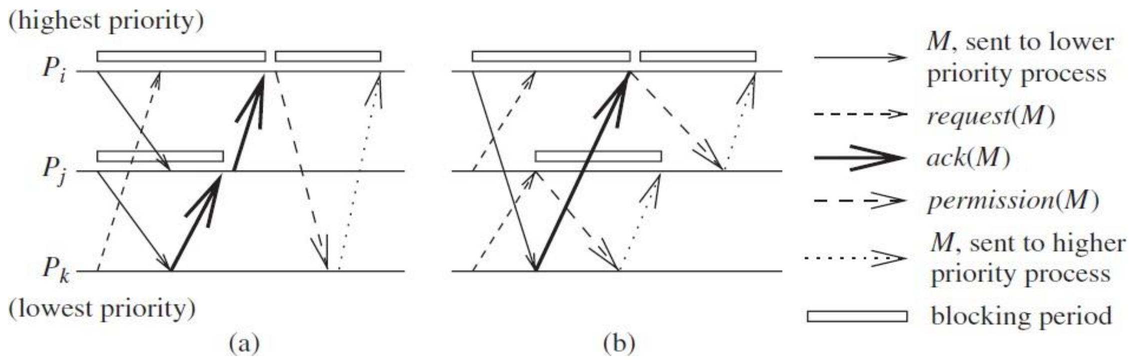
2. A send command, once enabled, remains enabled until it completes.
3. To prevent deadlock, process identifiers are used to break the crowns.
4. Each process attempts to schedule only one send event at any time.

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



Messages used to implement synchronous order. Pi has higher priority than Pj .
(a) Pi issues SEND(M). (b) Pj issues SEND(M).

- The message types used are:
(i) M – Message is the one i.e., exchanged between any two process during execution
(ii) ack(M) – acknowledgment for the received message M ,
(iii) request(M) – when low priority process wants to send a message M to the high priority process it issues this command.
(iv) permission(M) – response to the request(M) to low priority process from the high priority process.



(Examples showing how to schedule messages sent with synchronous primitives)

- A cyclic wait is prevented because before sending a message M to a higher priority process, a lower priority process requests the higher priority process for permission to synchronize on M, in a non-blocking manner.
- While waiting for this permission, there are two possibilities:
1. If a message M′ from a higher priority process arrives, it is processed by

a receive and ack(M′) is returned.Thus, a cyclic wait is prevented.
2. Also, while waiting for this permission, if a request(M′) from a lower priority process arrives, a permission(M′) is returned and the process blocks until M′ actually arrives.

Algorithm 6.1 A simplified implementation of synchronous order.
Code shown is for process Pi , $1 \leq i \leq n$.

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

(1) Pi **wants to execute SEND(**M**) to a lower priority process** Pj**:**

- Pi executes send(M) and blocks until it receives ack(M) from Pj. The send eventSEND(M) now completes.
- Any M′ message (from a higher priority processes) and request(M′) request for synchronization (from a lower priority processes) received during the blocking period are queued.

(2) Pi **wants to execute SEND(**M**) to a higher priority process** Pj**:**

(2a) Pi seeks permission from Pj by executing send(request(M)). (2b) While Pi is waiting for permission, it remains unblocked.

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

(ii) If a request(M′) arrives from a lower priority process Pk, Pi executes send(permission(M′)) to Pk and blocks waiting for the message M′. When M′arrives, the RECEIVE(M′) event is executed.

(2c) When the permission(M) arrives, Pi knows partner Pj is synchronized and Pi executes send(M). The SEND(M) now completes.

(3) **request(**M**) arrival at** Pi **from a lower priority process** Pj**:**

At the time a request(M) is processed by Pi, process Pi executes send(permission(M)) to Pj and blocks waiting for the message M. When M arrives, theRECEIVE(M) event is executed and the process unblocks.

(4) **Message M arrival at** Pi **from a higher priority process** Pj**:**

At the time a message M is processed by Pi, process Pi executes RECEIVE(M)(which is assumed to be always enabled) and then send(ack(M)) to Pj .

(5) **Processing when** Pi **is unblocked:**

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

Group communication

- Processes across a distributed system cooperate to solve a task. Hence there is need for group communication.
- A message broadcast is sending a message to all members.
- In Multicasting the message is sent to a certain subset, identified as a group.
- In unicasting is the point-to-point message communication.
- Broadcast and multicast is supported by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information.

- However, the hardware or network layer protocol assisted multicast cannot efficiently provide the following 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.
- If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a closed group algorithm.
- If the sender of the multicast can be outside the destination group, then the multicast algorithm is said to be an open group algorithm.
- Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms.
- Closed group algorithms cannot be used in in a large system like on-line reservation or Internet banking systems where client processes are short-lived and in large numbers.
- For multicast algorithms, the number of groups may be potentially exponential, i.e., $O(2^n)$.

Total order

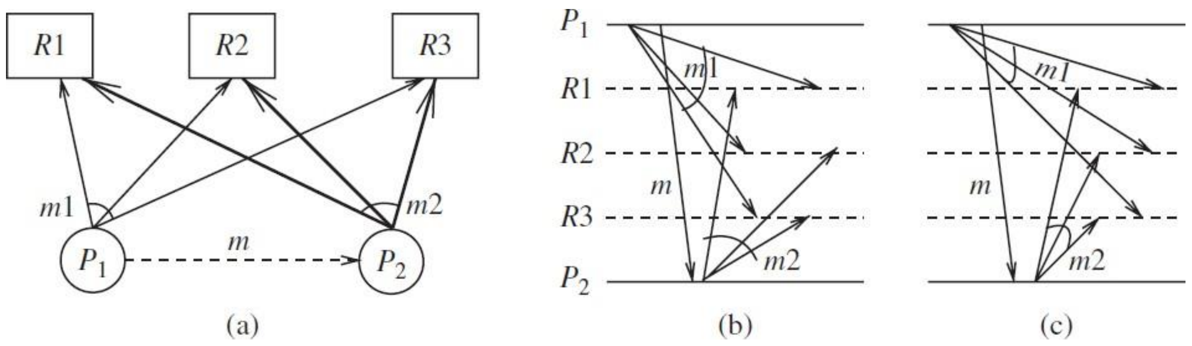
- For example of updates to replicated data would be logical only if all replicas see the updates in the same order.

Definition 6.14 (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.

Example

- The execution in Figure 6.11(b) does not satisfy total order. Even
- if the message m did not exist, total order would not be satisfied. The execution
- in Figure 6.11(c) satisfies total order.



Centralized algorithm for total order

- Algorithm Assumes all processes broadcast messages.
- It enforces total order and also the causal order in a system with FIFO channels.
- Each process sends the message it wants to broadcast to a centralized process.
- The centralized process relays all the messages it receives to every other process over FIFO channels.

Algorithm : centralized algorithm to implement total order & causal order of messages.

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

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

(2) When M(i,G) arrives from Pi at the central coordinator:
(2a) **send** M(i,G) to all members of the group G.

(3) When M(i,G) arrives at Pj 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

Three-phase distributed algorithm

- It enforces total and causal order for closed groups.
- The three phases of the algorithm are defined as follows:

Sender

Phase 1

- A process multicasts the message M to the group members with the
    - a locally unique tag and
    - the local timestamp

Phase 2

- Sender process awaits for the reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M.
- it is an non-blocking await i.e., any other messages received in the meanwhile are processed.
- Once all expected replies are received, the process computes the maximum of proposed timestamps for M, and uses the maximum as final timestamp.

Phase 3

- The process multicasts the final timestamp to the group members of phase 1.

Algorithm: Distributed algorithm to implement total order & causal order of messages. Code at Pi, 1 ≤ i ≤ n.

```
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 Pi, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts) // Phase 2 message sent by Pj , with revised timestamp, to
PiFINAL_TS(i, tag, ts)          // Phase 3 message sent by Pi, 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, tsj) from each process Pj ;
        (1e) ∀ j ∈ N, **do** temp_ts←max(temp_ts, tsj);
        (1f) **send** FINAL_TS(i, tag, temp_ts) to all
        processes;(1g) clock←max(clock, temp_ts).
(2) When REVISE_TS(M, j, tag, clk) arrives from Pj
        :(2a) priority←max_priority+1(clk);
        (2b) **insert** (M, tag, j, priority, undeliverable) in temp_Q;          // at end of queue
        (2c) **send** PROPOSED_TS(i, j, tag_ priority) to Pj .
(3) When FINAL_TS(j, x, clk) arrives from Pj :
        (3a) Identify entry Q_e in temp_Q, where Q_e.tag = x;
        (3b) **mark** Q_e.deliverable as true;
        (3c) Update Q_e.timestamp to clk and re-sort temp_Q based on the timestamp field;
        (3d) **if** (head(temp_Q)).tag = Q_e.tag **then**
        (3e)     **move** Q_e **from** temp_Q **to** delivery_Q;
        (3f)     **while** (head(temp_Q)).deliverable is true **do**
        (3g)             **dequeue** head(temp_Q) and insert in delivery_Q.
(4) When Pi removes a message (M, tag, j, ts, deliverable) from
        head(delivery_Qi):(4a) clock←max(clock, ts)+1.
```

Receivers

Phase 1

- The receiver receives the message with a tentative/proposed 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 (and the tag) back to the sender.
- The receiver then waits in a non-blocking manner for the final timestamp (correlated by the message tag).

