SS ZG 526: Distributed Computing (CS1)

Introduction

Prof. Geetha

Associate Professor, Dept. of Computer Sc. & Information Systems
BITS Pilani Hyderabad Campus

geetha@hyderabad.bits-pilani.ac.in

BITS Pilani
Hyderabad Campus

innovate achieve lead

Course Scope & Objectives

No	Objective
CO1	This course will cover various hardware architectures for building distributed systems, and their communication models.
CO2	This will help students understand the design aspects of various software applications that can be deployed on various distributed systems.
CO3	This will provide an understanding of the complexities and resource management issues that are critical in a large distributed system.
CO4	This course will cover algorithmic aspects of building/designing distributed systems in domains like IoT, P2P, Cluster, Grid computing etc.

innovate achieve lead

Course Material

Text Book

No.	Author(s), Title, Edition, Publishing House
T1	Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

Reference Books

No.	Author(s), Title, Edition, Publishing House
R1	John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.
R2	Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

innovate achieve lead

Course Evaluation Scheme

No.	Name	Type	Duration	Weight	Day, Date, Session, Time
EC-1	Quiz-I/ Assignment-I	Online	-	5%	September 10-20, 2020
	Quiz-II	Online		5%	October 20-30, 2020
	Assignment-II	Online		10%	November 10-20, 2020
EC-2	Mid-Semester Test	Closed Book	2 hours	30%	Saturday, 10/10/2020 (AN) 2 PM – 4 PM
EC-3	Comprehensive Exam	Open Book	3 hours	50%	Saturday, 28/11/2020 (AN) 2 PM – 5 PM

innovate achieve lead

Distributed Computing

innovate achieve lead

Modular Overview

- M1: Introduction to Distributed Computing
- M2: Logical Clocks & Vector Clocks
- M3: Global state and snapshot recording algorithms
- M4: Message ordering and Termination detection
- M5: Distributed Mutual Exclusion
- M6: Deadlock Detection
- M7: Consensus and Agreement Algorithm
- M8: Peer to Peer Computing and Overlay graphs
- M9: Cluster computing, Grid Computing
- M10: Internet of Things

What is a distributed system?

- A computing environment which deals with all forms of computing, information access, and information exchange across multiple processing platforms connected by computer networks

Ex: Banking systems; Communication systems, Information systems (WWW),Manufacturing and process control, Inventory Systems, General purpose automation systems etc,

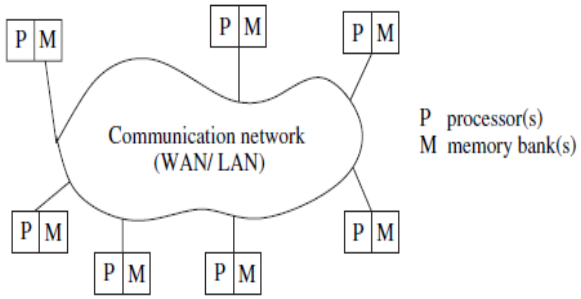
- So, what is a good definition of a distributed system?
- A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.
- Distributed systems have been in existence since many years

Characteristics of a distributed system

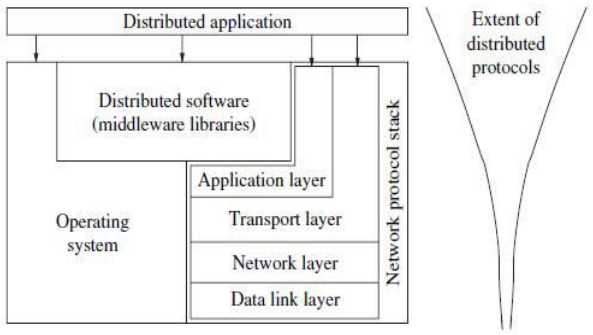
- Autonomous processors communicating over a communication network make a distributed system
  - ✓No common physical clock
  - ✓No shared memory
  - ✓Geographical separation
  - ✓Autonomy and heterogeneity

Ideal Distributed System?

Distributed System model



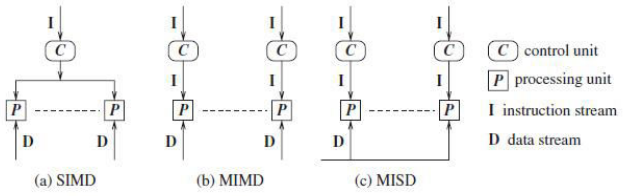
Software Components and their interaction in Distributed environment



Why do we need distributed systems?

- ✓ Inherently distributed computation
- ✓ Resource sharing
- ✓ Access to remote resources
- ✓ Increased performance/cost ratio
- ✓ Reliability
- ✓ Availability, integrity, fault-tolerance
- ✓ Scalability, Modularity and incremental expandability

Flynn's Taxonomy



1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data streams (SIMD)
   - Vector architectures
   - Multimedia extensions
   - Graphics processor units
3. Multiple instruction streams, single data stream (MISD)
   - No commercial implementation
4. Multiple instruction streams, multiple data streams (MIMD)
   - Tightly-coupled MIMD
   - Loosely-coupled MIMD

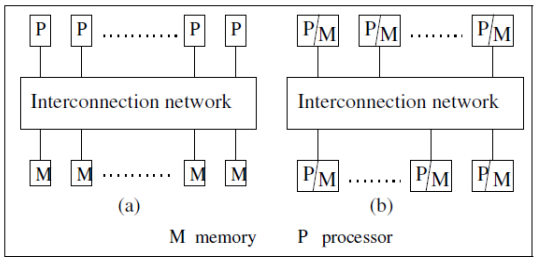
Distributed vs Parallel vs Multiprocessor systems

Parallel Systems

- Multiprocessor systems (direct access to shared memory, UMA model)
  - Interconnection network - bus, multi-stage sweitch
  - E.g., Omega, Butterfly, Clos, Shuffle-exchange networks
  - Interconnection generation function, routing function
- Multicomputer parallel systems (no direct access to shared memory, NUMA model)
  - bus, ring, mesh (w w/o wraparound), hypercube topologies
  - E.g., NYU Ultracomputer, CM* Conneciton Machine, IBM Blue gene
- Array processors (colocated, tightly coupled, common system clock)
  - Niche market, e.g., DSP applications

Contd..

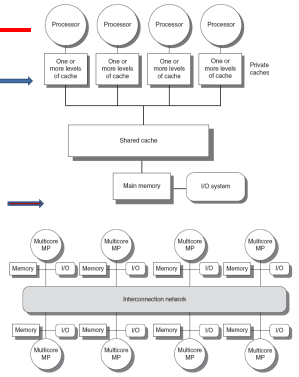
Multiprocessor/Multi computer Systems



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

Types of multiprocessing

- Centralized Shared Memory (CSM) or Symmetric multiprocessors (SMP)
  - Small number of cores
  - Share single memory with uniform memory latency (UMA)
- Distributed shared memory (DSM)
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks

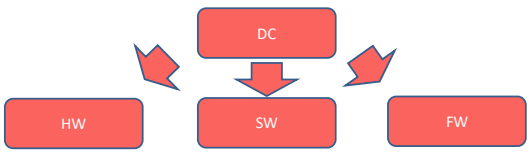


How does a distributed computing system work?

- The machines that are part of a distributed system may be computers, physical servers, virtual machines, containers, or any other node that can connect to the network, have local memory, and communicate by passing messages
- Two general ways that distributed systems function are:

1. Each machine works toward a common goal and the end-user views results as one cohesive unit
2. Each machine has its own end-user and the distributed system facilitates sharing resources or communication services

What all things constitute a distributed computing environment?



Tag cloud representation of distributed computing

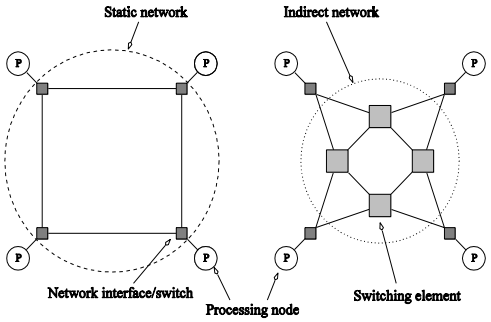
Interconnection networks in Distributed Computing

- Interconnection networks are composed of switching elements
- Topology is the pattern to connect the individual switches to other elements, like processors, memories and other switches
- A network allows exchange of data between processors in the distributed computing system

Types of Interconnection networks

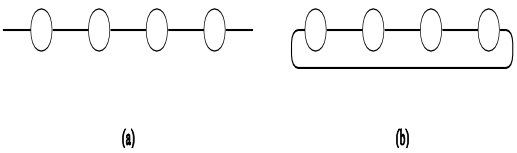
- **Direct connection networks** – Direct (static) networks have point-to-point connections between neighboring nodes. Ex:
  - Rings
  - Meshes
  - Cubes
- **Indirect connection networks** – Indirect (dynamic) networks have no fixed neighbors. The communication topology can be changed dynamically based on the application demands. Ex:
  - Bus networks
  - Multistage networks
  - Cross bars

Static and Dynamic Interconnection Networks



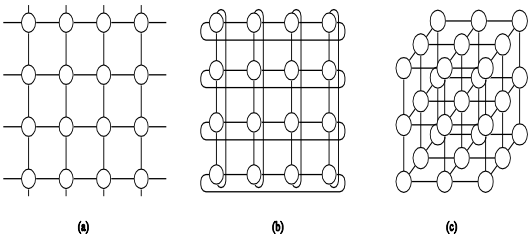
Classification of interconnection networks:
(a) static network (b) dynamic network

Network Topologies: Linear Arrays



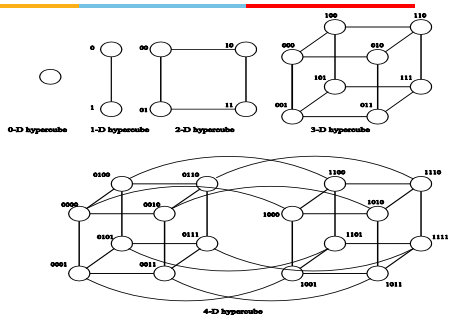
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

Network Topologies: Two- and Three Dimensional Meshes



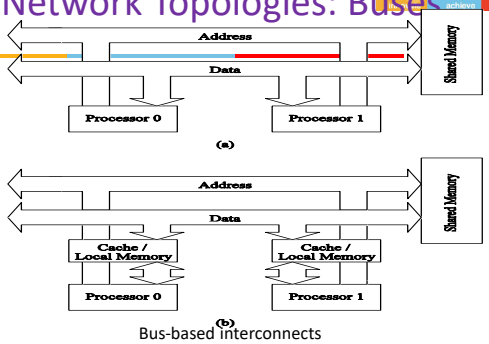
Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.
(3-D weather modelling, structural modelling physical simulations)

Network Topologies: Hypercubes and their Construction



Construction of hypercubes from hypercubes of lower dimension.

Network Topologies: Buses

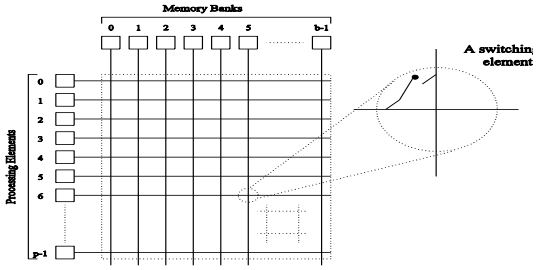


Bus-based interconnects
(a) with no local caches; (b) with local memory/caches.

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.

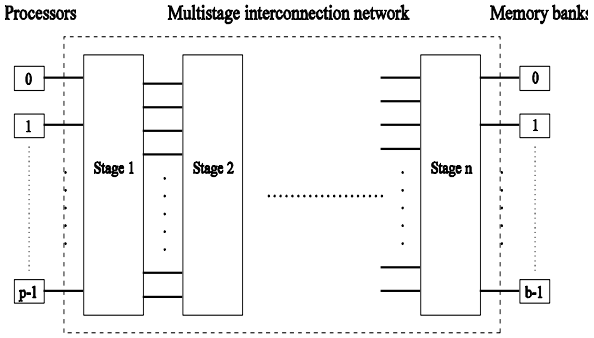
Network Topologies: Crossbars

A crossbar network uses an *p*×*m* grid of switches to connect *p* inputs to m outputs in a non-blocking manner.



A completely non-blocking crossbar network connecting *p* processors to b memory banks.

Network Topologies: Multistage Networks



The schematic of a typical multistage interconnection network.

Network Topologies: Multistage Omega Network

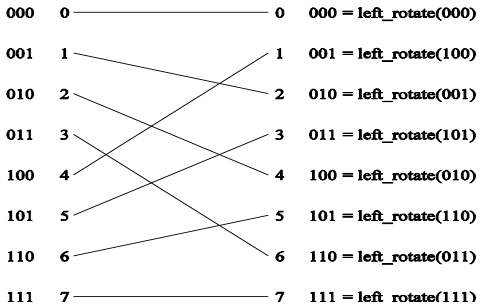
- One of the most commonly used multistage interconnects is the Omega network.
- This network consists of *log p* stages, where *p* is the number of inputs/outputs (processing nodes as well as memory banks)
- At each stage, input *i* is connected to output *j* if:

$$j = \begin{cases} 2i, & 0 \le i \le p/2 - 1 \\ 2i + 1 - p, & p/2 \le i \le p - 1 \end{cases}$$

- This is actually a left-rotation operation on the binary representation of i to obtain j

Network Topologies: Multistage Omega Network

Each stage of the Omega network implements a perfect shuffle as follows:



A perfect shuffle interconnection for eight inputs and outputs.

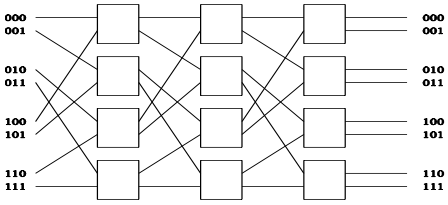
Network Topologies: Multistage Omega Network

- The perfect shuffle patterns are connected using 2×2 switches.
- The switches operate in two modes – crossover or pass through.

