06/02/2023

Module - 1

Syllabus

Distributed systems - definition, relation to computer systems components, motivation, primitives for distributed communication, design issues, challenges and applications, a model of distributed computation, distributed programs, model of distributed execution, model of communication networks, global state of distributed systems, cuts of distributed computations, past and future cones, consistent global states, models of process communication.

⇒ Distributed Systems.

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.

A distributed system can be characterized as a collection of mostly autonomous processes communicating over a communication network and have following features:

* No common physical clock
* No shared memory

* Geographical Separation
* Autonomy and heterogeneity.

→ Distributed Computation

Field of computer science that deals with the study of distributed system is called as distributed computation.

Relation to Computer System Components.

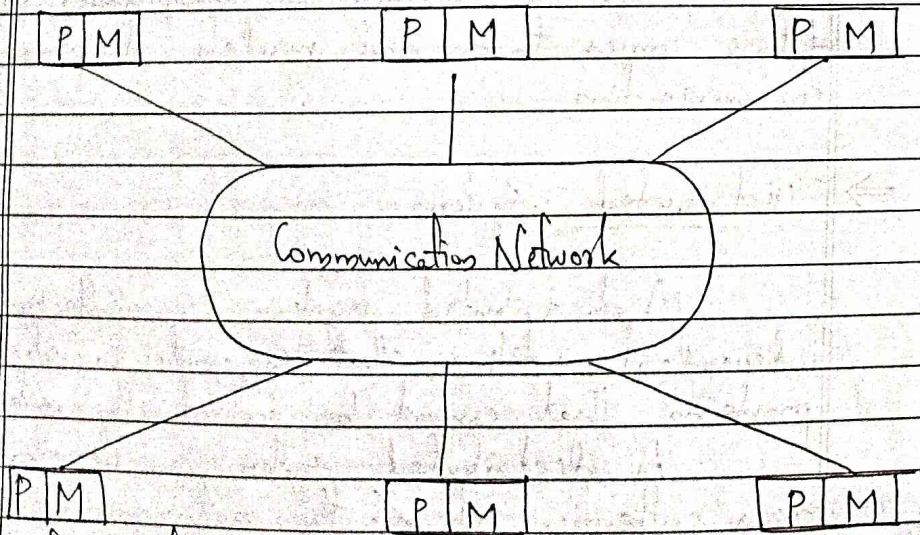


figure shows typical distributed systems.
P - Processor
M - Memory.

* Each computer has a memory processing unit and computers are connected by communication network.
* A distributed software is also termed as middleware.
* A distributed execution is a execution of processes across the distributed systems to collaboratively achieve a common goal.
* An execution is also termed as computation or a run.
* Distributed systems uses layered architecture to breakdows complexity of systems designs. The middleware is the distributed software that drive distributed system.

Fig: Interaction of software components at each process

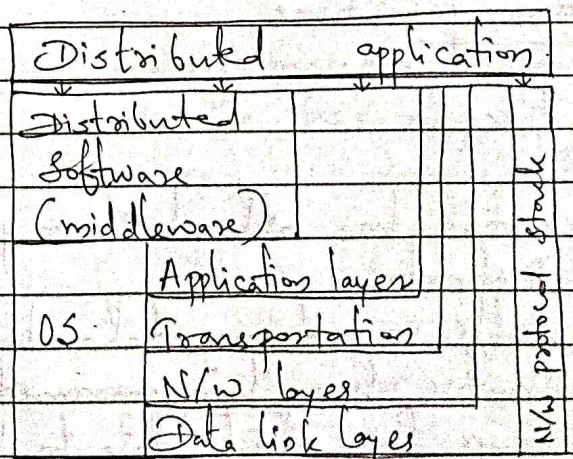


Fig: shows the interaction of this software here systems components at each processor

→ Motivation

* Inherently distributed computing. eg:- money transfer
* Resource sharing
* Access to geographically remote data and resources.
* Enhance reliability
* Increased performance / cost ratio
* Scalability
* Modularity and incremental expandability.

→ Primitives for distributive computation

Blocking / Non-blocking, Synchronous / Asynchronous primitives.

→ Communication primitives.

Sent and Receive

* Sent primitive has atleast two parameters:
  - Destination
  - Buffer - containing data to be sent.
* Receive primitive has two parameters:
  - Source from which data to be received
  - We use a buffer into which data is to be received.

* Two ways of sending data when sent primitive is imported.

* Buffered option - copies data from users buffer to kernel buffer then later copies from kernel buffer onto network.
* Unbuffered option - data is copied directly from user buffer onto the network.

7/02/23

→ Synchronous Primitive

A sent or receive primitive is synchronous if both sent and receive handshake with each other. Processing for sent primitive complete only after invoking processor learns that other corresponding receive primitive has also be invoked and receive operation has been completed. The processing for receive primitive completes when data to be received is copied into receivers user buffer.

Asynchronous Primitive

A sent primitive is said to be asynchronous

if control returns back to invoking process after data item to be sent has been copied out of user specified buffer.

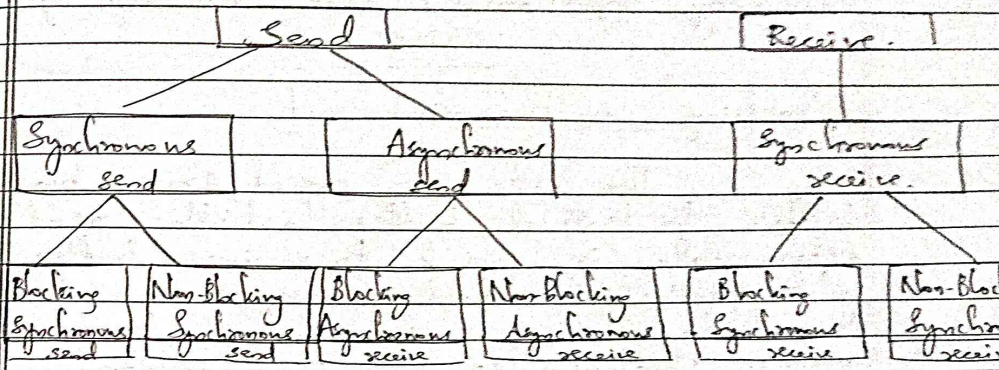
→ Blocking primitive

A primitive is blocking if control returns to invoking process after processing for primitive complete.

