DISTRIBUTED SYSTEMS

DISTRIBUTED SYSTEMS UNIT 1

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

1. Introduction

Definition – Distributed Systems

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

Characteristics of Distributed Systems

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

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

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

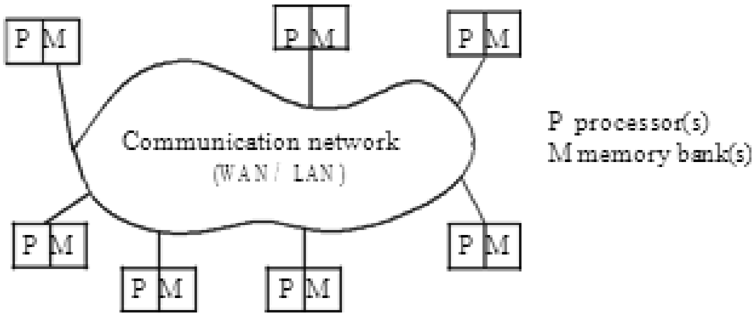
4. **Autonomy and heterogeneity** -> The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system.

1.2 Relation to computer system components

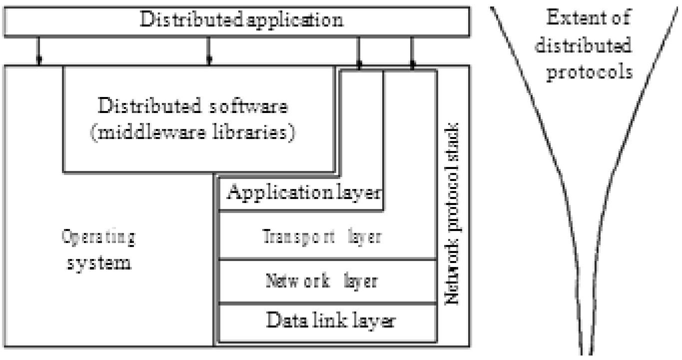
Each computer has a memory-processing unit and the computers are connected by a communication network. Figure shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning.

DISTRIBUTED SYSTEMS

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



<u>Interaction of the software components at each process</u>



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

1.3 Motivation

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

1. **Inherently distributed computations**

The computation is inherently distributed
Eg., <u>money transfer in banking</u>

2. **Resource sharing**

Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites. Further, they cannot be placed at a single site. Therefore, such resources are typically distributed across the system.
For example, <u>distributed databases such as DB2</u> partition the data sets across several servers