Two switching configurations of the 2 × 2 switch:
(a) Pass-through; (b) Cross-over.

Network Topologies: Multistage Omega Network

A complete Omega network with the perfect shuffle interconnects and switches can now be illustrated:

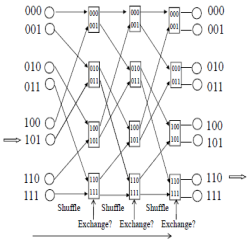


A complete omega network connecting eight inputs and eight outputs.

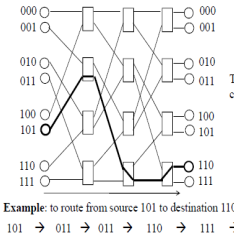
An omega network has *p/2 × log p* switching nodes, and the cost of such a network grows as *(p log p)*.

Routing in Omega network

((Unique route between a source and a destination - rules)



$$\text{Shuffle}(x_{q-1}, x_{q-2}, \ldots, x_0) = x_{q-2}, \ldots, x_0, x_{q-1}$$
$$\text{Exchange}(x_{q-1}, x_{q-2}, \ldots, x_0) = x_{q-1}, x_{q-2}, \ldots, \overline{x_0}$$

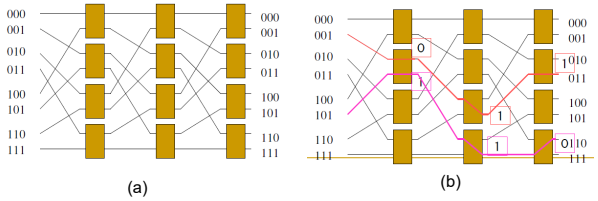


The destination $d_2 d_1 d_0$ is coded in message header

Example: to route from source 101 to destination 110 (101 xor 110 = 011)

101 → 011 → 011 → 110 → 111 → 111 → 110
shuffle straight shuffle exchange cross shuffle exchange cross

Omega network - example



1. **For the omega network given above in (a), find the number of switching elements**

Answer = ½ n$\log_2$n = 12

2. **How to apply the perfect shuffle technique for destination routing in (b)?**

First step is to take the xor of the source and destination, which will give the sequence.

Ex: 1 => 3

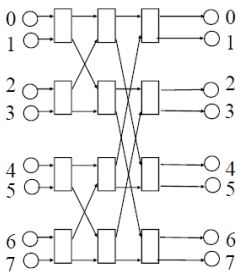
Answer: 001->010->010->100->101->011->011 (the sequence is straight-cross-straight)

5 => 6

Answer: 101->011->011->110->111->111->110 (the sequence is straight-cross-cross)

Network Topologies: Butterfly Networks

- A butterfly network is a technique to link multiple computers into a high-speed network
  - This form of multistage interconnection network topology can be used to connect different nodes in a multiprocessor system
  - Butterfly networks have lower diameter than other topologies like a linear array, ring and 2-D mesh. This implies that in butterfly network, a message sent from one processor would reach its destination in a lower number of network hops
  - Butterfly networks have higher bisection bandwidth than other topologies. This implies that in butterfly network, a higher number of links need to be broken in order to prevent global communication



A butterfly network

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

Coupling in MIMD Architectures

1. Tightly coupled multiprocessors (with UMA shared memory)
2. Tightly coupled multiprocessors (with NUMA shared memory or that communicate by message passing)
3. Loosely coupled multi computers (without shared memory) physically collocated
4. Loosely coupled multi computers (without shared memory and without common clock) that are physically remote

Parallelism

- 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
- Speedup (***S***) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements
- 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 is an aggregate measure of the percentage of time that all the processors are executing CPU instructions productively, as opposed to waiting for communication (either via shared memory or message-passing) operations to complete

- $S = T_{serial}/T_{parallel}$

Concurrency: Definition

- ✓ A broader term that means roughly the same as parallelism of a program, but is used in the context of distributed programs
- ✓ 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

Concurrency

- The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*
- If ***C(W)*** is the degree of concurrency of a parallel algorithm, then for a problem of size ***W***, no more than ***C(W)*** processing elements can be employed effectively

Granularity in Distributed Computing

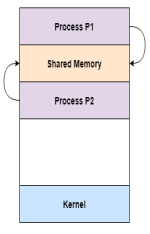
- **The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity**
- If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions, compared to the number of times the processors communicate either via shared memory or message passing and wait to get synchronized with the other processors
- Programs with fine-grained parallelism are best suited for tightly coupled systems
  - These typically include SIMD and MISD architectures, tightly coupled MIMD multiprocessors (that have shared memory), and loosely coupled multi computers (without shared memory) that are physically co located

Communication Paradigms for Distributed Computing

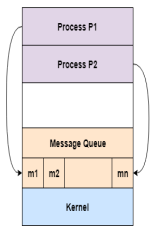
- Recap on interaction of software components in Distributed Computing (CORBA, RPC, DCOM,RMI etc.,)
- A distributed computation involves a number of processes communicating with one another and inter process communication mechanism is fairly complex
- many dimensions of variability in distributed systems like network topology, inter process communication mechanisms, failure classes, and security mechanisms
- Two major paradigms are:

1. **Shared Memory Process Communication Model**
2. **Message Passing Process Communication Model**

Shared Memory vs Message Passing



Shared Memory Model



Message Passing Model

Shared Memory Process Communication Model

Message Passing Process Communication Model

Message-passing and Shared Memory - emulation

- Emulating MP over SM:
  - Partition shared address space
  - Send/Receive emulated by writing/reading from special mailbox per pair of processes
- Emulating SM over MP:
  - 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

Implementing shared memory paradigm

- ❑ SystemV IPC
- ❑ POSIX IPC
- ❑ OpenMP API

Implementing Message passing paradigm

- ❑ MPI
- ❑ MPICH

Classification of primitives

- ❖ Synchronous (send/receive)
  - ❖ Handshake between sender and receiver
  - ❖ Send completes when Receive completes
  - ❖ Receive completes when data copied into buffer
- ❖ Asynchronous (send)
  - ❖ Control returns to process when data copied out of user-specified buffer

- ➢ Blocking (send/receive)
  - ➢ Control returns to invoking process after processing of primitive (whether sync or async) completes
- ➢ Nonblocking (send/receive)
  - ➢ Control returns to process immediately after invocation
  - ➢ Send: even before data copied out of user buffer
  - ➢ Receive: even before data may have arrived from sender

Non-blocking Primitive

nonblocking send primitive

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

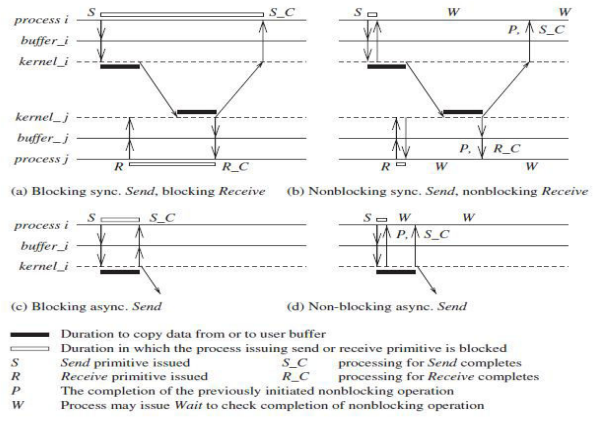
...

...

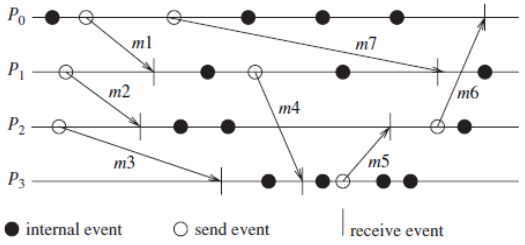
$Wait(handle_1, handle_2, \ldots, handle_k, \ldots, handle_m)$ //*Wait* always blocks

- ➢ When the Wait call returns, at least one of its parameters is posted
- ➢ 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/non blocking; Synchronous/asynchronous; send/receive primitives

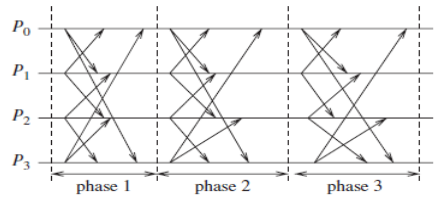


Asynchronous Executions in a Message-passing System



Processor synchrony ⟹ step

Synchronous Executions in a Message-passing System



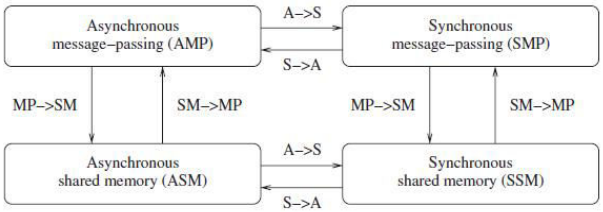
Synchronous execution in a message-passing system In any round/step/phase: (send | internal) • (receive | internal)•

Sync Execution(int k, n) //k rounds, n processes.

(1) for r = 1 to k do
(2) proc i sends msg to (i + 1) mod n and (i − 1) mod n;
(3) each proc i receives msg from (i + 1) mod n and (i − 1) mod n;
(4) compute app-specific function on received values.

System emulations: virtual synchrony

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

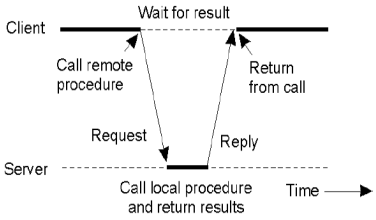


- Sync ↔ async, and shared memory ↔ msg-passing emulations
  Assumption: failure-free system
- System A emulated by system B
- If not solvable in B, not solvable in A
- If solvable in A, solvable in B

Distributed Communication Models

- Remote Procedure Call (RPC)
- Publish/Subscribe Model
- Message Queues

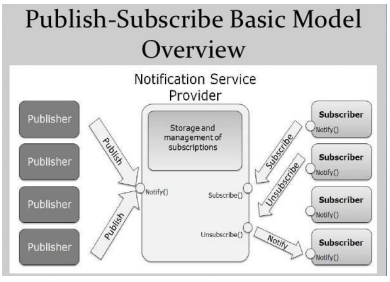
RPC between Client & Server



Source: http://web.cs.wpi.edu/~rek/DCS/D04/Communication.pdf

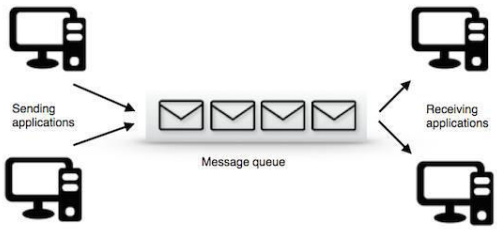
Real life example for RPC: Use of RPC in CERN Research

Publish/ Subscribe Model



Real life examples of Pub/Sub model: Google Cloud Pub/Sub

Distributed Message Queues



Real world example: ERP servers sending messages to CRM

Design Issues & Challenges – System Perspective

- Communication mechanisms
- Processes
- Naming
- Synchronization
- Data storage and access
- Consistency and replication
- Fault-tolerance
- Security
- Scalability and modularity
- API and transparency

Algorithmic Challenges

- Designing useful execution models and frameworks: to reason with and design correct distributed programs
- Dynamic distributed graph algorithms and routing algorithms
- Load balancing: to reduce latency, increase throughput, done dynamically
- Real-time scheduling: difficult without global view, network delays make task harder
- Performance modeling and analysis: Network latency to access resources must be reduced
- Synchronization/coordination mechanisms
- Group communication, multicast, and ordered message delivery
- Monitoring distributed events and predicates
- Distributed program design and verification tool
- Time and global state
- Debugging distributed programs
- Data replication, consistency models, and caching
- World Wide Web design: caching, searching, scheduling
- Distributed shared memory abstraction
- Reliable and fault-tolerant distributed systems

Recap Quiz

1. Which of the following is not one of the communication mechanisms in distributed computing?

(a) RPC (b) RMI (c) ROI (d) RTI

2. Let Ts = 10 time units be the sequential time of execution in a distributed system and let Tp = 8 time units be the parallel time. What is the speedup? Assume other delays are negligible.

(a) 1.25 (b) 1.0 (c) 0.8 (d) 1.8

3. In distributed computing processor synchrony is achieved through units of execution called

(a) Task (b) step (c) process (d) thread

4. What is the granularity of a distributed computing environment if the amount of computation is 100 units and the amount of communication is 36 units?