→ Non-Blocking primitive

A primitive is non-blocking if control returns back to invoking process immediately after invocation even though operation has not completed. For non-blocking sent control returned process even before data is copied out of user buffer. For non-blocking receive control returned process even before data may have arrived from centre.
For non-blocking primitive, a returns parameter on the primitive calls returns a system generated handle which can be later used to check status of the completion of call. The process can check for the completion of call in two ways:-

(i) It can keep checking if the handle has been flagged or posted
(ii) It can issue a ~~wait~~ wait with a list of handles as parameters.



operation.

W- process may issue wait to check completion of non-blocking operation.

8/02/2023

Blocking synchronous send, Blocking synchronous receive.

Blocking synchronous send

The data is copied from blocking buffer to kernel buffer and is then send over n/w. After data is copied to receiver buffer system and receive call has been issued. An acknowledgement is back to sender causes control to return to the process that invoke the send operation and complete the send.

Blocking receive

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

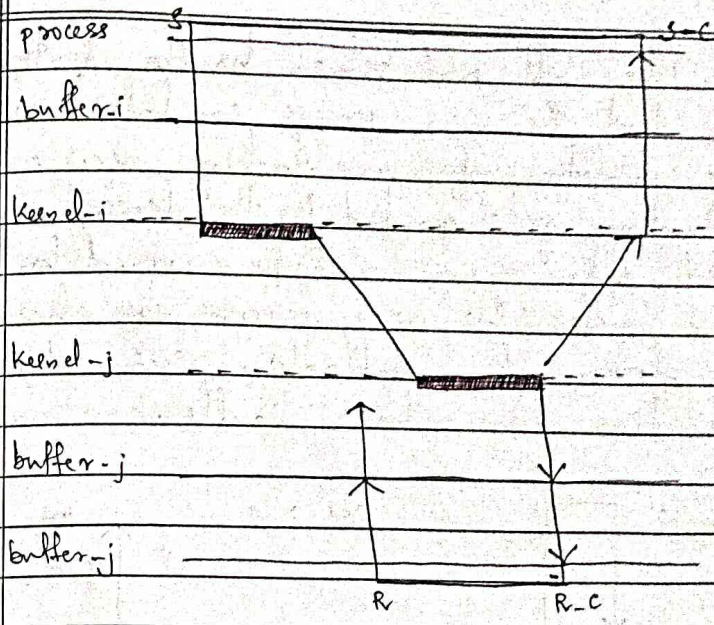


fig:- Blocking synchronous send, Blocking synchronous receive

→ Non-blocking synchronous send, non-blocking receive

Non-blocking synchronous send.

Control returns back to invoking process as soon as copy of data from user buffer to kernel buffer is initiated

Non-blocking receive

The receive call will cause the kernel to

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

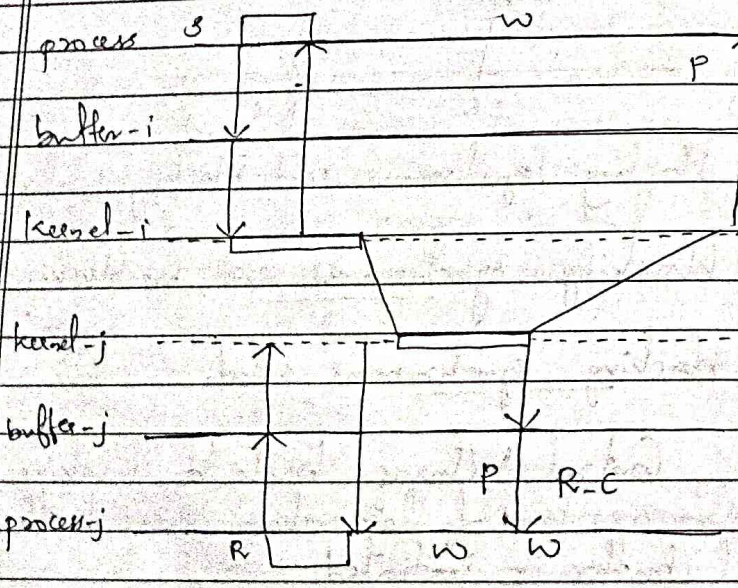


fig :- Non-Blocking Asynchronous Send,
Non-Blocking Asynchronous Receive.

13/02/23.

Blocking asynchronous send

The user process that invokes the send is blocked until the data is copied from user's buffer to kernel buffer. For unbuffered option, the user process that invoke the send is blocked until the data is copied from user's buffer to network.

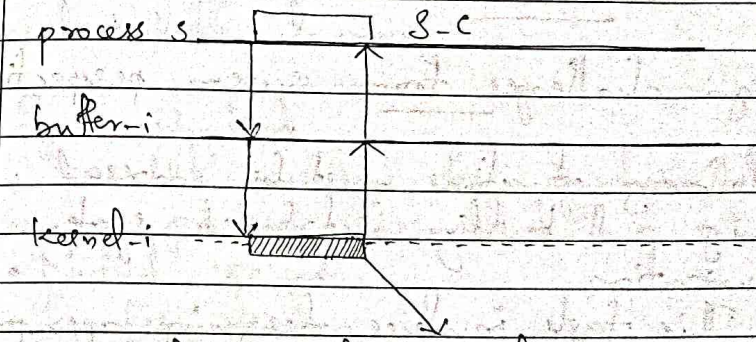
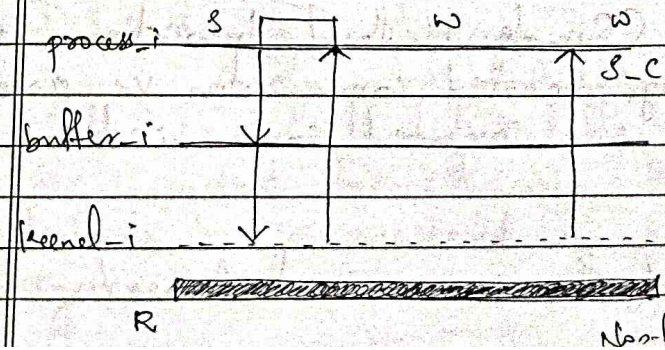


fig :- Blocking asynchronous send.