Phase 3

- In the third phase, the final timestamp is received from the multicaster.
- 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.

- If the message entry is at the head of the temp_Q, that entry, and all consecutive subsequent entries that are also marked as deliverable, are dequeued from temp_Q, and enqueued in deliver_Q in that order.

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.

Example An example execution to illustrate the algorithm is given in Figure 6.14. Here, A and B multicast to a set of destinations and C and D are the common destinations for both multicasts.

- Figure 6.14a. The main sequence of steps is as follows:

1. A sends a REVISE_TS(7) message, having timestamp 7. B sends a REVISE_TS(9) message, having timestamp 9.
2. C receives A's REVISE_TS(7), enters the corresponding message in temp_Q, and marksit as undeliverable; priority = 7. C then sends PROPOSED_TS(7) message to A.
3. D receives B's REVISE_TS(9), enters the corresponding message in temp_Q, and marksit as undeliverable; priority = 9. D then sends PROPOSED_TS(9) message to B.
4. C receives B's REVISE_TS(9), enters the corresponding message in temp_Q, and marks it as undeliverable; priority = 9. C then sends PROPOSED_TS(9) message to B.
5. D receives A's REVISE_TS(7), enters the corresponding message in temp_Q, and marks it as undeliverable; priority = 10. D assigns a tentative timestamp value of 10, which is greater than all of the timestamps on REVISE_TSs seen so far, and then sends PROPOSED_TS(10) message to A.

The state of the system is as shown in the figure.

• **Figure** 6.14**(b)** The main steps is as follows:

6. When A receives PROPOSED_TS(7) from C and PROPOSED_TS(10) from D, it computes the final timestamp as max(7, 10) = 10, and sends FINAL_TS(10) to C and D.
7. When B receives PROPOSED_TS(9) from C and PROPOSED_TS(9) from D, it computes the final timestamp as max(9, 9)= 9, and sends FINAL_TS(9) to C and D.
8. C receives FINAL_TS(10) from A, updates the corresponding entry in temp_Q with the timestamp, resorts the queue, and marks the message as deliverable. As the message is not at the head of the queue, and some entry ahead of it is still undeliverable, the message is not moved to delivery_Q.
9. D receives FINAL_TS(9) from B, updates the corresponding entry in temp_Q by marking the corresponding message as deliverable, and resorts the queue. As the message is at the head of the queue, it is moved to delivery_Q.
10. When C receives FINAL_TS(9) from B, it will update the corresponding entry in temp_Q by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the delivery_Q, and the next message (of A), which is also deliverable, is also moved to the delivery_Q.

11. When D receives FINAL_TS(10) from A, it will update the corresponding entry in temp_Q by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the delivery_Q.

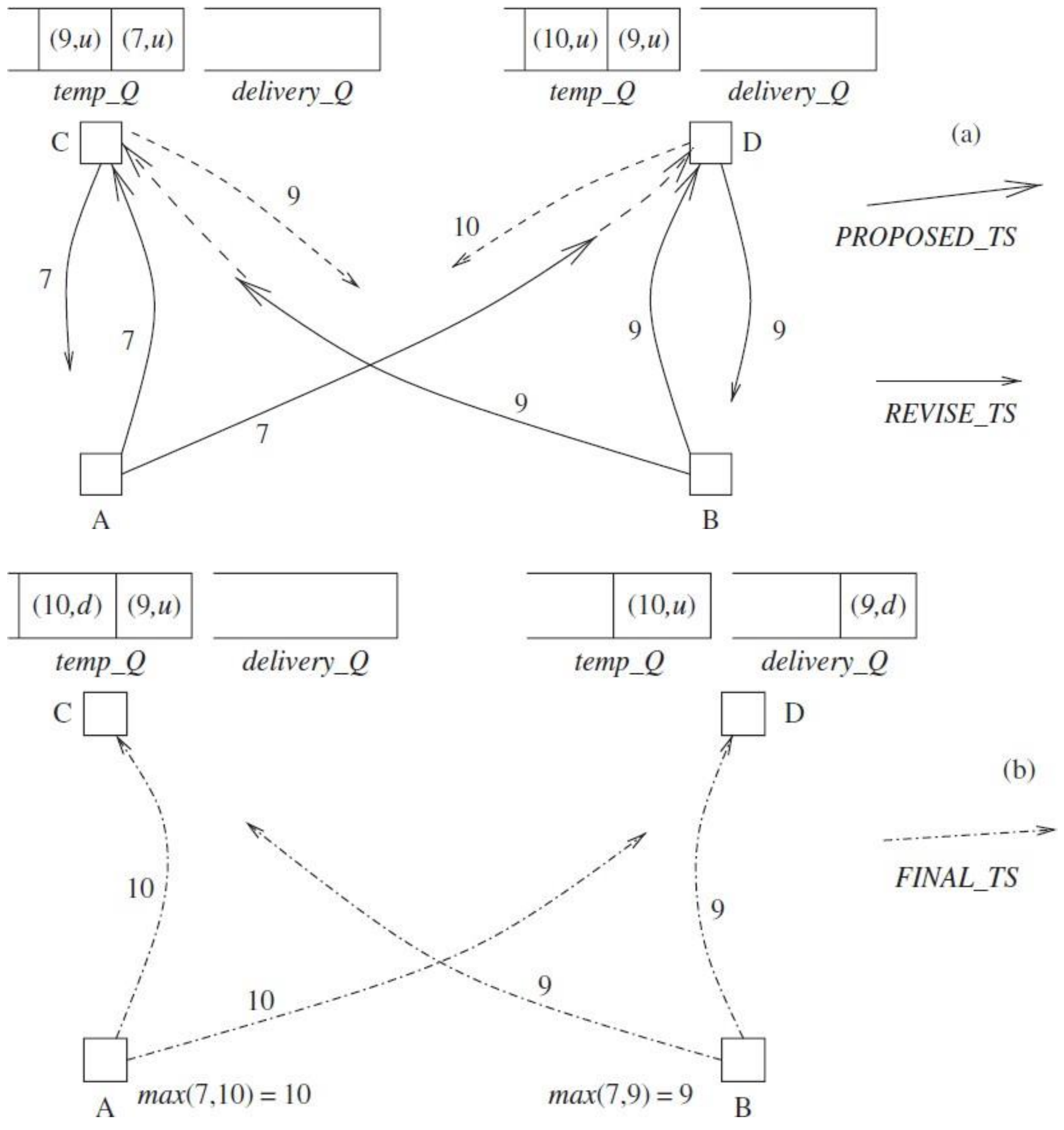


Figure 6.14 An example to illustrate the three-phase total ordering algorithm. (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages in the example.

Global state and snapshot recording Algorithms

Introduction

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

- Each component of a distributed system has a local state. The state of a process is the state of its local memory and a history of its activity.
- The state of a channel is the set of messages in the transit.
- The global state of a distributed system is the collection of states of the process and the channel.
- Applications that use the global state information are :
    - deadlocks detection
    - failure recovery,
    - for debugging distributed software
- If shared memory is available then an up-to-date state of the entire system is available to the processes sharing the memory.
- The absence of shared memory makes difficult to have the coherent and complete view of the system based on the local states of individual processes.
- A global snapshot can be obtained if the components of distributed system record their local states at the same time. This is possible if the local clocks at processes were perfectly synchronized or a global system clock that is instantaneously read by the processes.
- However, it is infeasible to have perfectly synchronized clocks at various sites as the clocks are bound to drift. If processes read time from a single common clock (maintained at one process), various indeterminate transmission delays may happen.
- In both cases, collection of local state observations is not meaningful, as discussed below.
- **Example:**
- Let S1 and S2 be two distinct sites of a distributed system which maintain bank accounts A and B, respectively. Let the communication channels from site S1 to site S2 and from site S2 to site S1 be denoted by C12 and C21, respectively.
- Consider the following sequence of actions, which are also illustrated in the timing
- diagram of Figure 4.1:
- Time t0: Initially, Account A=\$600, Account B=\$200, C12 =\$0, C21=\$0.
- Time t1: Site S1 initiates a transfer of \$50 from A to B. Hence,
  A= \$550, B=\$200, C12=\$50, C21=\$0.
- Time t2: Site S2 initiates a transfer of \$80 from Account B to A. Hence,
  A= \$550,B=\$120, C12 =\$50, C21=\$80.
- Time t3: Site S1 receives the message for a \$80 credit to Account A. Hence,
  A=\$630, B=\$120, C12 =\$50, C21 =\$0.
- Time t4: Site S2 receives the message for a \$50 credit to Account B. Hence,
  A=\$630, B=\$170, C12=\$0, C21=\$0.
- Suppose the local state of Account A is recorded at time t0 which is \$600 and the local state of Account B and channels C12 and C21 are recorded at time t2 are \$120, \$50, and \$80, respectively.
- Then the recorded global state shows \$850 in the system. An extra \$50 appears in the system.

- Reason: Global state recording activities of individual components must be coordinated.