3. Access to geographically remote data and resources

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

4. Enhanced reliability

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

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

5. Increased performance/cost ratio

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

6. Scalability

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

7. Modularity and incremental expandability

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

1.4 Relation to parallel multiprocessor/multicomputer systems

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

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

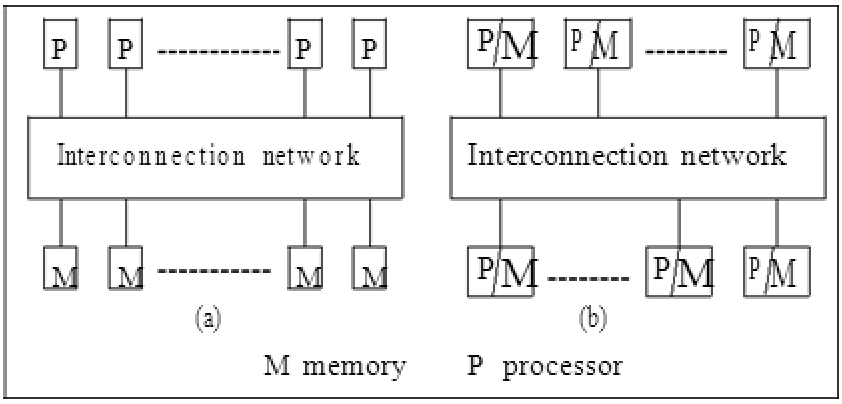
Characteristics of parallel systems

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

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

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

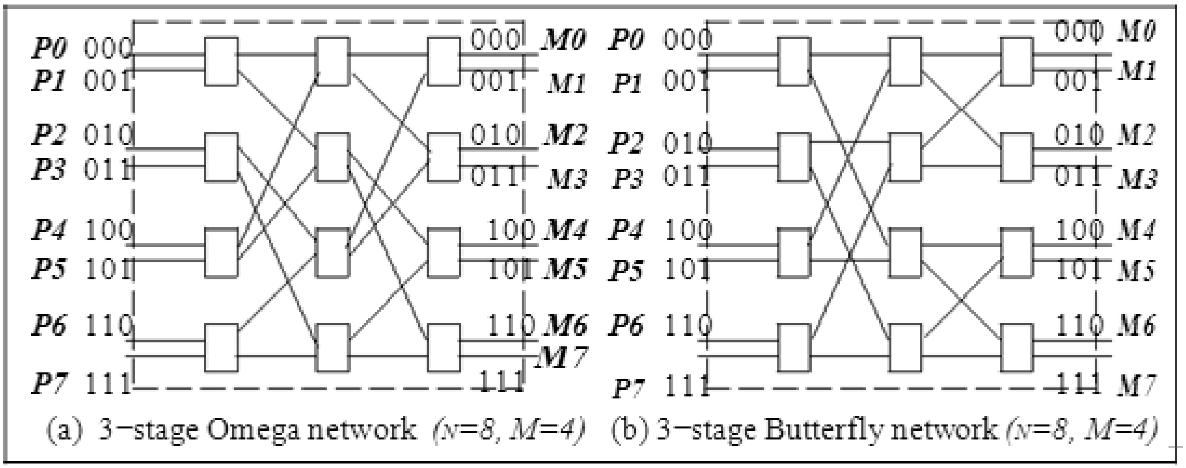
Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.



Omega network:

Figure(1.4) shows two popular interconnection networks – the Omega network and the Butterfly network, each of which is a multi-stage network formed of 2 ×2 switching elements. Each 2 ×2switch allows data on either of the two input wires to be switched to the upper or the lower output wire.

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



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) Butterfly network [10] for n = 8 processors P0–P7 and memory banks M0–M7.

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

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

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

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

Omega routing function

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

2. **Multicomputer parallel system**

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

Non-uniform memory access (NUMA) architecture

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

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

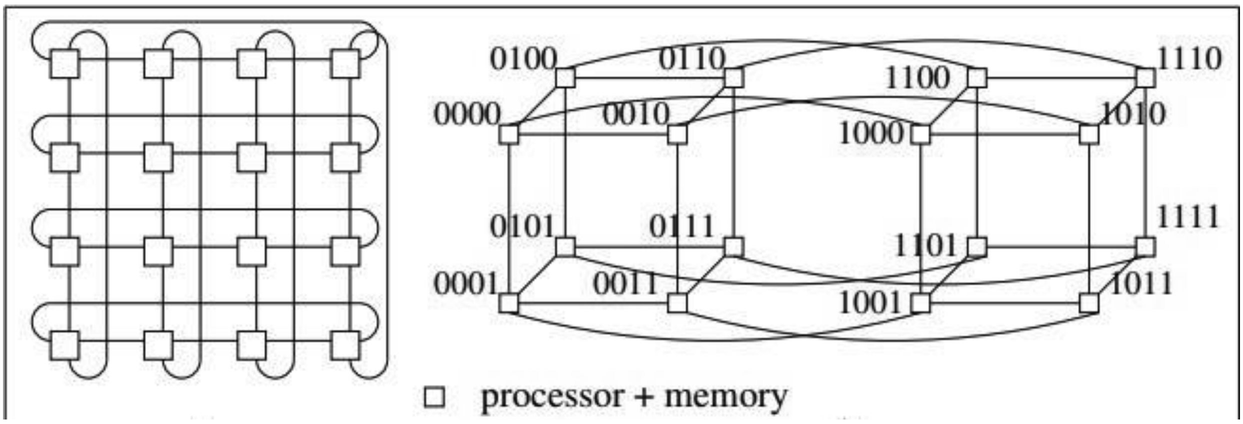


Figure (a) shows a wrap-around 4 × 4 mesh. For a k × k mesh which will contain k^2 processors, the maximum path length between any two processors is 2 k/2 − 1 . Routing can be done along the Manhattan grid.