Non-blocking asynchronous send

The user process that invokes the send is blocked until transfer of data from user buffer to kernel buffer is initiated (for unbuffered block until transfer from user buffer to network is initiated). Control returns to user process as soon as this transfer is initiated. The asynchronous send completes when data has been copied out of user buffer.

→ Design issues and challenges.

Design challenges from systems perspective.

The following issues must be addressed when designing and building distributed system.

* Communication
This task involves designing appropriate mechanisms for communication among processors in network. Some example mechanisms are :-
- RPC (Remote
- RMI (Remote Method Invocation
- Message oriented communication vs access oriented communication.

* Processes
Some of the issues involved are

- Management of processes and threads at client/server.
- Code migration.
- Design of software and mobile engine

* Naming.
Devising easy to use and robust scheme for names, identifiers and addresses is essential for locating resources and processes in a transparent and scalable manner.

* Synchronization
Mechanisms for synchronization or coordination among processes are essential.
- Mutual exclusion
- Leader election
- Synchronizing physical clocks
- Devising logical clocks etc

* Data storage and access
Schemes for data storage and access data in a fast and scalable manner across network are important for efficiency. File systems designs have to be re-considered in setting of distributed systems.

* Consistency and replication.

To provide fast access of data and to provide scalability replication of data is highly desirable. This leads to issues of managing replicas and dealing with consistency among replicas/caches in distributed setting.

* Fault Tolerence

Fault tolerence requires mainly is correct and efficient operation in case of failure in links, nodes and processors. Some mechanisms to provide fault tolerence are:

- Process resilience
- Reliable communication
- Distributed commit
- Checkpointing and recovery

* Security

Distributed systems security involves various aspect of cryptography, secure channels, access control, key management etc.

* API and transparency

API for communication and other specialized services is important for ease of use and wider

adoption of distributed system services by users. Transparency hides implementation policies from user. Different types of transparencies:-

- Access transparency

It hides difference in data representation on different system.

- Location transparency

Makes location of resources transparent to user

- Migration transparency.

Allows relocating resources without changing names.

- Relocation transparency

Ability to relocate resources as they are being accessed.

- Replication transparency

Doesnot let user become aware of a replication.

- Concurrency transparency

Deals with masking concurrent use of shared resources from the user.

- Failure transparency

Allowing user and application programs to complete their task despite the failure of hardware or software component

* Scalability and modularity
A system is described as scalable if it will remain effective when there is a significant increase in number of resources and number of users. Various techniques such as
- Replication
- Caching
- Asynchronous processing
- Helps to achieve scalability.

14/02/23.
→ A model of distributed computation

Distributed program

A distributed program consist of set of n asynchronous processors. $P_1, P_2 \dots P_n$ that communicate by message passing over communication n/w. We assume each process runs on different processor. Processor doesn't share global memory. c_{ij} denote channel from process P_i to process P_j, m_{ij} denote message send by P_i to P_j. These process don't share global memory. Communication delay is finite. Process execution and message transfers are asynchronous. The global state of distributed computation consist of state of channel of computation channel.

→ A model of distributed execution

Execution of process consist of sequential execution its actions. The actions are atomic and actions of process are modelled as 3 types of events.
(i) Internal event : It changes the state of process which occurs.
(ii) Message send event : A send event changes the st of process that sends the message and state of channel on which message is send.
(iii) Message receive event : A receive event changes the state of process that receives the message and state of channel on which message is received.

Let e_i^x denote x^{th} event at process P_i then send(m) and receive(m) denotes send and receive events. The occurence of events changes state of respective proces and channels.
The events at a process are linearly ordered by their order of occurence. The execution of process P_i produces a set of events $e_i^1, e_i^2 \dots, e_i^x$ and it is denoted

by Hi

$H_i = (h_i \rightarrow i)$

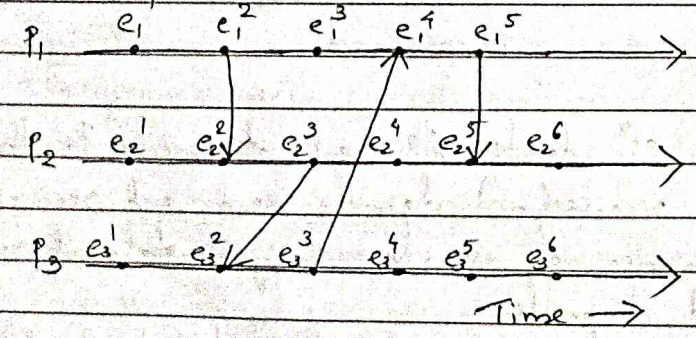
h_i - set of events produced by P_i

$\rightarrow i$ - binary relation defines linear order on these events, relation express causal dependencies events of P_i.

For every message m that is exchanged between 2 process, we have

$send(m) \rightarrow msg, rec(m)$

Relation $\rightarrow$ msg defines causal dependencies between pairs of corresponding send and receive event. The figure shows space-time diagram of distributed execution involving 3 process.



For any 2 event, $e_i \rightarrow e_j$ denotes the fact that event e_j is directly or transitively event e_j.

All information available at e_i is potentially access at e_j. For any 2 events, $e_i \not\rightarrow e_j$, event e_j do not directly or transitively depended on event e_i.

$e_j \not\rightarrow e_i$.

These events e_i and e_j is concurrent and the relation is denoted by $e_i \| e_j$.

15/02/23.

Models of communication network

There are several models of the service provided by communication network.

(i) FIFO Model

In FIFO model, each channel act as first in first out message queue and thus message ordering is preserved.