(a) 0.36 (b) 2.78 (c) 1.36 (d) 0.64

5. The maximum number of tasks that can be executed simultaneously at any time in a distributed computing environment is called the degree of

(a) Efficiency (b) granularity (c) consistency (d) concurrency

Recap Quiz key

Q1	Q2	Q3	Q4	Q5
d	a	b	b	d

Major references

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 1, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008.
- http://www.ois.com/Products/what-is-corba.html
- http://wiki.c2.com/?PublishSubscribeModel
- https://www.slideshare.net/ishraqabd/publish-subscribe-model-overview-13368808/5
- https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html
- https://en.wikipedia.org/wiki/Message_queue

Introduction

- concept of causality between events → important for design and analysis of parallel and distributed computing and operating systems
- causality is tracked using physical time
- But in distributed systems
  - not possible to have global physical time
  - possible to realize only an approximation of physical time
- way of measuring time
  - asynchronous distributed computations progress in spurts
  - logical time also advances in jumps
  - sufficient to capture the fundamental monotonicity property associated with causality in distributed systems

Introduction

- **monotonicity property: if event *a* causally affects event *b*, then timestamp of *a* is smaller than timestamp of *b***
- Ways to represent logical time:
  - scalar time
  - vector time

Introduction

- **Causality / Causal Precedence Relation**
- described among events in a distributed system
- useful in reasoning, analyzing, and drawing inferences about a computation
- helps to solve several problems in distributed systems
  - distributed algorithms design
  - tracking of dependent events
  - knowledge about progress
  - concurrency measure

Introduction

➢ **Distributed algorithms design**

knowledge of the causal precedence relation among events helps to:
- ensure liveness and fairness in mutual exclusion algorithms
- maintain consistency in replicated databases
- design correct deadlock detection algorithms

➢ **Tracking of dependent events**
- helps construct a consistent state for resuming re-execution
- in failure recovery, helps build a checkpoint
- in replicated databases, aids in detection of file inconsistencies

Introduction

➢ **Knowledge about progress - knowledge of causal dependency among events**
- helps measure progress of processes in distributed computation
- helps to discard obsolete information, garbage collection, and termination detection

➢ **Concurrency measure - knowledge regarding how many events are causally dependent**
- useful in measuring amount of concurrency
- events not causally related are concurrent
- analysis of causality tells about concurrency in program

Capturing Causality between Events

- in distributed systems
  - rate of event occurrence is of high magnitude
  - event execution time is of low magnitude
- physical clocks must be precisely synchronized
- in distributed computation
  - clocks accurate to a few tens of milliseconds are not sufficient
  - progress occurs in spurts
  - interaction between processes occurs in spurts
- logical clocks can accurately capture
  - causality relation between events produced by a program execution
  - fundamental monotonicity property

Logical Clocks

- every process has a logical clock
- logical clock is advanced using a set of rules
- each event is assigned a timestamp
- causality relation between events can be inferred from their timestamps
- timestamps follow the monotonicity property

Representing Logical Clocks

1. Lamport's scalar clocks
   - time is represented by non-negative integers
2. Vector clocks (Fidge, Mattern and Schmuck)
   - time is represented using a vector of non-negative integers

Real world applications of Logical clocks

- Scalar clocks => Amazon S3
- Vector clocks => Voldemort (Linkedin) Amazon Dynamo, Version control systems etc.

Self assessment Question:
- Why Lamport's clock cannot be used in blockchain technology?

Definition of Logical Clocks

- system of logical clocks consists of
  - time domain T
  - logical clock C
- elements of T form a partially ordered set over a relation $<$
- $<$:
  - happened before or causal precedence relation
  - analogous to 'earlier than' relation provided by physical time

Definition of Logical Clocks

- logical clock C
  - is a function that maps an event e in a distributed system to an element in the time domain T
  - denoted as $C(e)$
  - $C(e)$: timestamp of e
  - *C is defined as:* $C : H \mapsto T$
  - satisfies the following property –
    - for any 2 events e_i and e_j, $e_i \rightarrow e_j \implies C(e_i) < C(e_j)$.
    - monotonicity property
    - known as *clock consistency condition*

Definition of Logical Clocks

- system of clocks is *strongly consistent* if T and C satisfy the following:
  - for 2 events e_i and e_j,

$$e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$$

Implementing Logical Clocks

- Two issues need to be addressed
  - data structures local to every process to represent logical time
  - a protocol to update the data structures to ensure the consistency condition

Implementing Logical Clocks

Data Structures

- Each process p_i maintains two data structures
- A *local logical clock*
  - denoted by lc_i
  - helps process p_i measure its own progress
- A *logical global clock*
  - denoted by gc_i
  - represents p_i's local view of the logical global time
  - allows p_i to assign consistent timestamps to its local events
  - lc_i *is a part of* gc_i

Implementing Logical Clocks

Protocol - ensures consistent management of:

- a process's logical clock
- process's view of global time

- consists of the following 2 rules:
- **R1:** governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal)
- **R2:**
  - governs how a process updates its global logical clock to update its view of the global time and global progress
  - dictates what information about logical time is piggybacked in a message
  - how this information is used by the receiving process to update its view of the global time

Implementing Logical Clocks

- systems of logical clocks differ in
  - representation of logical time
  - the protocol to update the logical clocks
- all logical clock systems
  - implement rules **R1** and **R2**
  - ensure the fundamental monotonicity property associated with causality
  - provide users with some additional properties

Scalar Time - Definition

- representation was proposed by Lamport in 1978 to totally order events in distributed system
- in this representation, time domain is the set of non-negative integers
- C_i:
  - integer variable
  - denotes logical local clock of p_i and its local view of global time

Scalar Time - Definition

Rules for updating clock:

- **R1: Before executing an event (send, receive, or internal), process p_i executes** $C_i := C_i + d \qquad (d > 0)$

For each execution of **R1**,

- *d can have different value*
- value of *d* may be application-dependent

Usually $d = 1$

- helps to identify the time of each event uniquely at a process
- keeps rate of increase of *d* to lowest level

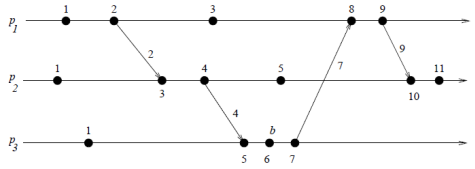
Scalar Time - Definition

Rules for updating clock (contd. from previous slide):

R2: When 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.

Evolution of Scalar Time



Space-time diagram of a distributed system

Scalar Time – Basic Properties

- Consistency
- Total Ordering
- Event Counting
- No Strong Consistency

Scalar Time – Basic Properties

Consistency Property

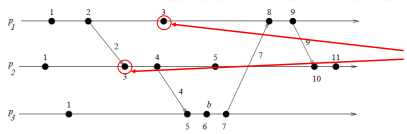
- for two events e_i and e_j, $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$
- scalar clocks satisfy monotonicity property
- hence, they also satisfy consistency property

Scalar Time – Basic Properties

Total Ordering

- Scalar clocks are used to totally order events in a distributed system
- Problem in totally ordering events – 2 or more events at different processes may have same timestamp

for 2 events e_1 and e_2, $C(e_1) = C(e_2) \Longrightarrow e_1 \parallel e_2$



same timestamp

Scalar Time – Basic Properties

Total Ordering

- Tie breaking mechanism is needed
- Tie breaking procedure:
  - process identifiers are linearly ordered
  - break tie among events with identical scalar timestamps on the basis of their process identifiers
  - lower process identifier implies higher priority
  - timestamp of an event is denoted by a tuple *(t, i)*
    - *t* → *time of occurrence*
    - *i* → *identity of the process where it occurred*

Scalar Time – Basic Properties

Total Ordering

- Total order relation $\prec$ on 2 events x and y with timestamps (h, i) and (k, j) respectively, is defined:

  $x \prec y \Leftrightarrow h < k$ **or** $(h = k$ **and** $i < j)$
- events that occur at the same logical scalar time are independent (i.e., they are not causally related)
- such events can be ordered using any arbitrary criterion without violating the causality relation →
- so, total order is consistent with the causality relation "→"
- Note:- $x \prec y \Rightarrow x \rightarrow y \vee x \,||\, y$

Scalar Time – Basic Properties

Total Ordering

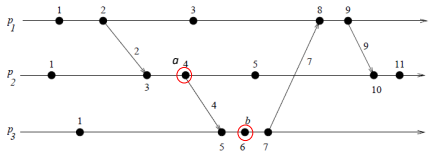
- Utility
  - total order is generally used to ensure liveness properties in distributed algorithms
    - Liveness property:
      - something good eventually happens
      - system makes progress, no starvation, programs terminate
  - requests are timestamped and served according to the total order based on these timestamps

Scalar Time – Basic Properties

Event Counting

- If the increment value $d = 1$, scalar time has the following property:
  - if event e has a timestamp h, then $h - 1$
    - is the minimum logical duration required before producing event e
    - counted in units of events
    - called the ***height*** of event e
  - implies that h - 1 events have been produced sequentially before event e, irrespective of the processes that produced these events

Scalar Time – Basic Properties

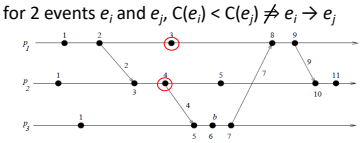


- height of event $a = 4 - 1 = 3$, i.e., 3 events precede a on the longest causal path ending at a
- height of event $b = 6 - 1 = 5$, i.e., 5 events precede b on the longest causal path ending at b

Scalar Time – Basic Properties

No strong consistency

- system of scalar clocks is not strongly consistent
- for 2 events e_i and e_j, $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$

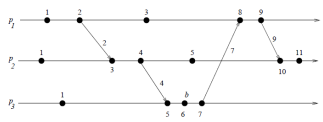


- scalar timestamp of 3rd event of p_1 < scalar timestamp of 3rd event of p_2
- but the former did not happen before the latter

Scalar Time – Basic Properties

No strong consistency

- scalar clocks are not strongly consistent because logical local clock and logical global clock of a process are combined into one
- results in the loss causal dependency information among events at different processes



- when p_2 receives the 1st message from p_1, p_2 updates its clock to 3
- forgets that the timestamp of the latest event at p_1 on which it depends is 2

Vector Time - Definition

- time domain is represented by a set of *n*-dimensional non-negative integer vectors
- each process p_i maintains a vector $vt_i[1..n]$
  - $vt_i[i]$ *: local logical clock of* p_i
  - describes progress of logical time at p_i
  - $vt_i[j]$*: represents process* p_i*'s latest knowledge of process* p_j*'s local time*
  - if $vt_i[j] = x$
    - p_i *knows that local time at* p_j *has progressed till* x
- vector vt_i constitutes p_i's view of the global logical time
- vt_i *is used to timestamp events*

Vector Time - Definition

Rules used by p_i for clock updating

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

$$vt_i[i] = vt_i[i] + d, \quad d > 0$$

Vector Time - Definition

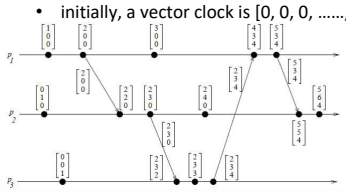
Rules used by p_i for clock updating

- **R2:**
  - each message *m* is piggybacked with the vector clock *vt* of the sender process at sending time
  - when p_i receives message (m, vt), it executes the following sequence of actions:
    1. update its global logical time as:
       $1 \le k \le n$: $vt_i[k] = max(vt_i[k], vt[k])$;
    2. execute R1;
    3. deliver the message *m*.

Vector Time - Definition

Rules used by p_i for clock updating

- **R2:**
  - timestamp associated with an event is the value of the vector clock of its process when the event is executed
  - initially, a vector clock is [0, 0, 0, ……, 0]



vector clock's progress with $d = 1$

Vector Time - Definition

Relations for comparing 2 vector timestamps *vh* and *vk*

- $vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$
- $vh \le vk \Leftrightarrow \forall x : vh[x] \le vk[x]$
- $vh < vk \Leftrightarrow vh \le vk$ *and* $\exists x : vh[x] < vk[x]$
- $vh \,||\, vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)$

Vector Time – Basic Properties

- Isomorphism
- Strong Consistency
- Event Counting

Vector Time – Basic Properties

Isomorphism

- if events in a distributed system are timestamped using a system of vector clocks, following property holds:
  - if 2 events *x* and *y* have timestamps *vh* and *vk*, respectively, then
    $x \rightarrow y \Leftrightarrow vh < vk$
    $x \,||\, y \Leftrightarrow vh \,||\, vk$
- there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps
- very powerful, useful, and interesting property of vector clocks

Vector Time – Basic Properties

Isomorphism

- if the process at which an event occurred is known, test for comparing 2 timestamps can be simplified as:
  - if events *x* and *y* respectively occurred at processes p_i and p_j and have timestamps *vh* and *vk* respectively, then

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

Vector Time – Basic Properties

Strong Consistency

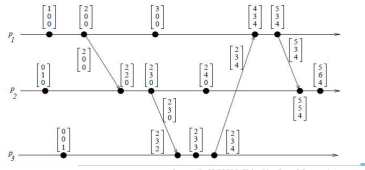
- system of vector clocks is strongly consistent
- examination of vector timestamps of 2 events can determine if the events are causally related
- dimension of vector clocks can't be less than *n* for this property to hold

(*n* = total no. of processes in the distributed computation)

Vector Time – Basic Properties

Event counting

- if *d* is always 1 in rule **R1** and $vt_i[i]$ is the i^{th} component of vector clock at process p_i
  - then $vt_i[i]$ = no. of events that have occurred at p_i until that instant

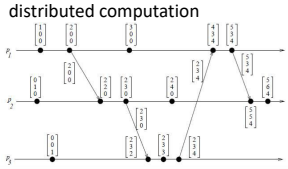


Vector Time – Basic Properties

Event counting

•if an event *e* has timestamp *vh*

- *vh*[*j*] = number of events executed by process p_j that causally precede *e*
- Σ *vh*[*j*] −1 : total no. of events that causally precede *e* in the distributed computation



Efficient Implementation of Vector Clocks

Why is efficient implementation necessary?