System model and definitions

System model

- The system consists of a collection of n processes, $p_1, p_2, \ldots, p_n$, that are connected by channels.
- There is no globally shared memory and processes communicate solely by passing messages (send and receive) asynchronously i.e., delivered reliably with finite but arbitrary time delay.
- There is no physical global clock in the system.
- The system can be described as a directed graph where vertices represents processes and edges represent unidirectional communication channels.
- Let C_{ij} denote the channel from process p_i to process p_j .
- Processes and channels have states associated with them.
- Process State: is the contents of processor registers, stacks, local memory, etc., and dependents on the local context of the distributed application.
- Channel State of C_{ij}: is SC_{ij} , is the set of messages in transit of the channel.
- The actions performed by a process are modeled as three types of events,
- internal events – affects the state of the process.
- message send events, and
- message receive events.
- For a message m_{ij} that is sent by process pi to process p_j, let send(m_{ij}) and rec(m_{ij})denote its send and receive events affects state of the channel, respectively.
- The events at a process are linearly ordered by their order of occurrence.
- At any instant, the state of process pi, denoted by LS_i, is a result of the sequence of all the events executed by p_i up to that instant.
- For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of eventsthat have taken process p_i to state LS_i.
- For an event e and a process state LS_i, $e \notin LS_i$ iff e does not belong to the sequence ofevents that have taken process pi to state LS_i.
- For a channel C_{ij} , the following set of messages will be:
- **Transit** : $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j \}$
- There are several models of communication among processes.
- In the FIFO model, each channel acts as a first-in first-out message queue hence, 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.
- In causal delivery of messages satisfies the following property:
  "for any two messages m_{ij} and m_{kj},
  if $send(m_{ij}) \rightarrow send(m_{kj})$, then $rec(m_{ij}) \rightarrow rec(m_{kj})$."
- Causally ordered delivery of messages implies FIFO message delivery.

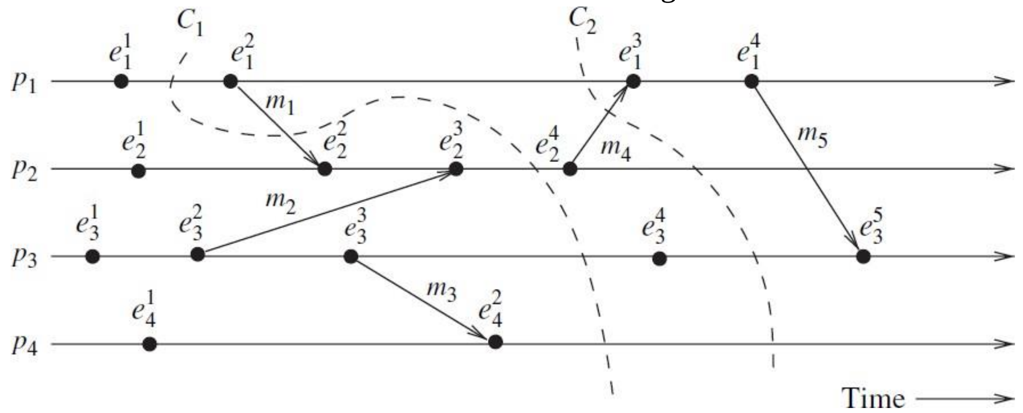
- The causal ordering model is useful in developing distributed algorithms and may simplify the design of algorithms.

A consistent global state

- The global state of a distributed system is a collection of the local states of
- the processes and the channels. Notationally, global state GS is defined as
  GS ={∪iLSi, ∪i,jSCij}.
- A global state GS is a consistent global state iff it satisfies the following two conditions:
  C1: send(mij)∈LSi ⇒ mij ∈SCij ⊕ rec(mij)∈LSj (⊕ is the Ex-OR operator).
  C2: send(mij) ∉LSi ⇒ mij ∉ SCij ∧ rec(mij) ∉ LSj .
- In a consistent global state, every message that is recorded as received is also recorded as sent. These are meaningful global states.
- The inconsistent global states are not meaningful ie., without send if receive of the respective message exists.

Interpretation in terms of cuts

- Cuts is a zig-zag line that connects a point in the space–time diagram at some arbitrary point in the process line.
- Cut is a powerful graphical aid for representing and reasoning about the global states of a computation.
- Left side of the cut is referred as PAST event and right is referred as FUTURE event.



- 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. Example: Cut C2 in the above figure.
- All the messages that cross the cut from the PAST to the FUTURE are captured in the corresponding channel state.
- If the flow is from the FUTURE to the PAST is inconsistent. Example: Cut C1.

Issues in recording a global state

- If a global physical clock is used then the following simple procedure is used to record a consistent global snapshot of a distributed system.
  - Initiator of the snapshot decides a future time at which the snapshot is to be taken and broadcasts this time to every process.
  - All processes take their local snapshots at that instant in the global time.

  - The snapshot of channel C_{ij} includes all the messages that process p_j receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot.

- However, a global physical clock is not available in a distributed system. Hence the following two issues need to be addressed to record a consistent global snapshot.
- **I1:** How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?
- Any message i.e., sent by a process before recording its snapshot, must be recorded in the global snapshot. (from **C1**).
- Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).
- **I2:** How to determine the instant when a process takes its snapshot.
- A process p_j must record its snapshot before processing a message m_{ij} that was sent byprocess p_i after recording its snapshot.
- These algorithms use two types of messages: computation messages and control messages. The former are exchanged by the underlying application and the latter are exchanged by the snapshot algorithm.

Snapshot algorithms for FIFO channels

Chandy–Lamport algorithm

- This algorithm uses a control message, called a marker.
- After a 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, marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot. This addresses issue **I1**.
- The role of markers in a FIFO system is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition **C2**.
- Since all messages that follow a marker on channel C_{ij} have been sent by process p_i after p_i has taken its snapshot, process p_j must record its snapshot if not recorded earlierand record the state of the channel that was received along the marker message. This addresses issue **I2**.
- **The algorithm**
- The algorithm is initiated by any process by executing the marker sending rule.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- **Algorithm 4.1** The Chandy–Lamport algorithm.

Marker sending rule for process p_i

```
(1) Process $p_i$ records its state.
(2) For each outgoing channel C on which a marker
    has not been sent, $p_i$ sends a marker along C
```

```
        before $p_i$ sends further messages along C.

    Marker receiving rule for process pj
    On receiving a marker along channel C:
        if $p_j$ 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 $p_{j,s}$ state was recorded
            and before $p_j$ received the marker along C
```

Correctness

- To prove the correctness of the algorithm, it is shown that a recorded snapshot satisfies conditions **C1** and **C2**.
- 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.
- Moreover, a process stops recording the state of an incoming channel when a marker is received on that channel.
- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij}, it actsas follows:
- If process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij}. Thus, condition **C1** is satisfied.

Complexity

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

UNIT III DISTRIBUTED MUTEX & DEADLOCK

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

DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS: INTRODUCTION

- Mutual exclusion is a fundamental problem in distributed computing systems.
- Mutual exclusion ensures that concurrent access of processes to a shared resource or data is serialized, that is, executed in mutually exclusive manner.

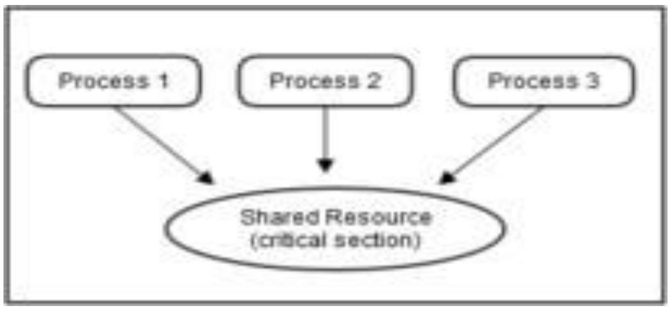


Figure 1: Three processes accessing a shared resource (critical section) simultaneously.

- 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.

There are three basic approaches for implementing distributed mutual exclusion:

1. **Token based approach**
2. **Non-token based approach**
3. **Quorum based approach**

1. In the **token-based approach**, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over. Mutual exclusion is ensured because the token is unique.

2. In the **non-token based approach**, two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next. A site enters the **Critical Section (CS)** when an assertion, defined on its local variables, becomes true. Mutual Exclusion is enforced because the assertion becomes true only at one site at any given time.

3. In the **quorum-based approach**, each site requests permission to execute the CS from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites

concurrently request access to the CS, one site receives both the requests and which is responsible to make sure that only one request executes the CS at any time.

OBJECTIVES OF MUTUAL EXCLUSION ALGORITHMS

1. **Guarantee mutual exclusion** (required)
2. **Freedom from deadlocks** (desirable)
3. **Freedom from starvation** -- every requesting site should get to enter CS in a finitetime (desirable)
4. **Fairness** -- requests should be executed in the order of arrivals, which would be based on logical clocks (desirable)
5. **Fault tolerance** -- failure in the distributed system will be recognized and therefore not cause any unduly prolonged disruptions (desirable)

PRELIMINARIES

We describe here,

1. **System model,**
2. **Requirements that mutual exclusion algorithms**
3. **Metrics we use to measure the performance of mutual exclusion algorithms.**

1. SYSTEM MODEL

- The system consists of N sites, **S1, S2, ..., SN.** We assume that a single process is running on each site.
- The process at site Si is denoted by pi.
- 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:
  1. **Requesting the Critical Section.**
  2. **Executing the Critical Section.**
  3. **Neither requesting nor executing the CS (i.e., idle).**
- In the ‘requesting the CS’ state, the site is blocked and can not make further requests for the CS. In the ‘idle’ state, the site is executing outside the CS.
- In 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.