(ii) Non-FIFO Model

In Non-FIFO model, channel act like a set in which setre process act messages and receive process and remove messages in random model.

(iii) Causal Ordering

This model is based on Lamport's happens before relation. A system that support causal ordering model satisfies the following properties.

For any two messages m_{ij} and m_{kj} if

send $(m_{ij}) \longrightarrow$ send (m_{kj})

then

rec $(m_{ij}) \longrightarrow$ rec (m_{kj})

This property ensures that those causally related messages destined to same destination are consistent.

CO $\subset$ FIFO $\subset$ Non FIFO

→ Models of process communication.

Two basic models of process communication.

(i) Synchronous communication model.

It is a blocking type where on message sent, the sender process blocks until the message has been received by reciver process. The sender process receives execution only after it learns that reciver process has accepted the message. Thus sender and receiver must synchronize to exchange message. The synchronous is simpler to handle and implement. poor performance

(ii) Asynchronous communication model.

* It is a non-blocking type.
* Sender and receiver dont synchronize to exchage message. After sending a message, the sender process doesnt wait for message to be delivered to receive a process. The message is buffered by the system and is delivered to receiver process when it is ready to accept a message.
* A buffer overflow may occur. It provides high parallelism.
* It requires more complex buffer management.
* Difficult to design, verify and implement distributed algorithm for asynchronous communication.

→ Cuts of distributed computing

In the space-time diagrams, of distributed

computation zigzag line joining 1 arbitrary point on each process line is termed as cut in computation. Such line slices space-time diagram and thus set of events in distributed computation into ~~past and future~~ PAST and FUTURE.

The PAST contains all events to left of cut and FUTURE contains all events of rights of cut.

Every cut corresponds to a global state and every global state can be graphically represented as cut in computation space-time digram.

→ Two types of cuts

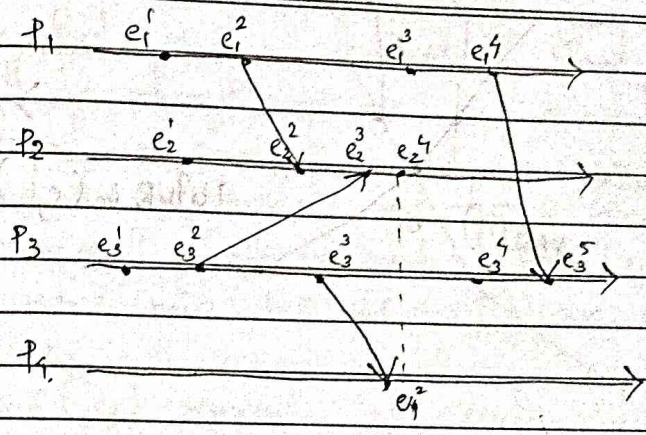
(i) Consistent cuts

A consistent global state corresponds to a cut in which every message received in the PAST of cut was send in the PAST of that cut.

(ii) Inconsistent cut.

All messages that cross the cut from PAST to FUTURE are in consistent global state.

If a message whose the crosses the cut from FUTURE to PAST



20/2/23

→ PAST and FUTURE cones of an Event.

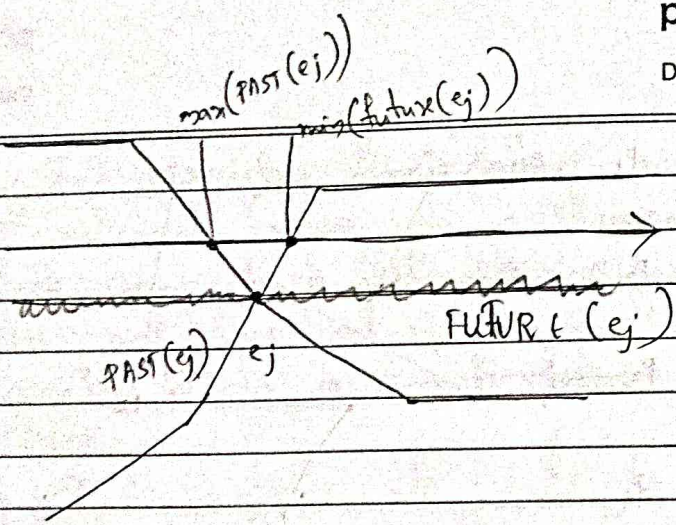
In a distributed computation an event e_i would have been affected only by all events e_j such that

$$e_j \rightarrow e_i$$

and all information available at e_j could be made accessible at e_i. All such events e_j belong to PAST of e_i.

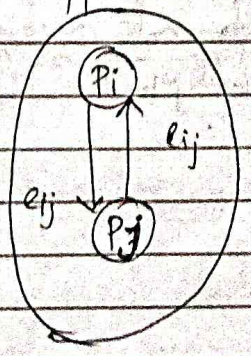
PAST (e_i) denote all events in the PAST of e_i in the computation $(H, \rightarrow)$ Then

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



Let PAST i (ej) we set of all those events of PAST(ej) that are on process Pi. Maximum element is denoted by max(PAST i(ej)).
It is the latest event at process Pi affected event ej. It will be always message send event. It is a consistent cut. It is referred as surface of PAST cone of ej. PAST(ej) represents all events on PAST light cone that effect ej.
FUTURE of an event ej denoted by FUTURE(ej) contains all events ei that are causally affected by ej.
In a computation, (H, →). FUTURE ej is defined as
FUTURE(ej) = { ei | ∀ei, ∈ H, ej → ei }
FUTURE i (ej) is set of those events of FUTURE(ej) that are on process Pi and min(FUTURE i(ej))

is the first event on process Pi that is affected by ej. It is always message receive event.
It denotes a consistent cut in computation. It is referred to as surface of future cone. All events at process Pi that is occured after max(PAST i (ej)) but before min(FUTURE i (ej)) are concurrent with ej.
FUTURE(ej) represents all events in the FUTURE light cone that is affected by ej.
Set H - PAST(ej) - future(ej) are concurrent with each other.
Global state of distributed system is a collection of local state of its components namely processes and communication channel.
The state of process at any time is defined by contents of processor registers, process stack, local memory immense on local context of distributed application.



global state.

