Mutual Exclusion

- Fundamentals of distributed systems is concurrency and collaboration.
- Simultaneous access to the same resources
- To prevent concurrent access, corrupt the resources or make it inconsistent
- Distributed mutual exclusion algorithms can be classified into two different categories:
  - → Token based → avoid starvation, deadlocks, lost of token (process holding it is crashed)
  - → Permission based

Critical Section → Piece of code for which we need to ensure there is atmost one process executing it at any point of time

Each process can call three functions
- Enter: To enter the critical section
- AccessResource: To run the critical section
- Exit: To exit the critical section

Semaphores
→ An integer that can only be accessed via two special functions: wait(S), signal(S)

In distributed systems we cannot share variables like semaphores
DSystems → Process communicates by exchanging messages
Need to guarantee 3 properties:
- Safety - At most one process executes in CS at any time
- Liveness - Every request for a CS is granted eventually
- Ordering - Requests are granted in the order they were made

In permission based approach → Centralized Algorithm, Decentralized Algorithm, Distributed Algorithm

Centralized Algorithm

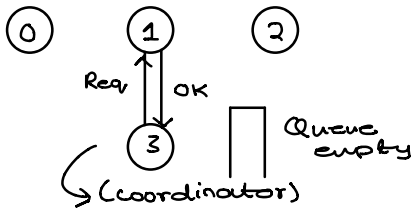
System model → Each pair of processes is connected by reliable channels
- → Messages are eventually delivered to recipient and in FIFO order
- → Process do not fail

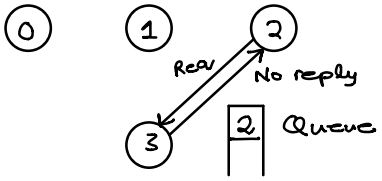
Central solution

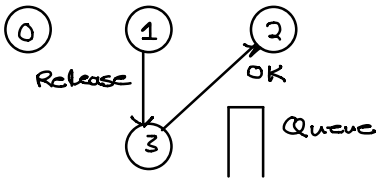
- Select a central master or leader

Master keeps
- → A queue of waiting requests from processes who wish to enter/access the CS
- → A special token which allow its holder to access CS

Actions of any process in group
- enter()
  - send a request to master
  - wait for token from master
- exit()
  - send back token to master







Bandwidth: The total number of messages sent in each enter and exit operation

Entry → 2 msgs, Exit → 1 msg

Synchronization Delay: The time interval between one process exiting the critical section and the next process entering it (when there is only one process waiting)

- 2 msgs

Master → single point of failure, bottleneck

Unable to detect dead coordinator

Token Ring Algorithm

Logical ring, each process is assigned a position in the ring

ordering does not matter

Each process must know who is next in the line after itself

Acquire token and use shared resources, else forward

- only one process has the token, starvation cannot occur
- Token lost, regenerated token is difficult to detect

Bandwidth per entry: 1 msg by requesting process, N msgs throughout system.

per exit: 1 msg

Synchronization delay → best case enter() process is successor of process in exit()

Worst case → is predecessor

Distributed Algorithm – Ricart-Agrawala's Algorithm

- To access a shared resource, the process builds a message containing the name of resource, its process number and its current logical time. Then it sends the msg to all processes

Three cases for replies:

1) If receiver already has access to the resource, it simply does not reply. Instead, it queues the request
2) If receiver is not accessing the resource and does not want to access the resource, it sends back OK msg to sender
3) If the receiver wants to access the resource but has not yet done so, it compares the timestamp of incoming msg with the one contained in msg that it has sent everyone. The lowest one wins. Sends an OK if incoming timestamp is lower.

Msg include <Pi, Ti, R>

Enter CS at Pi

- Set state to wanted
- Multicast Request <Ti, Pi, R> to all processes
- Change state to held and enter CS

On receipt of Request $\langle T_j, P_j, R \rangle$ at P_j

If state == held
- Add request to queue

If state == wanted && $(T_j, j) < (T_i, i)$
- Add request to queue

Else
- Send Reply

Issues

- Number of msgs required per entry: $2(n-1)$ more traffic
- No single point of failure, if a process crashes, no reply, interpreted as denial of permission
- Slow & performance bottleneck

Decentralized Algorithm

- Resource is assumed to be replicated n times
- Each replica has its own coordinator
- Process wants to access the resource, needs to get majority of votes: $m > n$, where m is # of votes
- If process gets less than m votes, back off for a randomly chosen time

Issues

- Coordinators are not maintaining any history, in case of reset or failure they can vote to different process
- for violation required k coordinator to reset
- Many nodes want to access resource, no will get m votes
- Starvation

Election Algorithms

- Many algorithms allow/require one process to act as coordinator, or perform a special fn.

