Week 1

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 24/Jul/2021

NOTE THAT THIS PAGE HAS DIAGRAMS THAT ARE BEST VISIBLE IN LIGHT MODE

Topics Covered

1. What is a flipped mode course?
2. What is a distributed system?
3. Motivations for a distributed system
4. Coupling
5. Parallel Systems
6. UMA (Uniform Memory Access) Model
7. Omega Network (An example of UMA)

What is a flipped mode course?

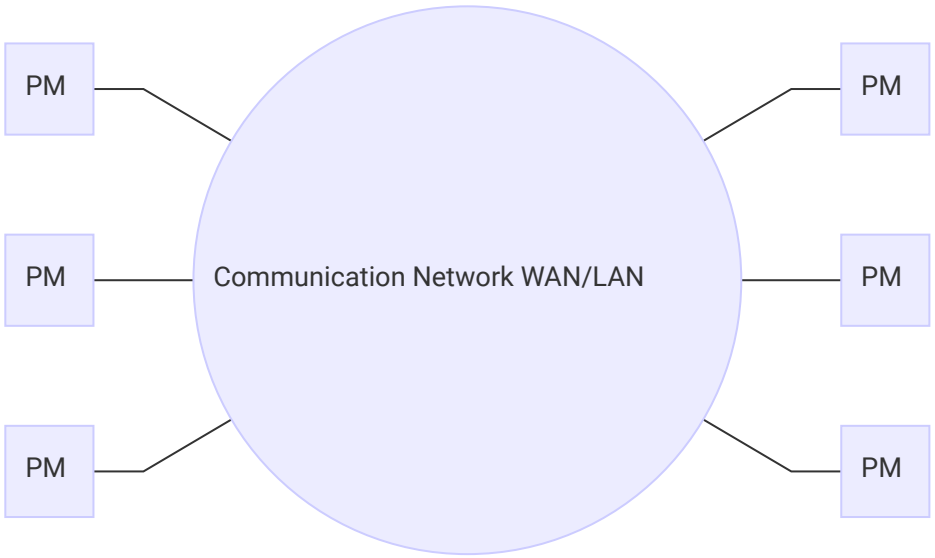
A course where there is content from a course ware and a live lecture

the events
Quizzes and Assignments will be done online in elearn portal

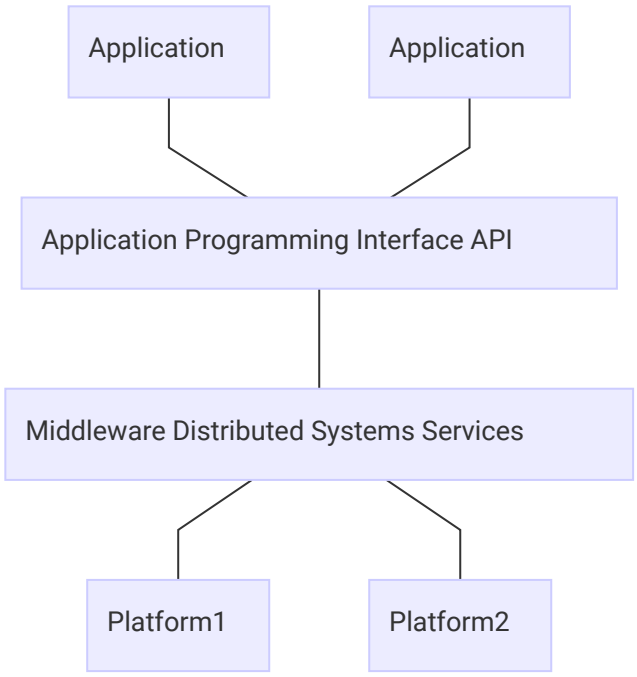
2 Quizzes (MCQ type) predetermined time slots
1 Assignment
1 Mid Sem (Half of the modules) (Theoretical)
1 Comprehensive (All modules) (Theoretical)

What is a distributed system?

Collection of independent individual entities (Can function on it's own) to solve a given task collectively



1. No common physical clock(system clock)
2. No shared memory - employs message passing for communication
3. Geographical separation - All the nodes taking part in the problem solving can be placed in different locations geographically
4. Autonomy and heterogeneity
    a. Each node is a fully functioning independent system irrespective of being taking part in a distributed system.
    b. The processors of the nodes are loosely coupled
    c. Different processor speeds and operating systems are allowed
    d. And despite all these differences they cooperate with one another



Middleware drives the distributed system and the heterogeneity at a platform level is abstracted by APIs.

Common Object Request Broker Architecture (CORBA)
Remote Procedure Call (RPC)
Distributed Component Object Model (DCOM)
Remote Method Invocation (RMI)

Motivations for a distributed system

1. Share resources
2. Access to resources from different geographical locations (Like AWS Cloud servers at different locations or even a work from home situation can be covered)
3. Increased performance/cost ratio, since there is a large resource pool and programs can be written in a way to efficiently use that pool to get tasks done more quickly
4. Reliability, in the sense that since the system is distributed, even if one system breaks down, there is still a degree of availability of resources.
5. Scaling, in the sense that it is easy to increase performance by merely adding more nodes to a distributed system.

6. Modularity and Incremental Expandability.

Coupling

High coupling: Homogeneous modules and hence have more restrictions imposed on these systems
Low coupling: Heterogeneous modules and hence more flexibility is gained

Parallel Systems

1. Multiprocessor systems:
    - Direct access to shared memory area/address space
    - Usually do not have a common system clock
    - Eg, Omega, Butterfly Networks
2. Multicomputer parallel systems
    - There are multiple processors but no direct access to shared memory/address space
    - There can be more than one nodes, but most likely are not geographically separated
    - Eg, IBM Blue gene, CM* Connection Machine
3. Array Processors
    - Collocated
    - Tightly Coupled
    - Common system clock

UMA (Uniform Memory Access) Model

1. Direct access to shared memory
2. **Access latency**: Waiting time to complete an access to any memory location from any processor
3. Access latency is same for all processors
4. Processors remain in close proximity
5. Connected by an interconnection network
6. Processors are of the same type

Omega Network (An example of UMA)

2x2 switching elements
data can be sent on any one for the input wires
n-input and n-output network uses
$log_2(n)$ stages

Week 2

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 31/Jul/2021

NOTE THAT THIS PAGE HAS DIAGRAMS THAT ARE BEST VISIBLE IN LIGHT MODE

Topics Covered

Module 1

1. Multicomputer parallel systems
    a. NUMA (Non Uniform Memory Access) Model
    b. Wraparound mesh interconnection in an MPS
    c. Hypercube interconnection in an MPS
2. Distributed/Parallel Computing Jargon
    a. Parallelism/Speedup
    b. Parallel/Distributed Program Concurrency
3. Distributed Communication Models
    a. RPC (Remote Procedure Call)
    b. Publish/Subscribe

Module 2

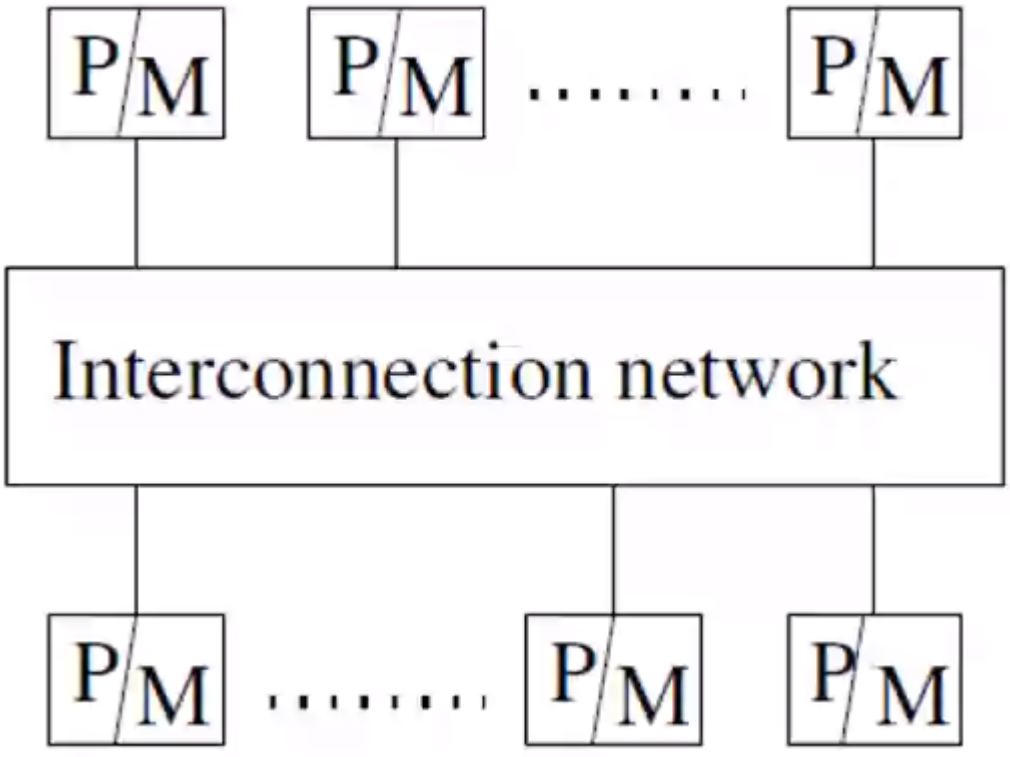
1. Preliminary understandings
2. Notations
3. Space time diagram
    a. What are these letters?
    b. Causal precedence/dependency or happens before relation
    c. Concurrent events

Module 1

Multicomputer parallel systems

- Processors **do not have direct access to shared mem**.
- Usually **do not have a common clock**.
- Processors are in **close proximity**. (THIS IS DIFFERENT FROM A DS)
- Very **tightly coupled**. (THIS IS DIFFERENT FROM A DS)
- Connected by an interconnection network.
- Communicate via Message passing APIs/common address space. (THIS IS DIFFERENT FROM A DS as DS can only use MP APIs).
- Typically this is used for high performance computing use cases.

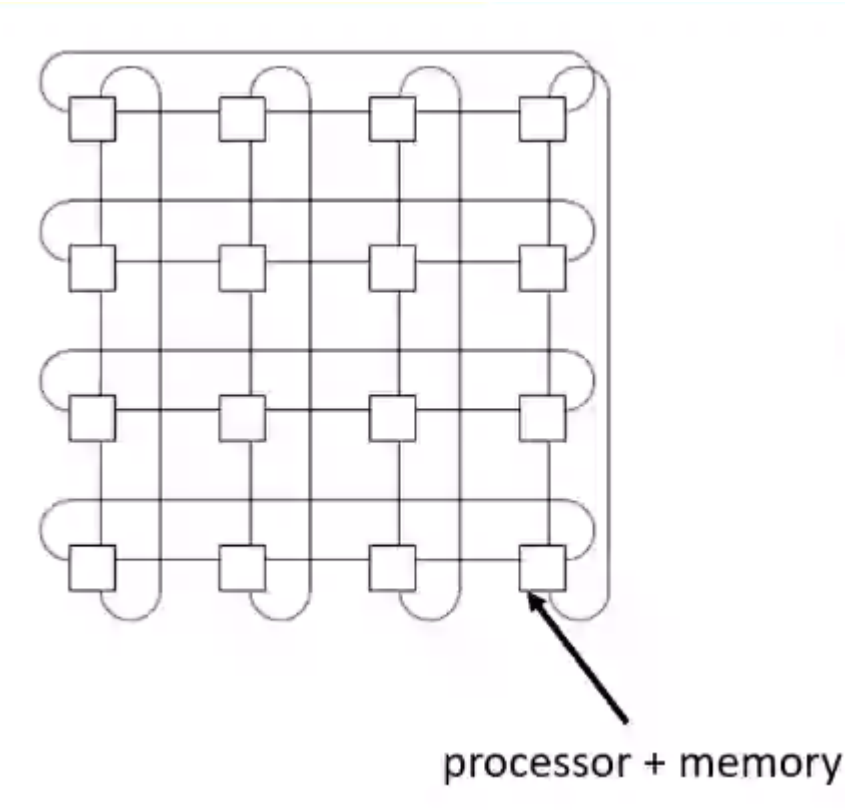
NUMA (Non Uniform Memory Access) Model



- Above you can see that the individual mem components are not clubbed together like the UMA Model (See here Week1DC#UMA Uniform Memory Access Model).
- The processor can still access the mem components but have a different access latency
- non-uniform memory access - Different processor can have different access latency

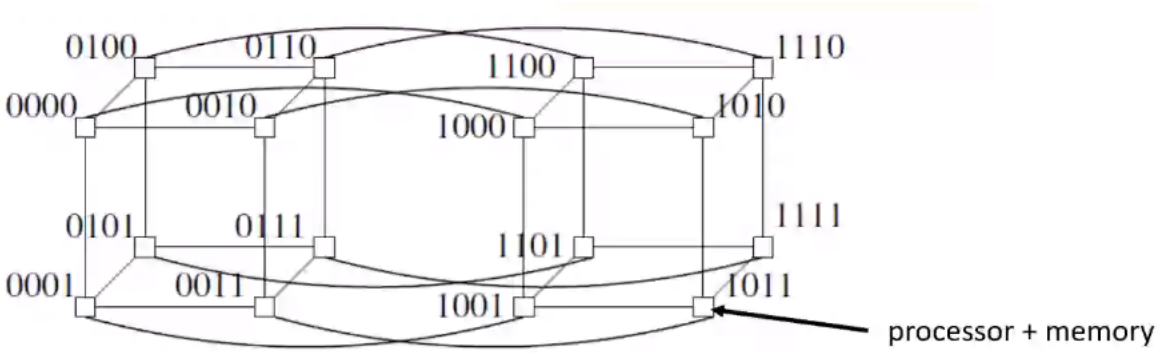
- Multicomputer system having a **common address space**.
- Latency to access various shared memory location from the different processors varies.