State of channel is given by set of messages in transit to the channel. The occurrence of event change state of respective processes and channels thus causes transition in global system state.
Let LS_i^x denote state of process P_i after the occurrence of event e_i^x and before event e_i^{x+1}
LS_i^0 - initial state of process P_i
$send(m) \leq LS_i^x$
$rec(m) \notin LS_i^x$
$\exists y \ 1 \leq y < x$
$e_i^y = send(m)$
$\forall y \ 1 \leq y \leq x \quad e_i^y \neq rec(m)$
$SC_{ij}^{x,y}$ denote state of channel C_{ij}
$SC_{ij}^{x,y} = \{ m_{ij} \mid send(m_{ij}) \leq LS_i^x \wedge rec(m_{ij}) \leq LS_j^y \}$

State of channel depends upon state of processes connects. Thus channel state $SC_{ij}^{x,y}$ denotes all messages that P_i send up to event e_i^x and which process P_j had not received until event e_j^y
$GS = \{ \cup_i LS_i, \cup_{j,k} SC_{jk}^{-yj, zk} \}$

27/2/23

MODULE - 2

→ Leader Election Algorithm

Global state and snapshot recording algorithm

The global state of distributed system is a collection of local state and its components.
Recording of global state of distributed system is an important paradigm and it find applications in several aspect of distributed system design.
for example, in detection of stable properties such as dead lock and termination.
Global state of system is examined for certain properties.
i) for failure recovery
ii) for debugging distributed software
Absence of shared memory necessitates way of getting coherent and complete view of system based on local state of individual process.

→ Issues in recording global state

* If a global physical clock available, the following simple procedure could be used to record a consistent global

snapshot of distributed system. In this the initiator of the snapshot collection decides a future time at which snapshot is to be taken and broadcast this time to every processes. All process take their local snapshots at that instance in the global time.

A global physical clock is not available in distributed system and the following two issues need to be addressed in recording of a consistent global snapshot of distributed system.

(i) How to distinguish the msgs to be recorded in the snapshot (in a channel state/process state) from those not to be recorded.

(ii) How to determine the instant when a process takes it's snapshot.

Snapshot Algorithm for FIFO channels

Chandy-Lamport Algorithm

* This algorithm uses a control message call a marker. After a site has recorded it's snapshot it sends a marker along all of its outgoing channels before sending out any more messages. Since

channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot (channel state/process state) from those not to be recorded in the snapshot.

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_i

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_i.

Election algorithms choose a process from group of processor to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted.

Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then this number is send to every active process in the distributed system.

We have two election algorithms for two different configurations of distributed systems.

1. The Bully Algorithm
2. The Ring Algorithm.

* Distributed algorithms require one process to act as a coordinator or initiator. To decide which process becomes the coordinator different types of algorithms are used.
* Election algorithms are meant for electing a coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.
* The goal of an election algorithm is to ensure that when an election starts it concludes with all the processes agreeing on who the coordinator should be.
* Therefore, whenever initiated, an election algorithm basically finds out which of the currently active processes has the highest priority number and then informs this to all other active processes.

→ Bully Algorithm

* This algorithm was proposed by Garcia-Molina.
* When the process notices that the coordinator is no longer responding to requests, it initiates an election.

A process, P, holds an election as follows:

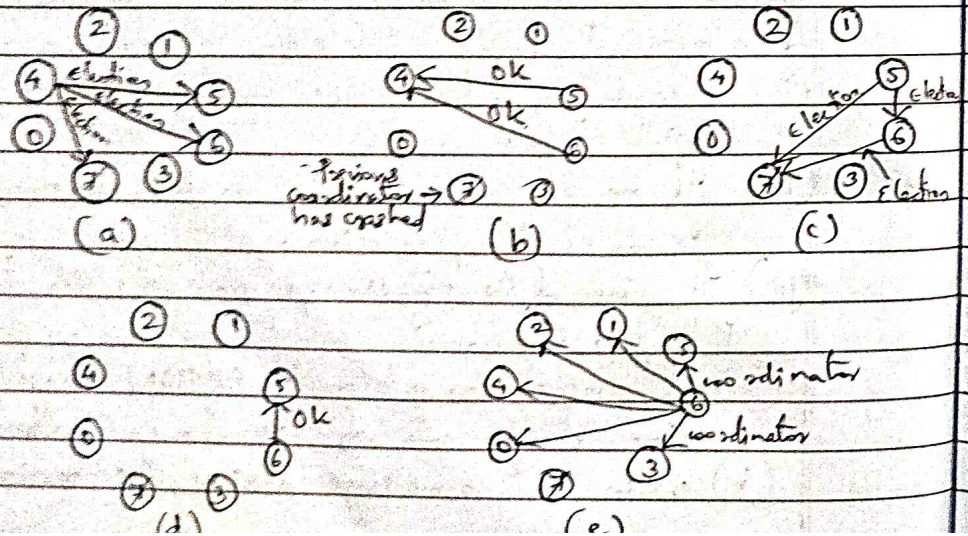
(i) P sends an ELECTION message to all processes with higher number.
(ii) If no one responds, P wins the election and becomes the coordinator
(iii) If one of the higher-ups answers, it takes over. P's job is done.

(a) A process can get an ELECTION message at any time from one of its lower numbered colleagues.
(b) When such a message arrives, the receiver sends an

OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one.

(c) All processes give up except one that is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

(d) If a process that was previously down comes back up it holds an election. If it happens to be the highest numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm".



→ Ring Algorithm

* This algorithm uses a ring but its election but does not use any tokens. In this algorithm it is assumed that the processes are physically or logically ordered so that each processor knows its successor.

1) When any process notices that a coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down the sender skips over the successor and goes to the next member along the ring until a process is located.