→ Algorithms for coordinator selection

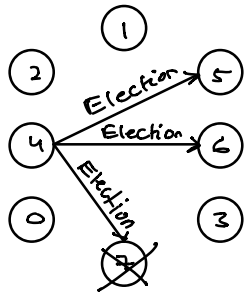
1) The Bully Algorithm

Process notices that coordinator is no longer responding to requests

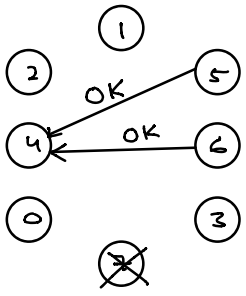
It initiates an election as follows
- P sends an election msg to all processes with higher number
- If no one responds, P wins the election & becomes coordinator
- If one of higher_ups answer, it takes over. P's job is done

The biggest guy in town always win - bully algorithm

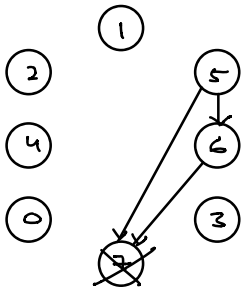
i)



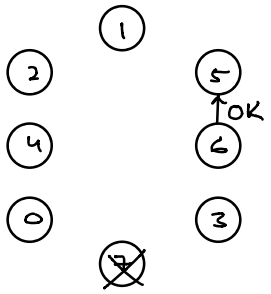
ii)



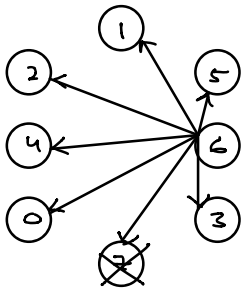
iii)



iv)



v)

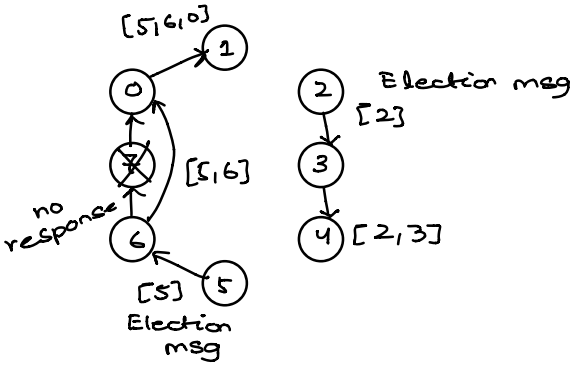


- 4 holds election
- 5 & 6 respond
- 4 stops, 5 & 6 hold election, 6 responds
- 6 tells 5 to stop and tells everyone that it is the coordinator (v)

7 → prev. coordinator and is crashed

2) Token Ring Algorithm

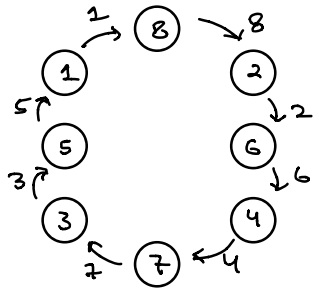
- The processes are arranged in an ring.
- When any process notices that coordinator is not responding, it builds an election msg containing its own process nbr & sends it to its successor.
- At each step successor adds its own nbr to list and makes itself a candidate
- The msg reaches the process that started this and checks for highest nbr and that becomes the coordinator
- Then msg is changed to coordinator and is sent to circulate the ring so everyone knows who is the coordinator.



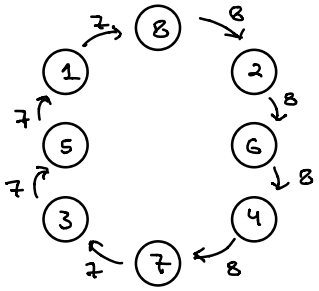
- Can have two elections at once like 5 & 2.