Wraparound mesh interconnection in an MPS



- The above diagram shows a 2d wraparound mesh
- The above diagram corresponds to a $4x4$ mesh containing 16 nodes.
- Each node is connected to each and every node around it. The leaf nodes wrap around to the opposite leaf node of their respective row or column, hence the name **Wrap Around Configuration**.
- If any of the nodes go down, the wrap around connection helps in maintaining the connection.
- The connection can be bidirectional through a bus or any interconnection logic for that matter.

Hypercube interconnection in an MPS



- A k dimensional hypercube has 2^k processor and mem units (nodes)
- Each node has at most k connections
- Each dimension is associated with a bit position in the label
- Labels of two adjacent nodes differ in the k^{th} bit position for the dimension k
- From node 0000 we can see that each direction differs by a single bit. bottom is 0001, right is 0010 and the adjacent one is 0100
- We can by applying the above rules get to know the label for the other adjacent nodes
- The shortest path between any two processors is called *Hamming distance*
- consider two bit labels 0110 and 1001, the hamming distance would be 4 since all 4 bits differ.
- consider two bit labels 0111 and 0110, the hamming distance would be 1 since only 1 bit differs.
- The above calculations can be used to connect the nodes according to the label
- In the case of such an interconnection the $hamming\ distance <= k$
- Even if some of the nodes go down, there will be another path present to keep the connection alive. **This provides fault tolerance and congestion control mechanism**
- **Routing of messages happen hop by hop**

Distributed/Parallel Computing Jargon

Parallelism/Speedup

$$Speedup = \frac{T(1)}{T(n)}$$

- $T(1)$ = Time taken on a single processor
- $T(n)$ = Time taken with n processors
- Measure of the relative speedup of a specific program on a given machine
- This depends on:

    a. Number of processors

b. Mapping of the code to the processors

Parallel/Distributed Program Concurrency

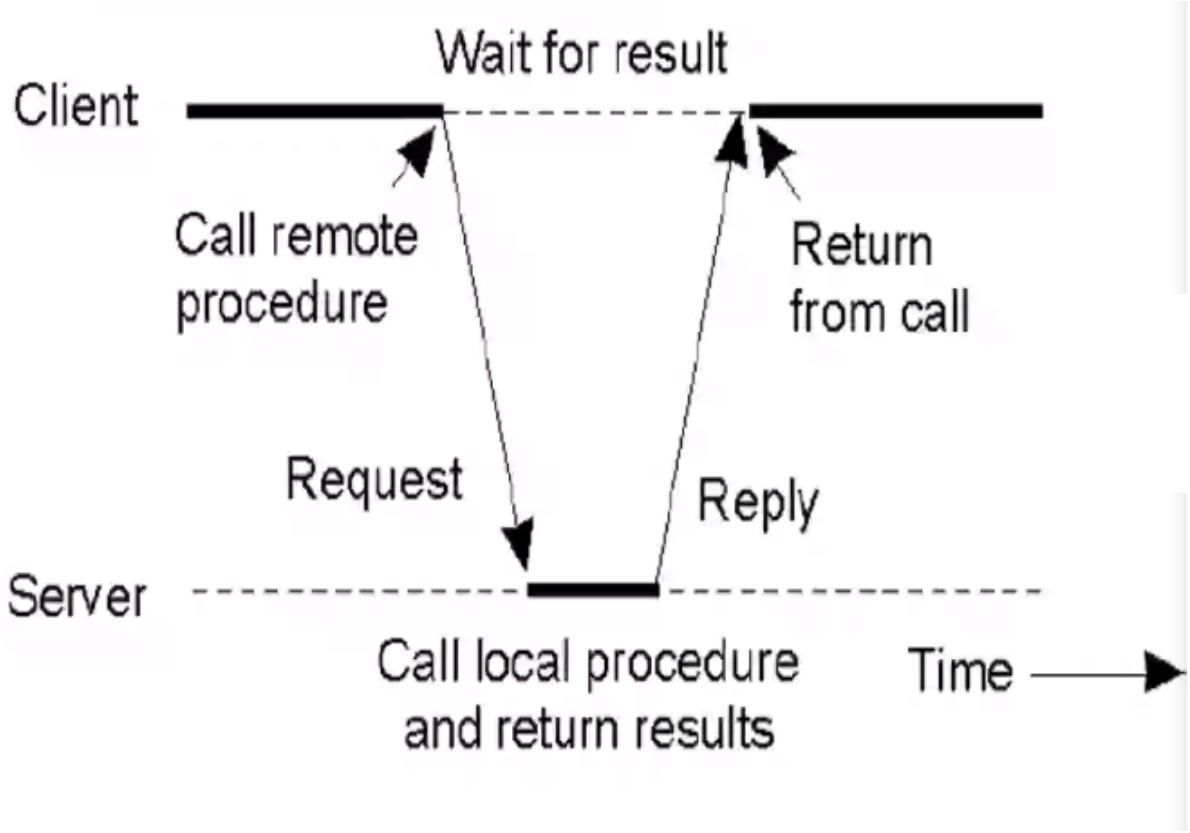
$$Parallel\ Program\ Concurrency = \frac{number\ of\ local\ operations}{total\ number\ of\ operations}$$

- $number\ of\ local\ operations$ -> Non communication and non shared memory access operations
- $total\ number\ of\ operations$ -> All operations including communication or shared memory access

Distributed Communication Models

RPC (Remote Procedure Call)

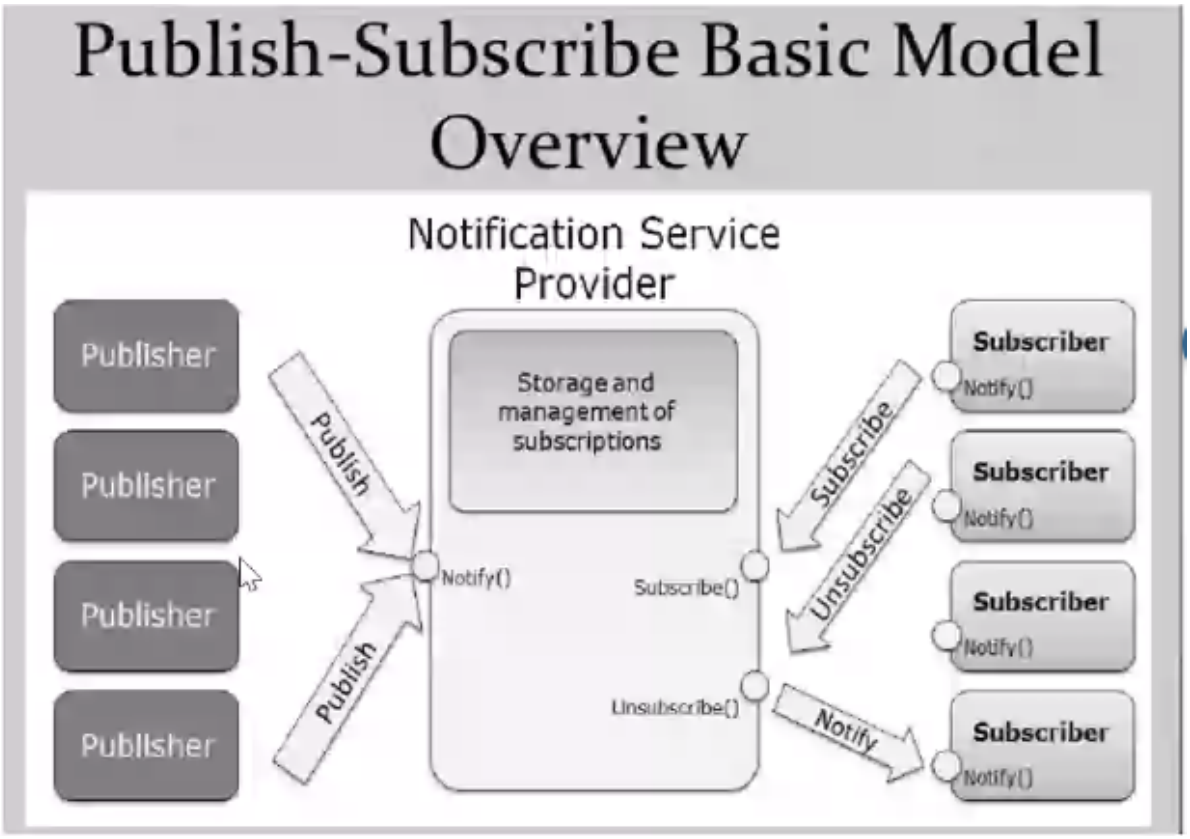
This topic is covered in a pre reading session here: PreReadingWeek1DC#RPC Remote Procedure Call



- The client procedure calls a client stub to pass params
- The client marshals the params (makes a common representation of the params), builds the

message, and calls the local OS
- The local OS sends the message to the remote OS
- The server's remote OS gives the message to a server stub
- The server stub de-marshals the params(Converts common form to server OS specific form) and calls the desired server routine
- The server computes the result and hands it to the server stub
- The server stub marshals the result into a message and calls the local OS
- The server OS sends the message to the client OS
- The client OS receives the message and sends it to the client stub
- The client stub demarshals the result, and execution returns to the client

Publish/Subscribe



- We have two entites mainly:
- **Publishers**: Those who create information and publish them to a service (Through a `Notify()` call)
- **Subscriber**: Those who consume the content published by publishers it had subscribed to.
- The Subscriber gets data asynchronous though a notification, and they can continue their tasks until they receive notifications.
- The subscriber has an option to unsubscribe as well

Module 2

Preliminary understanding

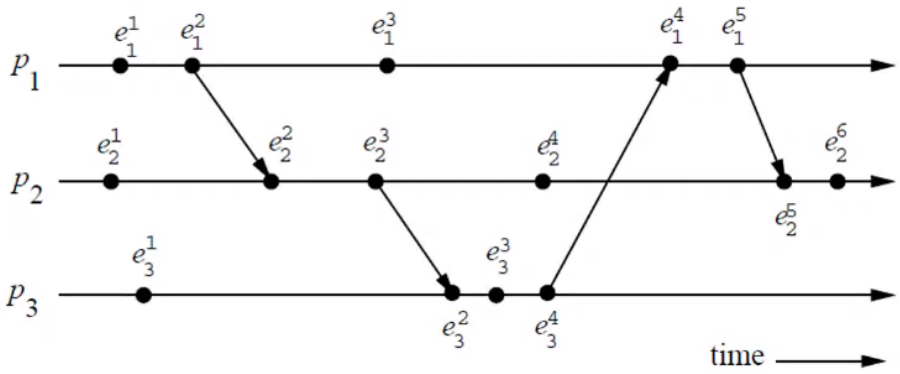
1. DS consists of a set of processors.
2. There is an intercommunication network.
3. Delay in communication is finite but is not predictable.
4. Processors do not share a common global memory.
5. Uses asynch message passing.
6. No common physical global clock.
7. Possibility of messages being delivered out of order.
8. Messages may be lost, garbled or duplicated due to timeouts and retransmissions.
9. There is always a possibility of communication and processor failure.

Notations

- C_{ij} : Channel from process p_i to process p_j
- m_{ij} : message sent by p_i to p_j
- Global state GS of a distributed computation consists of:
  - States of the processes: Local memory state
  - States of the communication channels: set of messages in transit

  Global state is explained in detail here PreReadingWeek1DC#Global state of a DS
  - Occurrences of events e
  - Causes changes in respective processes states
  - Causes changes in channels states
  - Causes transition in global system state

Space time diagram

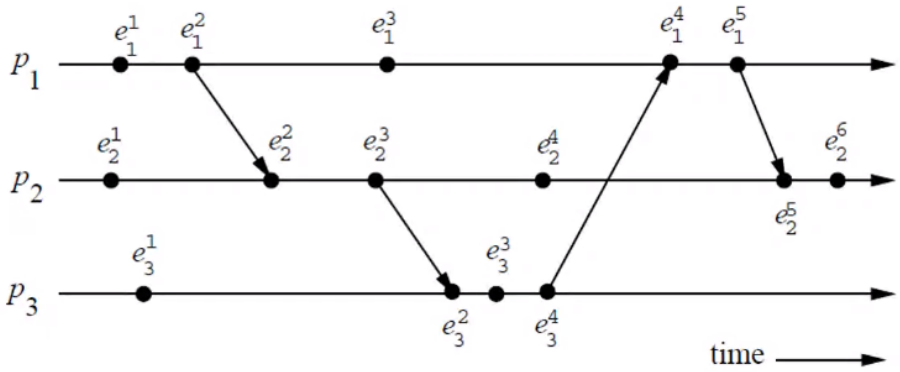


$\bullet e_1^1 \rightarrow e_3^3$
$\bullet e_3^3 \rightarrow e_2^6$

What are these letters?