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

<u>Hamming distance</u>

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

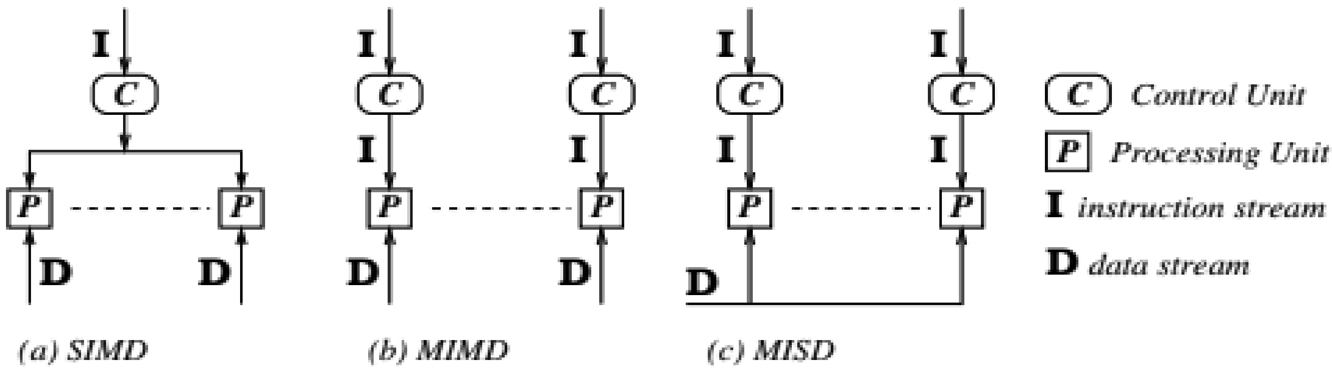
3. <u>Array processors</u>

- **<u>Array processors</u>** belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock (but may not share memory and communicate by passing data using messages).
- Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing belong to this category.
- These applications usually involve a large number of iterations on the data. This class of parallel systems has a very niche market.

DISTRIBUTED SYSTEMS

Flynn's Taxonomy

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



SISD: Single Instruction Stream Single Data Stream (traditional)

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

SIMD: Single Instruction Stream Multiple Data Stream

This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items.

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

MISD: Multiple Instruction Stream Single Data Stream

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

- E.g., visualization

MIMD: Multiple Instruction Stream Multiple Data Stream

- In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems.
- There is no common clock among the system processors.

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

Coupling, parallelism, concurrency, and granularity

- **Coupling**

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

- When the degree of coupling is high (low), the modules are said to be tightly (loosely) coupled.
- SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.
- Various MIMD architectures in terms of coupling:

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

Parallelism or speedup of a program on a specific system

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

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

Concurrency of a program

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

Granularity of a program

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

1.5 Message-passing vs. Shared Memory

- Shared memory systems are those in which there is a (common) shared address space throughout the system.
- Communication among processors takes place via shared data variables, and control variables for synchronization among the processors.

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

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

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

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

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

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

Primitives for distributed communication

Blocking/non-blocking, synchronous/asynchronous primitives

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

Synchronous primitive(send/receive)

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

Asynchronous primitive (send)

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

Blocking primitive (send/receive)

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

Nonblocking primitive (send/receive)