- when no. of processes in a distributed computation is large
  - vector clocks will require piggybacking of huge amount of information in messages
  - messages are required for disseminating time progress and updating clocks
  - message overhead grows linearly with the no. of processes
  - message overhead is not affected by the no. of events occurring at the processors

Singhal–Kshemkalyani's Differential Technique

- based on the observation - between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change
- this is more likely when the number of processes is large
  - reason: only a few of them will interact frequently by passing messages
- fundamental idea behind the technique
  - when process p_i sends a message to process p_j, it piggybacks only those entries of its vector clock that differ since the last message sent to p_j

Singhal–Kshemkalyani's Differential Technique

•if entries $i_1, i_2, ..., i_{n_1}$ of the vector clock at p_i have changed to $v_1, v_2, ..., v_{n_1}$, respectively, since the last message sent to p_j ,

•then p_i piggybacks a compressed timestamp of the form

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

to the next message to p_j

- on receiving this message, p_j updates its vector clock as follows:

$vt_j[i_k] = max(vt_j[i_k], v_k)$ *for* $k = 1, 2,...., n_1$

Singhal–Kshemkalyani's Differential Technique

- **Benefit:**
  - reduces message size, communication bandwidth and buffer (to store messages) requirements
- **Worst case:**
  - every element of the vector clock has been updated at p_i since the last message sent to p_j
  - next message from p_i to p_j will need to carry the entire vector timestamp of size n
- **Average case: size of the timestamp on a message will be less than n**

Singhal–Kshemkalyani's Differential Technique

•Requirements for implementation:

•each process must remember the vector timestamp in the message last sent to every other process

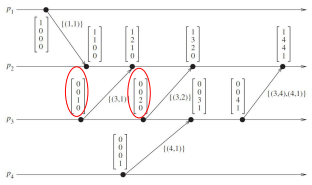
•communication channels follow FIFO discipline for delivery of messages

Singhal–Kshemkalyani's Differential Technique

- in general, the timestamp is of the form:

$\{(p_1, latest\_value), (p_2, latest\_value),...\}$

p_i *indicates* p_i^{th} *component of the vector clock has changed*



- 2nd message from p_3 to p_2 contains a timestamp {(3, 2)}
- informs p_2 that the 3rd component of the vector clock has been modified and the new value is 2

Singhal–Kshemkalyani's Differential Technique

Benefits:

- cost of maintaining vector clocks in large systems can be substantially reduced
- especially if the process interactions exhibit temporal or spatial localities
- useful in applications
  - causal distributed shared memories
  - distributed deadlock detection
  - enforcement of mutual exclusion and localized communications

Fowler–Zwaenepoel's Direct-Dependency Technique

- reduces the size of messages by transmitting only a scalar value
- no vector clocks are maintained on-the-fly
- process only maintains information regarding direct dependencies on other processes
- vector time for an event represents transitive dependencies on other processes
- vector time for event is constructed off-line from a recursive search of the direct dependency information at processes

Fowler–Zwaenepoel's Direct-Dependency Technique

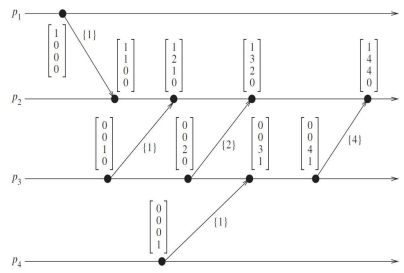
- *p_i maintains a dependency vector D_i*
- initially, $D_i[j] = 0$ for $j = 1,.....,n$
- *D_i is updated using the following rules:*

1. When an event occurs at p_i, $D_i[i] = D_i[i] + 1$ (i.e., increment the vector component corresponding to its own local time by one)

2. When p_i sends a message to p_j , it piggybacks the updated value of $D_i[i]$ in the message

3. When p_i receives a message from p_j with piggybacked value d, p_i updates its dependency vector as follows: $D_i[j] = max\{D_i[j], d\}$

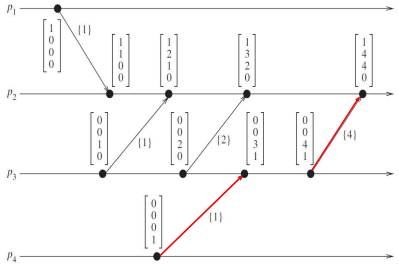
Fowler–Zwaenepoel's Direct-Dependency Technique



Fowler–Zwaenepoel's Direct-Dependency Technique

- *D_i reflects only direct dependencies*
- at any instant, $D_i[j]$ denotes the sequence no. of the latest event on p_j that ***directly*** affects the current state
- this event may precede the latest event at p_j that *causally* affects the current state

Fowler–Zwaenepoel's Direct-Dependency Technique



- when p_4 sends a message to p_3, it piggybacks a scalar that indicates the direct dependency of p_3 on p_4 because of this message
- then p_3 sends a message to p_2 piggybacking a scalar to indicate the direct dependency of p_2 on p_3 because of this message
- *p_2 is indirectly dependent on p_4 since p_3 is dependent on p_4*

Fowler–Zwaenepoel's Direct-Dependency Technique

- transitive (indirect) dependencies are not maintained by this method
- transitive dependencies
  - can be obtained only by recursively tracing the direct dependency vectors of the events off-line
  - involves computational overhead and latencies
- this method is ideal only for those applications
  - that do not require computation of transitive dependencies on the fly
  - eg. applications: causal breakpoints, asynchronous checkpoint recovery

Fowler–Zwaenepoel's Direct-Dependency Technique

- saves cost considerably
- not suitable for applications that require on-the-fly computation of vector timestamps

Physical Clock Synchronization

- no global clock or common memory
- each processor has its own internal clock and its own notion of time
- clocks can drift apart by several seconds per day, accumulating significant errors over time
- clock rates are different, may not remain always synchronized
- for most applications and algorithms that run in a distributed system, need to know time in one or more contexts:
  - time of the day at which an event happened on a specific machine in the network
  - time interval between two events that happened on different machines in network
  - relative ordering of events that happened on different machines in network

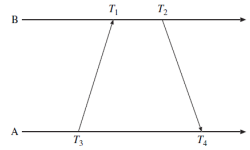
Physical Clock Synchronization

- Clock synchronization –
  - ensuring that physically distributed processors have a common notion of time
  - performed to correct clock skew in distributed systems
  - synchronized to an accurate real-time standard like UTC (Universal Coordinated Time)
- Offset –
  - difference between the time reported by a clock and the real time
  - offset of the clock C_a is given by $C_a(t) - t$ ($C_a(t)$ – time of clock)
  - offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$

Network Time Protocol

- uses offset delay estimation method
- involves a hierarchical tree of time servers
- primary server at the root synchronizes with the UTC
- 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

Network Time Protocol



- Each NTP message includes the latest three timestamps T_1, T_2, and T_3
- T_4 is determined upon arrival
- Peers A and B can independently calculate delay and offset using a bidirectional message stream

- T_1, T_2, T_3, T_4 - values of the 4 most recent timestamps as shown
- assume that clocks A and B are stable and running at the same speed
- $a = T_1 - T_3$, $b = T_2 - T_4$
- 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

$\theta = (a+b)/2$

$\delta = a - b$

Network Time Protocol

- A pair of servers 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).
- Each peer maintains pairs (O_i, D_i), where:
  - O_i – measure of offset (θ)
  - D_i – transmission delay of two messages (δ)
- Offset corresponding to the minimum delay is chosen
- Assume that message m takes time t to transfer and m' takes t' to transfer
- 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)
  - $A(t) = B(t) + O$
- Then, $T_{i-2} = T_{i-3} + t + O$, $T_i = T_{i-1} - O + t$

Network Time Protocol

- Assume t = t'
- Offset O_i can be estimated as $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - Ti)/2$
- Round-trip delay is estimated as $D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- 8 most recent pairs of (O_i, D_i) are retained
- Value of O_i that corresponds to minimum D_i is chosen to estimate O

Recap Quiz

1. If an event e has a timestamp h in scalar clock, then the minimum logical duration required before producing event e is
(a) h (b) h+1 (c) h – 1 (d) 1/h
2. If the height of an event b = 5, then how many events would have preceded b on the longest causal path ending at b if we use a scalar clock?
(a)4 (b) 5 (c) 3 (d) 2
3. In the tie breaking mechanism used in total ordering in scalar clocks, a lower process identifier implies ___ priority
(a) Equal (b) lower (c) higher (d) none of the above
4. Which of the following is not a mechanism to represent logical time in distributed computing systems?
(a) Scalar clock (b) vector clock (c) matrix clock (d)list clock
5. Consider the event counting in vector clocks. if an event e has timestamp vh, then the umber of events executed by process pj that causally precede e will be
(a) vh[j] (b)vh[j-1] (c) vh[j-1] (d)vh[0]
6. Let the total number of processes n = 8, in a distributed environment. Then the dimension of the vector clocks can't be _____ 8 for strong consistency
(a) < (b) > (c)<= (d) >=
7. Which of the following would ensure the efficient implementation of vector clocks if the process interactions exhibit temporal or spatial localities in a distributed system?
(a)Singhal-Kshemkalyani (b)Fowler-Zwaenepoel (c) physical clock synchronization (d) NTP
8. While implementing NTP in distributed computing systems, if A's local clock time is A(t) and B's local clock time is B(t), then the Offset O is
(a) A(t) + B(t) (b) A(t) * B(t) (c) A(t)/B(t) (d) A(t) – B(t)
9. Fowler-Zwaenepoel approach does not maintain ___ dependencies
(a) Commutative (b) associative (c) transitive (d) none of the above
10. In distributed computing systems, the clock skew can be corrected by _____ clock synchronization
(a) Scalar (b) vector (c) matrix (d) physical

Recap Quiz - key

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
c	b	c	d	a	a	a	d	c	d

Major References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 3, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

Introduction – global state and snapshot

- Recording the global state of a distributed system on-the-fly is required for analyzing, testing, or verifying properties associated with distributed executions
- problems in recording global state
  - lack of a globally shared memory
  - lack of a global clock
  - message transmission is asynchronous
  - message transfer delays are finite but unpredictable
- problem is non-trivial

Global state and snapshot contd..

- Every distributed system component has a local state
- state of a process is characterized by
  - state of its local memory
  - history of process activity
- channel state is characterized by the set of messages sent along the channel less the set of messages received along the channel
- Global state of a distributed system is a collection of the local states of its components
- Snapshot is the state of a system at a particular point in time

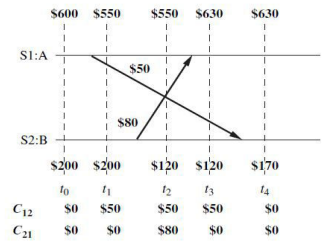
Global state and snapshot contd..

- **benefit of shared memory** - up-to-date state of the entire system is available to the processes sharing the memory
- absence of shared memory requires ways of getting a coherent and complete view of the system based on the local states of individual processes
- meaningful global snapshot can be obtained
  - if the components of the distributed system record their local states at the same time
  - requires local clocks at processes to be perfectly synchronized
  - existence of global system clock that could be instantaneously read by the processes

Global state and snapshot contd..

- technologically infeasible to have perfectly synchronized clocks at various sites
- clocks are bound to drift
- reading time from a single common clock maintained at one process will not work
- indeterminate transmission delays during read operation cause processes to identify various physical instants as the same time
- collection of local state observations will be made at different times
- may not be meaningful

global state and snapshot - a challenging scanario



System Model

- system consists of a collection of n processes, p_1, p_2,..., p_n connected by channels
- no globally shared memory
- processes communicate solely by passing messages
- no physical global clock in the system
- message send and receive are asynchronous
- message delivery is reliable, occurs within finite time but has arbitrary time delay

System Model contd..

- system can be
  - represented as a directed graph
  - vertices represent processes
  - edges represent unidirectional communication channels
- both processes & channels have states
- process state consists of contents of
  - processor registers
  - stacks
  - local memory

System Model contd..

- process state is highly dependent on the local context of the distributed application
- C_{ij}: channel from process p_i to process p_j
- SC_{ij}:
  - state of channel C_{ij}
  - consists of in-transit messages
- 3 types of events
  - internal events
  - message send events
  - message receive events

System Model contd..

- send(m_{ij}): send event of message m_{ij} from p_i to p_j
- rec(m_{ij}): receive event of message m_{ij} from p_i to p_j
- occurrence of events
  - changes the states of respective processes and channels
  - causes transitions in global system state
- internal event: changes the state of the process at which it occurs
- send event changes:
  - state of the process that sends the message
  - state of the channel on which the message is sent
- events at a process are linearly ordered by their order of occurrence

System Model contd..

- receive event:
  - changes state of the receiving process
  - state of the channel on which the message is received
- LS_i: state of process p_i
- at an instant t, LS_i: state of p_i as a result of the sequence of events executed by p_i up to t
- an event $e \in LS_i$ iff e belongs to the sequence of events that have taken p_i to LS_i
- $e \notin LS_i$ iff e does not belong to the sequence of events that have taken p_i to LS_i

System Model contd..

- for a channel C_{ij} , in-transit messages are:
  - $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$
- if a snapshot recording algorithm records the states of p_i and p_j as LS_i and LS_j , respectively,
  - it must record the state of channel C_{ij} as $transit(LS_i, LS_j)$

System Model contd..

- Several models of communication among processes exist
- FIFO model:
  - each channel acts as a first-in first-out message queue
  - message ordering is preserved by a channel
- non-FIFO model:
  - channel acts like a set
  - sender process adds messages to the channel in a random order
  - receiver process removes messages from the channel in a random order

A Consistent Global State

- global state GS is a ***consistent global state*** iff it satisfies the following two conditions:
  - **C1:** $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$ ($\oplus$: XOR)
  - **C2:** $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$

A Consistent Global State

- **Condition C1 states the law of conservation of messages:**
  - every message m_{ij} that is recorded as sent in the local state of sender p_i must be captured
    - in the state of the channel C_{ij}
    - or in the collected local state of the receiver p_j
- **Condition C2 states that:**
  - in the collected global state, for every effect, its cause must be present
  - if a message m_{ij} is not recorded as sent in the local state of p_i, then it must neither be present in the state of the channel C_{ij} nor in the collected local state of the receiver p_j

Chandy–Lamport Approach

- uses a control message called **marker**
- after a process has recorded its snapshot, it sends a marker along all of its outgoing channels before sending out any more messages
- all messages that follow a marker on a channel have been sent by the sender after it took its snapshot
- channels are FIFO
- marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded
- a process must record its snapshot no later than when it receives a marker on any of its incoming channels

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 p_j

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

Chandy–Lamport Algorithmic approach

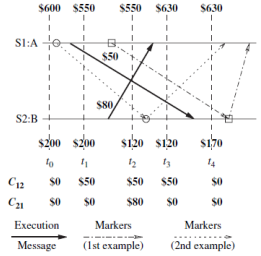
Putting together recorded snapshots

- global snapshot creation at initiator:
  - each process can send its local snapshot to the initiator of the algorithm
- global snapshot creation at all processes:
  - each process sends the information it records along all outgoing channels
  - each process receiving such information for the first time propagates it along its outgoing channels
  - all the processes can determine the global state

Chandy–Lamport Algorithm contd..