- p_i denotes the processes in the space axis
- e_i^j denotes the j^{th} event in process p_i
- Events that do not have arrows are
- The arrows denote the direction of the message sent.
  - e_1^2 is a message sending event
  - e_3^2 is a message receive event

Causal precedence/dependency or happens before relation



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

- Relation '$\rightarrow$' denotes flow of information in a distributed computation
- $e_i \rightarrow e_j$ implies that all the information at e_i is accessible at e_j
- In the above example we can see that e_1^1 has a complete path to e_3^3 hence we can be certain that $e_1^1 \rightarrow e_3^3$

Concurrent events

Week 3

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 07/Aug/2021

Topics Covered

1. Models of Communication Networks
    a. FIFO Model
    b. Non FIFO Model
    c. Causal Ordering Model
2. Global State of a DS
    a. Consistent Global State
3. Cuts of a DC
    a. Consistent vs inconsistent cut
4. Logical Time
    a. Scalar Time
        i. Notations
        ii. Rules for updating scalar clock
        iii. Height of an event
    b. Vector Time
        i. Notations
        ii. Rules for updating vector clock
        iii. Event Counting

Models of Communication Networks

FIFO Model

- Each channel acts as a FIFO queue
- Since the messages are sent in a queue, ordering is preserved by the channel
- If P_1 sends m_1, m_2 to P_2 then P_2 receives those two messages as a queue in the same order

Non FIFO Model

- The channel acts as a set of messages
- Since it is a set, the messages do not necessarily have the order maintained
- Sender process adds messages to channel
- Receiver process removes messages from it
- This helps in relieving the communication network from the overhead of maintaining the message ordering

Causal Ordering Model

- This is based on the happens before relation.
- Such a system must satisfy:
  - If there are two messages m_{ij} and m_{kj}, then if $Send(m_{ij}) \rightarrow Send(m_{kj})$, then it must also follow $Rec(m_{ij}) \rightarrow Rec(m_{kj})$
  - Note that the messages are coming to the same process.
- Causally ordered messages implies FIFO message delivery
- $CO \subset FIFO \subset NonFIFO$

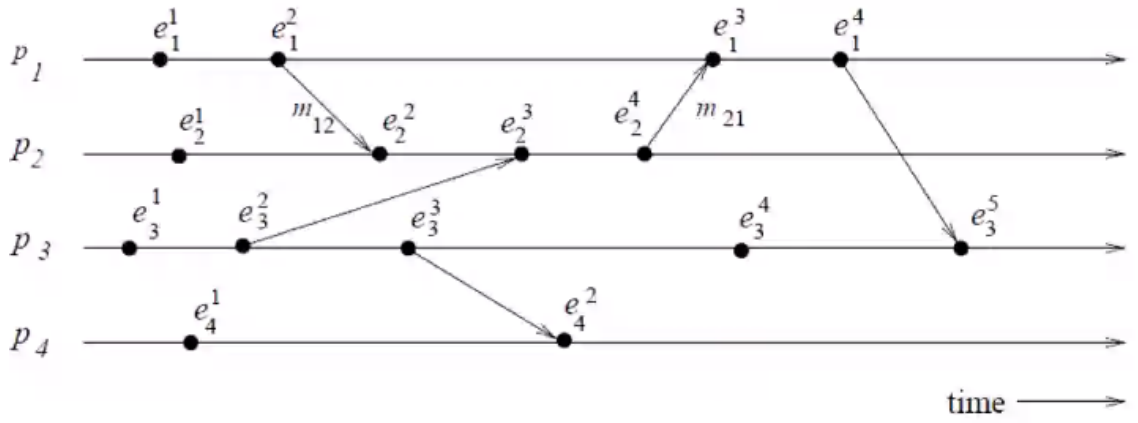
Global State of a DS

More details here Week2DC#Notations.
- LS_i^x : State of process p_i after the occurrence of event e_i^x and before e_i^{x+1}
- SC_{ij} : State of channel C_{ij}

Consistent Global State

- A state is meaningful, meaning that **every message that is recorded as received is also recorded as sent**

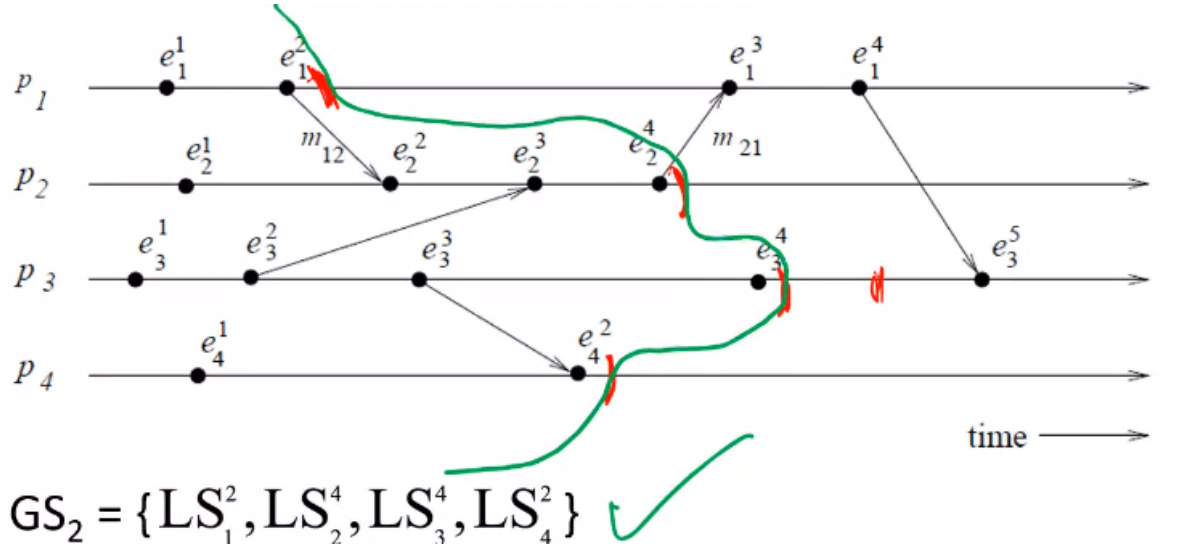
- For all m_{ij}, $send(m_{ij}) \notin LS_i^x \implies m_{ij}/notinSC_{ij}rec(m_{ij}) \notin LS_j^y$



$$GS_2 = \{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$$

  - The above space time diagram is consistent, because when we see all the LS that define the GS all the sends are recorded before the receives of that message
  - Consider a GS of $LS_1^3, LS_2^3, LS_3^4, LS_4^2$, here the GS is inconsistent, because the receive of message m_{21} is recorded but not the send.

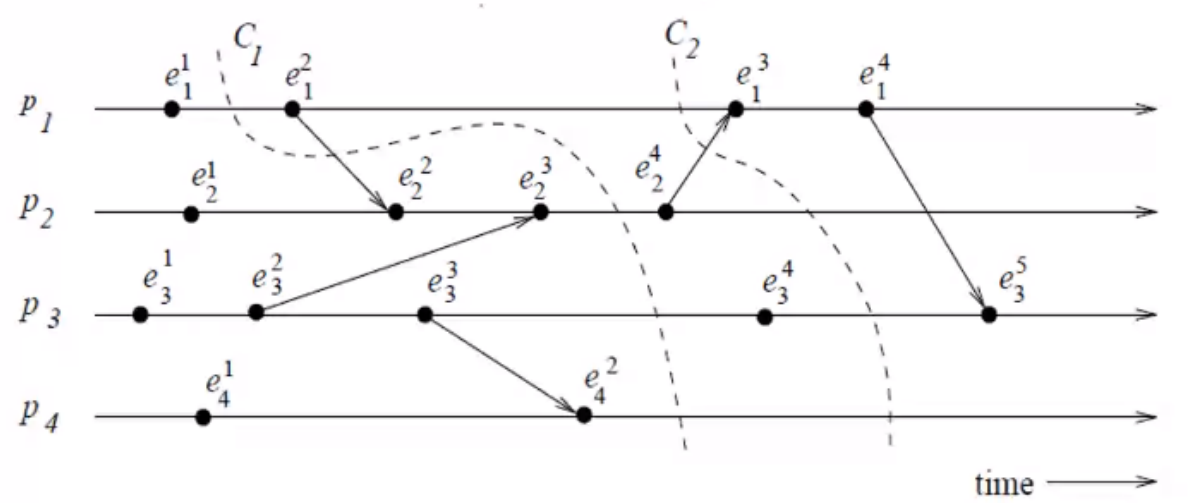
Cuts of a DC



$$GS_2 = \{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$$

- Any zig-zag line drawn that cuts all the processes in a space time diagram
- The events in a DS is partitioned into:
- Past: Contains all events left of the cut
- Future: Contains all events right of the cut
- A cut is a representation of the GS at those points in the cut and processes

Consistent vs inconsistent cut



- **Consistent cut**:
- Every message received in the PAST of the cut was sent in the PAST of that cut
- All messages that cross the cut from the PAST to the FUTURE are **in transit**
- The above image $C2$ is an example since all the sends and receives happen in the PAST and the message between e_2^4 to e_1^3 is in transit.
- **Inconsistent cut**:
- If the receive is recorded in the PAST and the send is in the FUTURE then it is an inconsistent cut
- In the above image $C1$ is inconsistent since $send(e_1^2)$ is in the FUTURE and $rec(e_2^2)$ is in the past.

Logical Time

- To show a sense of time in the sense of
- Logical time follows Monotonicity property: if event α has happened before event β then the timestamp of event α is less that the timestamp of event β
- Every process has a LC
- LC is advanced using a set of rules
- each event is assigned as timestamp
- There are two ways to represent logical time:
  - Scalar Time
  - Vector Time

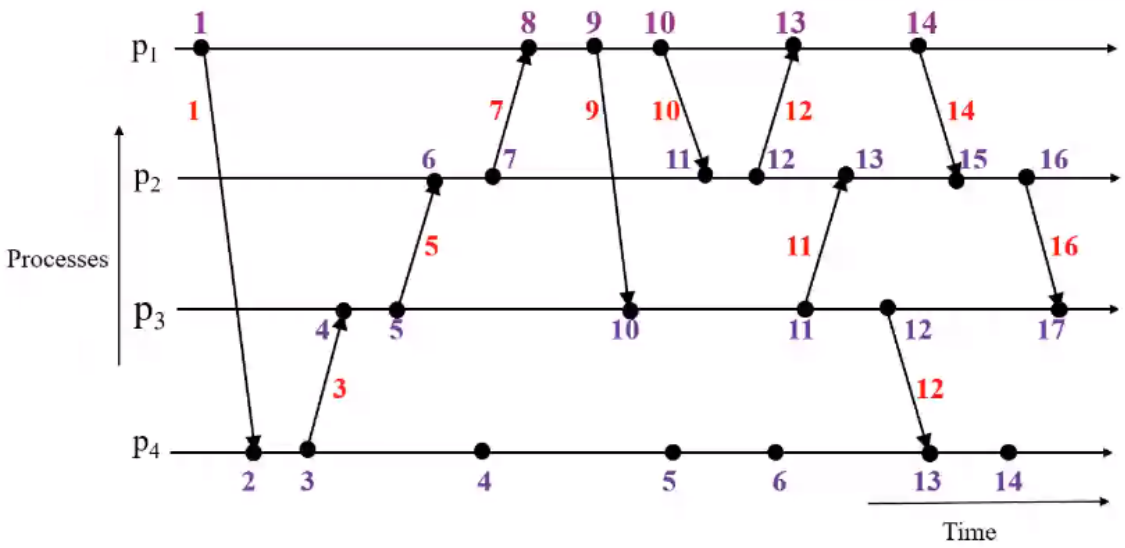
Scalar Time

Notations

- This is covered in the pre reading content here: PreReadingWeek1DC#Scalar Time
- Time domain is the set of non negative integers
- C_i : Integer variable, denotes the logical clock of p_i

Rules for updating scalar clock

- **R1**: Before executing an event, process p_i executes
  - $C_i = C_i + d \ (d > 0)$
  - d can be any value but usually is 1
- **R2**: When process p_i receives a message with a timestamp C_{msg}, it executes the following actions:
  - $C_i = mac(C_i, C_{msg})$
  - execute **R1**
  - deliver the message
- Whenever there is an internal event the **R1** is executed
- The above rules can be seen in action in the steps below:



Height of an event

- Height is defined as the number of events that causally precedes it

- If $d = 1$, the height of a given event is 1 minus the timestamp at that event.