11/3/23

A ring based election algorithm

Each process Pi has a communication channel to the next process in the ring P(i+1) |N|. All messages are send clockwise around the ring. The goal of this algorithm is to elect a single process called the coordinator. Initially

1. Every process is marked as a non-participant in an election. Any process can begin an election, the

starting process marks itself as a participant and place its identifier in a message to its neighbour.

2. A process receives a message and compare it with it's own. If the arrived identifier is larger, it passes on the message.

3. If arrived identifier is smaller and receiver is not a participant, substitute its own identifier in the message and forward it. If doesn't forward the message if it is already a participant

4. On forwarding of any case, the process marks itself as a participant.

5. If the received identifier is that of receiver itself then this process identifier must be the greatest and it becomes the coordinator.

6. The coordinator marks itself as non-participant set elected; and sends the elected message to its neighbour enclosing its id.

7. When a process receives elected message, it marks itself as non-participant and sets its variable elected; and forwards the message.

→ Termination Detection

A fundamental problem in distributed system is to be determined if a distributed computation has terminated

A Distributed computation is considered to be globally terminated if every process is locally terminated and there is no message in transmit b/w any processes.

A locally terminated state is a state in which process has finished its computation and will not restart any action unless it receives a message.

Two distributed computations taking place in distributed systems are

1. Underlined computation :- basic messages
2. Termination detection algorithm - control messages

A termination detection algorithm must ensure the following.

(i) Execution of termination detection algorithm can't indefinitely delay underline computation, ie execution of termination detection algorithm must not freeze underline algorithm.

(ii) Termination detection algorithm must not require addition of new communication channels b/w processors.

→ System model of distributed computing

* Communication is asynchronous.
* Messages send over same communication channel may not obey FIFO ordering.
* A distributed computation has following characteristics
(a) Process can be in one or two stages states
(i) Active (Busy) - where it is doing local computation.
(ii) Idle (Passive) - where process has temporarily finish execution of local computation and reactivated only only on the receipt of ~~memory~~ message from another process.

(b) An active process can become idle at any time (complete local computation & processed all received messages).

(c) An idle process can become active only on the receipt of message from another process.

(d) Only active processes can send messages.

(e) A message can be received by a process when the process is in active or idle state.

(f) The sending of a message and receipt of a message occur as atomic actions.

Assignment topic

6/3/23

Vector time.

⇒ Logical time

→ A framework for a system of logical clocks.

* A system of logical clock consist of the time domain $T and logical clock, C.
* Elements of $T form a partially ordered set over a relation ≤, this relation is usually called happen before or causal ordering.
* Distributed system may have no physically synchronous global clock so a logical clock allows global ordering on events from different processes in such systems.
* The logical clock C is a function that maps an event E in a distributed system to an element in time domain T, denoted as C(E) and called time stamp

of e and is defined as
$C : H \mapsto T$
such that following property is satisfied.
For two events e_i and e_j
$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$, this is called clock consistency condition.
when T and C satisfy following condition for two events, $e_i \neq e_j$
$e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$
The system of clock is said to be strongly consistent.

→ Implementing logical clock

It requires addressing two issues.
(i) Data structure local to every process to represent logical time.
(ii) Protocol to update data structure to ensure consistency condition.
Each process P_i maintains data structure that allow following two capabilities.
(1) A local logical clock denoted by LC_i that helps process P_i measure it's own progress.
(2) A logical global clock denoted by GC_i that is representation of Process P_i's local view of logical global time. This allows the process to assign consistent time stamp to it's local events.
The protocol consist of following rules:
R_1: This rule governs how local logical clock is updated by a process when it executes an event [send, receive or internal]
R_2: This rule governs how process updates it's global logical clock to update its view of global time and global progress.

Scalar time

This representation was proposed by Lamport as an attempt to totally order events in distributed system.
Time domain in this representation is the set of non-negative integers.
Rules R_1 & R_2 to update clocks are as follows
R_1: Before executing an event [send, receive, internal] Process P_i executes the following.
$C_i = C_i + d \quad (d > 0)$
Every time R_1 is executed d can have different value & this value may be application dependent
R_2: Each message piggybacks the clock value of its sender at sending time. When a

process P_i receive a message with time stamp C_{msg} message it executes the following actions.

1. $C_i = \max(C_i, C_{msg})$
2. execute R_i
3. Deliver the message.

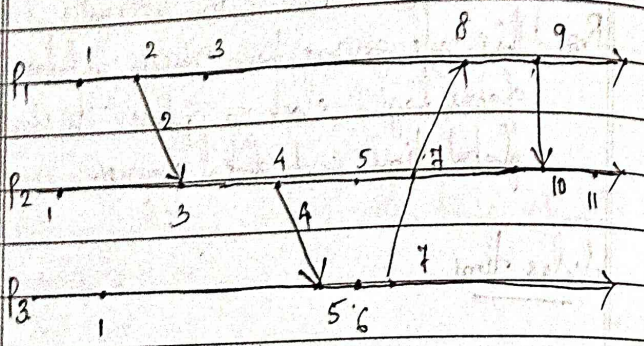


fig: shows the evolution of scalar time with $d=1$.

→ Termination detection by weight throwing

* In termination detection by weight throwing, a process called controlling agent monitors the computation.
* A 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.
* The weight at each process is zero and the weight at the controlling agent is 1.
* The computation starts when the controlling agent sends a basic message to one of the processes.
* The process becomes active and the computation starts.
* A non-zero weight $W(0<W\leq 1)$ is assigned to each process in the active state and to each message in transit in the following manner.
* 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.
* Thus, the 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.
* The controlling agent concludes termination if its weight becomes 1.

Notation.

* The weight on the controlling agent and a process is in general represented by W.
* B(DW): A basic message B is sent as a part of the computation, where DW is the weight assigned to it.
* C(DW): A control message C is sent from a process

to the controlling agent where DW is the weight assigned to it.

Formal description

The algorithm is defined by the following four rules.

Rule 1: The controlling agent or an active process may send a basic message to one of the processes, say P, by splitting its weight W into W_1 and W_2 such that $W_1 + W_2 = W$, $W_1 > 0$ and $W_2 > 0$. It then assigns its weight $W := W_1$ and sends a basic message $B(DW := W_2)$ to P.