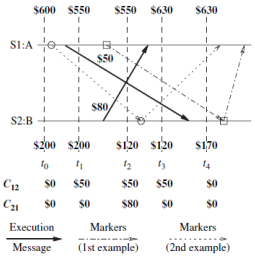
- algorithm can be initiated by any process by executing the marker sending rule
- **termination criterion** - each process has received a marker on all of its incoming channels
- if multiple processes initiate the algorithm concurrently
  - each initiation needs to be distinguished by using unique markers

Chandy-Lamport Algorithm-scenario1



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

Chandy-Lamport Algorithm-scenario2



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

Snapshot Algorithms for Non-FIFO Channels

- FIFO system
  - ensures that all messages sent after a marker on a channel will be delivered after the marker
- in a non-FIFO system
  - a marker cannot be used to differentiate messages into those to be recorded in the global state from those not to be recorded in the global state
- non-FIFO algorithm by Lai and Yang
  - use message piggybacking to distinguish computation messages sent after the marker from those sent before the marker

Lai–Yang Algorithm

- **fulfills the role of marker using a coloring scheme**
- **Coloring Scheme:**
- every process is initially white
- process turns red while taking a snapshot
- equivalent of the "marker sending rule" is executed when a process turns red
- every message sent by a white process is colored white
  - a white message is a message that was sent before the sender of that message recorded its local snapshot

Lai–Yang Algorithm

- **Coloring Scheme:**
- every message sent by a red process is colored red
  - a red message is a message that was sent after the sender of that message recorded its local snapshot
- every white process takes its snapshot no later than the instant it receives a red message

Lai–Yang Algorithm contd..

- **Coloring Scheme:**
- when a white process receives a red message, it records its local snapshot before processing the message
- ensures that
  - no message sent by a process after recording its local snapshot is processed by the destination process before the destination records its local snapshot
- an explicit marker message is not required
- marker is piggybacked on computation messages using a coloring scheme

Lai–Yang Algorithm contd..

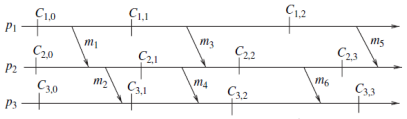
- Lai–Yang algorithm fulfills this role of the marker for channel state computation as follows:
  - every white process records a history of all white messages sent or received by it along each channel
  - when a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot
  - initiator process evaluates transit(LS_i, LS_j) to compute the state of a channel C_{ij} as:
    - SC_{ij} = {white messages sent by p_i on C_{ij}} − {white messages received by p_j on C_{ij}}
    - = $\{m_{ij} \mid send(m_{ij}) \in LS_i\} - \{m_{ij} \mid rec(m_{ij}) \in LS_j\}$

Necessary and sufficient conditions for consistent global snapshots

- local checkpoint – saved intermediate state of a process during its execution
- global snapshot –
  - set of local checkpoints one from each process
- $C_{p,i}$ - i^{th} ($i \geq 0$) checkpoint of process p_p , assigned the sequence number i
- i^{th} checkpoint interval of p_p - all computation performed between $(i-1)^{th}$ and i^{th} checkpoints (and includes $(i-1)^{th}$ checkpoint but not i^{th}).

Necessary and sufficient conditions for consistent global snapshots

- a causal path exists between checkpoints $C_{i,x}$ and $C_{j,y}$ if a sequence of messages exists from $C_{i,x}$ to $C_{j,y}$ such that each message is sent after the previous one in the sequence is received
- a zigzag path between two checkpoints is a causal path, however, allows a message to be sent before the previous one in the path is received



Necessary and sufficient conditions for consistent global snapshots

- **necessary condition for consistent snapshot** - absence of causal path between checkpoints in a snapshot
- **necessary and sufficient conditions for consistent snapshot** - absence of zigzag path between checkpoints in a snapshot

Recap Quiz

1. Which of the following is not a type of event in distributed computing environments?
(a) Internal (b) external (c) message send (d) message receive

2. If the message ordering is preserved in the distributed computing environment, then this system model is called
(a)non-FIFO (b) LIFO (c) queue (d) FIFO

3. The control message used in Chandy-Lamport algorithm is called
(a) Master (b) snapshot (c) marker (d) checkpoint

4. Message piggybacking is used in snapshot algorithms for ____ channels
(a)non-FIFO (b) LIFO (c) queue (d) FIFO

5. The Lai-Yang algorithm for non-FIFO channels uses _____ instead of a marker
(a) Checkpointing (b) colouring (c) creating (d) initiating

Recap Quiz - key

Q1	Q2	Q3	Q4	Q5
b	d	c	a	c

Reference

Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 4, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008.

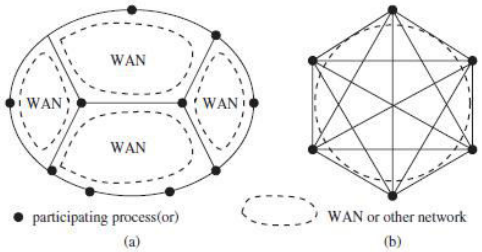
Distributed Computing (CS 3 contd..)

Distributed Computing Terminology and Basic Algorithms

Prof. Geetha
Associate Professor, Dept. of Computer Sc. & Information Systems
BITS Pilani Hyderabad Campus
geetha@hyderabad.bits-pilani.ac.in

BITS Pilani
Hyderabad Campus

Topological abstraction in distributed computing



Centralized and distributed algorithms

- **centralized algorithm –**
  - predominant amount of work is performed by one (or possibly a few) processors
  - other processors play a relatively smaller role in accomplishing the joint task
  - other processors usually request or supply information
- **distributed algorithm -** each processor plays an equal role in sharing the message overhead, time overhead, and space overhead

Symmetric and asymmetric algorithms

- **symmetric algorithm -** algorithm in which all the processors execute the same logical functions
- **asymmetric algorithm** - algorithm in which different processors execute logically different functions

Synchronous and asynchronous systems

- **synchronous system -** satisfies the following properties:
  - a known upper bound on the message communication delay
  - a known bounded drift rate for the local clock of each processor with respect to real-time
  - a known upper bound on the time taken by a process to execute a logical step in the execution
- **asynchronous system** - a system in which none of the above 3 properties of synchronous systems are satisfied

Failure models

- **Failure model -** specifies the manner in which the component(s) of the system may fail
- **Process failure models:**
  - **Fail-stop** - a properly functioning process may fail by stopping execution from some instant, other processes can learn that the process has failed
  - **Crash** - a properly functioning process may fail by stopping to function from any instance, other processes do not learn of this crash
  - **Byzantine or malicious failure, with authentication** - a process may exhibit any arbitrary behavior, authentication can be used to detect forgery
  - **Byzantine or malicious failure** - a process may exhibit any arbitrary behavior, no authentication techniques are applicable

Notation & Definitions

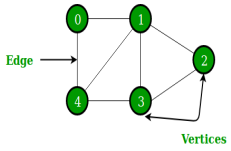
- undirected unweighted graph G = (*N, L*) represents topology
- *n* = |*N*|
- *l* = |*L*|
- diameter of a graph –
  - minimum number of edges that need to be traversed to go from any node to any other node
  - diameter = $max_{i,j \in N}$ {length of the shortest path between *i* and *j*}
  - For a tree embedded in the graph, its depth is denoted as h

Model

- Node Information:
  - Adjacent links
  - Neighboring nodes.
  - Unique identity.
- Graph model: undirected graph.
  - Nodes V
  - Edges E
  - Diameter D

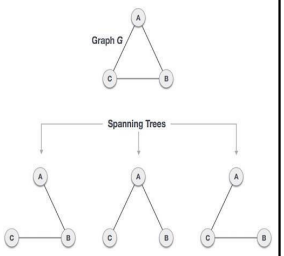
Graph data structure

- A Graph is a non-linear data structure consisting of nodes and edges
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph
- The formal definition is : *A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes*
- In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.
- Graphs are used to solve many real-life problems. Graphs are used to represent networks
- In distributed computing, the network of processes/processors can be modeled as a graph



Constructing Minimal Spanning Trees (MST) from Graphs

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..
- Thus every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices
- The distributed minimum spanning tree (MST) problem involves the construction of a minimum spanning tree by a distributed algorithm, in a network where nodes communicate by message passing



Model Definition

- Synchronous Model:
  - There is a global clock.
  - Packet can be sent only at clock ticks.
  - Packet sent at time *t* is received by *t+1*.
- Asynchronous model:
  - Packet sent is eventually received.

Complexity Measures

- Message complexity:
  - Number of messages sent (worst case).
  - Usually assume "small messages".
- Time complexity
  - Synchronous Model: number of clock ticks
  - Asynchronous Model:
    - Assign delay to each message in (0,1].
    - Worse case over possible delays

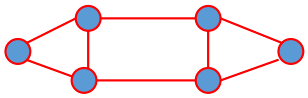
Problems in Distributed Computing

- Broadcast - Single initiator:
  - The initiator has an information.
  - At the end of the protocol all nodes receive the information
  - Example : Topology update.
- Spanning Tree:
  - Single/Many initiators.
  - On termination there is a spanning tree.
- BFS tree:
  - A minimum distance tree with respect to the originator.

Basic Flooding: Algorithm

- Initiator:
  - Initially send a packet p(info) to all neighbors.
- Every node:
  - When receiving a packet p(info):
    - Sends a packet p(info) to all neighbors.

Basic Flooding: Example



Basic Flooding - complexity

- Time Complexity
  - Synchronous (and asynchronous):
    - Node at distance *d* receives the info by time *d*.
- Message complexity:
  - There is no termination!
  - Message complexity is unbounded!
  - How can we correct this?

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

- ❑ algorithm proceeds in rounds (hence, synchronous)
- ❑ assumes a designated root node
- ❑ root initiates
  - ❑ the algorithm
  - ❑ a flooding of QUERY messages to identify tree edges
- ❑ processes must produce a spanning tree rooted at the root node
- ❑ each process P_i ($P_i \neq$ root) should output its own parent for the spanning tree
- ❑ spanning tree is a BFS spanning tree

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

Local Variables maintained at each P_i

- **int** *visited*
- **int** *depth*
- **int** *parent*
- **set of int** *Neighbors*

The root initiates a flooding of QUERY messages in the graph to identify tree edges. The parent of a node is that node from which a QUERY is first received; if multiple QUERYs are received in the same round, one of the senders is randomly chosen as the parent

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding – contd..

Initially at each P_i

- *visited* = 0
- *depth* = 0
- *parent* = NULL
- *Neighbors* = set of neighbors

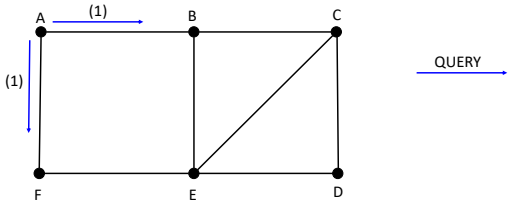
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..

Algorithm for P_i
Round r = 1
if P_i = root **then**
  visited = 1
  depth = 0
  send QUERY to *Neighbors*
if P_i receives a QUERY message **then**
  visited = 1
  depth = r
  parent = root
  plan to send QUERYs to *Neighbors* at next round

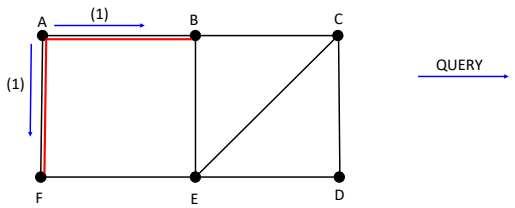
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

Algorithm for P_i
Round r > 1 and r <= *diameter*
if P_i planned to send in previous round **then**
  P_i sends QUERY to *Neighbors*
if *visited* = 0 **then**
  if P_i receives QUERY messages **then**
    visited = 1
    depth = r
    parent = any randomly selected node from which QUERY was received
    plan to send QUERY to *Neighbors* \ {senders of QUERYs received in r}

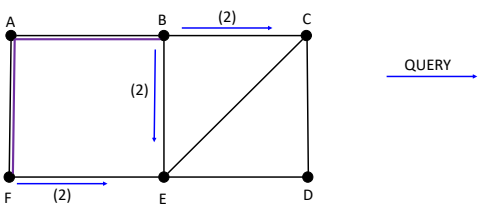
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding



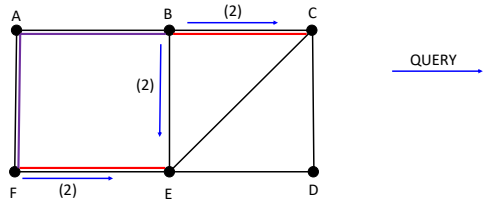
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..



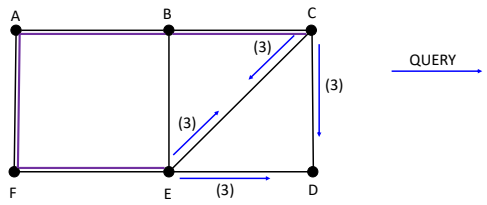
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..



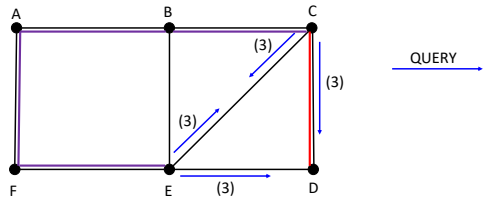
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..



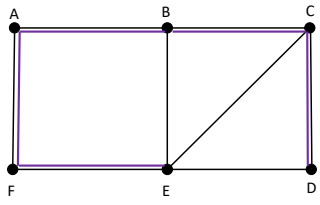
Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..



Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..

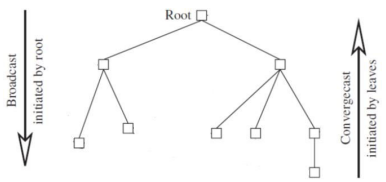


Synchronous Single-Initiator Spanning Tree Algorithm using Flooding contd..



Broadcast and Convergecast on a Tree

A spanning tree is useful for distributing (via a broadcast) and collecting (via a convergecast) information to/from all the nodes



Broadcast and Convergecast on a Tree

A broadcast algorithm on a spanning tree can be specified by 2 rules:

- ***BC1:***
  - The root sends the information to be broadcast to all its children.
  - Terminate.
- ***BC2:***
  - When a (non-root) node receives information from its parent, it copies it and forwards it to its children.
  - Terminate.