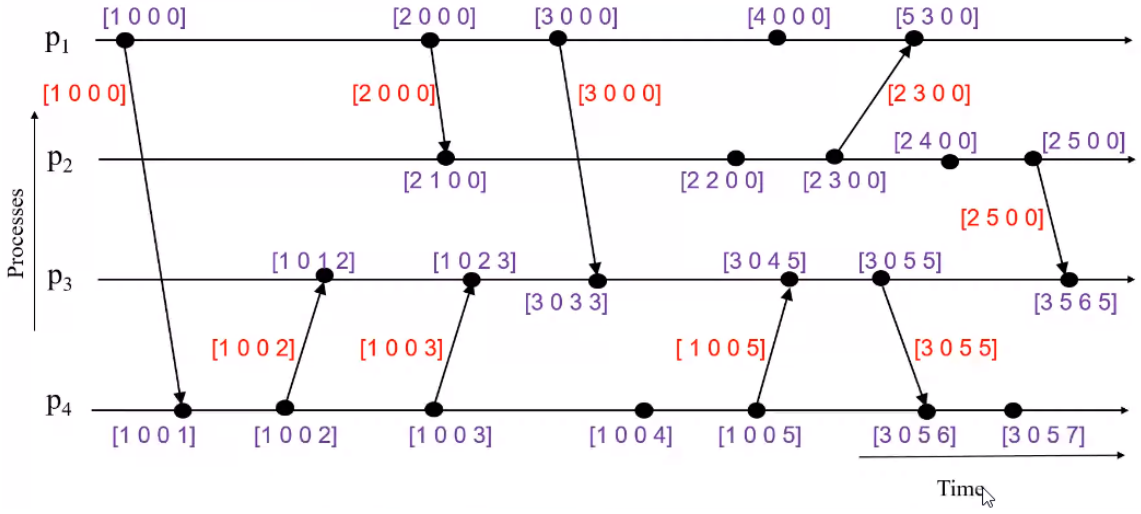
Vector Time

Notations

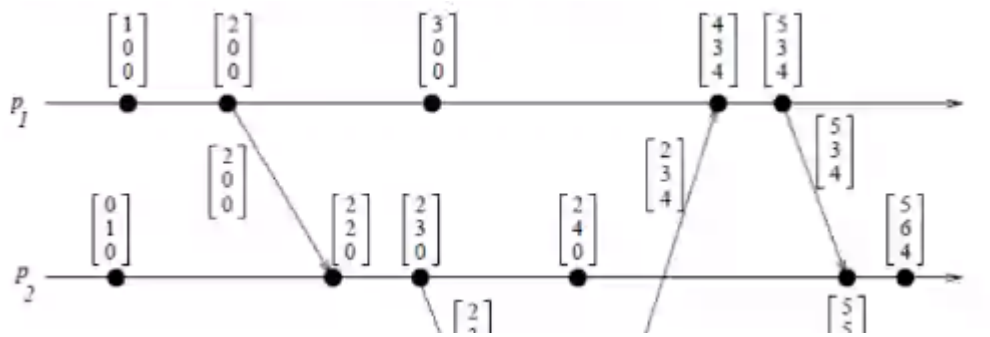
- Time domain is represented by a set of n-dimensional non-negative integer vectors. (Can be a row or a column vector)
- $vt_i[i], vtj[j]$

Rules for updating vector clock

- **R1**: First take the element wise max of the vector received and the local vector
- **R2**: Add 1 to the local clock index of the vector decided above
- The above rules can be seen in action in the steps below:



Event Counting


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

Week 4

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

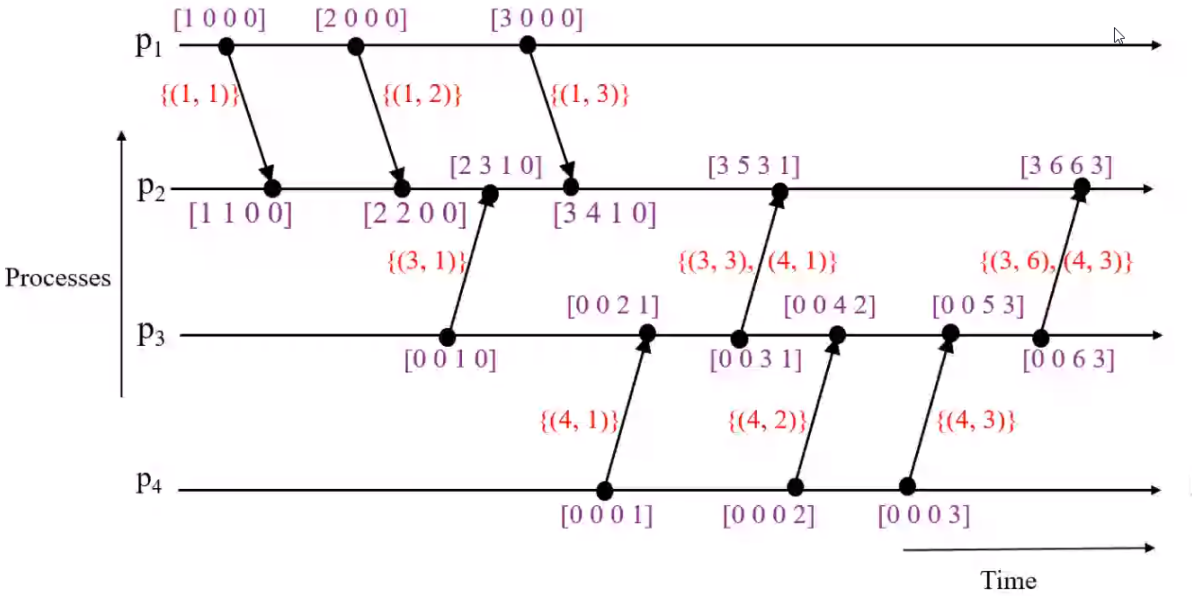
BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 21/Aug/2021

Topics Covered

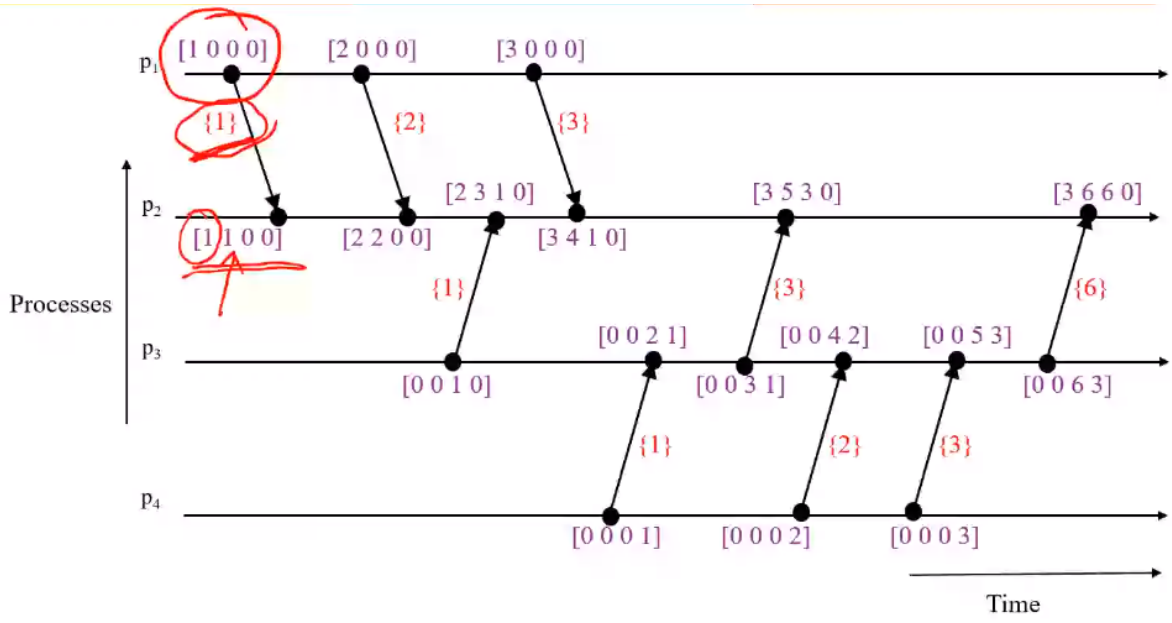
1. Singhal-Kshemakalyani's Differential Technique
2. Fowler-Zwaenepoel's Direct-Dependency Technique
3. Problems in recording global state
4. Some Key Concepts of Consistency and Channel State
5. Snapshot Recording Algorithms
    a. Chandy Lamport Algorithm (For FIFO)
    b. Lai and Yang algorithm (For NON-FIFO)

Singhal-Kshemakalyani's Differential Technique



1. The message contains a list of tuples that follows the given syntax: $\{(index, val)\}$, where $index$ = Process that has the clock value of = val.
2. We see that a process sends only the details of the clocks that are changed on that given process only.
3. This method considerably saves the amount of data that is sent between processed in the initial timing

Fowler-Zwaenepoel's Direct-Dependency Technique



1. Here we send only the local clock value of the process to the receiver
2. Dependency to other process that is directly not connected by messages is lost.
3. Direct dependence relationship is preserved

Problems in recording global state

1. Lack of a globally shared memory
2. Lack of a global clock
3. Message transmission is asynchronous
4. message transfer delays are finite but unpredictable

Some Key Concepts of Consistency and Channel State

- 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 $transit(LS_i, LS_j)$
- For C_{ij}, intransit messages are:

$$transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$$

- global state GS is a ***consistent global state*** iff:
  - **C1:** every message m_{ij} that is recorded as sent in the local state of sender p_i must be captured either in the state of C_{ij} or in the collected local state of the receiver p_j
    - **C1 states the law of conservation of messages:**
  - **C2:** $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$
    - **Condition C2 states that for every effect, its cause must be present**

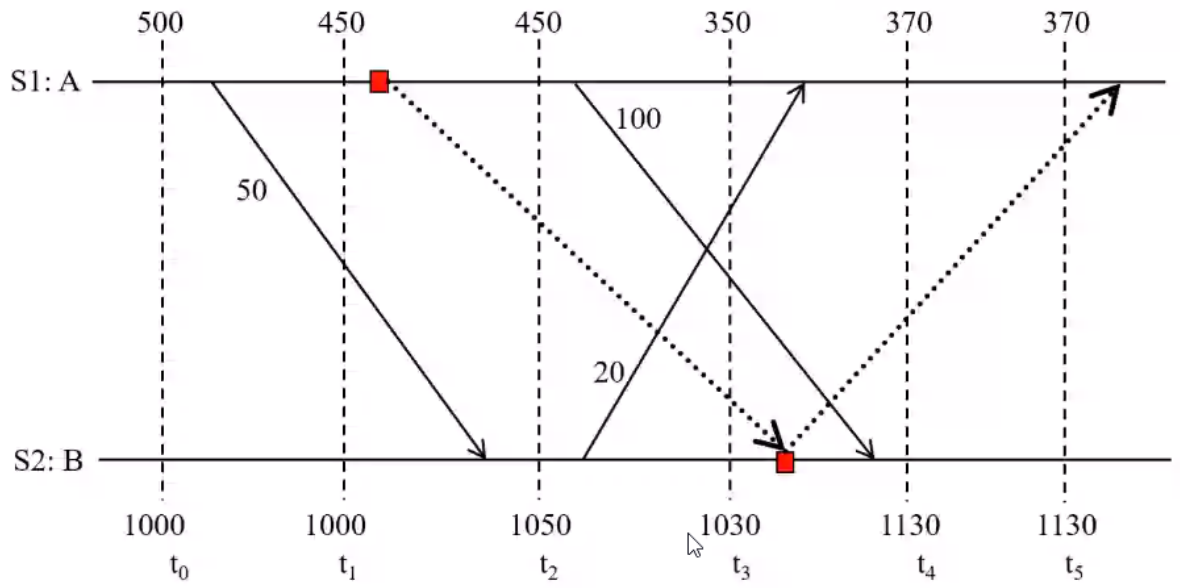
Snapshot Recording Algorithms

Chandy Lamport Algorithm (For FIFO)

Process p_i records its state
For each outgoing channel C on which a marker has not been sent, p_j sends a marker along C

before p_i sends further messages along C

On receiving a marker along channel C:
if p_j has not recorded its state then
Record the state of C as empty set
Execute the "marker sending rule"
else
Record the state of C as the set of messages received along C after p_j state was recorded and before p_j received the marker along C



The vertical dashed lines are the amount of moneyt present between users A and B.
The arrow shows a transaction between A and B

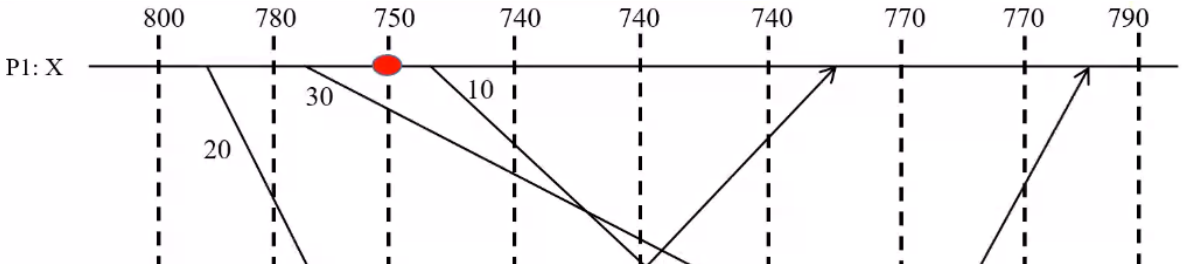
Let $S1$ sends a marker to $S2$ through C_{12} , so
$S1$ => saves 450 as the local state
$S2$ => Since it has not saved its local state, it will record the value of C_{12} as empty and save the local state as 1030
After it has recorded $S2$, Since S1 has already set its local state, it will store the
After $S1$ has recorded its local state

Lai and Yang algorithm (For NON-FIFO)

1. Since markers cannot be used in NON FIFO channels we use piggybacking, so that messages sent after and before the marker are distinguished.
2. In Lai and Yang we use coloring scheme to do the piggy backing

3. every process is initially white
4. process turns red while taking a snapshot
5. equivalent of the "marker sending rule" is executed when a process turns red
6. Every message sent by a white process is colored white
    a. A white message is a message that was send before the sender of that message recorded its local snapshot
7. Every message sent by a red process is colored white
    a. A white message is a message that was send before the sender of that message recorded its local snapshot

- 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} | send(m_{ij}) ∈ LS_i} – {m_{ij} | rec(m_{ij}) ∈ LS_j}



Week 4 (cont.)