Note:

- We assume that channels reliably deliver all messages, sites do not crash, and the network does not get partitioned.

- Some mutual exclusion algorithms are designed to handle such situations. Many algorithms use Lamport-style logical clocks to assign a timestamp to critical section requests.
- Timestamps are used to decide the priority of requests in case the of a conflict.
- A general rule followed is that the **smaller the timestamp of a request**, the **higher itspriority to execute the CS.**
- We use the following notations:
    - **N** denotes the **number of processes or sites** involved in invoking the criticalsection,
    - **T** denotes the average **Message Time Delay**,
    - and **E** denotes the average critical section **Execution Time**.

2. REQUIREMENTS OF MUTUAL EXCLUSION ALGORITHMS

A mutual exclusion algorithm should satisfy the following properties:

a. 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.
b. Liveness Property: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive. 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.
c. Fairness: Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS.

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

3. PERFORMANCE METRICS

The performance of mutual exclusion algorithms is generally measured by the following four metrics:
a. Message complexity: It is the number of messages that are required per CS execution by a site.
b. Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS (sees Figure 9.1).

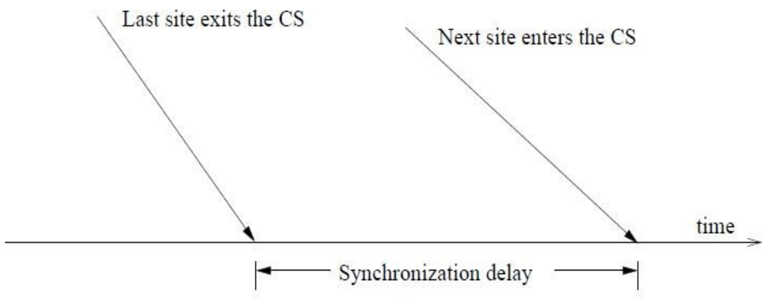


Figure 9.1: Synchronization Delay

c. Response time: It is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2).

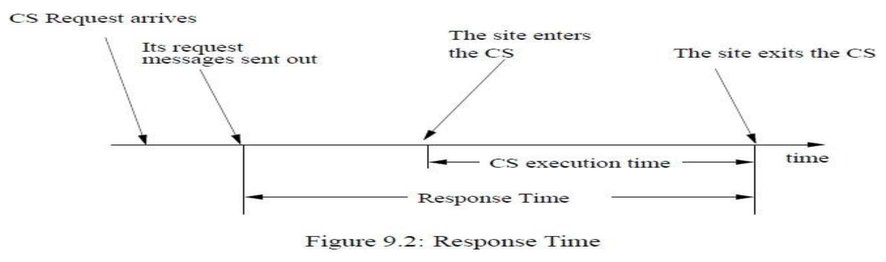


Figure 2: Response Time

d. System throughput: It 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, then the throughput is given by the following equation:

System Throughput=1/(SD+E)

Generally, the value of a performance metric fluctuates statistically from request to request and we generally consider the average value of such a metric.

Low and High Load Performance: The load is determined by the arrival rate of CS execution requests. Two special loading conditions, viz., **"low load"** and **"high load".**

- Under **low load** conditions, there is seldom <u>more than one request for the critical section present in the system simultaneously</u>.
- Under **heavy load** conditions, <u>there is always a pending request for critical section ata site</u>.

.

LAMPORT'S ALGORITHM

- The algorithm is fair in the sense that a request for CS is executed in the order of their timestamps and time is determined by logical clocks.
- When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp.
- The algorithm executes CS requests in the increasing order of timestamps.
- Every site Si keeps a queue, **request_queuei**,

This algorithm requires communication channels to deliver messages the FIFO order.

<u>**The Algorithm**</u>

1. 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(tsi, i)** message from site Si, places site Si'sRequest on request_queuej and it returns a time stamped REPLY message to Si.

2. Executing the critical section:

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

L1: Si has received a message with timestamp larger than (tsi, i) from all other sites.

L2: Si's request is at the top of request_queuei.

3. Releasing the critical section:

- Site Si, upon exiting the CS, removes its request from the top of its request queue and broadcasts a time stamped RELEASE message to all other sites.

- When a site Sj receives a RELEASE message from site Si, it removes Si's request from its request queue. When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. Clearly, when a site receives a REQUEST, REPLY or RELEASE message, it updates its clock using the timestamp in the message.

Correctness

Theorem 1: Lamport's algorithm achieves mutual exclusion.

Proof: **Proof is by contradiction**. Suppose two sites Si and Sj are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites concurrently. This implies that at some instant in time, say t, both Si and Sj have their own requests at the top of their request_queues and condition L1 hold at them. Without loss of generality, assume that Si's request has smaller timestamp than the request of Sj. From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of Si must be present in request_queuej when Sj was executing its CS. This implies that Sj's own request is at the top of its own request_queue when a smaller timestamp request, Si's request, is present in there quest_queuej – a contradiction!! Hence, Lamport's algorithm achieves mutual exclusion.

Theorem 2: Lamport's algorithm is fair.

Proof: A distributed mutual exclusion algorithm is fair if the requests for CS are executed in the order of their timestamps. The proof is by contradiction. Suppose a site Si's request has a smaller timestamp than the request of another site Sj and Sj is able to execute the CS before Si. For Sj to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, Sj has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites. But request_queueat a site is ordered by timestamp, and according to our assumption Si has lower timestamp. So Si's request must be placed ahead of the Sj's request in the request_queuej . This is a contradiction. Hence Lamport’s algorithm is a fair mutual exclusion algorithm.

An Example

In Figures 9.3 to 9.6, we illustrate the operation of Lamport’s algorithm. In Figure 9.3, sites S1 andS2 are making requests for the CS and send out REQUEST messages to other sites. The time stamps of the requests are (1, 1) and (1, 2), respectively. In Figure 9.4, both the sites S1 and S2 have received REPLY messages from all other sites. S1 has its request at the top of its request_queue but site S2 does not have its request at the top of its request_queue. Consequently, site S1 enters the CS. In Figure 9.5, S1 exits and sends RELEASE messages to allother sites. In Figure 9.6, site S2 has received REPLY from all other sites and also received a RELEASE message from siteS1. Site S2 updates its request_queue and its request is now at thetop of its request_queue. Consequently, it enters the CS next.

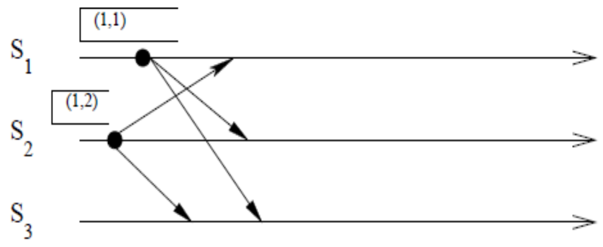


Figure 9.3: Sites S_1 and S_2 are Making Requests for the CS.

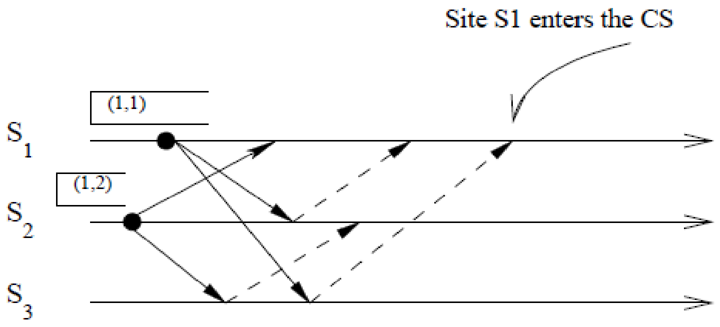


Figure 9.4: Site S_1 enters the CS.

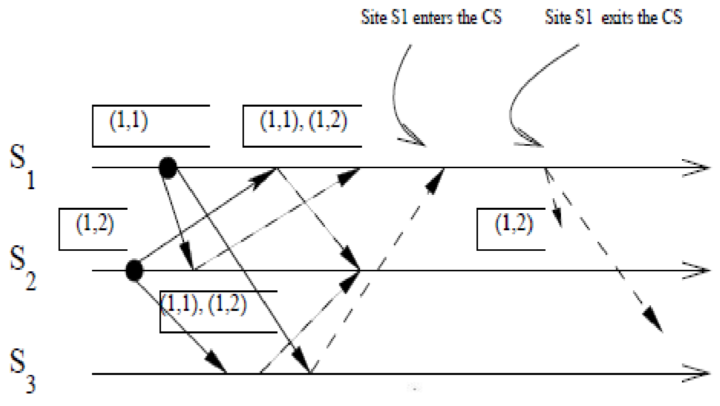


Figure 9.5: Site S_1 exits the CS and sends RELEASE messages.

