Distributed Systems

Leader Election Algorithm

definition
process to determine a manager or a coordinator among a set of process on distributed networks

Assumptions:
- Each process knows all other process
- Each P has unique identifier
- P with highest identifier is chosen as leader

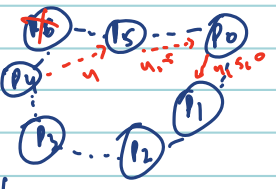
When to elect?
- when system initialises or fails
- process gets no response for some time

Reason to elect leader (Motivation)
- Process synchronisation (who performs what)
- fault tolerance (system Recovery)
- Resource Allocation
- Load Balancing (by distrib tasks by leader)

Algorithms

Logical Ring Algorithm

interconnected nodes arranged in ring structure



Steps:
- P6 fails
- P4 observes failure
- P4 passes id 4 to 5
- P5 passes 4,5 to P0
- P0 passes 4,5,0 to P1
- P1 passes 4,5,0,1 to P2
- P2 passes 4,5,0,1,2 to P3
- P3 passes 4,5,0,1,2,3 to P4

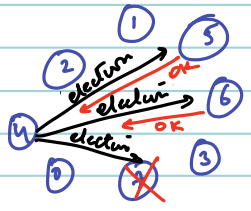
(Ring msg passing)

→ P4 receives message back (4,5,0,1,2,3) and knows the ring is traversed

→ P4 picks highest id in the list and sends coordinator message to all the process in ring 1 by 1 (Ring message passing)

- ↑ latency, slower Election ✗
- Simple implementation ✓

Bully Algorithm



Steps: Process P (eg 4) detects failure of leader (eg 7)
- P_4 sends 'ELECTION' message to all the Process with ↑ id
  - → If no response then P_4 declares itself as leader
  - → else, (response from higher ID)
- P > P_4 send OK msg & take over election process
- Now P who sent OK, sent election msg to ↑er IDs and this occurs until no ↑er ID exists
- Process that receives no OK message is declared itself leader
- Sends coordinator msg to all P
- all P recognise the new leader

→ Quicker than Logical Ring ✓

→ High message overhead ✗

Tree Algorithm
- process arranged in tree structure

2

Physical vs **Logical**

- measure real time
- Requires periodic sync with time server or GPS to ensure accuracy across DS
- Ideal for real time scenarios; logging timestamp in DB

- provides sequence of events in system w/o real clock
- Using lamport clock (Inc count), which don't require real time sync.
- Ideal for message passing where event order matters

In any system, clock synchronisation is important ie (all nodes should have same time), which can be affected by Network delays, varying clock speed, lack of central authority

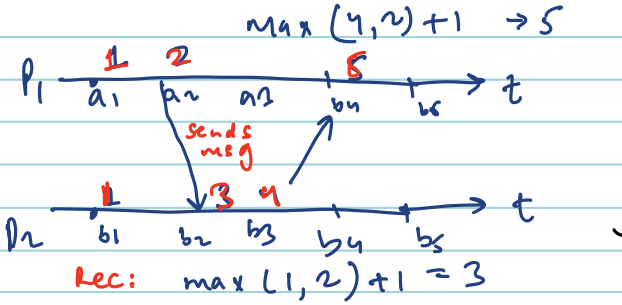
Time in Distributed Systems

Problems: Clock synch. fault Tolerance consistency Event Ordering

Lamport clock (Capture happens before Relation)

maintains logical clock.

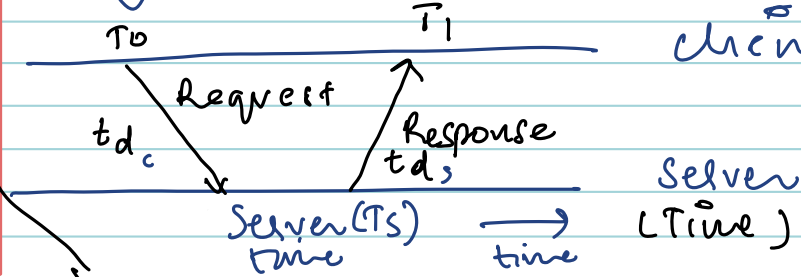
update (max (per, sender) +1)

- Each process maintains its local counter (Lamport clock)
- clock updates during internal event, message Sending & recieving msg [attaches personal timestamp to msg)
- 

even though P1 & P2 might not have synchronised clock, the logical clock allows them to agree on the order of events.