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 22/Aug/2021

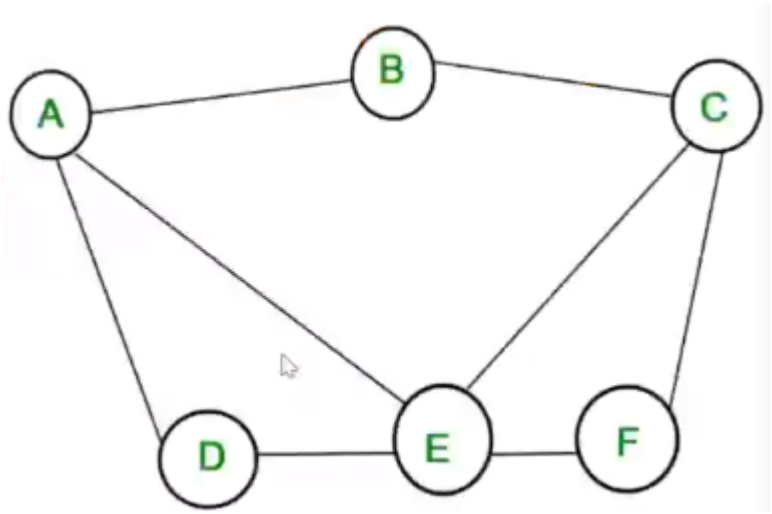
Topics Covered

1. Terminology and Basic Algorithms
    a. Notations and Definitions
    b. Synchronous Single-Initiator Spanning Tree Algorithm using Flooding
        i. Algorithm Design
            i. Struct for each process P_i
            ii. Algorithm pseudo code
            iii. Algorithm in action
    c. Broadcast and Convergecast Algorithm on a Tree
        i. Broadcast Algorithm
        ii. Convergecast Algorithm
        iii. Complexity
2. Message Ordering

Terminology and Basic Algorithms

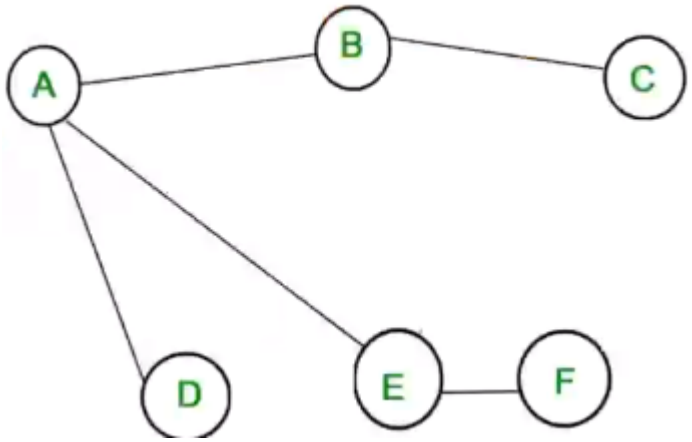
Notations and Definitions

- **Undirected unweighted graph** $G = (M, L)$, represents topology of an example graph such as the one shown below:



- Vertices are **nodes**
- Edges are **channels**
- $n = |N|$, Cardinality of set of all nodes $N = [A, B, C, D, E, F]$
- $l = |L|$, Cardinality of set of all edges $L = [AB, BC, CF, DE, EF, AD, AE, CE]$
- 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 where j belongs to the set N. We basically find all the minimum distances from i to all j and the diameter is the maximum value
  - In the above example let $i = A$ and $j = [A, B, C, D, E, F]$ then we can say that the max value of the lengths is 2 (In case of the path length for A to F), so the $diameter = 2$

- **A spanning tree** is a tree where the graph does not have any edges that will cause cycles in it. An example for a spanning tree for the above example is shown below:



- As a rule of thumb, a spanning tree for a graph with n nodes, will have $n-1$ edges
- So in the example graph has 6 nodes and the number of edges in the spanning tree is 5.

Synchronous Single-Initiator Spanning Tree Algorithm using Flooding

- Algorithm executes steps **synchronously** (When a message is sent, the sender waits till all the other nodes receives the messages)
- **Root** of the graph initiates the algorithm (An arbitrary node can be selected as the root)
- **QUERY messages are flooded**. This means that first the root node will send a message to all its nearby nodes, and inturn those nodes will send it to its neighbors and so on.
- The final spanning tree will have the root node as the initial arbitrary node selected in the graph.
- Each process $P_i (P_i \neq root)$ should output its own parent for the spanning tree. **In distributed computing we interchangeably use nodes and processes**

Algorithm Design

STRUCT FOR EACH PROCESS P_I

Variables maintained at each P_i	Initial variable values at each P_i
int *visited*	0
int *depth*	0
int *parent*	NULL

Variables maintained at each P_i	Initial variable values at each P_i
set of int *Neighbors*	set of neighbors

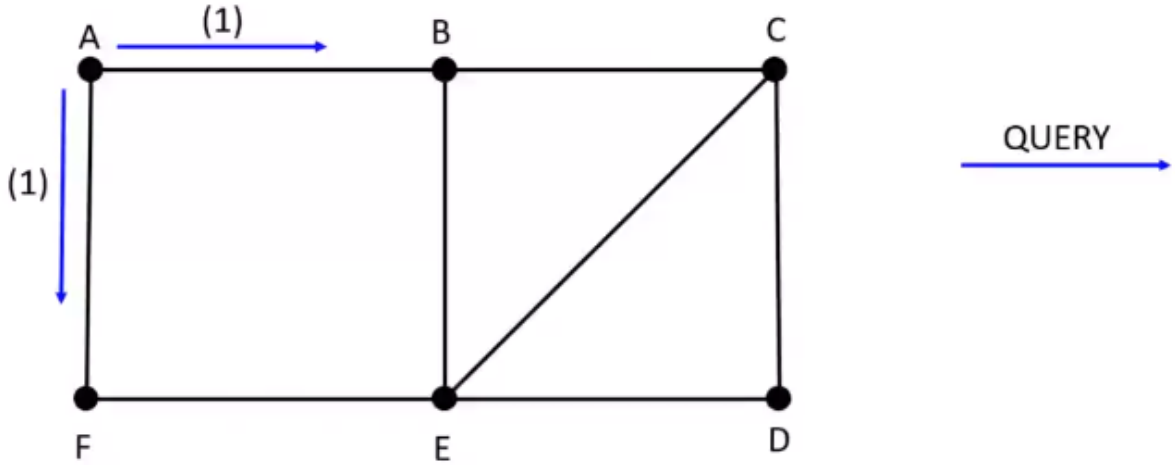
ALGORITHM PSEUDO CODE

```
Algorithm for Pi

When Round r = 1
if Pi = root then
    visited = 1
    depth = 0
    send QUERY to Neighbors
if Pi receives a QUERY message then
    visited = 1
    depth = r
    parent = root
    plan to send QUERY to Neighbors at the next round

When Round r > 1 and r <= diameter
if Pi planned to send in previous round then
    Pi sends QUERY to Neighbors
if Pi receives QUERY messages then
    visited = 1
    depth = r
    parent = any randomly selected nodes from which Query was receives plan to
send QUERY to Neighbors but not to any nodes from which send was received
```

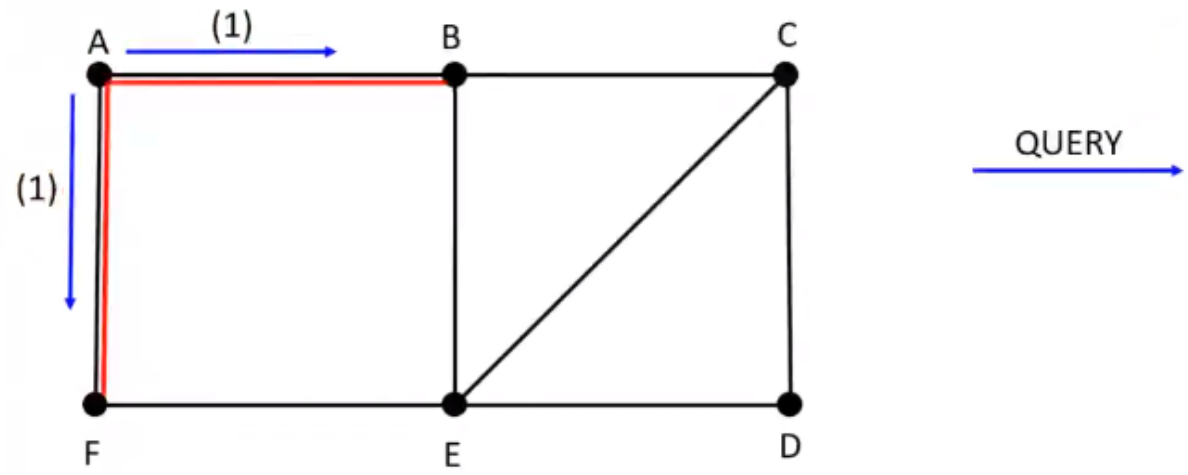
ALGORITHM IN ACTION

Round 1 :

Root A sends $QUERY$ to neighbors $[B, F]$

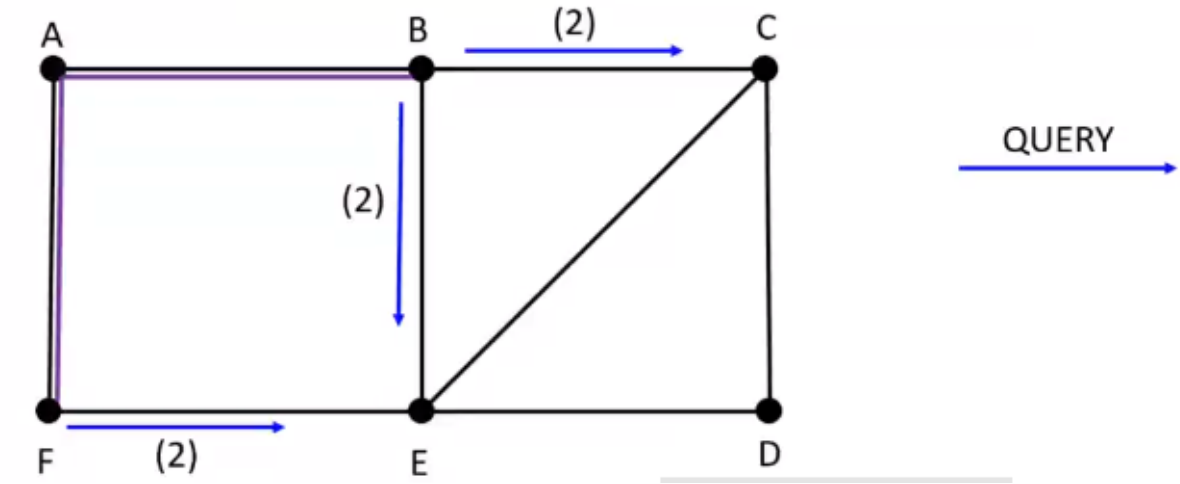


B and F set A as parent and plans to send $QUERY$ to its neighbors

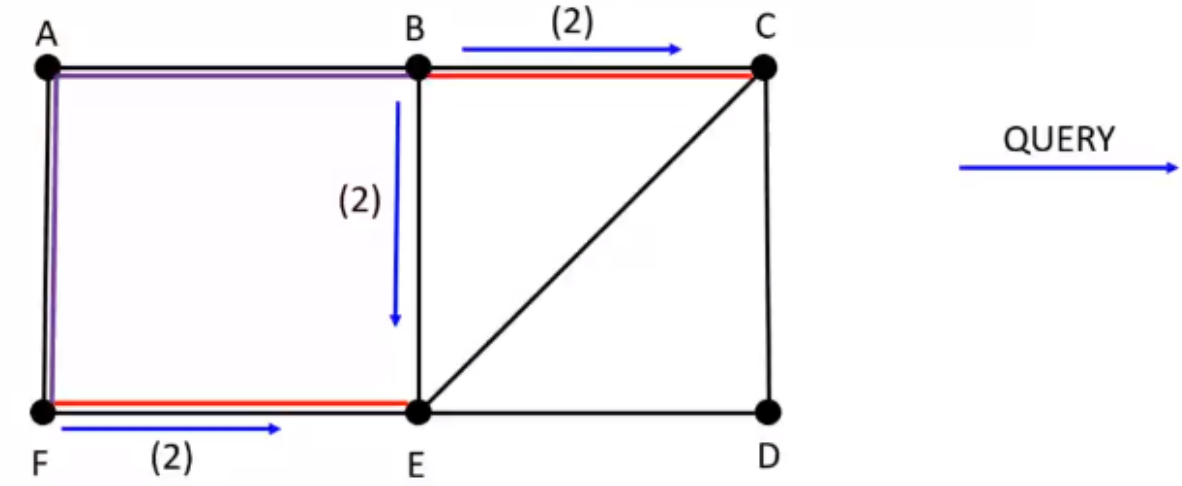


Round 2 :