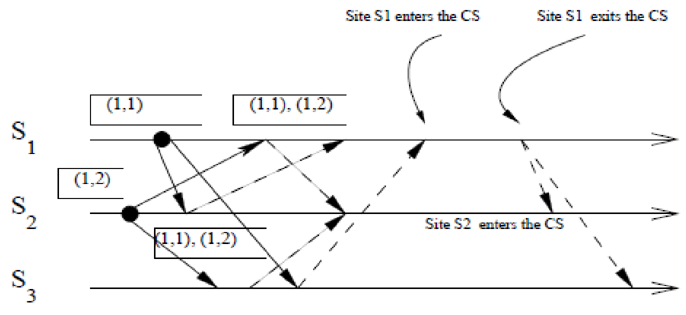


Figure 9.6: Site S_2 enters the CS

Performance

for each CS invocation

(N-1)
REQUEST
(N-1)
REPLY
(N-1) RELEASE,

Total 3(N-1) messages, synchronization delay Sd = average delay

RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm assumes the **communication channels are FIFO**.

- The algorithm uses two types of messages: **REQUEST** and **REPLY.**
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section.
- A process sends a REPLY message to a process to give its permission to that process.
- **Processes use Lamport-style logical clocks to assign a timestamp** to critical section requests. Timestamps are used to decide the priority of requests in case of conflict – if a process pi that is waiting to execute the critical section, receives a REQUEST message from process pj, then if the priority of pj's request is lower, pi defers the REPLY to pj and sends a REPLY message to pj only after executing the CS for it spendingrequest.

 Otherwise, pi sends a REPLY message to pj immediately, provided it is currently not executing the CS. Thus, if several processes are requesting execution of the CS, the highest priority request succeeds in collecting all the needed REPLY messages and gets to execute the CS.

Each process pi maintains the **Request-Deferred array**, RDi, the size of which is the same as the number of processes in the system. Initially, ∀i ∀j: RDi[j]=0. Whenever pi defer the request sent by pj, it sets RDi[j]=1 and after it has sent a REPLY message to pj, it sets RDi[j]=0.**Note:** Deferred – Postponed the request / waiting

ALGORITHM

1. Requesting the critical section:

(a) When a site Si wants to enter the CS, it broadcasts a time stamped 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 Si's request's timestamp is smaller than site Sj's own request's timestamp. otherwise, the reply is deferred and Sj sets RDj[i]=1

2. 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.

3. Releasing the critical section:

(d) When site Si exits the CS, it sends all the deferred REPLY messages: ∀j if RDi[j]=1, then send a REPLY message to Sj and set RDi[j]=0.

When a site receives a message, it updates its clock using the timestamp in the message. Also, when a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request. In this algorithm, a site's REPLY messages are blocked only by sites which are requesting the CS with higher priority (i.e., smaller timestamp).Thus, when a site sends out differed REPLY messages, site with the next highest priority request receives the last needed REPLY message and enters the CS. Execution of the CS requests in this algorithm is always in the order of their timestamps.

An Example

Figures 9.7 to 9.10 illustrate the operation of Ricart-Agrawala algorithm. In Figure 9.7, sites S1 and S2 are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Figure 9.8, S2 has received REPLY messages from all other sites and consequently, it enters the CS. In Figure 9.9, S2 exitsthe CS and sends a REPLY message to site S1. In Figure 9.10, site S1 has received REPLY fromall other sites and enters the CS next.

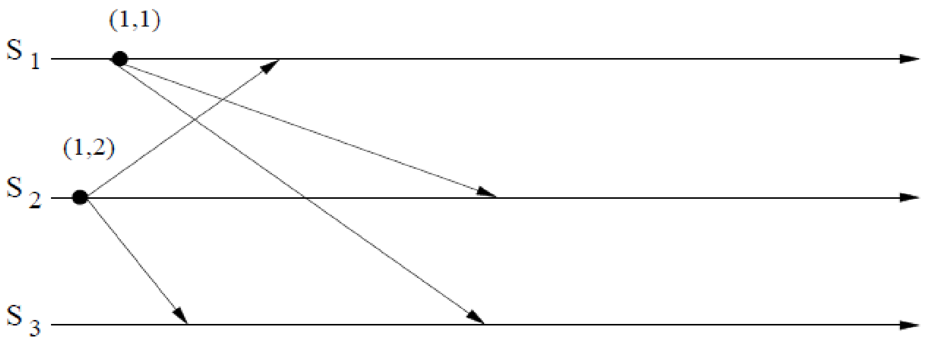


Figure 9.7: Sites S_1 and S_2 are making request for the CS

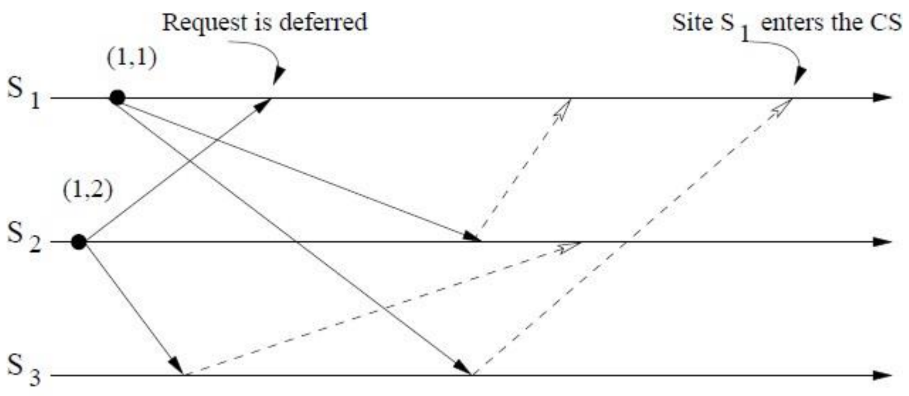


Figure 9.8: Site S_1 enters the CS

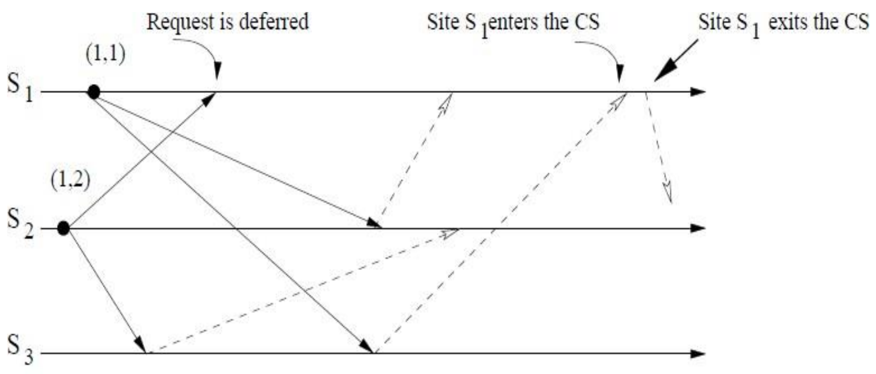


Figure 9.9: Site S_1 exits the CS and sends a REPLY message to S_2's deferred request

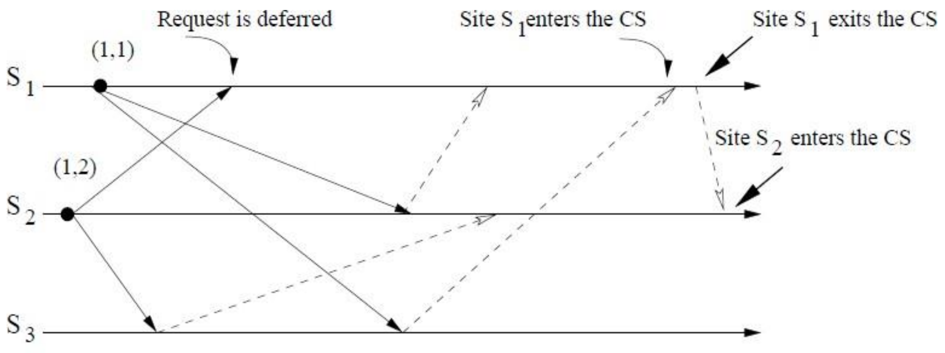


Figure 9.10: Site S_2 enters the CS

Performance

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

MAEKAWA'S ALGORITHM

Maekawa's Algorithm is quorum (subset) based approach to ensure mutual exclusion in distributed systems. As we know, 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**.

In this algorithm:

- 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 criticalsection.
- A site send a **RELEASE** message to all other site in its request set or quorum upon exitingthe critical section.

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

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

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

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

M4: Any site S_j is contained in K number of R_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| = √N. Since there is at least one common site between the request sets of any two sites (condition M1), every pair of sites has a common site which mediates conflicts between the pair. A site can have only one outstanding REPLY message at any time; that is, it grants permission to an incoming request if it has not granted permission to some other site. Therefore, mutual exclusion is guaranteed. This algorithm requires delivery of messages to be in the order they are sent between very pair of sites. Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm. ConditionM3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion. Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have "equal responsibility" in granting permission to other sites.

<u>ALGORITHM</u>

In Maekawa's algorithm, a site Si executes the following steps to execute the CS.

1. Requesting the critical section

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

(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.

2. Executing the critical section

(c) Site Si executes the CS only after it has received a REPLY message from everysite in Ri.

3. Releasing the critical section

(d) After the execution of the CS is over, site Si sends a RELEASE (i) message toevery site in Ri.

(e) When a site Sj 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.

If the queue is empty, then the site updates its state to reflect that it has not sentout any REPLY message since the receipt of the last RELEASE message.