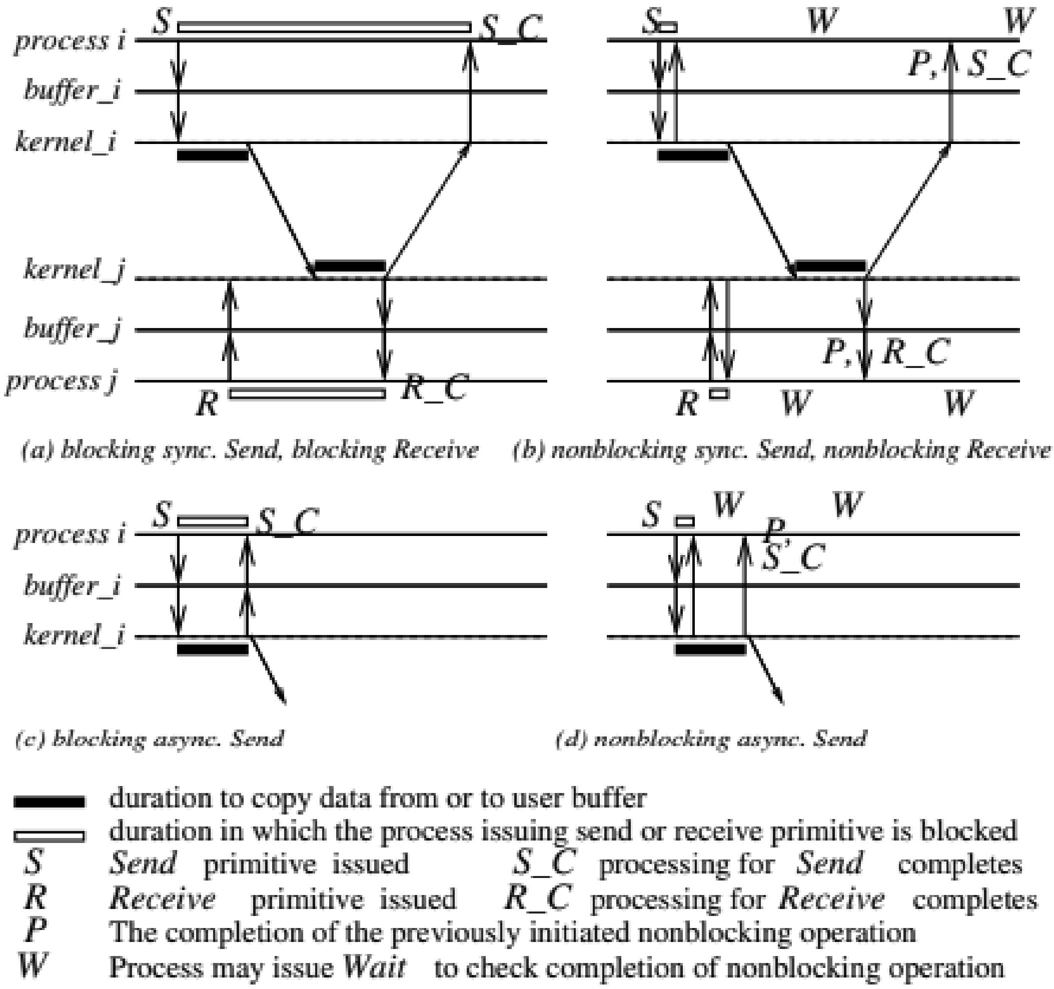
- A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.
- Send: even before data copied out of user buffer
- Receive: even before data may have arrived from sender

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

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

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

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



Processor synchrony

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

Libraries and standards

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

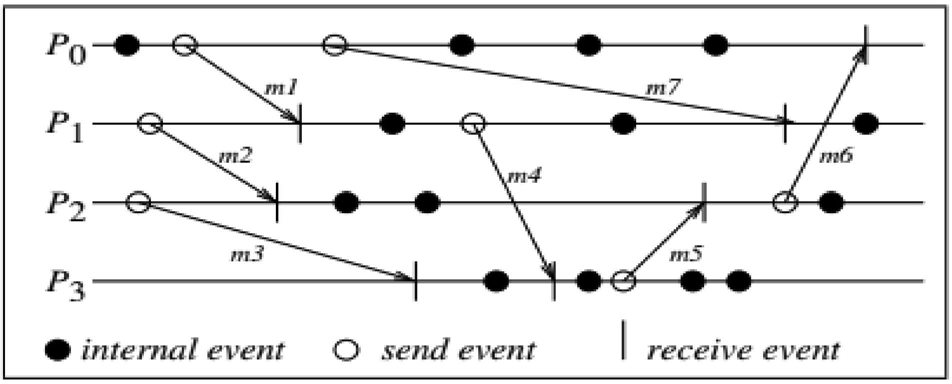
- CORBA (common object request broker architecture) and DCOM (distributed component object model) are two other standardized architectures with their own set of primitives

1.7 Synchronous versus asynchronous executions

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

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

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



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

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

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

There is a hurdle to having a truly synchronous execution

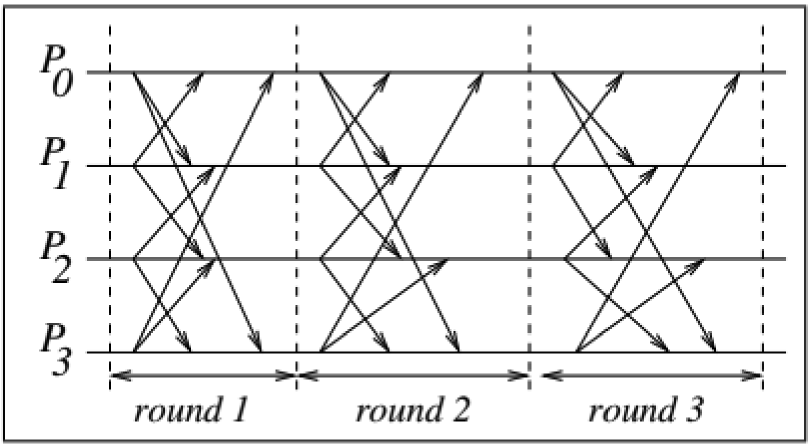
- It is practically difficult to build a completely synchronous system, and have the messages delivered within a bounded time.