B sends $QUERY$ to neighbors $[C, E]$ and F sends $QUERY$ to neighbors $[E]$

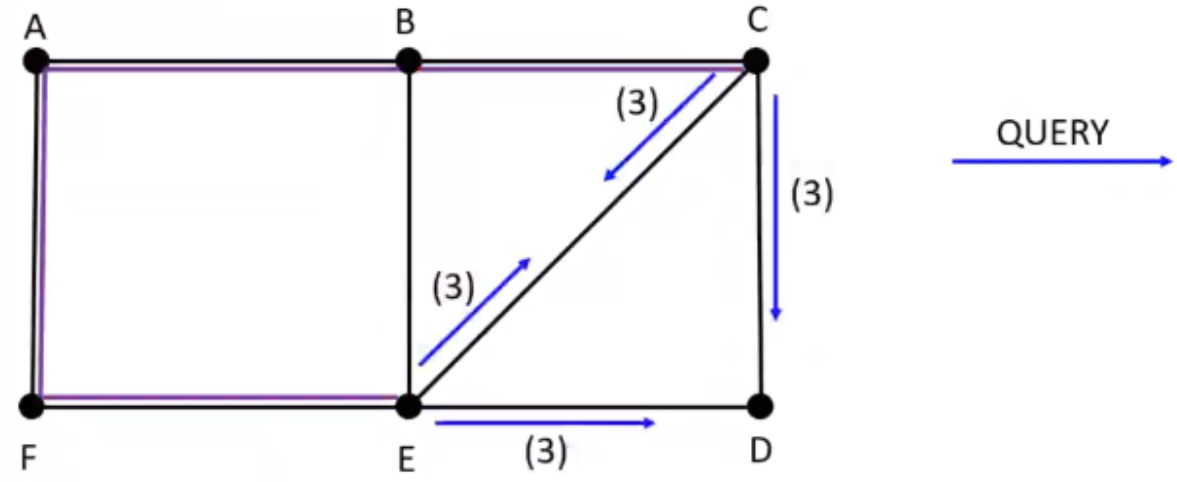


E randomly chooses F as parent and C chooses B as parent and both E and F plans to send $QUERY$ to its neighbors

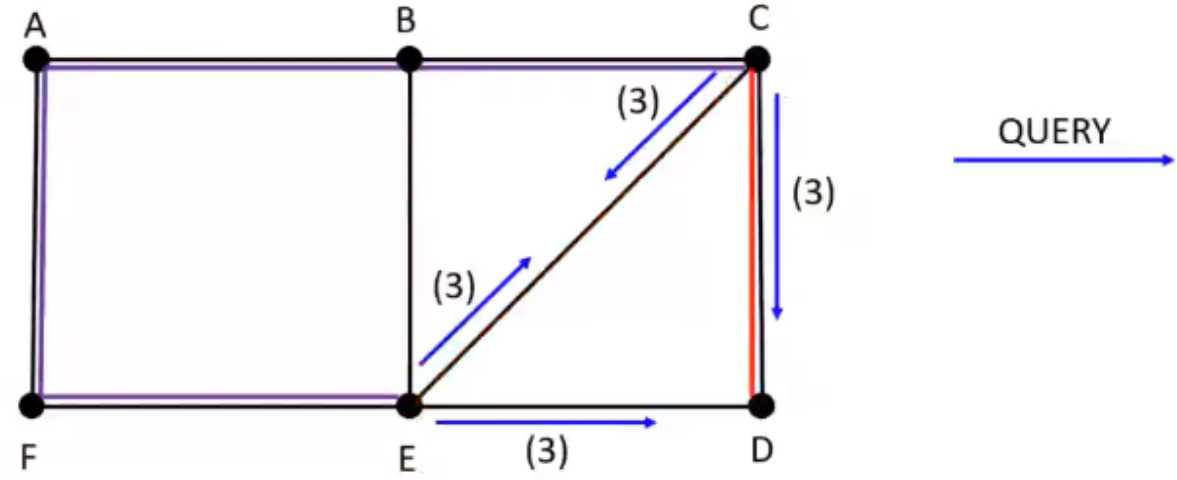


Round 3 :

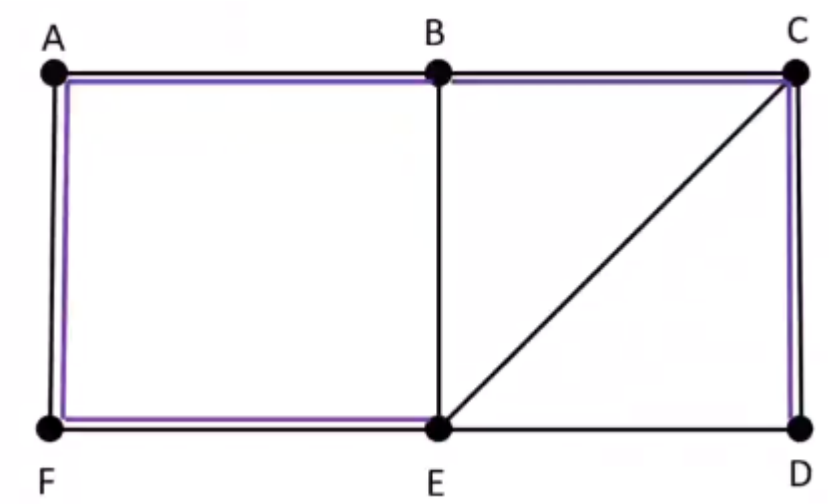
E sends $QUERY$ to neighbors $[C, D]$ and C sends $QUERY$ to neighbors $[E, D]$



Since C and E are already visited, the $QUERY$ is ignored in those cases and D randomly chooses C as parent over E



The spanning tree generated is as follows :

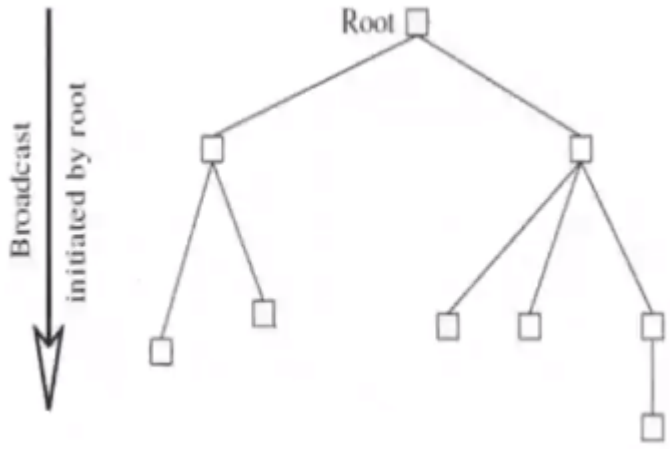


The above spanning tree has 6 nodes and 5 edges

Broadcast and Convergecast Algorithm on a Tree

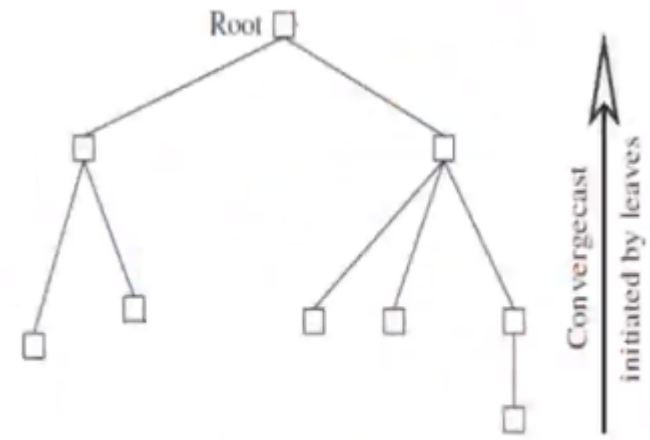
A spanning tree is useful for distributing (via **Broadcast**) and collecting information (via **Convergecast**) to and from all the nodes

Broadcast Algorithm



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

Convergecast Algorithm



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

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 $\mathcal{O}(n)$

Message Ordering

Group Communication

- **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

Causal Order

- Causal Order is explained here: Week3DC#Causal Ordering Model

- 2 criteria must be satisfied by causal ordering protocol
  - 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 messages at a process
      - Event at which the message is given to the application process
  - Liveness:
    - A message that arrives at a process must be eventually be delivered to the process

Raynal-Schiper-Toueg Algorithm

- Each message M should carry a log of
  - All other messages
  - Send causally before M's send event, and sent to the same destination dest(M)
- Log can be examined to ensure when it is safe to deliver a message
- Channels are assumed to be FIFO

Local Variables

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

The Algorithm

1. **Message Send Event**, where P_i wants to send message M to P_j:
   a. $send(M, SENT)$ to P_j (Here $SENT$ array is the log which will be used to determine if the message needs to be buffered or not)
   b. $SENT[i, j] = SENT[i, j] + 1$
2. **Message Arrival Event**, when $(M, SENT_j)$ arrives at P_i from P_j:
   a. deliver M to P_i when for each process x,

b. $DELIV_i[x] \geq SENT_j[x, i]$

c. $\forall x, y, SENT_i[x, y] = max(SENT_i[x, y], SENT_j[x, y])$

d. $DELIV_i[j] = DELIV_i[j] + 1$

Algorithm Complexity

- Space complexity at each process: $\mathcal{O}(n^2)$ integers
- Space overhead per message: n^2 integers
- Time complexity at each process for each send and deliver event: $\mathcal{O}(n^2)$

Algorithm in action

Assume following steps have occurred till now
- P_1 sent 3 messages to P_2
- P_1 sent 4 messages to P_3
- P_2 sent 5 messages to P_1
- P_2 sent 2 messages to P_3
- P_3 sent 4 messages to P_2

Variable state
Assuming $SENT$ of all P is aware of all the other values of $SENT$, $SENT$ of P_1:

```
| 0   | 3   | 4   |
| 5   | 0   | 2   |
| 0   | 4   | 0   |
```

Assume the following values for $DELIV$ arrays
$DELIV_1 = [0, 4, 0]$
$DELIV_2 = [3, 0, 4]$
$DELIV_3 = [3, 2, 0]$

MESSAGE EVENTS:
Now if, P_1 sends m_1 to P_2
1. $SEND$ event from P_1
1. $SENT_1$

```
| 0   | 3   | 4   |
| 5   | 0   | 2   |
| 0   | 4   | 0   |
```

2. Send $(m_1, SENT_1)$
3. Updated $SENT_1$

```
| 0   | 3+1 | 4   |    | 0   | 4   | 4   |
| 5   | 0   | 2   | => | 5   | 0   | 2   |
| 0   | 4   | 0   |    | 0   | 4   | 0   |
```

2. $RECEIVE\, m_1$ from P_1 at P_2
1. $DELIV_2[1] \geq SENT_1[1, 2]$
2. $DELIV_2[3] \geq SENT_1[3, 2]$
3. Deliver m_1 to P_2
4. $DELIV_2[1] = DELIV_2[1] + 1 = 4$
5. $DELIV_2 = [4, 0, 4]$
6. $SENT_2$

```
| 0   | 3   | 4   |
| 5   | 0   | 2   |
| 0   | 4   | 0   |
```

Now if, P_3 sends m_2 to P_2
1. $SEND$ event from P_3
1. $SENT_3$

```
| 0   | 3   | 4   |
| 5   | 0   | 2   |
| 0   | 4   | 0   |
```

2. Send $(m_2, SENT_3)$
3. **Updated** $SENT_3$

```
| 0   | 3   | 4   |    | 0   | 3   | 4   |
| 5   | 0   | 2   | => | 5   | 0   | 2   |
| 0   | 4+1 | 0   |    | 0   | 5   | 0   |
```

2. $RECEIVE\, m_2$ from P_3 at P_2
1. $DELIV_2[1] \geq SENT_3[1, 2]$
2. $DELIV_2[3] \geq SENT_3[3, 2]$
3. Deliver m_2 to P_2

4. $DELIV_2[3] = DELIV_2[3] + 1 = 5$
5. $DELIV_2 = [4, 0, 5]$
6. $SENT_2$

Week 5

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 28/Aug/2021

Topics Covered

1. Message Ordering (cont.)
    a. Birman-Schiper-Stephenson Protocol
        i. P_i sends a message m to P_j
        ii. P_j receives a message m to P_i
2. Distributed Mutual Exclusion
    a. Types of Approaches
    b. Quorum Based
    c. Requirements of Mutual Exclusion Algorithms
    d. Performance Metrics
    e. Lamport's Algorithm
        i. Requesting the Critical Section
        ii. Executing the Critical Section
        iii. Releasing the Critical Section
        iv. Example Problem

Message Ordering (cont.)