Cristian Algorithm (Synchronising clocks in DS)

On dist system clock synchronisation is needed and done using a central time server

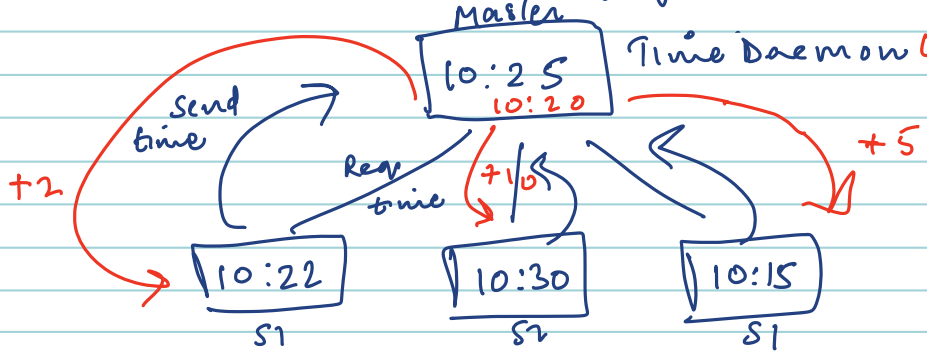


Symm assumption $td_c = td_s$

$$T_{new} = T_s + \frac{(T_1 - T_0)}{2}$$

Berkeley clock :

(Clock synchronisation method in DS to synchronise clocks of all nodes in Network. w/o Relying on central time server



Master → Polls (Req)
↓
Slave → sends time personal
↓
Calculate wtavg w/o delays
↓
Master sends adjustment value to slave
↓
each clock adjust time accordingly

(3)

Requirements → Safety (Only 1 P in CS at a time), Liveness (Stable absence of deadlock & starvation), fairness (each P has fair chance to access critical sec)

Mutual Exclusion

Definition: Concept that ensures that at a time only one process access the critical section of code on shared resource, to avoid race conditions.

Critical section

Shared Resource
file, memory, printer

P accesses resource in serial way

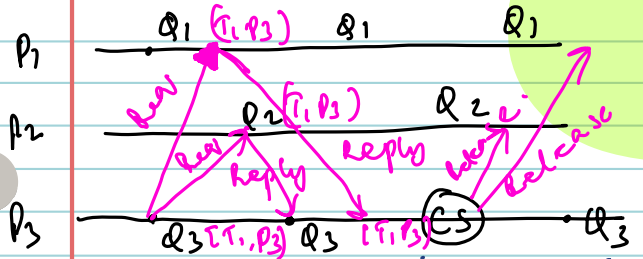
To achieve ME

permission based approach

Lamport Algo for Mutual Exclusion

3 Messages
- Request
- Reply
- Release

(Fifo order)



each process maintain logical clock

each process maintains Queue to enter CS

Request → Timestamp sent to all P. Increment LC when send. Process added to local queue (send)

Reply → Acknowledges with reply (All P) add req to local queues in fifo order

Enter critical section:
- P with least TS enter CS
- After recieving reply from all other P.