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

Virtual Synchrony

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

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



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

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

Sync vs async executions

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

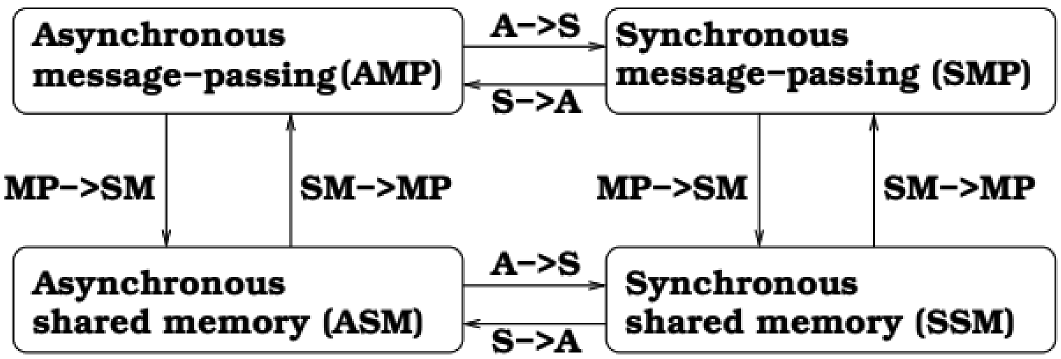
- Difficult to build a truly synchronous system; can simulate this abstraction
- Virtual synchrony:
  - async execution, processes synchronize as per application requirement;
  - execute in rounds/steps
- Emulations:
  - Async program on sync system: trivial (A is special case of S)
  - Sync program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

System Emulations

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

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

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



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

- If solvable in A, solvable in B

1.8 Design issues and challenges

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

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

1.8.1 Distributed systems challenges from a system perspective

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

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

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

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

Synchronization

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

Data storage and access

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

Consistency and replication

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

Distributed systems security

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

API for communications, services: ease of use

Transparency: hiding implementation policies from user

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

Scalability and modularity

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

1.8.2 Algorithmic challenges in distributed computing

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

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

Dynamic distributed graph algorithms and routing algorithms

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

Time and global state

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

Synchronization/coordination mechanisms

Some examples of problems requiring synchronization:

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

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

Group communication, multicast, and ordered message delivery

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

Monitoring distributed events and predicates

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

Distributed program design and verification tools

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

Debugging distributed programs

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

Data replication, consistency models, and caching

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

World Wide Web design: caching, searching, scheduling

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

Distributed shared memory abstraction

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

Consistency models:

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

- Weaker models than strict consistency of uniprocessors

Reliable and fault-tolerant distributed systems

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

Replication and replica management

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

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

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

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

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

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

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

1.8.3 Applications of distributed computing and newer challenges

Mobile systems

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

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

- Processors embedded in and seamlessly pervading environment

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

Publish/subscribe, content distribution

- Filtering information to extract that of interest

Distributed agents

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

Distributed data mining

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

Grid computing

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

Security

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

1.9 A Model of Distributed Computations

A Distributed Program

- A distributed program is composed of a set of n asynchronous processes, p_1, p_2, ..., p_i , ..., p_n.
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.
- Withoutlossofgenerality, we assumethat each processisrunning ona different processor.
- Let C_{ij} denote the channel from process p_i to process p_j and let m_{ij} denote a message sent by p_i to p_j.
- The message transmission delay is finite and unpredictable.

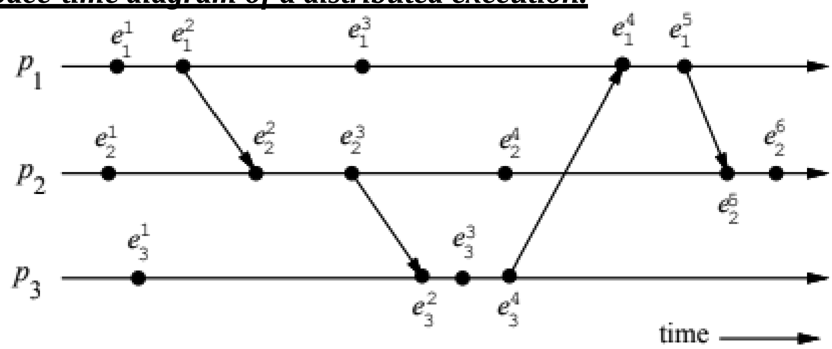
1.10 A Model of Distributed Executions

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

denoteitssendandreceiveevents, respectively.