Birman-Schiper-Stephenson Protocol

- C_i = Vector clock of P_i
- $C_i[j]$ = j^{th} element of C_i

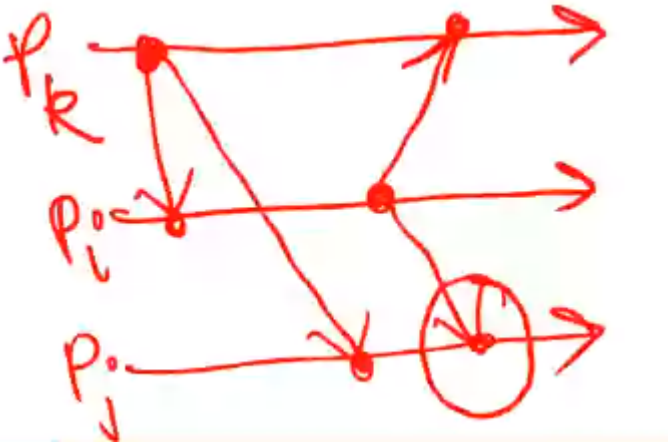
- tm = Vector timestamp for message m, stamped after local clock is incremented.
- **NOTE**, we also assume that **all messages taking part in this algorithm is a broadcast**

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

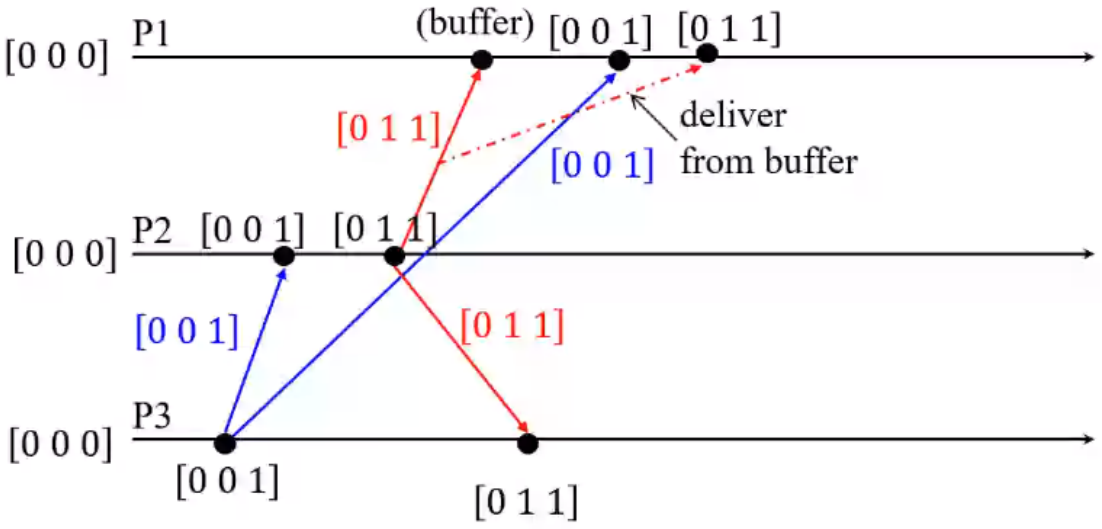
P_j receives a message m to P_i

- When $P_j (j \neq i)$ reveives m with timestamp tm, it delays message delivery until:
  - $C_j[i] = tm[i] - 1$, P_j has received all preceding messages sent by P_i
  - $\forall k \leq n$ 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



  Here we need to check if P_j also received P_k message before message from P_i is processed
- 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

Example Problem



P_3 executes a broadcast
- $C_3 = [0, 0, 1]$
- $tm_1 = [0, 0, 1]$

P_2 receives message m_1
- $C_2 = [0, 0, 0]$
- **Condition checks:**

1. $C_2[3] = tm_1[3] - 1$ is **TRUE**
2. $C_2[i] \geq tm_1[i]\ \forall i \leq n$ and $i \neq 3$ is **TRUE**

- $C_2[i] = max(C_2[i], tm_1[i]), \forall i$
- $C_2 = [0, 0, 1]$

P_2 sends message m_2
- $C_2 = [0, 1, 1]$
- $tm_2 = [0, 1, 1]$

P_3 receives message m_2
- $C_3 = [0, 0, 1]$
- **Condition checks:**

1. $C_3[2] = tm_2[2] - 1$ is **TRUE**
2. $C_3[i] \geq tm_2[i]\ \forall i \leq n$ and $i \neq 2$ is **TRUE**

- $C_3[i] = max(C_3[i], tm_2[i]), \forall i$
- $C_3 = [0, 1, 1]$

P_1 receives message m_2
- $C_1 = [0, 0, 0]$
- **Condition checks:**

1. $C_1[2] = tm_2[2] - 1$ is **TRUE**
2. $C_1[i] \geq tm_2[i]\ \forall i \leq n$ and $i \neq 2$ is **FALSE**
- m_2 is added to **BUFFER**

P_1 receives message m_1
- $C_1 = [0, 0, 0]$
- **Condition checks:**
1. $C_1[3] = tm_1[3] - 1$ is **TRUE**
2. $C_1[i] \geq tm_1[i]\ \forall i \leq n$ and $i \neq 3$ is **TRUE**
- $C_1[i] = max(C_1[i], tm_1[i]), \forall i$
- $C_1 = [0, 0, 1]$

Deliver m_2 from BUFFER to P_1
- $C_1 = [0, 0, 1]$
- **Condition checks:**
1. $C_1[2] = tm_2[2] - 1$ is **TRUE**
2. $C_1[i] \geq tm_2[i]\ \forall i \leq n$ and $i \neq 2$ is **TRUE**
- $C_1[i] = max(C_1[i], tm_1[i]), \forall i$
- $C_1 = [0, 1, 1]$

NOTE, in the exam make sure that all the above steps are also written in the paper, annotating the diagram alone will lead to getting partial marks

Distributed Mutual Exclusion

Types of Approaches

- Token based
- Assertion based
- From Recorded Lecture
- Quorum based

Quorum Based

- Each site requests permission to execute the CS from a subset of sites 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

Requirements of Mutual Exclusion Algorithms

- Safety Property: At any instant, only one process can execute the CS
- Liveness Property: A site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS
- Fairness:
  - Each process gets a fair chance to execute the CS
  - CS execution requests are executed in order of their arrival time
  - Time is determined by a logical clock

Performance Metrics

From Recorded lectures

Lamport's Algorithm

- Every site S_i keeps a queue, $request\_queue_i$
- $request\_queue_i$ contains mutual exclusion requests ordered by their timestamps
- FIFO
- CS requests are executed in increasing order of timestamps

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 $request\_queue_i$.
- When site S_j receives the $REQUEST(ts_i, i)$ message from site S_i, it places site S_j's request on $request\_queue_j$ and returns a timestamped **REPLY** message to S_i

Executing the Critical Section

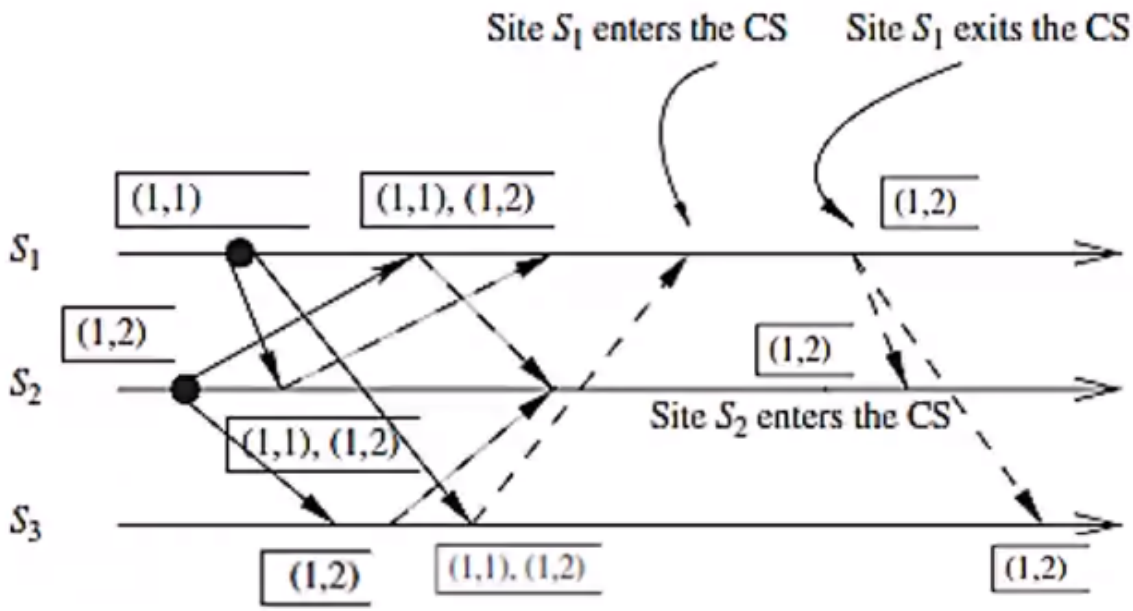
Site S_i enters the CS when the following conditions hold:
- **L1:** S_i has received a message with timestamp larger than (ts_i, i)
- **L2:** S_i's request is at the top of $request\_queue_i$

Releasing the Critical Section

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped **RELEASE** message to all other sites

- When a site S_j receives a **RELEASE** message from site S_i, it removes S_i's request from its request queue

Example Problem



1. S_1 and S_2 are requesting for the CS
2. at S_2
    - $request\_queue_2 = [(1, 2)]$
3. at S_1
    - $request\_queue_1 = [(1, 1)]$
4. S_1 Receive request for CS from S_2:
    - $request\_queue_1 = [(1, 1), (1, 2)]$ (This priority is calculated based on the index value i)
    - S_1 sends a **REPLY** to S_2
5. S_3 Receive request for CS from S_2:
    - $request\_queue_3 = [(1, 2)]$
    - S_3 sends a **REPLY** to S_2
6. S_3 Receive request for CS from S_1:
    - $request\_queue_3 = [(1, 1), (1, 2)]$
    - S_3 sends a **REPLY** to S_1
7. S_1 enters the CS

8. S_1 sends **RELEASE** message
9. S_2 removes $(1, 1)$ from $request\_queue_2$
10. S_2 enters the CS
11. S_3 removes $(1, 1)$ from $request\_queue_3$

Week 6

Lecturer: Barsha Mitra, BITS Pilani, Hyderabad Campus

BARSHA.MITRA@HYDERABAD.BITS-PILANI.AC.IN

Date: 4/Sep/2021

Topics Covered

1. Distributed Mutual Exclusion (Cont.)
    a. Ricart Agrawala Algorithm
        i. Requesting the critical section
        ii. Executing the critical section
        iii. Releasing the critical section
        iv. Example Problem
    b. Maekawa's Algorithm
        i. Problem Example
        ii. Algorithm Steps
        iii. Deadlocks
    c. Raymond's Tree Based Algorithm
        i. Example
        ii. Data Structures
        iii. Steps of algorithm
    d. Suzuki-Kasami's Broadcast Algorithm

Distributed Mutual Exclusion (Cont.)

Ricart Agrawala Algorithm

- Communication channels are not required to be FIFO

- $REQUEST$ and $REPLY$ messages
- Each process p_i maintains the request deferred array RD_i
- Size of RD_i = no. of processes in the system
- Initially, $\forall i \; \forall j : RD_i[j] = 0$
- 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 $RD_i[j] = 0$

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a timestamped $REQUEST$ message to all other sites
- 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 requests's timestamp. Otherwise, the reply is deferred and S_j sets $RD_j[i] = 1$

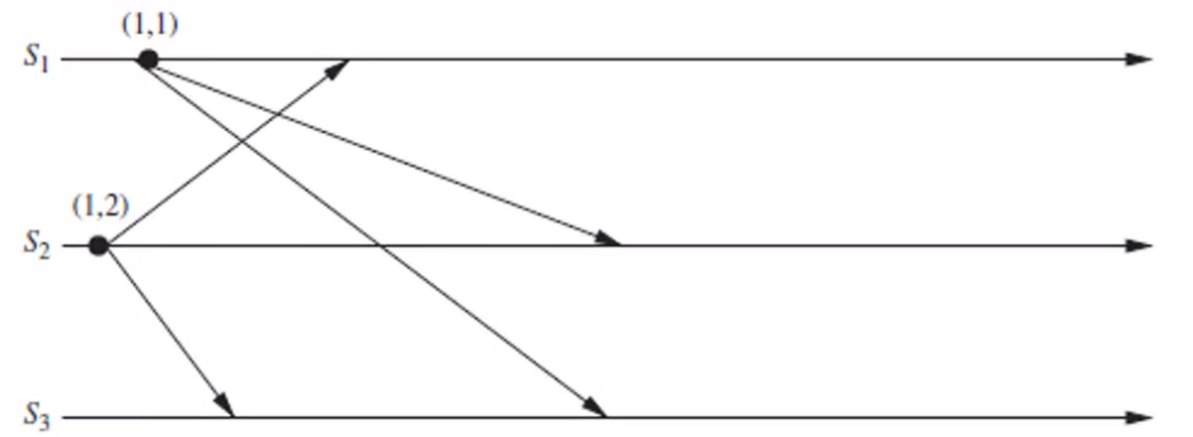
Executing the critical section

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

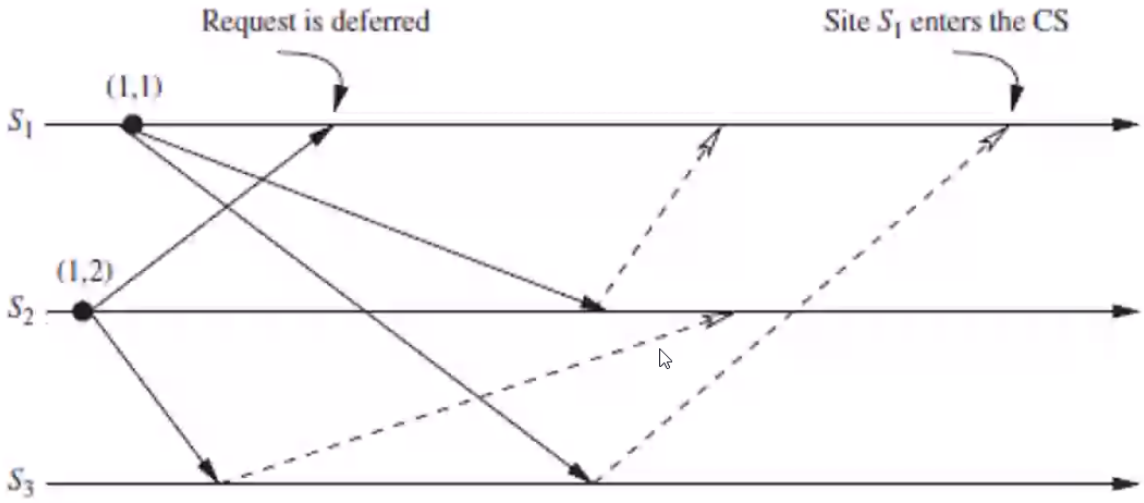
When site S_i exits the CS, it sends all the deferred REPLY messages:
$\forall \; j$ if $RD_i[j] = 1$, then S_i sends a REPLY message to S_j and sets $RD_i[j] = 0$