Exit CS: Send release msg to all P. Remove req from local msg stamps

3(N-1) msg / critical section

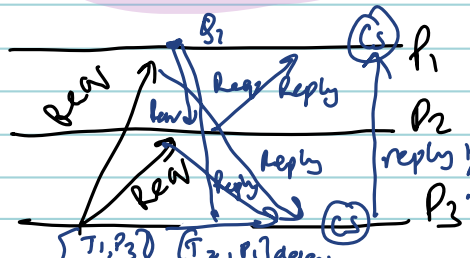
Ricart Agarwala Algo for Mutual Exclusion

By Glenn Ricart & Ashok Agarwala

extension of Lamport Algo.

follows permission based approach

2 messages → Request, Reply



Request: Send req to all P to access CS

Reply: Only when P is
- P is not using CS
- Not req CS, & has timestamp less than the requested Proc

To use CS: Only when all send reply

To release CS: Send reply to previous def req which requested CS

2(N-1) msg / critical section

Deadlock Avoiding condition

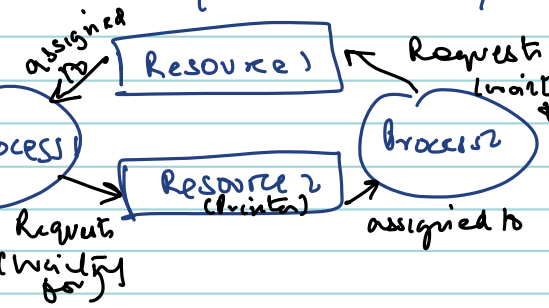
all conditions should be true for deadlock

1. Mutual Exclusion
2. Hold & wait
3. No preemption
4. Circular wait

COFFMAN conditions

"No progress"

definition: occurs when set of P each hold some resources & request access to other resources and none of them can proceed



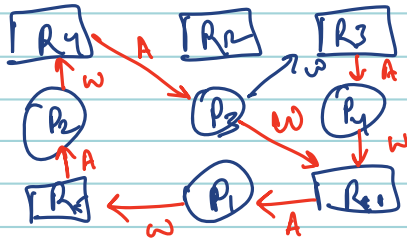
Deadlock

Deadlock Detection & Recovery

Involves identifying & resolving situations where multiple process are waiting indefinitely for resources held by other resulting in Deadlock.

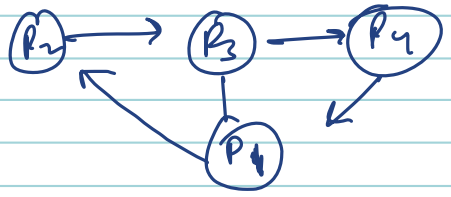
Example:
- Wait for Graph (WFG)
- Chandy Misra Algo.
- Resource Allocation Graph (RAG)

RAG
Resource Allocation Graph (Predicts)
"shows state of the system"



cyclic predicts the deadlock

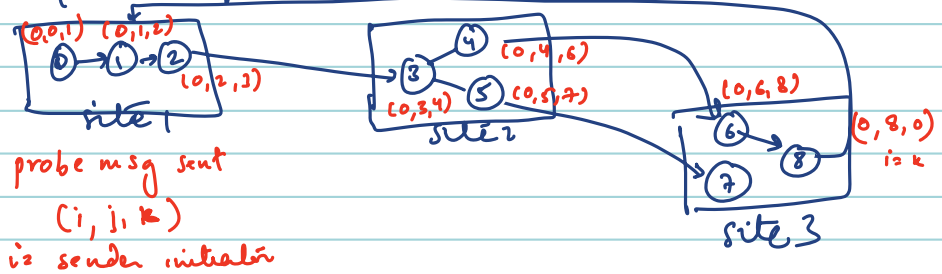
Wait for Graph (ensures) Deadlock
Same as RAG, but only process not Resources
Represents dependencies b/w processes



Chandy Misra Haas Algo: Edge chasing probe based problem

PROBE used (for distributed systems).