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

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



<u>Causal Precedence Relation</u>

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

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

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

- Note that the relation → is nothing but Lamport's "happens before" relation.
- For any two events e_i and e_j , if $e_i \rightarrow e_j$, then event e_j is directly or transitively dependent on event e_i. (Graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space-time diagram that starts at e_i and ends at e_j.)
- For example, in Figure 2.1, $e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$.
- The relation → denotes flow of information in a distributed computation and $e_i \rightarrow e_j$ dictates that all the information available at e_i is potentially accessible at e_j .
- For example, i2n Figure 2.1, event e_2^6 has the knowledge of all other events shown in the figure.
- For any two events e_i and e_j, $e_i \not\rightarrow e_j$ denotes the fact that event e_j does not directly or transitively dependent on event e_i. That is, event e_i does not causally affect event e_j.
- In this case, event e_j is not aware of the execution of e_i or any event executed after e_i on the same process.
- For example, in Figure 2.1, $e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$.

Note the following two rules:

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

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

Concurrent Events

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

Logical vs. Physical Concurrency

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

instant in physical time.

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

1.11 Models of communication networks

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

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

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

1.12 Global State of a Distributed System

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

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

Notations

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

A Channel State

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

The state of a channel is defined as follows:

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

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

Global State

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

A Consistent Global State

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

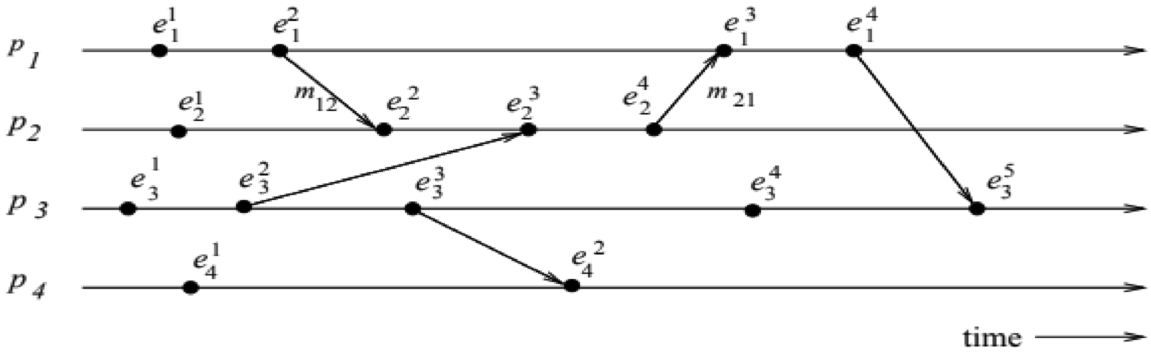
A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

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

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

An Example

Consider the distributed execution of Figure



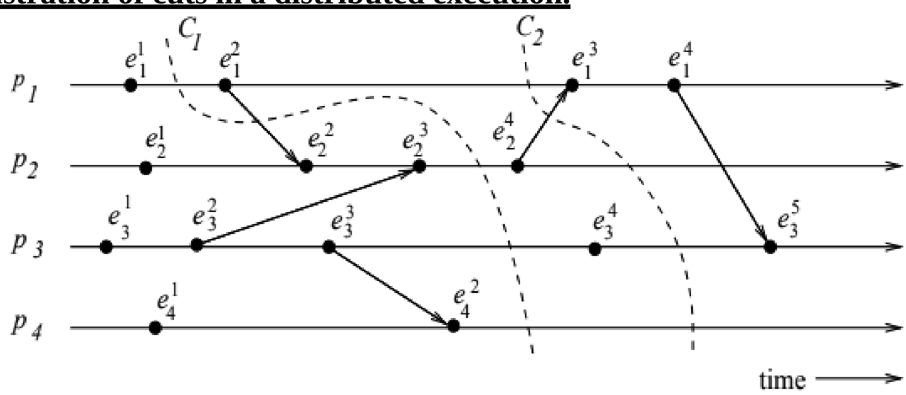
- A global state $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent because the state of p_2 has recorded the receipt of message m_{12}, however, the state of p_1 has not recorded its send.
- A global state GS_2 consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all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

1.13 Cuts of a Distributed Computation

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

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

Figure: Illustration of cuts in a distributed execution.