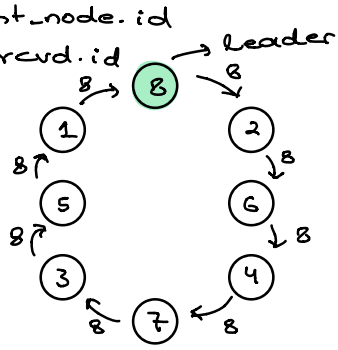
3) Chang_Roberts Algorithm

- Every process sends an election msg with its id to left if it has not seen a msg with higher id
- Forward any msg with an id greater than own id to left
- If a process receives own msg, it is the leader
- It is uniform: Number of processes need not be known to the algorithm



- if msg_rcvd.id > current_node.id then forward msg_rcvd.id



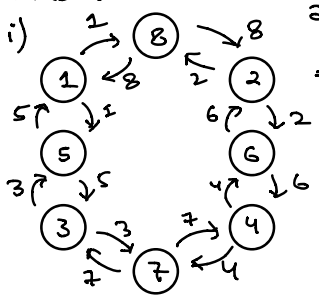


4) **Hirschberg-Sinclair Algorithm**

- Assume ring is bidirectional
- Carry out elections on increasingly larger sets
- Pi is a leader in phase $r = 0, 1, 2, \ldots$ iff it has the largest node id of all nodes that are at a distance 2^r or less from it; to establish that it sends probing msg to both sides

Phase 1:

i)

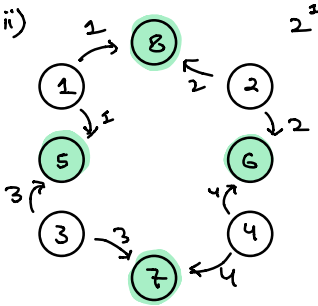


$2^1 = 2$ (adjacent neighbour)

$\Rightarrow$ if rcvd.id > current.id send OK

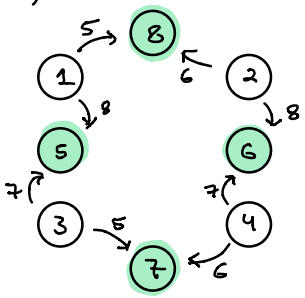
If node receives both replies:
it becomes temporary leader & next phase starts

ii)

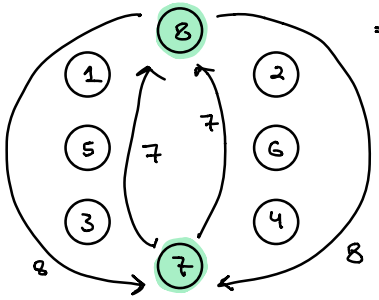


$2^2 = 2$ (neighborhood)

iii)



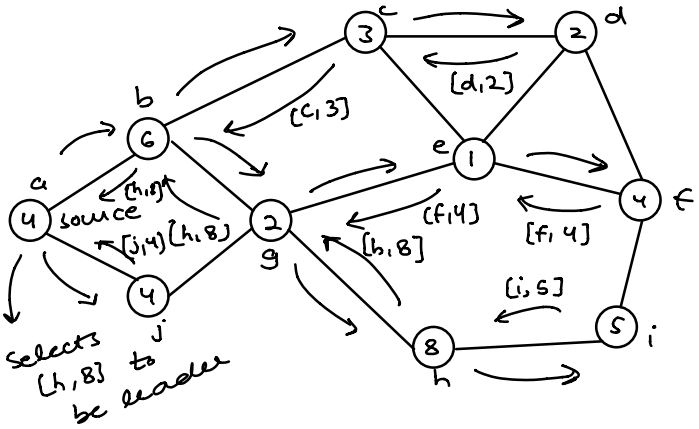
iv) Phase 2: 2^2 (4-neighborhood)



$\Rightarrow$ So, node 8 becomes the leader

5) **Election in Wireless Networks**

- Unreliable and processes may move
- Network topology constantly changing
  - > Consider static adhoc network
- Any node starts by sending out an election msg to neighbors
- When a node receives an election msg for first time, it forwards to neighbors and designates sender as its parent
- It then waits for responses from its neighbors
  - → Responses may carry source info
- When node election msg for second time, it just OKs it



Simulation - I

→ A system that represents behavior of another system over time.
→ The system being simulated is called a physical system
Physical system maybe an actual system or a hypothetical one

Physical system → contains some notion of state and evolve over time
The simulation must provide:

- a representation of state of physical system
- Some means of changing this representation of model
- Some representation of time

Physical time → physical time in system
Simulation time → Abstraction used by simulation to model physical time
Wall-clock time → Time during execution of the simulation
Real time execution → Simulation executions where advances in simulation time are paced by wall clock time
Scaled Real time → Simulation time advances faster or slower than wall clock time by some constant factor is refered as scaled-real time
As-fast-as possible → Progression of simulation time is not paced with wall clock time

Parallel Simulation → Execution of single simulation program on a collection of tightly coupled processors
Distributed Simulation → Execution of a single simulation program on collection of loosely coupled processors
Simulation Model → Classified as

- Continuous Model: State of system changes over time. Modelling weather conditions, air pressure & voltage
- Discrete Model: Changing state at discrete points in simulation time

Discrete Models → Time Stepped, Event Stepped

- State change may occur b/w time steps
- Use small time steps to minimize errors
- This is inefficient since many time steps even if no state change

Compute a new value for state variables at each time step, it is more efficient to update state when something interesting occurs

⇒ A discrete event simulation consist of logical processes (LPs) that interact by exchanging time stamped messages

- Each LP process events in non-decreasing timestamp order - local causality constraint
- If each LP adheres to local causality constraint, then the parallel/dist. execution yield same results as sequential execution

Sequential discrete event simulation utilizes three data structures

- state variables
- Event list
- A clock

Start simulation → Initialize state variables, generating initial events

Stop simulation → Last event processed, End on simulation time

In parallel and distributed simulation,

- Each LP process events both generated locally or by other LPs in time stamp order
- Failure could affect computation of event in its past

Processing out of order events - causality error

Problem ensuring that events are processed in timestamp order - Synchronization problem

Sync. algo ensures that execution results of parallel & distributed and sequential simulation should exactly the same

Conservative Synchronization Protocol

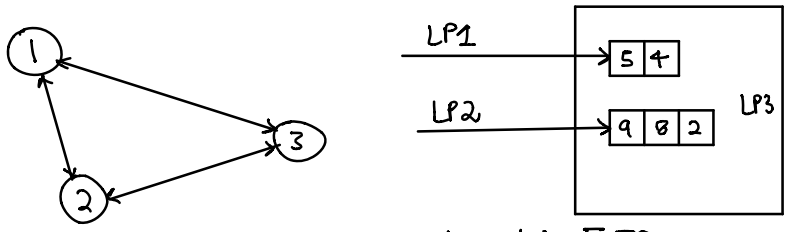
```
While( simulation not over)
    wait until FIFO contains at least one msg
    Remove smallest stamped msg M from FIFO
    Clock = time stamp of M
    Process M
```

LPs communicate by sending non-decreasing timestamped msgs

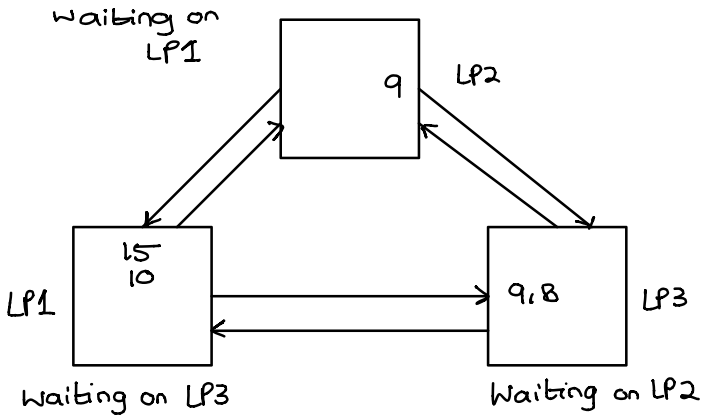
Each LP keeps a static FIFO channel for each LP with incoming communication

Each FIFO channel (input channel, IC) has a clock c_i that ticks according to the timestamp of topmost msg, if any, otherwise it keeps time stamp of last msg.