Rule 2: On the receipt of the message B(DW), process P adds DW to its weight W ($W := W + DW$). If the receiving process is in the idle state, it becomes active.

Rule 3: A process switches from the active state to the idle state at any time by sending a control message $C(DW := W)$ to the controlling agent and making its weight $W := 0$.

Rule 4: On the receipt of a message C(DW), the controlling agent adds DW to its weight ($W := W + DW$). If $W = 1$, then it concludes that the computation has terminated.

8/3/22

→ A spanning tree based termination detection algorithm

The algorithm description.

The algorithm work as follows:

1. Initially, each leaf process is provided with a token. The set S is used for book-keeping to know which processes have the tokens. Here S will be the set of all leaves in the tree.
2. Initially all processes and tokens are white. As explained above, coloring helps the root know if a message-passing was involved in one of the subtrees.
3. When a leaf node terminates, it sends the token it holds to its parent process.
4. A parent process will collect the tokens sent by each of its children. After it has received a token from all of its children and after it has terminated, the parent process sends a token to its parent.
5. A process turns black when it sends a message to some other process. This is coloring scheme helps a process remember that it has sent a message

When a process terminates, if it is black, it sends a black token to its parent.

6. A black process turns back to white after it has sent a black token to its parent.

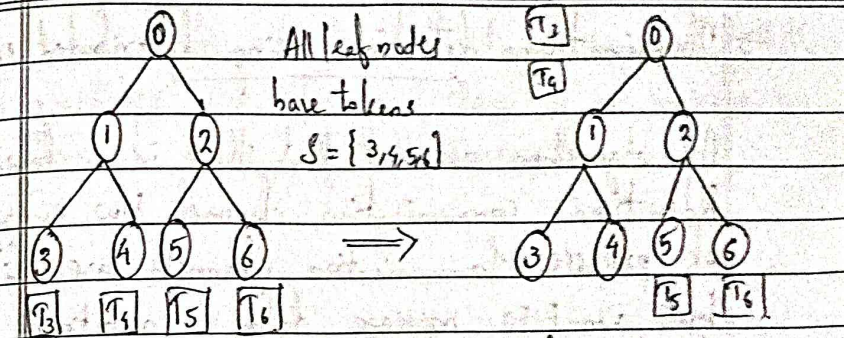
7. A parent process, holding a black token (from one of its children), sends only a black token to its parent, to indicate that a message-passing was involved in its subtree.

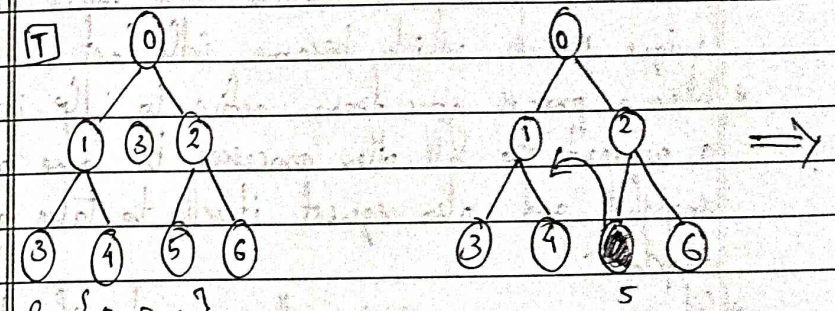
8. Tokens are propagated to the root in this fashion. The root, upon receiving a black token, will know that a process in the tree had sent a message to some other process. Hence, it restarts the algorithm by sending a Repeat signal to all its children.

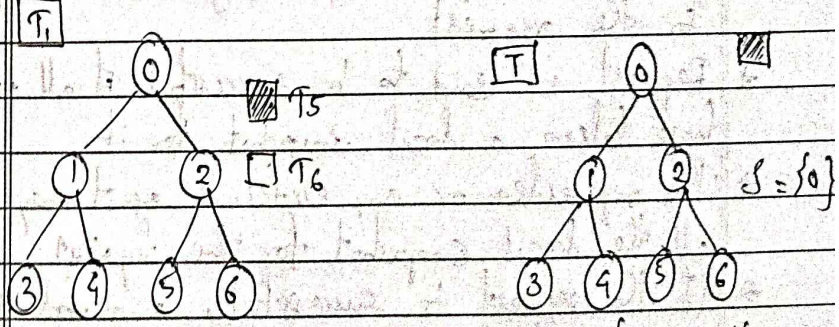
9. Each child of the root propagates the Repeat signal to each of its children and so on, until the signal reaches the leaves.

10. The leaf nodes restart the algorithm on receiving the Repeat signal.

11. The root concludes that termination has occurred if:
(a) it is white;
(b) it is idle; and
(c) it has received a white token from each of its children.







→ Termination detection using distributed snapshot

The algorithm assumes that there is a logical bidirectional communication channel b/w every pair of processes. Communication channels are reliable but non-FIFO message delay is arbitrary but finite.

The main idea behind the algorithm is as follows:

* When a computation terminates there must exist a unique process which became idle last.
* When a process goes from active to idle it issues a request to all other processes to take a local snapshot and also request itself to take a local snapshot.
* When a process receives the request if it agrees that requester becomes idle before itself it grants the request by taking a local snapshot for the request
* Request is said to be successful if all processes have taken a local snapshot for it
* The requester or any external agent may collect all the local snapshot for the request.
* If the request is successful, a global snapshot can be of the request thus be obtained and the recorded state will indicate termination of the

communication. In the recorded snapshot all the processors are idle and there is no message in transit to any of the message.

13/3/23

Module - 3

→ Distributed Mutual Exclusion Algorithm.

* Mutual exclusion is a distributed system state that only one process is allowed to execute critical section at any given time.
* Message passing is the means for implementing distributed mutual exclusion.
* Three basic approaches for implementing distributed mutual exclusion.
  - Token based approach
  - Non-token based approach
  - Quorum based approach.