Correctness

Theorem 3: Maekawa's algorithm achieves mutual exclusion.

Proof: Proof is by contradiction. Suppose two sites Si and Sj are concurrently executing the CS. This means site Si received a REPLY message from all sites in Ri and concurrently site Sj was able to receive a REPLY message from all sites in Rj. If Ri ∩ Rj= {Sk}, then site Sk must havesent REPLY messages to both Si and Sj concurrently, which is a contradiction. 2

Performance

Note that the size of a request set is √N. Therefore, an execution of the CS requires √N REQUEST,√N REPLY, and √N RELEASE messages, resulting in 3√N messages per CS execution. Synchronization delay in this algorithm is 2T. This is because after a site Si exits the CS, it first releases all the sites in Ri and then one of those sites sends a REPLY message to the next site that executes the CS. Thus, two sequential message transfers are required between two successive CS executions. As discussed next, Maekawa's algorithm is deadlock-prone. Measures to handle deadlocks require additional messages.

Problem of Deadlocks

Maekawa's algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps [14, 22]. Thus, a site may send a REPLY message to a site and later force a higher priority request from another site to wait.

Without the loss of generality, assume three sites Si, Sj, and Sk simultaneously invoke mutual exclusion. Suppose Ri ∩ Rj= {Sij}, Rj∩ Rk= {Sjk}, and Rk∩ Ri= {Ski}. Since sites do not send REQUEST messages to the sites in their request sets in any particular order and message delays are arbitrary, the following scenario is possible: Sij has been locked by Si (forcing Sj towait at Sij), Sjk has been locked by Sj(forcing Sk to wait at Sjk), and Ski has been locked by Sk(forcing Si to wait at Ski). This state represents a deadlock involving sites Si, Sj, and Sk.

Handling Deadlocks

Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock (unless the former has succeeded in acquiring locks on all the needed sites). A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrive sand waits at a site because the site has sent a REPLY message to a lower priority request. Deadlock handling requires the following three types of messages:

Deadlock handling requires the following three types of messages:

FAILED: A FAILED message from site Si to site Sj indicates that Si cannot grant Sj's requestbecause it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from Si to Sj indicates that Si would like to find out from Sjif ithas succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site Si to Sj indicates that Si is returning the permission to Sj(toyield to a higher priority request at Sj).

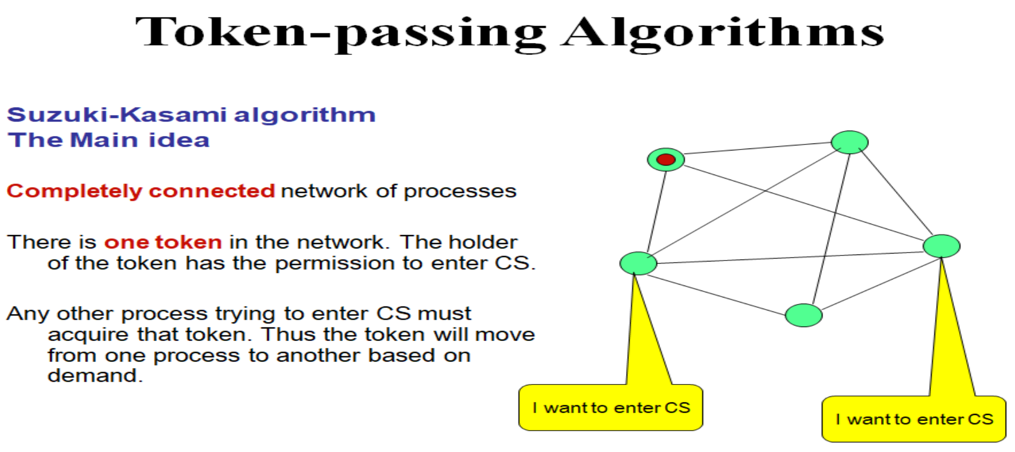
Details of how Maekawa's algorithm handles deadlocks are as follows:

• When a REQUEST(ts, i) from site Si blocks at site Sj because Sj has currently granted permission to site Sk, then Sj sends a FAILED(j) message to Si if Si's request has lower priority. Otherwise, Sj sends an INQUIRE(j) message to site Sk.• In response to an INQUIRE(j) message from site Sj, site Sk sends a YIELD(k) message to Sj provided Sk has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new GRANT from it.• In response to a YIELD(k) message from site Sk, site Sj assumes as if it has been released by Sk, places the request of Sk at appropriate location in the request queue, and sends a GRANT(j) to the top request's site in the queue.
Thus, Maekawa-type algorithms require extra messages to handle deadlocks and may exchange these messages even though there is no deadlock. Maximum number of messages required per CS execution in this case is 5√N.

SUZUKI-KASAMI'S BROADCAST ALGORITHM

- **Suzuki–Kasami algorithm** is a token-based algorithm for achieving mutual exclusionin distributed systems.
- 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 request for critical section contains a sequence number. This sequence numberis used to distinguish old and current requests.



Token-passing Algorithms

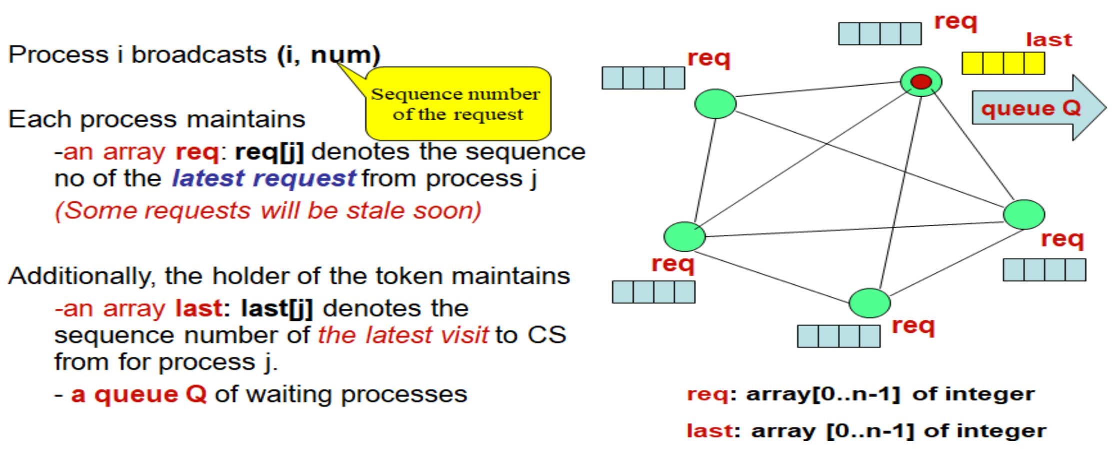
Suzuki-Kasami algorithm
The Main idea

Completely connected network of processes

There is one token in the network. The holder of the token has the permission to enter CS.

Any other process trying to enter CS must acquire that token. Thus the token will move from one process to another based on demand.

Suzuki-Kasami Algorithm


Process i broadcasts **(i, num)**

Each process maintains

-an array **req**: **req[j]** denotes the sequence no of the *latest request* from process j

(Some requests will be stale soon)

Additionally, the holder of the token maintains

-an array **last**: **last[j]** denotes the sequence number of *the latest visit* to CS from for process j.

- **a queue Q** of waiting processes

req: array[0..n-1] of integer

last: array [0..n-1] of integer

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

The basic idea underlying this algorithm may sound rather simple, however, there are the following two design issues must be efficiently addressed:

1. How to distinguishing an outdated REQUEST message from a current REQUEST message:

Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied. If a site can not determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it. This will not violate the correctness; however, this may seriously degrade the performance by wasting messages and increasing the delay at sites that are genuinely requesting the token. Therefore, appropriate mechanisms should implemented to determineif a token request message is outdated.

2. **How to determine which site has an outstanding request for the CS:** After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them. The problem is complicated because when a site Si receives a token request message from a site Sj, site Sj may have an outstanding request for the CS. However, after the corresponding request for the CS has been satisfied at Sj, an issue is how to inform site Si (and all other sites) efficiently about it. Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: A REQUEST message of site Sjhas the form REQUEST(j, n) where n (n=1, 2,
...) is a sequence number which indicates that site Sjis requesting its nth CS execution.
A site Si keeps an array of integers RNi[1..N] where RNi[j] denotes the largest sequence number received in a REQUEST message so far from site Sj. When site Si receives a REQUEST (j, n)message, it sets RNi[j]:=max(RNi[j], n). Thus, when a site Si receives a REQUEST (j, n) message, the request is outdated if RNi[j]>n. Sites with outstanding requests for the CS are

determined in the following manner: The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site Sj executed most recently. After executing its CS, a site Si updates LN[i]:=RNi[i] to indicate that its request corresponding to sequence number RNi[i] has been executed. Token array LN[1..N] permits a site to determine if a site has an outstanding request for the CS. Note that at site Siif RNi[j]=LN[j]+1, then site Sj is currently requesting token. After executing the CS, a site checks this condition for all the j' s to determine all the sites which are requesting the token and places their id's in queue Q if these id's are not already present in the Q. Finally, the site sends the token to the site whose id is at the head of the Q.

ALGORITHM

1. Requesting the critical section

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

(b) When a site Sj receives this message, it sets RNj[i] to max(RNj[i], sn). If Sj has the idle token, then it sends the token to Si if RNj[i]=LN[i]+1.