Broadcast and Convergecast on a Tree

- A convergecast algorithm collects information from all the nodes at the root node in order to compute some global function
- It is initiated by the leaf nodes of the tree, usually in response to receiving a request sent by the root using a broadcast

Broadcast and Convergecast on a Tree

The convergecast algorithm is specified as follows:

- ***CVC1:***
  - Leaf node sends its report to its parent.
  - Terminate.
- ***CVC2:***
  - At a non-leaf node that is not the root: When a report is received from all the child nodes, the collective report is sent to the parent.
  - Terminate.
- ***CVC3:***
  - At the root: When a report is received from all the child nodes, the global function is evaluated using the reports.
  - Terminate.

Broadcast and Convergecast on a Tree

Complexity

- each broadcast and each convergecast requires $n - 1$ messages
- each broadcast and each convergecast requires time equal to the maximum height h of the tree, which is $O(n)$

Applications:

- computation of minimum of integer variables associated with the nodes of an application
- leader election

Synchronizers

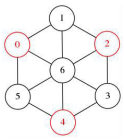
- It is difficult to design algorithms for asynchronous systems
- **solution** – simulate synchronous behavior on an asynchronous system
- **synchronizers** - transformation algorithms to run synchronous algorithms on asynchronous systems
- synchronizer signals to each process when it is sure that all messages to be received in the current round have arrived and it is safe to proceed to the next round

Synchronizers

- **Simple synchronizer:**
  - requires each process to send every neighbor only one message in each round
  - if no message is to be sent in the synchronous algorithm, an empty dummy message is sent in the asynchronous algorithm
  - if more than one messages are sent in the synchronous algorithm, they are combined into one message in the asynchronous algorithm
  - in any round, when a process receives a message from each neighbor, it moves to the next round
- **α, β, and γ synchronizers:** (rounds, rooted spanning tree; n/w of clusters)
  - use the notion of process safety
  - a process i is safe in round r if all messages sent by i in round r have been received

Maximal Independent Set

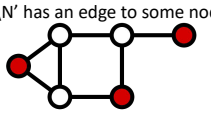
- For a graph (N, L), an independent set of nodes N', where N'⊂ N, is such that for each i and j in N', (i, j)∉ L
- An independent set N' is a maximal independent set if no strict superset of N' is an independent set
- A graph may have multiple maximal independent sets
- Adjacent nodes must not be chosen



source - http://www.martinbroadhurst.com/greedy-max-independent-set-in-c.html

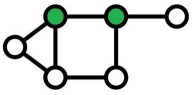
Connected dominating set

- A dominating set of graph (N, L) is a set N' ⊆ N such that each node in N \N' has an edge to some node in N'



source - https://en.wikipedia.org/wiki/Dominating_set

- A connected dominating set (CDS) of (N, L) is a dominating set N' such that the subgraph induced by the nodes in N' is connected



source - https://www.google.com/search?q=connected+dominating+set&source=lnms&tbm=isch&sa=X&ved=0ahUKEwi48raDoPPcAhVENY8KHd_1AkMQ_AUICigB&biw=1920&bih=966#imgrc=B3d-ZBKurdlQLM:

Reference

❑Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 5, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

Recap Quiz

1. In a graph, the minimum number of edges that need to be traversed to go from any node to any other node is called
(a) Diameter (b) radius (c) perimeter (d) circumference

2. The Single-Initiator Spanning Tree Algorithm using Flooding is synchronous because
(a) There is a global clock (b) algo works in rounds (c) there are local clocks (d) spanning tree is used

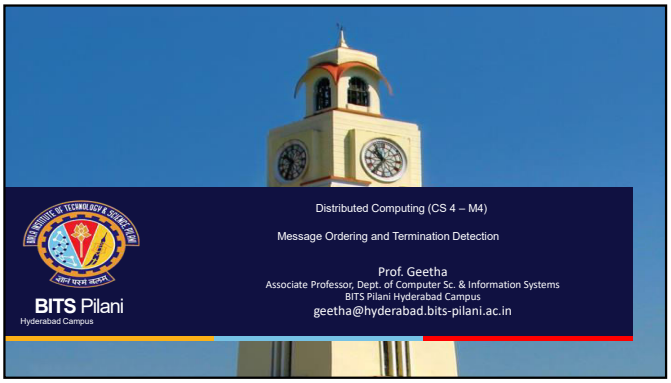
3. A Convergecast algorithm is initiated by ___ node of the tree
(a) Root (b) intermediate (c) leaf (d) none

4. Let h be the maximum height of the tree. Then the time required for broadcast and convergecast algorithms will be
(a) h (b) logh (c) 2h (d) h^2

5. The number of maximal independent sets a graph can have in graph algorithms is
(a) 0 (b) 1 (c) any number (d) nil

Recap Quiz - key

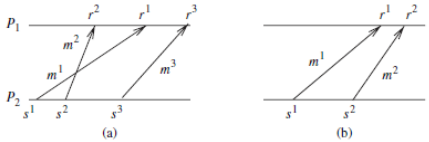
Q1	Q2	Q3	Q4	Q5
a	b	c	a	c



Message ordering paradigms

- ❑ **FIFO**
  - ❑ each channel acts as a first-in first-out message queue
  - ❑ message ordering is preserved by channel
- ❑ **non-FIFO**
  - ❑ channel acts like a set
  - ❑ sender process adds messages to channel
  - ❑ receiver process removes messages from it
- ❑ **Causal Order** - for m_{ij} and m_{kj}, if send(m_{ij}) → send(m_{kj}),
- ❑ then rec(m_{ij}) → rec(m_{kj})
- ❑ **Synchronous Order** - all the communication between pairs of processes uses synchronous send and receive

Asynchronous executions (A-executions)



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

FIFO execution

$$\text{for all } (s, r) \text{ and } (s', r') \in \mathcal{T}, (s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'.$$

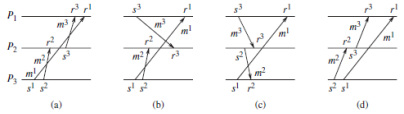
- FIFO logical channel over a non-FIFO channel would use a separate numbering scheme to sequence the messages on each logical channel
- The sender assigns and appends a *sequence_num, connection_id* tuple to each message
- The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence

Causally Ordered (CO) executions

$$\text{for all } (s, r) \text{ and } (s', r') \in \mathcal{T}, (r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'.$$

- If two send events s and s' are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events r and r' occur in the same order at all common destinations.
- If s and s' are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false

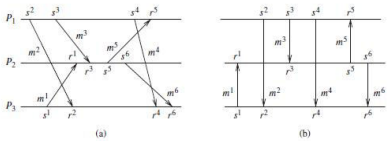
Illustration of causally ordered executions



(a) Not a CO execution. (b),(c), and (d) CO executions.

Synchronous (SYNC) execution – S execution

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



(a) Execution in an asynchronous system (b) Equivalent instantaneous communication

- 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 occurring instantaneously and atomically
- In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines

Group Communication

- Group communication needs to be supported
- message broadcast - sending a message to all members in the distributed system
- multicasting - a message is sent to a certain subset, identified as a group, of the processes in the system
- unicasting - point-to-point message communication

Group Communication

- **closed group algorithm** –
  - when a multicast algorithm requires the sender to be a part of the destination group
- **open group algorithm** –
  - when the sender of the multicast can be outside the destination group

- multicast algorithms
  - number of groups may be potentially exponential, i.e., $O(2^n)$

Causal Order

•Two criteria must be satisfied by a causal ordering protocol:

- Safety
- Liveness

Causal Order

- **Safety**
- a message M arriving at a process may need to be buffered until all system-wide messages sent in the causal past of the send(M) event to the same destination have already arrived
- distinction is made between
  - arrival of a message at a process
  - event at which the message is given to the application process
- **Liveness**
- A message that arrives at a process must eventually be delivered to the process

Raynal–Schiper–Toueg Algorithm

- each message M should carry a log of
  - all other messages
  - or their identifiers, sent causally before M's send event, and sent to the same destination dest(M)
- log can be examined to ensure whether it is safe to deliver a message
- channels are FIFO

Allows each send event to unicast, multicast, or broadcast a message in the system.

Raynal–Schiper–Toueg Algorithm contd..

local variables:

- **array of int** SENT[1 …… n, 1 …… n] (n x n array)
  - $SENT_i[j, k]$ = no. of messages sent by P_j to P_k as known to P_i
- **array of int** DELIV [1 …… n]
  - $DELIV_i[j]$ = no. of messages from P_j that have been delivered to P_i
  - DELIV[j] = no. of messages sent by P_j that are delivered locally

(1) **send event**, where P_i wants to send message M to P_j :
(1a) **send** (M, SENT) to P_j
(1b) SENT[i, j] = SENT[i, j] + 1

Raynal–Schiper–Toueg Algorithm

(2) **message arrival**, when (M, ST) arrives at P_i from P_j :

(2a) **deliver** M to P_i when for each process x,

(2b) $DELIV[x] \geq ST[x, i]$

(2c) $\forall$ x, y, $SENT[x, y] = \max(SENT[x, y], ST[x, y])$

(2d) DELIV[j] = DELIV[j] + 1

Raynal–Schiper–Toueg Algorithm

Complexity:

- space requirement at each process: $O(n^2)$ integers
- space overhead per message: n^2 integers
- time complexity at each process for each send and deliver event: $O(n^2)$

Raynal–Schiper–Toueg Algorithm

- P_1 sent 4 messages to P_2
- P_2 sent 3 messages to P_1
- P_1 → $DELIV_1[0\ 2]$ // 2 messages from P_2 have been delivered to P_1
- P_2→ $DELIV_2[4\ 0]$ // all 4 messages from P_1 have been delivered to P_2
- Now if,
  - P_1 sends a message to P_2 → no buffering required
  - P_2 sends a message to P_1→ buffering required

Birman-Schiper-Stephenson Protocol

- A message is delivered to a process only if the message preceding it has been delivered
- buffer is needed for pending deliveries
- each message has a vector that contains information for the recipient to determine if another message preceded it
- all messages are broadcast

Birman-Schiper-Stephenson Protocol

- C_i = vector clock of P_i
- $C_i[j] = j^{th}$ element of C_i
  - contains P_i's latest value for the current time in P_j
- tm = vector timestamp for message m
- stamped after local clock is incremented

Birman-Schiper-Stephenson Protocol

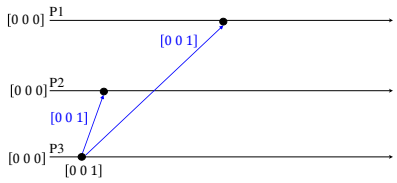
- P_i sends a message m to P_j
- P_i increments $C_i[i]$
- P_i sets the timestamp tm = C_i for message m

Birman-Schiper-Stephenson Protocol

P_j receives a message m from P_i

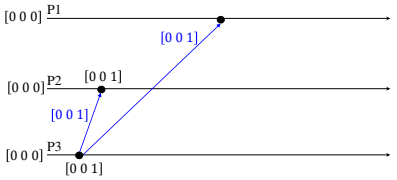
- when P_j ($j \neq i$) receives m with timestamp tm, it delays message delivery until:
  - $C_j[i] = tm[i] - 1$ //P_j has received all preceeding messages sent by P_i
  - $\forall\ k <= n$ and $k \neq i$, $C_j[k] \geq tm[k]$ //P_j has received all the messages that were received at P_i from other processes before P_i sent m
- when m is delivered to P_j, update P_j's vector clock
  $\forall i$, $C_j[i] = \max(C_j[i], tm[i])$
- check buffered messages to see if any can be delivered

Birman-Schiper-Stephenson Protocol

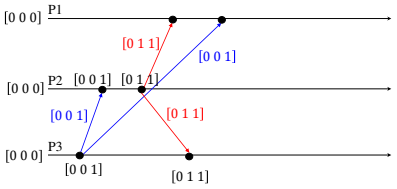


vectors in blue indicate timestamps of messages

Birman-Schiper-Stephenson Protocol

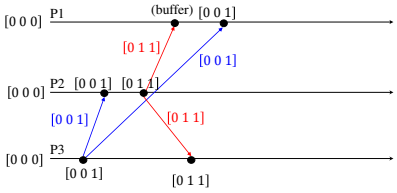


Birman-Schiper-Stephenson Protocol

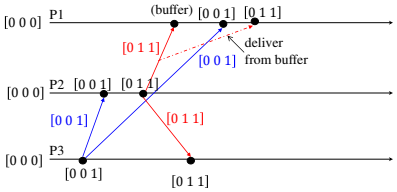


vectors in red indicate timestamps of messages

Birman-Schiper-Stephenson Protocol



Birman-Schiper-Stephenson Protocol



Total Order

- requires that all messages be received in the same order by the recipients of the messages
- 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

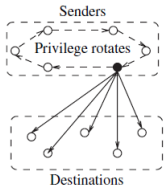
Classification of Application-Level Multicast Algorithms

- **Communication history-based algorithms**
- use a part of the communication history to guarantee ordering requirements
- eg.,
  - RST (Raynal, Schiper. and Toueg), KS (Kshemkalyani and Singhal) algorithms -> causal ordering
  - Lamport's algorithm -> scalar timestamps
  - NewTop protocol – extension of Lamport's algorithm

Classification of Application-Level Multicast Algorithms

- **Privilege-based algorithms**
- A token circulates among sender processes
- The token carries the sequence number for the next message to be multicast
- only the token-holder can multicast
- after a multicast send event, the sequence number is updated
- destination processes deliver messages in the order of increasing sequence numbers
- senders need to know the other senders, hence closed groups are assumed
- can provide total ordering and causal ordering using closed group configuration
- not scalable as do not permit concurrent send events

Classification of Application-Level Multicast Algorithms



Privilege-based algorithm

Classification of Application-Level Multicast Algorithms

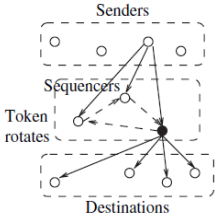
Moving sequencer algorithms

- sender sends the message to all the sequencers to multicast a message
- sequencers circulate a token among themselves
- token carries
  - a sequence number
  - a list of all the messages for which a sequence number has already been assigned – such messages have been sent already
- when a sequencer receives the token
  - it assigns a sequence number to all received but un-sequenced messages
  - it sends the newly sequenced messages to the destinations
  - inserts these messages into the token list
  - passes the token to the next sequencer

Classification of Application-Level Multicast Algorithms

Moving sequencer algorithms contd..