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

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

1.14 Past and Future Cones of an Event

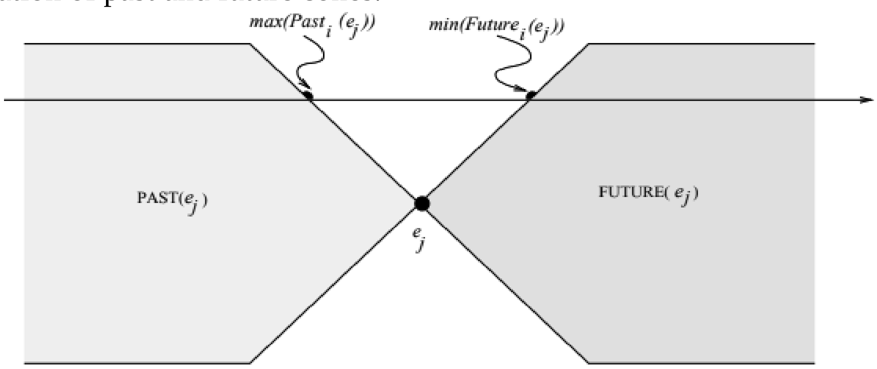
Past Cone of an Event

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

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

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

Figure: Illustration of past and future cones.



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

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

Future cone of an Event

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

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

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

1.15 Models of Process Communications

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

1.16 Logical Time

Introduction

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

1.17 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 <.
- Relation < is called the ***happened before* or *causal precedence***. Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time.
- The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T, denoted as C(e) and called the timestamp of e, and is defined as follows:

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

  such that the following property is satisfied:

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

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

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

Implementing Logical Clocks

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

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

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

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

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

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

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

1.18 Scalar Time

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

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

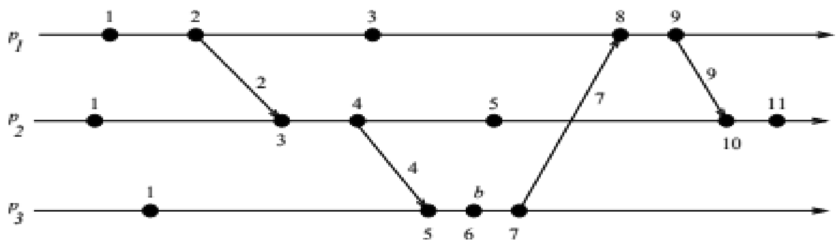
R2: Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:

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

- Figure shows evolution of scalar time.

Evolution of scalar time:

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



Basic Properties

Consistency Property

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

Total Ordering

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

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

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

Event counting

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

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

No Strong Consistency

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

1.19 Vector Time

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

describes the logical time progress at process p_i.

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

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

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

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

R1: Before executing an event, process p_i updatesitslocallogicaltimeas 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 the receipt of such a message *(m,vt)*, process p_i executes the following sequence of actions:

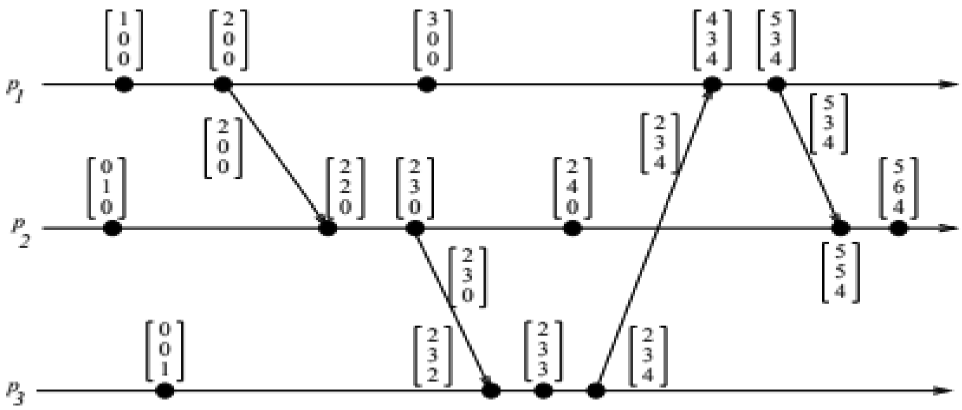
1. ***Update its global logical time as follows:***
   $1 \le k \le n : vt_i[k] := \max(vt_i[k], vt[k])$
2. ***Execute R1.***
3. ***Deliver the message m.***