2. Executing the critical section

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

3. Releasing the critical section Having finished the execution of the CS, site Si takes thefollowing actions:

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

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

(f) If the token queue is nonempty after the above update, Si deletes the top site id from the token queue and sends the token to the site indicated by the id. Thus, after executing the CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS). Note that Suzuki-Kasami's algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of symmetric algorithm: "no site possesses the right to access its CS when it has not been requested".

Correctness

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

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

Proof: Token request messages of a site Si reach other sites in finite time. Since one of these sites will have token in finite time, site Si's request will be placed in the token queue in finite time. Since there can be at most N − 1 requests in front of this request in the token queue, site Si will get the token and execute the CS in finite time. 2

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

DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS INTRODUCTION

A deadlock is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

We can consider two types of deadlock:

1. Communication deadlock occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A.
2. A resource deadlock occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources. We will not differentiate between these types of deadlock since we can consider communication channels to be resources without loss of generality.

"A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set."

Deadlock deals with various components **like deadlock prevention, deadlock avoidance other then deadlock detection**.

- **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 hold the needed resource.
- In the **deadlock avoidance** approach to distributed system, 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. To resolve the deadlock, we have to abort a deadlocked process.

SYSTEM MODEL

- A distributed system consists of a set of processors that are connected by a communication network.
- The communication delay is finite but unpredictable.
- A distributed program is composed of a set of n asynchronous processes p1, p2, . . . , pi, . . . , pn that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.

- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.
- The system can be modelled as a directed graph in which vertices represent the processes and edge represent unidirectional communication channels.

We make the following assumptions:

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

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

Wait-For-Graph (WFG)

- In distributed systems, the state of the system can be modelled by **directed graph, called a wait for graph (WFG)**.
- In a WFG, nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Figure 10.1 shows a WFG, where process P11 of site 1 has an edge to process P21 of site 1 and P32 of site 2 is waiting for a resource which is currently held by process P21. At the same time process P32 is waiting on process P33 to release a resource. If P21 is waiting on process P11, then processes P11, P32 and P21 form a cycle and all the four processes are involved in a deadlock depending upon the request model.

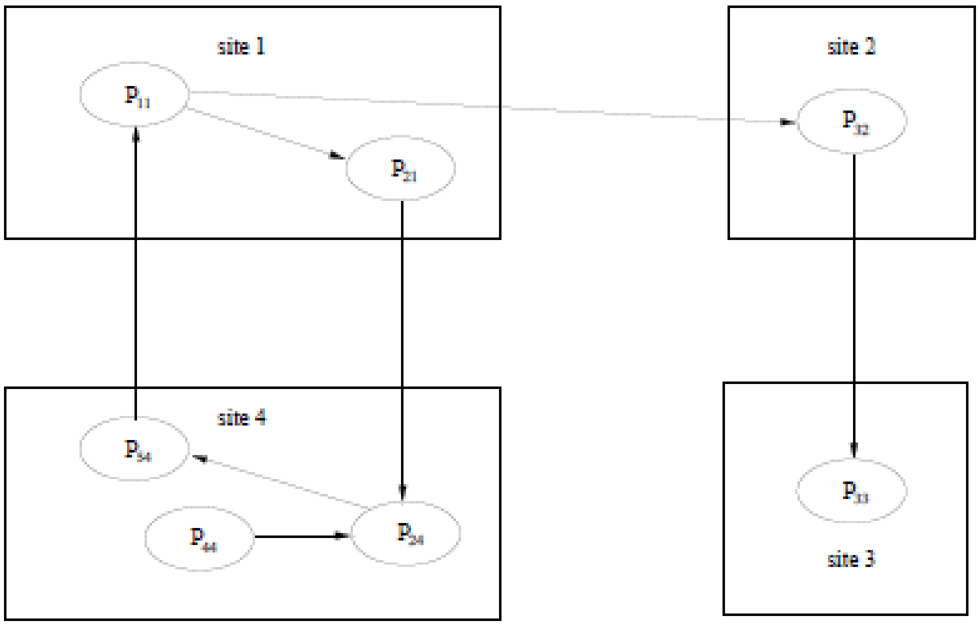


Figure 10.1: Example of a WFG

PRELIMINARIES

Deadlock Handling Strategies
There are three strategies for handling deadlocks,

1. **Deadlock Prevention,**
2. **Deadlock Avoidance,**
3. **Deadlock Detection.**

Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter site communication involves a finite and unpredictable delay. Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by pre-empting a process which holds the needed resource. This approach is highly inefficient and impractical in distributed systems. In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system). However, due to several problems, deadlock avoidance is impractical in distributed systems.

Issues in Deadlock Detection

Deadlock handling using the approach of deadlock detection entails addressing two basic issues:

1. **Detection of existing deadlocks**
2. **Resolution of detected deadlocks.**

1. Detection of Deadlocks

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

(i) Progress (No undetected deadlocks): The algorithm must detect all existing deadlocks in finite time. Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

(ii) Safety (No false deadlocks): The algorithm should not report deadlocks which do not exist (called phantom or false deadlocks). In distributed systems where there is no global memory and there is no global clock, it is difficult to design a correct deadlock detection algorithm because sites may obtain out of date and inconsistent WFG of the system. As a result, sites may detect a cycle which never existed but whose different segments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

2. 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. Note that several deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph. Therefore, when a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system. If this information is not cleaned in timely manner, it may result in detection of phantom deadlocks. Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

MODELS OF DEADLOCKS

- Distributed systems allow many kinds of resource requests. A process might require a single resource or a combination of resources for its execution. This section introduces a hierarchy of request models starting with very restricted forms to the ones with no restrictions whatsoever. This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

The Single Resource Model

The single resource model is the simplest resource model in a distributed system, here a process can have at most one outstanding request for only one unit of a resource. Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock. In a later section, an algorithm to detect deadlock in the single resource model is presented.

The AND Model

In the AND model, a process can request for more than one resource simultaneously and the
request is satisfied only after all the requested resources are granted to the process. The 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. Consider the example WFG described in the Figure 10.1. Process P11 has two outstanding resource requests. In case of the AND model, P11shall become active from idle state only after both the resources are granted. There is a cycle P11->P21->P24->P54->P11 which corresponds to a deadlock situation.

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. Consider process P44 in Figure 10.1. It is not a part of any cycle but is still deadlocked as it is dependent on P24 which is deadlocked. Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

The OR Model

In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted. 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. To make it more clear, consider Figure 10.1. If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied. After P32 finishes execution and releases its resources, process P11 can continue with its processing.

In the OR model, the presence of a knot indicates a deadlock. In a WFG, a vertex v is in a knot if for all u:: u is reachable from v : v is reachable from u. No paths originating from a knot shall have dead ends.

A deadlock in the OR model can be intuitively defined as follows : A process Pi is blocked if ithas 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 idle to active state on receiving a grant 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. Intuitively, a set of processes S is deadlocked if all the processes in S are permanently blocked. To formally state that a set of processes is deadlocked, the following conditions hold true:

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, and
3. No grant message is in transit between any two processes in set S.

We now show that a set of processes S shall remain permanently blocked in the OR model if

the above conditions are met. 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. Note that no grant message can be expected from any process in S because they are all blocked. Also, the third condition states that no grant messages in transit between any two processes in set S. So, all the processes in set S are permanently blocked.
Hence, deadlock detection in the OR model is equivalent to finding knots in the graph. Note that, there can be a process deadlocked which is not a part of a knot. Consider the Figure 10.1 where P44 can be deadlocked even though it is not in a knot. So, in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot.

The AND-OR Model
A generalization of the previous two models (OR model and AND model) is the AND-OR model. In the AND-OR model, a request may specify any combination of and and or in the resource request. For example, in the AND-OR model, a request for multiple resources can be of the form x and (y or z). The requested resources may exist at different locations. To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock. However, this is a very inefficient strategy. Efficient algorithms to detect deadlocks in AND-OR model are discussed in Herman
.

The $\binom{p}{q}$ Model
Another form of the AND-OR model is the $\binom{p}{q}$ model (called the P-out-of-Q model) which allows a request to obtain any k available resources from a pool of n resources. Both the models are the same in expressive power. However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request.

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

Unrestricted Model
In the 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 (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication). Hence, these algorithms can be used to detect other stable properties as they deal with this general model. But, 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.

KNAPP’S CLASSIFICATION OF DISTRIBUTED DEADLOCK DETECTION

Algorithms

- Distributed deadlock detection algorithms can be divided into four classes : path-pushing, edge-chasing, diffusion computation, and global state detection.

Path-Pushing Algorithms

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

Edge-Chasing Algorithms

In an edge-chasing algorithm, the presence of a cycle in a distributed graph structureis 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 simply discards this message and continues. Only blocked processes propagate probe messages along their outgoing edges. An interesting variation of this method can be found in Mitchell [36], where probes are sent upon request and in the opposite direction of the edges.
Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short. Examples of such algorithms include Chandy et al., Choudhary et al., Kshemkalyani-Singhal , and Sinha-Natarajan algorithms.