If process requests for resource and fails or times out, then it sends probe msg to all the processes holding one more of its requested resource



probe msg sent
(i, j, k)
i = sender initially
j = sender
k = receiver
i = k → deadlock,

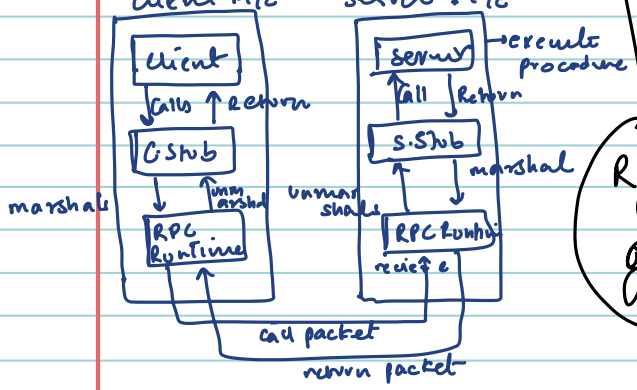
if i = k then deadlock

Solution: abort the process to break deadlock

RPC

Remote Procedure Call

Principle: protocol enables a program to execute functions on remote server as if it is run on its own.



RPC Runtime: manages transmission of msgs b/w n/w across client & server.

RPC & gRPC

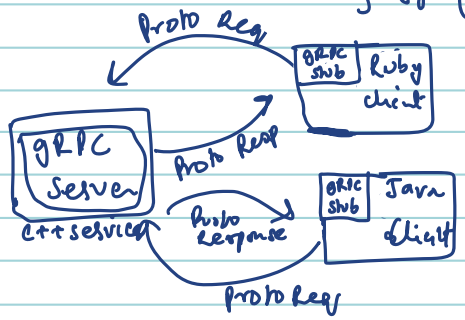
gRPC

google Remote Procedure Call

HTTP/2 Protobufs

• framework for implementing RPC with HTTP/2

Features
- high performance [uses HTTP/2] [uses Protobuf]
- Bi-directional streaming
- Cross language support
- Strongly Typed [define services & messages types using Protobuf] Can be automatically converted to multiple languages ensuring strong typing.



stub → serialises the request using protobuf & sends to gRPC server

RPC v/s REST

REST - Representational state Transfer

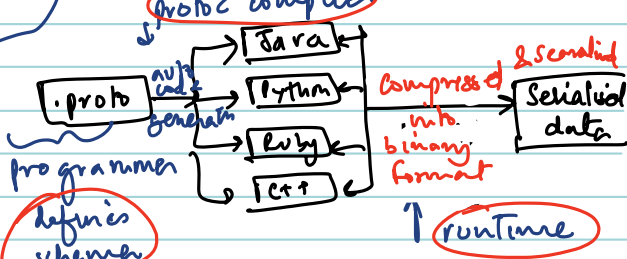
RPC	REST
• Procedure based approach	• Resource based Approach using HTTP methods
• Stateful for long lived connections	• stateless each request has all the necessary info & is not dependent ∴ concurrent process
• encode HTTP 4xx, 5xx	• Defined specifically

- HTTP/2 instead of HTTP/1
- Protobuf instead of XML or JSON

HTTP 2: Allows real time communication & improved n/w efficiency

- bidirectional streaming (msg 2 way, no wait, real time)
- multiplexing (single TCP connection [many messages] instead of opening several connection for each req.)

Protobuffers: Use Binary data representation enabling data packets to transport faster than other formats like JSON/XML



gRPC Services: Types

Unary : C1 : S1 (authentication/auth)
Server Streaming : C1 : Sn (notifications)
Client Streaming : Cn : S1 (video upload)
Bidir Streaming : Cn : Sn (Chat)

service abc {
rpc ServerStreaming calls (req msg) returns (stream Response) }

example: code:

```
syntax = "proto3";
message RequestMessage {
        string name = 1;
        int32 id = 2;
    }
message ResponseMessage {
        string marks = 3;
    }
service state {
    rpc Unarycall (RequestMessage) returns (ResponseMessage);
    }
```

	RPC	gRPC
1	Use various protocols (HTTP, TCP) etc	(1) Uses HTTP/2 for comm^n
2	support unary calls [streaming support	(2) support 4 types of services Unary server streaming client streaming Bi-directional
3	For serialisation can use formats such as - XML, JSON	(3) for serialisat^n, uses protobuf
4	Manual code Generat^n	(4) Automatic code gen