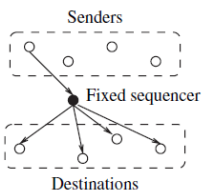
- destination processes deliver messages received in the order of increasing sequence number
- guarantee total ordering



Classification of Application-Level Multicast Algorithms

Fixed sequencer algorithms

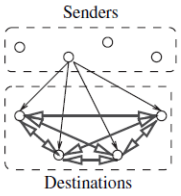
- simplified version of the previous class of algorithms
- a single sequencer is present
- centralized algorithms
- eg., propagation tree approach, ISIS sequencer, Amoeba, Phoenix, Newtop's asymmetric algorithm



Classification of Application-Level Multicast Algorithms

Destination agreement algorithms

- destinations receive the messages with some limited ordering information
- destination processes then exchange information among themselves to define an order
- Two sub-classes
  - uses timestamps
  - uses an agreement or consensus protocol among the processes



Termination detection using distributed snapshots

- consistent snapshot of a distributed system captures stable properties
- termination of a distributed computation is a stable property
- if a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation
- all processes will have become idle
- there will be no in-transit messages

Termination detection by weight throwing

- a process called controlling agent monitors the computation
- communication channel exists between each of the processes and the controlling agent and also between every pair of processes
- initially, all processes are in the idle state
- weight at each process is zero and the weight at the controlling agent is 1
- computation starts when the controlling agent sends a basic message to one of the processes
- process becomes active and the computation starts
- non-zero weight W (0 <W ≤ 1) is assigned to each process in the active state

Termination detection by weight throwing contd..

- when a process sends a message, it sends a part of its weight in the message
- when a process receives a message, it add the weight received in the message to its weight
- sum of weights on all the processes and on all the messages in transit is always 1
- when a process becomes passive, it sends its weight to the controlling agent in a control message, which the controlling agent adds to its weight
- controlling agent concludes termination if its weight becomes 1

Spanning-tree-based termination detection

- Assumes N processes P_i, $0 \le i \le N$
- processes → nodes, channels → edges
- uses a fixed spanning tree of the graph with process P_0 at its root which is responsible for termination detection
- P_0 communicates with other processes to determine their states
- messages used for this purpose are called signals
- all leaf nodes report to their parents, if they have terminated
- an intermediate node will report to its parent when it has completed processing and all of its immediate children have terminated, and so on……
- root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated

Recap Quiz

1. In the multicast algorithms, the number of groups may grow to the order of
(a) logn (b) $\log^2 n$ (c) n^2 (d) 2^n

2. In which of the following message ordering paradigms, the channel acts as a set?
(a) Synchronous (b) causal order (c) FIFO (d) non-FIFO

3. In the Raynal–Schiper–Toueg Algorithm, the channel is organized as
(a) Synchronous (b) causal order (c) FIFO (d) non-FIFO

4. Which of the following is not a classification of application-level multicast algorithms?
(a) Fixed sequencer (b) moving sequencer (c) source agreement (d) destination agreement

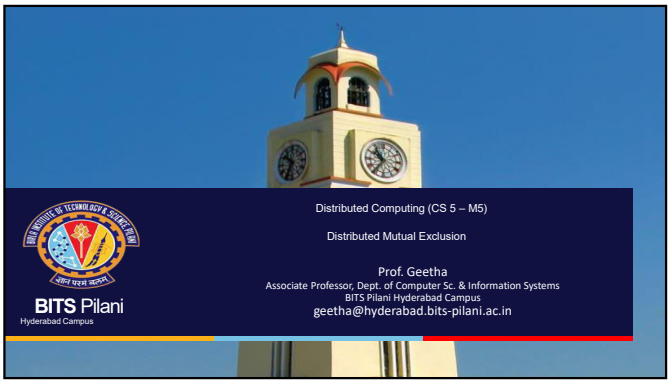
5. The messages used in the spanning tree based termination detection algorithms are called
(a) Markers (b) signals (c) triggers (d) events

Recap Quiz - key

Q1	Q2	Q3	Q4	Q5
d	d	c	c	b

References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 6, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).
- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 7, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008(Reprint 2013).
- https://courses.csail.mit.edu/6.006/fall11/rec/rec14.pdf



What is mutual exclusion?

- Mutual Exclusion also known as Mutex was first identified by Dijkstra
- When a process is accessing a shared variable, it is said to be in a critical section (code segment)
- When no two processes can be in Critical Section at the same time, this state is known as Mutual Exclusion which is a property of concurrency control and is used to prevent race condition (happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute)

Mutual Exclusion Devices:-

Locks, recursive locks, semaphores, monitor, message passing etc.,

Mutual exclusion in synchronization

- **Mutual exclusion** is a property of process synchronization which states that "no two processes can exist in the critical section at any given point of time"
- Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition
- During concurrent execution of processes, processes need to enter the critical section (or the section of the program shared across processes) at times for execution
- It might so happen that because of the execution of multiple processes at once, the values stored in the critical section become inconsistent; in other words, the values depend on the sequence of execution of instructions – also known as a race condition
- The primary task of process synchronization is to get rid of race conditions while executing the critical section

Mutual Exclusion in Distributed Computing

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions; the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e., only one process is allowed to execute the critical section at any given instance of time

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved

In Distributed systems, we neither have shared memory nor a common physical clock and therefore we can not solve mutual exclusion problem using shared variables. To solve the mutual exclusion problem in distributed systems, message passing is used

A site in distributed system does not have the complete information of the state of the system due to lack of shared memory and a common physical clock

Mutual Exclusion

- for distributed systems –
  - decision as to which process is allowed access to the CS next is made by message passing
  - must deal with unpredictable message delays and incomplete knowledge of the system state

Mutual Exclusion

- ensures that concurrent access of processes to a shared resource or data is serialized
- executed in a mutually exclusive manner
- for distributed system –
  - only one process is allowed to execute the critical section (CS) at any given time
  - semaphores or a local kernel cannot be used to implement mutual exclusion

Three approaches for mutual exclusion in distributed computing systems

- **Token based**
- **Non-token based**
- **Quorum based**

Token-Based Approach

- a unique token is shared among the sites
- token is also called PRIVILEGE message
- a site is allowed to enter its CS if
  - it possesses the token
  - it continues to hold the token until the execution of the CS is over
- mutual exclusion is ensured because the token is unique

Non-Token-Based Approach

- two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next
- a site enters the CS when an assertion becomes true
- mutual exclusion is enforced because the assertion becomes true only at one site at any given time

Quorum-Based Approach

- each site requests permission to execute the CS from a subset of sites
- subset of sites is called quorum
- quorums are formed in such a way that when two sites concurrently request access to the CS
  - at least one site receives both the requests
  - this site is responsible to make sure that only one request executes the CS at any time

System Model for Mutual Exclusion

- ✓ system consists of N sites, S_1, S_2,…., S_N
- ✓ without loss of generality, assume that a single process is running on each site
- ✓ process at site S_i is p_i
- ✓ processes communicate asynchronously over an underlying communication network
- ✓ any process wishing to enter the CS
  - ✓ requests all other or a subset of processes by sending REQUEST messages
  - ✓ waits for appropriate replies before entering the CS
- ✓ while waiting the process is not allowed to make further requests to enter the CS

System Model for Mutual Exclusion contd..

- ❑ site can be in one of the following 3 states:
  - ❑ requesting the CS
  - ❑ executing the CS
  - ❑ neither of the 2
- ❑ "requesting the CS" state - site is blocked and cannot make further requests for the CS
- ❑ "idle" state - site is executing outside the CS
- ❑ for 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 called *idle token state*

System Model for mutual exclusion contd..

- ❖ 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
- ❖ nature of channels (FIFO or not) is algorithm specific
- ❖ assume that:
  - ❖ channels reliably deliver all messages
  - ❖ sites do not crash
  - ❖ network does not get partitioned
- ❖ **timestamps are used to decide the priority of requests in case of a conflict**
- ❖ ***general rule - smaller the timestamp of a request, the higher its priority to execute the CS***

System Model contd..

Notations:

- ❑ N - number of processes or sites involved in invoking the critical section
- ❑ T - average message delay
- ❑ E - average critical section execution time

Requirements of mutual exclusion algorithms

❑ **No Deadlock:**
Two or more sites should not endlessly wait for any message that will never arrive

❑ **No Starvation:**
Every site which wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other sites are repeatedly executing the critical section

❑ **Fairness:**
Each site should get a fair chance to execute the critical section. Any request to execute the critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system

❑ **Fault Tolerance:**
In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption

Properties of Mutual Exclusion Algorithms

A mutual exclusion algorithm should satisfy:

- ❖ Safety Property – absolutely necessary
- ❖ Liveness Property – important
- ❖ Fairness – important

Requirements of Mutual Exclusion Algorithms

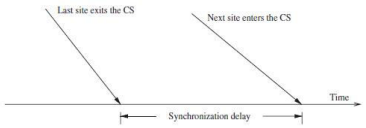
- **Safety property**
  - at any instant, only one process can execute the critical section
  - absolutely necessary property
- **Liveness property**
  - absence of deadlock and starvation
  - a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS
  - every requesting site should get an opportunity to execute the CS in finite time

Mutual Exclusion Algorithms

- **Fairness**
  - each process gets a fair chance to execute the CS
  - CS execution requests are executed in order of their arrival in the system
  - time is determined by a logical clock

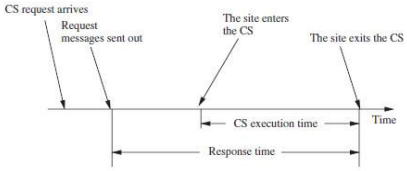
Performance Metrics

- **Message complexity** - number of messages that are required per CS execution by a site
- **Synchronization delay** –
  - after a site leaves the CS, the time required before the next site enters the CS
  - one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS



Performance Metrics

- **Response time** – time interval a request waits for its CS execution to be over after its request messages have been sent out
- **System throughput** -
  - rate at which the system executes requests for the CS
  - SD is Synchronization Delay
  - E is average critical section execution time
  - **System throughput = 1/(SD+E)**



Performance Metrics

Low and high load performance –

- "low load" – not more than one request for the critical section present in the system simultaneously
- "high load" -
  - there is always a pending request for critical section at a site
  - after having executed a request, a site immediately initiates activities to execute its next CS request
  - a site is seldom in the idle state

Lamport's Algorithm

- time is determined by logical clocks
- when a site processes a request for the CS, it
  - updates its local clock
  - assigns the request a timestamp
- algorithm is fair - executes CS requests in the increasing order of timestamps
- every site S_i keeps a queue, $request\_queue_i$
- $request\_queue_i$ contains mutual exclusion requests ordered by their timestamps

Lamport's Algorithm

- algorithm requires communication channels to deliver messages in FIFO order
- when a site removes a request from its request queue
  - its own request may come at the top of the queue
  - enables it to enter the CS
- when a site receives a REQUEST, REPLY, or RELEASE message
  - it updates its clock using the timestamp in the message

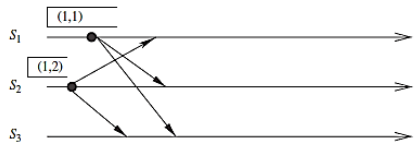
Lamport's Algorithm

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

Lamport's Algorithm

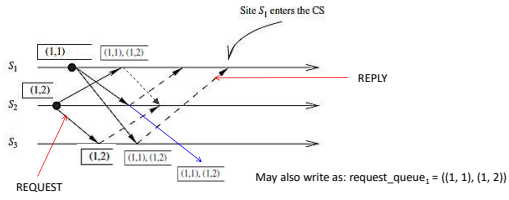
- **Executing the critical section:**
- Site S_i enters the CS when the following two conditions hold:
  - **L1**: S_i has received a message with timestamp larger than (ts_i, i) from all other sites
  - **L2**: S_i's request is at the top of the $request\_queue_i$
- **Releasing the critical section:**
- • Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites
- • When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue

Lamport's Algorithm



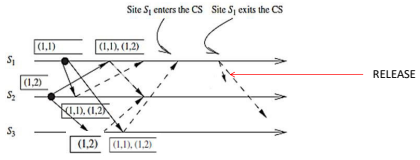
Sites S_1 and S_2 make requests for the CS

Lamport's Algorithm



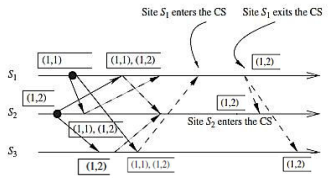
Site S_1 enters the CS

Lamport's Algorithm



Site S_1 exits the CS and sends RELEASE messages

Lamport's Algorithm



Site S_2 enters the CS

Lamport's Algorithm

- **Correctness**
- *Lamport's algorithm achieves mutual exclusion*
- *Lamport's algorithm is fair - requests for CS are executed in the order of their timestamps*
- **Performance**
- For each CS execution, requires (N − 1) REQUEST messages, (N − 1) REPLY messages, and (N − 1) RELEASE messages
- Requires 3(N − 1) messages per CS invocation
- Synchronization delay in the algorithm is T (T = avg. message delay)

Ricart–Agrawala Algorithm

- communication channels are not required to be FIFO
- uses two types of messages:
  - REQUEST
  - 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

Ricart–Agrawala Algorithm

•timestamps are used to decide the priority of requests in case of conflict –

- if a process p_i that is waiting to execute the CS receives a REQUEST message from process p_j , then if the priority of p_j's request is lower, p_i defers the REPLY to p_j and sends a REPLY message to p_j only after executing the CS for its pending request
- otherwise, p_i sends a REPLY message to p_j immediately, provided it is currently not executing the CS