Diffusing Computations Based Algorithms

In diffusion computation based distributed deadlock detection algorithms; deadlock detection computation is diffused through the WFG of the system. These algorithms make use of echo algorithms to detect deadlocks. This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock. The main feature of the superimposed computation is that the global WFG is implicitly reflected in the structure of the computation. The actual WFG is never built explicitly.
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. Queries are discarded by a running process and are echoed back by blocked processes in the following way: 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 (to its successors in the WFG). 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. Examplesof these types of deadlock detection algorithms are Chandy-Misra-Haas algorithm for OR model and Chandy-Herman algorithm .

Global State Detection Based Algorithms

Global state detection based deadlock detection algorithms exploit the following facts: (i) A consistent snapshot of a distributed system can be obtained without freezing the underlying computation and (ii) a consistent snapshot may not represent the system state at any moment in time, but 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. Examples of these types of algorithms include Bracha- Toueg , Wang et al., and Kshemkalyani-Singhal algorithms.

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

Mitchell and Merritt's algorithm belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG. When a probe initiated by a process comes back to it, the process declares deadlock. The algorithm has many good features like:
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 process which 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.

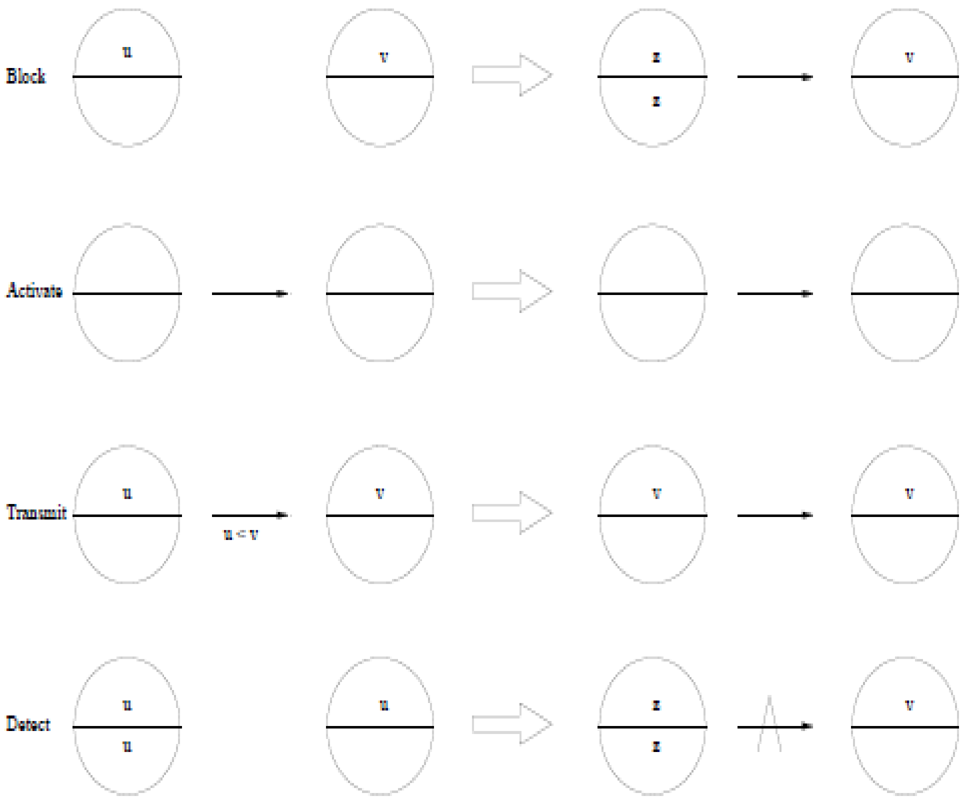


Figure 10.2: The four possible state transitions

Each node of the WFG has two local variables, called labels: a private label, which is unique

to the node at all times, though it is not constant, and a public label, which can be read by other processes and which may not be unique. Each process is represented as u/v where u and u are the public and private labels, respectively. Initially, private and public labels are equal for each process.
A global WFG is maintained and it defines the entire state of the system.The Algorithm is defined by the four state transitions shown in Figure 10.2, where z = inc(u, v), and inc(u, v) yields a unique label greater than both u and v labels that are not shown do not change. Block creates an edge in the WFG. Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for. Activate denotes that a process has acquired the resource from the process it was waiting for. Transmit propagates larger labels in the opposite direction of the edges by sending a probe message. Whenever a process receives a probe which is less then its public label, then it simply ignores that probe. Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
Mitchell andMerritt showed that every deadlock is detected. Next, we show that in the absence of spontaneous aborts, only genuine deadlocks are detected. As there are no spontaneous aborts, we have following invariant:

For all processes u/v: u <= v
Proof. Initially u = u for all processes. The only requests that change u or v are

1) Block: u and v are set such that u = v.
(2) Transmit: u is increased.

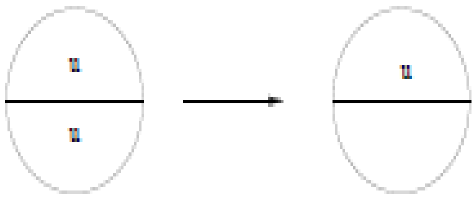
Hence, the invariant.

From the previous invariant, we have the following lemmas.

Lemma 12. For any process u/v, if u > u, then u was set by a Transmit step.

Theorem 13. If a deadlock is detected, a cycle of blocked nodes exists.

Proof. A deadlock is detected if the following edge p →p' exists:



We will prove the following claims:
(1) u has been propagated from p to p' via a sequence of Transmits.
(2) P has been continuously blocked since it transmitted u.
(3) All intermediate nodes in the transmit path of (l), including p', have been continuously blocked since they transmitted u.

From the above claims, the proof for the theorem follows as discussed below:

From the invariant and the uniqueness of private label u of p’ : u < v. By Lemma 4.1, u was set by a Transmit step. From the semantics of Transmit, there is some p” with private label u and public label w. If w = u, then p” = p, and it is a success. Otherwise, if w <u, we repeat the argument. Since there are only processes, one of them is p. If p is active then it indicates thatit has transmitted u else it is blocked if it detects deadlock. Hence upon blocking it incremented its private label. But then private and public labels cannot be equal. Consider a process which has been active since it transmitted u. Clearly, its predecessor is also active, as Transmits migrate in opposite direction. By repeating this argument, we can show p has been active since it transmitted u. The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted. This algorithm has two phases. The first phase is almost identical to the algorithm. In the second phase the smallest priority is propagated around the circle, The propagation stops when one process recognizes the propagated priority as its own.

Message Complexity
Now we calculate the complexity of the algorithm. 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.

CHANDY-MISRA-HAAS ALGORITHM FOR THE AND MODEL

We now discuss Chandy-Misra-Haas’s distributed deadlock detection algorithm for AND model that is based on edge-chasing. The algorithm uses a special message called probe, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process Pi and it is being sent by the home site of
process Pj to the home site of process Pk. 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 Pj is said to be dependent on another process Pk if there exists a sequence of processes Pj,Pi1, Pi2, ..., Pim, Pk such that each process except Pk in the sequence is blocked and each process, except the Pj, holds a resource for which the previous process in the sequence is waiting. Process Pj is said to be locally dependent upon process Pk if Pj is dependent upon Pk and both the processes are on the same site.

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

The Algorithm

The following algorithm is executed to determine if a blocked process is deadlocked:

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

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

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

```
dependent_k(i) = true;
if k=i
    then declare that $P_i$ is deadlocked
else for all $P_m$ and $P_n$ such that
    (a') $P_k$ is locally dependent upon $P_m$, and
    (b') $P_m$ is waiting on $P_n$, and
    (c') $P_m$ and $P_n$ are on different sites,
    send a probe (i, m, n) to the home site of $P_n$
end.
```

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

Performance Analysis

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

CHANDY-MISRA-HAAS ALGORITHM FOR THE OR MODEL

We now discuss Chandy-Misra-Haas distributed deadlock detection algorithm for OR model that is based on the approach of diffusion-computation. A blocked process determines if it is deadlocked by initiating a diffusion computation. Two types of messages are used in a diffusion computation: query(i, j, k) and reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process Pi and are being sent from process Pj to process Pk.

Basic Idea

A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message). 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 (called the engaging query), then it propagates the query to all the processes in its dependent set and sets a local variable numk(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 waitk(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 numk(i) only if waitk(i) holds. A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query. The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.

The Algorithm

The algorithm works as follows:

```
**Initiate a diffusion computation for a blocked process $P_i$:**
    send query(i, i, j) to all processes $P_j$ in the dependent set $DS_i$ of $P_i$;
    $num_i$(i):= $|DS_i|$; $wait_i$(i):= true;

**When a blocked process $P_k$ receives a query(i, j, k):**
    if this is the engaging query for process $P_i$
        then send query(i, k, m) to all $P_m$ in its dependent set $DS_k$;
        $num_k$(i):= $|DS_k|$; $wait_k$(i):= true
        else if $wait_k$(i) then send a $reply$(i, k, j) to $P_j$.

**When a process $P_k$ receives a reply(i, j, k):**
    if $wait_k$(i)
        then begin
            $num_k$(i):= $num_k$(i) $-$ 1;
            if $num_k$(i)= 0
                then if i=k then declare a deadlock
                else send reply(i, k, m) to the process $P_m$
                    which sent the engaging query.
```

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

Performance Analysis

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