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

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

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

<u>An Example of Vector Clocks</u>



<u>Comparing Vector Timestamps</u>

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

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

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

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

Basic Properties of Vector Time

Isomorphism

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

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

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

Strong Consistency

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

Event Counting

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

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

Applications

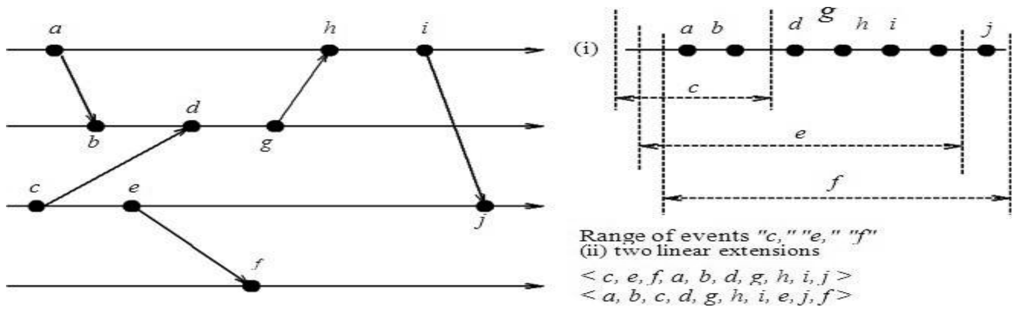
- Distributed debugging,
- Implementations of causal ordering,
- Communication and causal distributed shared memory,
- Establishment of global breakpoints
- Determining the consistency of checkpoints in optimistic recovery

Size of vector clocks

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

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

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



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

1.20 Physical Clock Synchronization: NTP

Motivation

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

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

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

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

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

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

Definitions and Terminology

Let C_a and C_b be any two clocks.

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

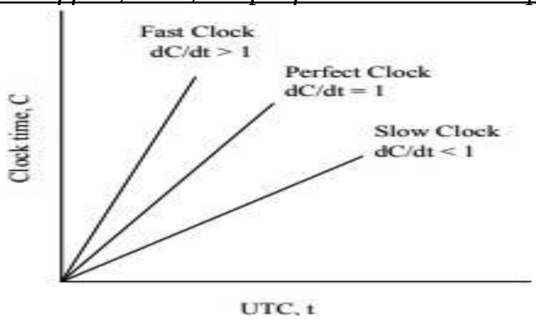
Clock Inaccuracies

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

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

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

Figure illustrates the behavior of fast, slow, andperfect clockswith respect to UTC.



Offset delay estimation method

The ***Network Time Protocol (NTP)*** which is widely used for clock synchronizationonthe Internet usesthe ***OffsetDelayEstimation*** method.

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

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

Clock offset and delay estimation:

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

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

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

Offset and delay estimation.

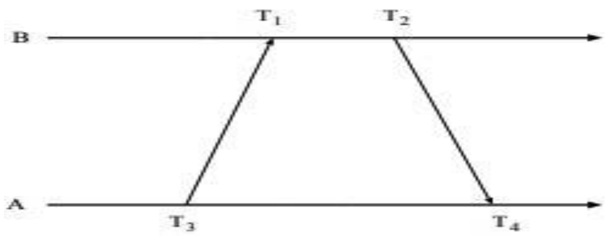


Figure 3.6: Offset and delay estimation.

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

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

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

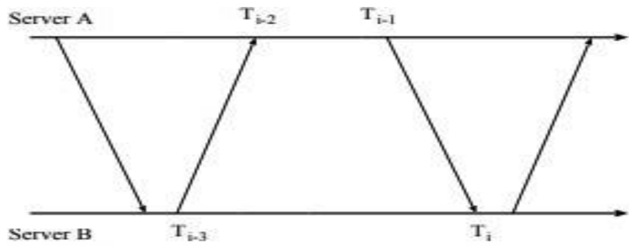


Figure 3.7: Timing diagram for the two servers.

DISTRIBUTED SYSTEMS

QUESTIONS:

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