•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

Ricart–Agrawala Algorithm

- each process p_i maintains the request-deferred array, RD_i
- size of RD_i = no. of processes in the system
- initially, ∀i ∀j: $RD_i[j] = 0$
- whenever p_i defers the request sent by p_j, it sets $RD_i[j] = 1$,
- after it has sent a REPLY message to p_j , it sets $RD_i[j] = 0$
- when a site receives a message, it updates its clock using the timestamp in the message
- when a site takes up a request for the CS for processing, it
  - updates its local clock
  - assigns a timestamp to the request
- execution of the CS requests is always in the order of their timestamps

Ricart–Agrawala Algorithm

Requesting the critical section:

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

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

Ricart–Agrawala Algorithm

Executing the critical section:

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

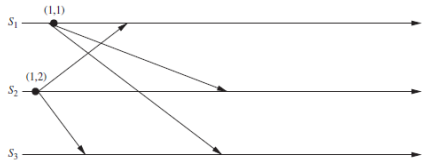
Releasing the critical section:

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

Correctness

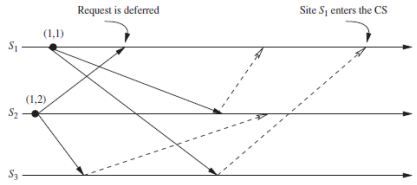
•*Ricart–Agrawala algorithm achieves mutual exclusion*

Ricart–Agrawala Algorithm



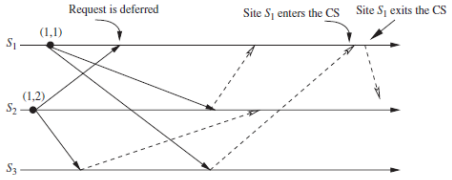
Sites S_1 and S_2 each makes a request for the CS

Ricart–Agrawala Algorithm



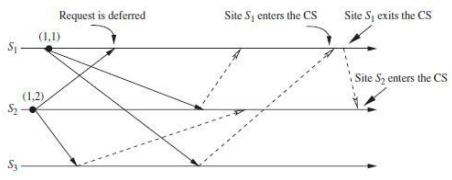
Site S_1 enters the CS

Ricart–Agrawala Algorithm



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

Ricart–Agrawala Algorithm



Site S_2 enters the CS

Ricart–Agrawala Algorithm

Performance

- For each CS execution, requires (N − 1) REQUEST messages and (N − 1) REPLY messages
- requires 2(N − 1) messages per CS execution
- synchronization delay in the algorithm is T

Maekawa's Algorithm

- quorum-based mutual exclusion algorithm
- request sets for sites (i.e., quorums) 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: $(\forall i : 1 \leq i \leq N :: |R_i| = K$ for some $K)$
  - M4: Any site S_j is contained in K number of R_i's, $1 \leq i, j \leq N$
- Maekawa showed that $N = K(K - 1) + 1$
- This relation gives $|R_i| = K = \sqrt{N}$ (square root of N)

Maekawa's Algorithm

- ❑ 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
  - ❑ mutual exclusion is guaranteed
- ❑ requires delivery of messages to be in the order they are sent between every pair of sites

Maekawa's Algorithm

- ❑ conditions M1 and M2 are necessary for correctness
- ❑ M3 ($\forall i : 1 \leq i \leq N :: |R_i| = K$ for some K) states that the size of the requests sets of all sites must be equal
  - ❑ equal amount of work to invoke mutual exclusion
- ❑ M4 (Any site S_j is contained in K number of R_i's, $1 \leq i, j \leq N$) enforces that exactly the same number of sites should request permission from any site
  - ❑ equal responsibility

Maekawa's Algorithm

Requesting the critical section:
(a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i.
(b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.

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

Maekawa's Algorithm

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

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

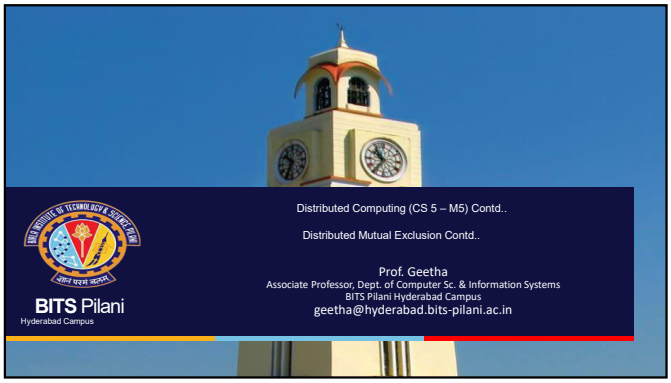
Maekawa's Algorithm

Correctness: *Maekawa's algorithm achieves mutual exclusion*

Performance:
- size of a request set is $\sqrt{N}$
- an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution
- synchronization delay 2T

Reference

❖Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 9, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (reprint: 2013).



Distributed Computing (CS 5 – M5) Contd..

Distributed Mutual Exclusion Contd..

Prof. Geetha
Associate Professor, Dept. of Computer Sc. & Information Systems
BITS Pilani Hyderabad Campus
geetha@hyderabad.bits-pilani.ac.in

BITS Pilani
Hyderabad Campus

Suzuki–Kasami's Broadcast 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
- ❖ The site that 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 CS execution
- ❖ Two design issues exist:
  - ❖ how to distinguish an outdated REQUEST message from a current REQUEST message ?
  - ❖ how to determine which site has an outstanding request for the CS?

Suzuki–Kasami's Broadcast Algorithm

How to distinguish an outdated REQUEST message from a current REQUEST message ?

- ❑ a site may receive a token request message after the corresponding request has been satisfied
- ❑ if a site cannot determine if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it

Suzuki–Kasami's Broadcast Algorithm

- ➢ **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
- ➢ after the corresponding request for the CS has been satisfied at S_j, an issue is how to inform site S_i and all other sites efficiently

Suzuki–Kasami's Broadcast Algorithm

- ❖ outdated REQUEST messages are distinguished from current REQUEST messages in the following manner:
  - ❖ a REQUEST message of site S_j has the form REQUEST(j, sn) where sn (sn = 1, 2.....) is a sequence number that indicates that site S_j is requesting its sn^{th} CS execution
  - ❖ a site S_i keeps an array of integers $RN_i[1, \ldots, n]$ where $RN_i[j]$ is the largest sequence number received in a REQUEST message so far from site S_j
  - ❖ when site S_i receives a REQUEST(j, sn) message, it sets $RN_i[j] = \max(RN_i[j], sn)$
  - ❖ when a site S_i receives a REQUEST(j,sn) message, the request is outdated if $RN_i[j] > sn$

Suzuki–Kasami's Broadcast Algorithm

- 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 S_j executed most recently
  - after executing its CS, a site S_i updates LN[i] = RN_i[i] to indicate that its request corresponding to sequence number RN_i[i] has been executed
  - token array LN[1, … ,n] permits a site to determine if a site has an outstanding request for the CS
  - **at site S_i, if RN_i[j] = LN[j] + 1, then site S_j is currently requesting a token**
  - after executing the CS, a site checks this condition for all the j's to determine all the sites that are requesting the token and places their IDs in queue Q if these IDs are not already present in Q
  - finally, the site sends the token to the site whose ID is at the head of Q

Suzuki–Kasami's Broadcast Algorithm

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

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

Executing the critical section:
(c) Site S_i executes the CS after it has received the token

Suzuki–Kasami's Broadcast Algorithm

Releasing the critical section: Having finished the execution of the CS, site S_i takes the following actions:
(d) It sets LN[i] element of the token array equal to RN_i[i].

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

(f) If the token queue is non-empty after the above update, S_i deletes the top site ID from the token queue and sends the token to the site indicated by the ID

Suzuki–Kasami's Broadcast Algorithm

- ✓ **Correctness**
- ✓ mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution
- ✓ a requesting site enters the CS in finite time

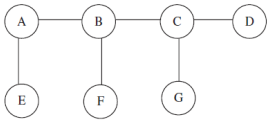
- ✓ **Performance**
- ✓ if a site holds the token, no message is required
- ✓ if a site does not hold the token, N messages are required
- ✓ synchronization delay is 0 or T

Raymond's Tree-Based Algorithm

- ❑ uses a spanning tree of the computer network to reduce the number of messages exchanged per critical section execution
- ❑ assumes that the underlying network guarantees message delivery
- ❑ time or order of message arrival cannot be predicted
- ❑ all nodes of the network are completely reliable
- ❑ a spanning tree of a network of N nodes will be a tree that contains all N nodes
- ❑ a minimal spanning tree is a spanning tree with minimum cost
- ❑ cost function is based on the network link characteristics

Raymond's Tree-Based Algorithm

- ➢ messages between nodes traverse along the undirected edges of the tree
- ➢ node needs to hold information about and communicate only to its immediate-neighboring nodes



- tree is a spanning tree of 7 nodes A, B, C, D, E, F, and G
- node C holds information about and communicates only to nodes B, D, and G
- node C does not need to know about the other nodes A, E, and F for the operation of the algorithm

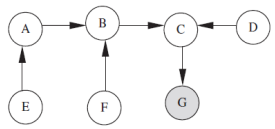
Raymond's Tree-Based Algorithm

- only one node can be in possession of the privilege (called the privileged node) at any time
  - except when the privilege is in transit from one node to another in the form of a PRIVILEGE message
- when there are no nodes requesting for the privilege, it remains in possession of the node that last used it

Raymond's Tree-Based Algorithm

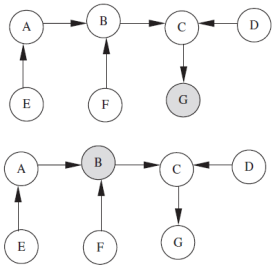
- each node maintains a HOLDER variable that provides information about the placement of the privilege in relation to the node itself
- a node stores in its HOLDER variable the identity of a node that it thinks has the privilege or leads to the node having the privilege
- HOLDER variables of all nodes maintain directed paths from each node to the node in possession of the privilege
- for two nodes X and Y, if $HOLDER_X = Y$, the undirected edge between X and Y can be redrawn as a directed edge from X to Y

Raymond's Tree-Based Algorithm



- if node G holds the privilege, figure can be redrawn
- shaded node represents the privileged node
- $HOLDER_A = B$ (as the privilege is located in a sub-tree of A denoted by B)
- $HOLDER_B = C$
- $HOLDER_C = G$
- $HOLDER_D = C$
- $HOLDER_E = A$
- $HOLDER_F = B$
- $HOLDER_G$ = self

Raymond's Tree-Based Algorithm



- node B does not hold the privilege and wants to execute the CS
- B sends a REQUEST message to $HOLDER_B$, i.e., C, which in turn forwards the REQUEST message to $HOLDER_C$, i.e., G
- privileged node G, if it no longer needs the privilege, sends the PRIVILEGE message to its neighbor C, which made a request for the privilege, and resets $HOLDER_G$ to C
- C forwards the PRIVILEGE to B, since it had requested the privilege on behalf of B
- C also resets $HOLDER_C$ to B

Raymond's Tree-Based Algorithm

- **Data Structures**
- HOLDER
  - possible values "self " or the identity of one of the immediate neighbors
- USING
  - possible values true or false
  - indicates if the current node is executing the critical section
- ASKED
  - possible values true or false
  - indicates if node has sent a request for the privilege
  - prevents the sending of duplicate requests for privilege

Raymond's Tree-Based Algorithm

Data Structures

- REQUEST_Q
  - FIFO queue that could contain "self " or the identities of immediate neighbors as elements
  - REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege
  - maximum size of REQUEST_Q of a node is the number of immediate neighbors + 1 (for "self ")

Raymond's Tree-Based Algorithm

Correctness

The algorithm guarantees the following:

- ❖ **mutual exclusion**
- ❖ **deadlock is impossible**
- ❖ **starvation is impossible**

Raymond's Tree-Based Algorithm

Cost and performance analysis

The algorithm exchanges to execute the CS

- *O*(log N) messages under light load
- approximately four messages under heavy load,

N being the number of nodes in the network

Recap Quiz

Q1. The system throughput in distributed computing systems in terms of synchronization delay SD and average execution time E at the CS can be expressed as
(a) SD + E (b) 1/(SD + E) (c) 1/SD + 1/E (d) SD – E
Q2. The Lamport's mutual exclusion algorithm requires the channel to deliver messages in _____
(a) FIFO (b) non-FIFO (c) causal order (d) none of the above
Q3. The Lamport's mutual exclusion algorithm requires ____ messages per CS invocation
(a) 2(N-1) (b) (N-1) (c) 4(N-1) (d) 3(N-1)
Q4. The Ricart-Agrawala algorithm uses ____ to assign a timestamp to critical section requests
(a) Scalar clock (b) vector clock (c) NTP (d) none of the above
Q5. In which of the following mutual exclusion algorithms, each process maintains a deferred array?
(a) Maekawa (b)Lamport's (c) Ricart-Agrawala (d) Suzuki-Kasami
Q6. Consider the performance metrics of mutual exclusion algorithms. If there is always a pending request for critical section at a site, then it is called
(a) Low load (b) high load (c) medium load (d) none of the above
Q7. Which of the following is a quorum based distributed mutual exclusion algorithm?
(a) Maekawa (b)Lamport's (c) Ricart-Agrawala (d) Suzuki-Kasami
Q8. The name of the variable that provides information about the placement of the privilege in relation to the node itself in Raymond's tree based algorithm is called
(a) Marker (b) signal (c) assertion (d) holder
Q9. If the number of sites is N = 25, then according to the Maekawa algorithm, then the number of request sets of sites, K will be
(a) 5 (b) 10 (c) 15 (d) 25
Q10. Which of the following is a token based distributed mutual exclusion algorithm?
(a) Maekawa (b)Lamport's (c) Ricart-Agrawala (d) Suzuki-Kasami

Recap Quiz - Key

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
b	a	d	a	c	b	a	d	a	d

Reference

❖Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 9, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).