Example Problem



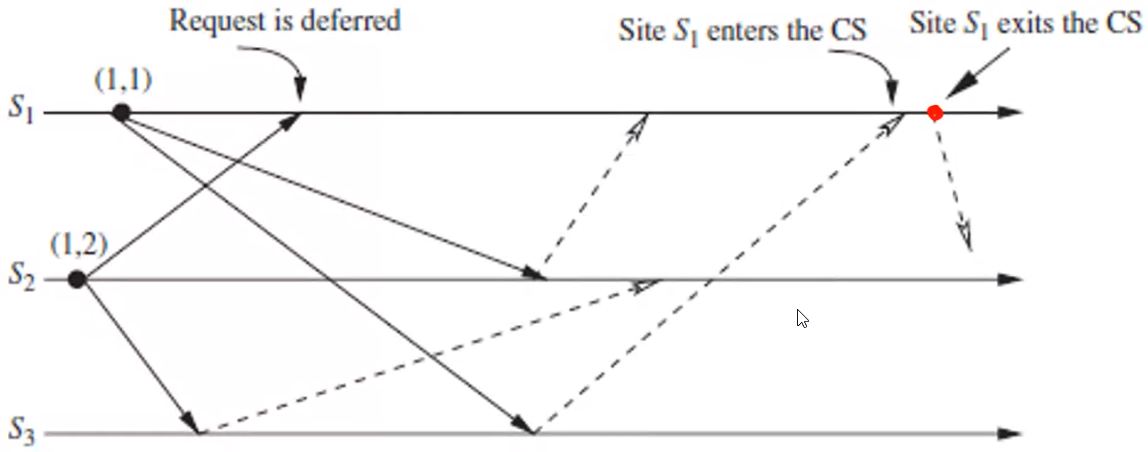
Sites S_1 and S_2 each make a request for the CS

- S_1 has timestamp $(1, 1)$
- S_2 has timestamp $(1, 2)$



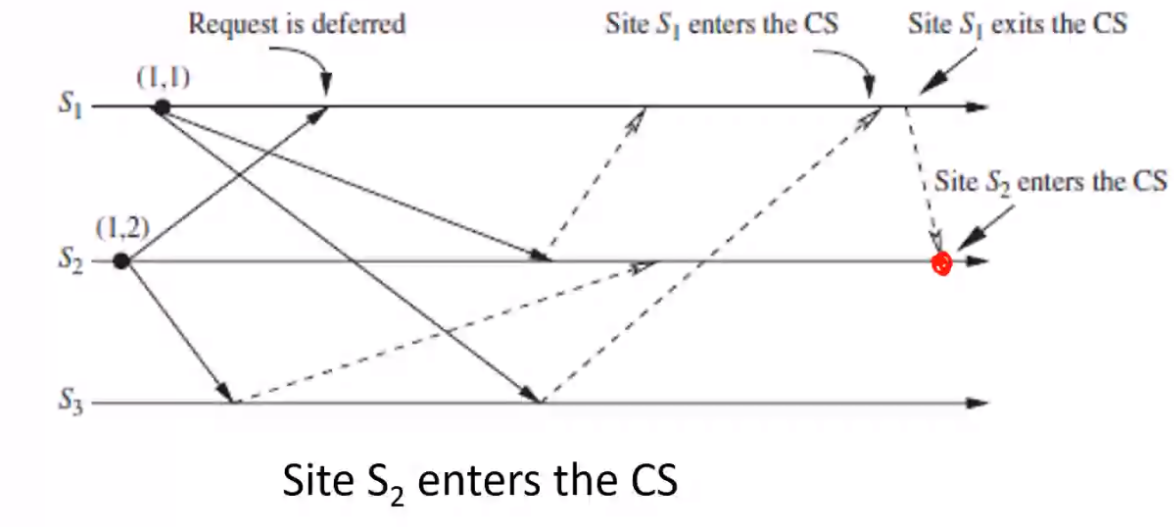
Site S_1 enters the CS

- S_3 receives S_2 request so sends $REPLY$ (dotted line) to S_2
- S_1 defers S_2 request since it needs to get CS
- S_2 sends S_1 $REPLY$ since S_1 has higher priority
- S_3 sends S_1 $REPLY$
- S_1 enters CS since it received $REPLY$ from all other sites
- At this point $RD_1 = [0, 1, 0]$



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

- S_1 finishes executing the CS
- S_1 sends $REPLY$ to S_2 deferred $REQUEST$
- At this point $RD_1 = [0, 0, 0]$



Site S_2 enters the CS

- S_2 enters the CS since S_2 has received $REPLY$ from all the sites

Performance - requires 2(N - 1) messages per CS execution

Maekawa's Algorithm

- It is a quorum based mutual exclusion algorithm (Explained here Week5DC#Quorum Based)
- Request sets for sites are constructed to satisfy the following conditionsL

    - M1: $\forall i \forall j, i \neq j, q \leq i, j \leq N :: R_i \cap R_j \neq \phi$
    - 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$ \le N$
- Maekawa showed that $N = K(K-1) + 1$
- This relation gives $|R_j| = K = \sqrt{(N)}$
- Uses $REQUEST$, $REPLY$ and $RELEASE$ messages
- Performance -> $3\sqrt{(N)}$ messages per CS invocation

Problem Example

If $N = 7$
$K = 3$
since there are 7 Sites, there will be 7 Request Sets (R_1 to R_7) each of size $K = 3$

Let us take $R_1 = S_1, S_3, S_4$, then:
- The Request Set $R_2 = S_2, S_5, S_6$ is wrong since M1 is not satisfied
- The Request Set $R3 = S_3, S_4$ is wring since M3 is not satisfied

If the following Request sets are taken:
- $R_1 = S_1, S_3, S_4$
- $R_2 = S_1, S_2, S_5$
- $R_3 = S_1, S_2, S_3$
- $R_4 = S_1, S_4, S_6$
Then, it is wrong since S_1 is in 4 places and according to M4 only K (3) number of occurrences are allowed

Algorithm Steps

From Recorded Lecture

Deadlocks

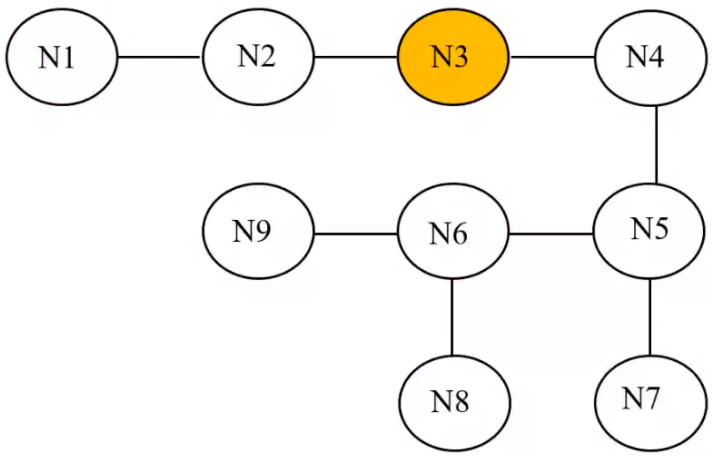
From Recorded Lecture

Raymond's Tree Based Algorithm

- Uses a spanning tree of the network
- Each node maintains a $HOLDER$ variable that provides information about the placement of the privilege in relation to the node itself

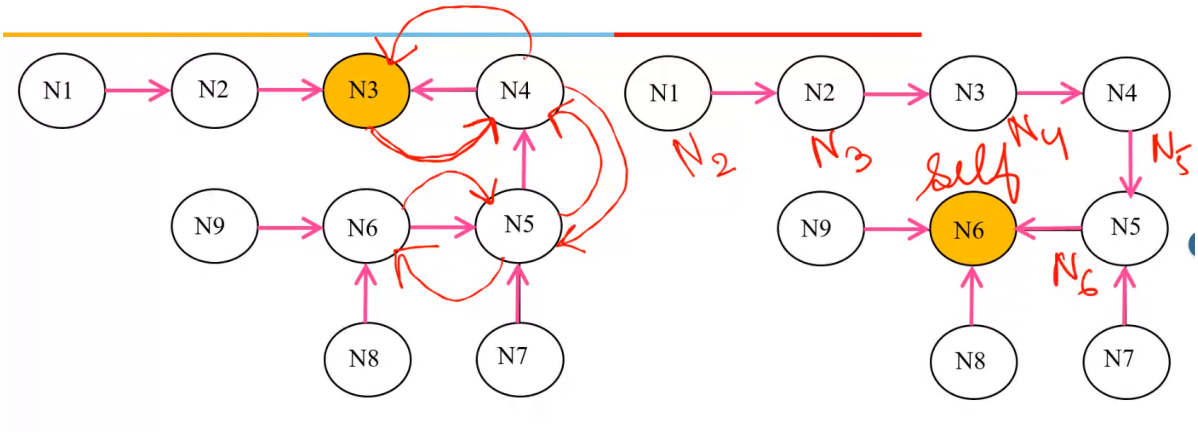
- A node stores in its $HOLDER$ variable the identity of a node that it thinks has the privilege or leads to the node having the privilege
- For 2 nodes X and Y, if $HOLDER_X = Y$ the undirected edge between X and Y can be redrawn as a directed edge from X to Y
- Node containing the privilege is also known as the root node

Example



- $\text{HOLDER}_{\text{N1}} = \text{N2}$
- $\text{HOLDER}_{\text{N2}} = \text{N3}$
- $\text{HOLDER}_{\text{N3}} = \text{self}$
- $\text{HOLDER}_{\text{N4}} = \text{N3}$
- $\text{HOLDER}_{\text{N5}} = \text{N4}$
- $\text{HOLDER}_{\text{N6}} = \text{N5}$
- $\text{HOLDER}_{\text{N7}} = \text{N5}$
- $\text{HOLDER}_{\text{N8}} = \text{N6}$
- $\text{HOLDER}_{\text{N9}} = \text{N6}$

- Messages between nodes traverse along undirected edges of the tree
- Node needs to hold information about and communicate only to its immediate neighboring nodes
- Here $N3$ holds privilege to itself



- Consider $N6$ sends a $REQUEST$, then it travels along the directed edge
- When $N3$ receives $REQUEST$ to give privilege, and it is not using it, then $N3$ will send message back to whoever sent the $REQUEST$ and that will further send it down till the node that sent the original $REQUEST$.
- When that happens the $HOLDER$ variable for each node changes it when it forwards the

$REQUEST$ back to the original node

Data Structure

- $HOLDER$
- $USING$
  - 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
- $REQUEST_Q$
  - FIFO queue that can 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 $immediateneighbors + 1$ (for "self")

Steps of algorithm

From Recorded Lectures

Suzuki-Kasami's Broadcast Algorithm

- Broadcasts a $REQUEST$ message for the token to all other sites
- 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
- Distinguish outdated and current REQUEST messages
- $REQUEST(j, sn)$ where sn is a sequence number that indicates that S_j is requesting its sn^{th} CS execution
- S_i keeps an array of integers $RN_i[1, \ldots, n]$ where $RN_i[i]$ is the largest sequence number received in a $REQUEST$ message so far from S_j

- When S_i receives a $REQUEST(j, sn)$ message, it sets $RN_i[j] = max(RN_i[j], sn)$
- When S_i receives a $REQUEST(j, sn)$ message, the request is outdated if $RN_i[j] > sn$

- sites with outstanding requests for the CS are determined in the following manner:
  - 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 S_j executed most recently
  - after executing its CS, a S_i updates LN[i] = RN_i[i]
  - token array LN[1, … ,n] permits a site to determine if a site has an outstanding request for the CS
  - at S_i, if $RN_i[j] = LN[j] + 1$, then 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 i.d.'s in queue Q if these i.d.'s are not already present in Q
  - finally, the site sends the token to the site whose i.d. is at the head of Q

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

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

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

Releasing the critical section: Having finished the execution of