→ System Model

* N sites $S_1, S_2 \ldots S_n$
* Single process running on each site, the process at site S_i denoted by P_i. Processes communicate asynchronously.
* Process send request message to enter critical section and wait for reply before next entering critical section.
* A site can be in any of the three state.

(i) Requesting the critical section - site block and cannot make further request
(ii) Executing critical section (CS)
(iii) Neither requesting nor executing CS (NCE) - Site is executing outside CS.

* At any instant site may have several pending request for CS, site queues up this request and serves them one at a time. No assumptions regarding communication channel if they are FIFO or not.

Assume channels reliably deliver messages, do not crash and network doesn't get partitioned

→ Requirements of mutual exclusion algorithm

* It should satisfy the following properties
  (a) Safety property - At any instant only one process can execute CS
  (b) Liveness property - This property states absence of deadlock & starvation
  (c) Fairness - each process must get a fair chance to execute CS.

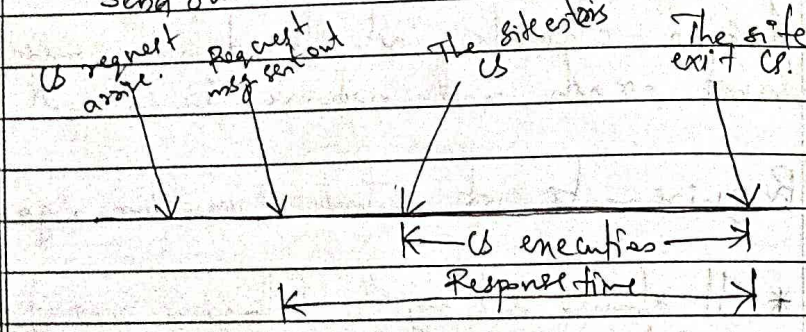
→ Performance Matrix

Four matrix

(i) Message Complexity - No. of messages that are required per critical section execution by a site.

(ii) Synchronization delay - After a site leaves the critical section, it is the time required before next site enters critical section.

(iii) Response time - This is the time interval a request wait for its critical section execution to be over after its request messages have been send out.



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

This is the rate at which system execute request for critical section.

SD - Synchronization Delay.
E - Average Critical Section execution time

⇒ Lamport's algorithm for mutual exclusion (Non token based algorithm)

* Requesting the critical section
- When a site S_i wants to enter the CS, it broadcast a REQUEST (ts_i, i) message to all other site 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.

* Executing the critical section
Site S_i enter 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 request-queue

* 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.

→ Ricart-Agrawala algorithm

If

→ Ricart-Agrawala algorithm (Non-token based)

* If a process p_i that is waiting to execute the critical section 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.

* Each process p_i maintains the request-deferred array, RDi, the size of which is the same as the number of processes in the system.
* Initially, $\forall i \forall j : RDi[j] = 0$.

* 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 $RDj[i] := 1$.

* 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 sends a REPLY message to S_j and sets $RD_i[j] := 0$.

14/3/23

Quorum Based Mutual Algorithm

* This algorithm reduce msg complexity of invoking mutual exclusion by having sites as permission from only a subset of site

→ ## Maekawa's Algorithm

It was the first quorum based mutual exclusion algorithms.
This algorithm requires delivery of message to be tested in order they are send b/w every pair of sites.

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.

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~~ deletes that entry from the queue. If the queue is empty, then the site updates its state of reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

15/3/23

Quorums or subset of each process.

$P_1 = (3, 5)$ // process 3 and 5 are in quorum of P_1

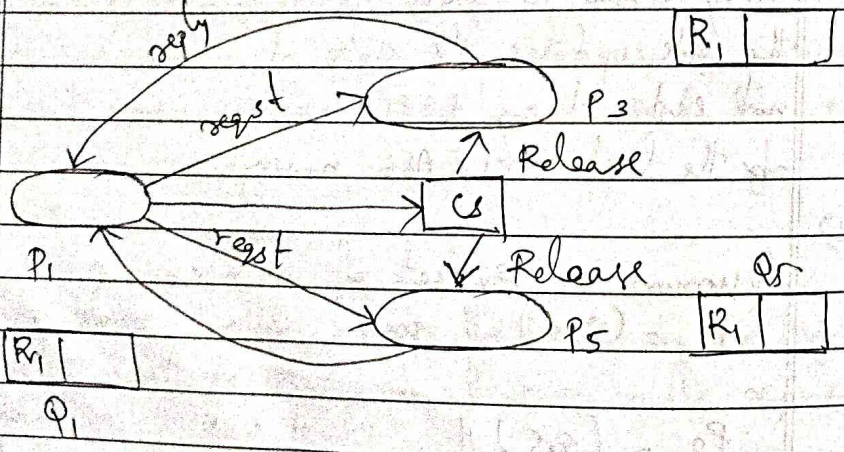
$P_2 = (4, 5)$ // " 4 & 5 " of P_2

$P_3 = (2, 3, 4)$ // " 2, 3 & 4 " of P_3

$P_4 = (3, 4)$
$P_5 = (2, 3)$
$P_6 = (1, 3)$

→ Rules

1) The intersection of two group ≠ Null
ie, atleast any one process should be presented in both subsets.
2) A process can be part of any group
3) A single process cannot be considered as a group.
4) If a process want to access CS then it should send request msg to all members in its quorum & then it should receive reply from all members in quorum.



→ Drawback

* Chance of deadlock
* Requests are not prioritized by timestamp.

→ Token based algorithm

Suzuki - Kasami's broadcast algorithm.

* Requesting the CS

(a) If requesting the site S_i doesn't have token then it increments its sequence number $RN_i[i]$ and sends a REQUEST(i, s_n) msg to all the sites ("s_n" is updated value of $RN_i[i]$).
(b) When a site S_j receives this msg it sets $RN_j[i]$ to max $(RN_j[i], s_n)$. If S_j has the idle token, then it sends the token to S_i if $RN_j[i] > LN[i] + 1$.

* Executing the CS

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