Deadlock Avoidance



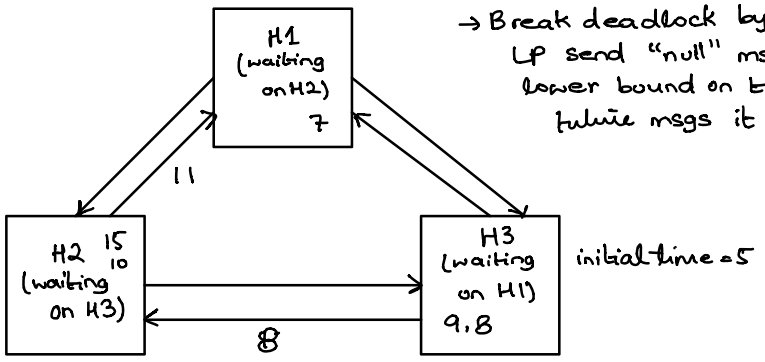
Message arriving on each link stored in FIFO queue



→ Deadlock situation, each LP is waiting on the incoming link

Assumption

- Logical processes (LPs) exchanging time stamped events (msgs)
- Msgs sent on each link are sent in time stamp order
- Network provides reliable delivery, preserves order



→ Break deadlock by having each LP send "null" msgs indicating a lower bound on time stamp of future msgs it could send.

initial time = 5

Minimum delay = 3 unit

- H3 = 5
- H3 sends null to H2 with timestamp 8
- H2 sends null to H1 with timestamp 11
- H1 may now process msg with timestamp 7

Chandy/Misra/Bryant "Null Msg" Algorithm

While (simulation is not over)
  wait until each FIFO contains at least one msg
  remove smallest time stamped event from its FIFO
  process that event
  send null msgs to neighbouring LPs with time stamp indicating a lower bound on future messages sent to that LP (current time plus lookahead)
END WHILE

→ Null msg algorithm relies on "lookahead" (minimum delay)
→ If lookahead is small, there maybe many null messages

⇒ The main disadvantage of Chandy/Misra/Bryant algo is large number of NULL messages - avoid deadlock situation

Deadlock Detection
- Diffusing distributed computations
- Dijkstra/Scholten algorithm (signaling protocol)

Deadlock Recovery

System Model

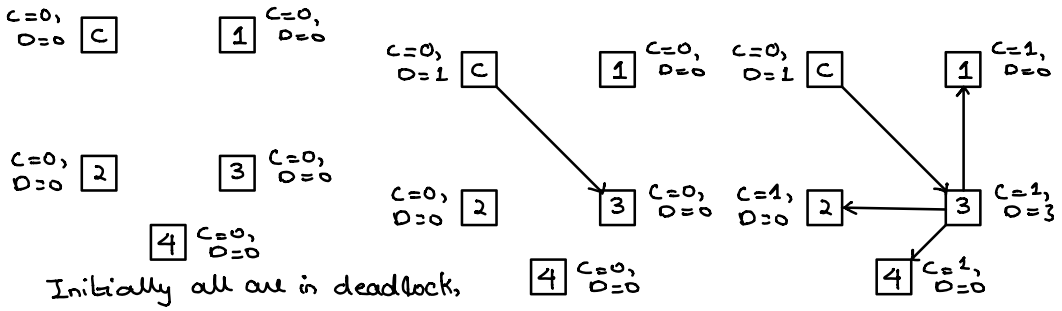
- No null msgs
- Allow simulation to execute until deadlock occurs
- Provide a mechanism to detect deadlock
- Provide a mechanism to recover from deadlock

Diffusing Computation (Dijkstra/Scholten)

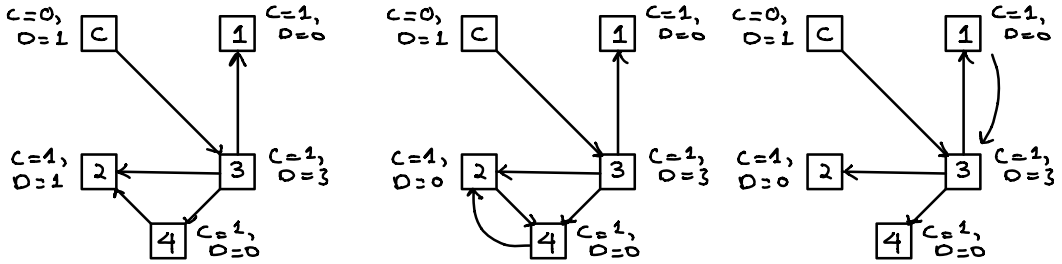
- Computation consists of a collection of processes that communicate by exchanging messages
- Receiving a msg triggers computation, may result in sending/receiving more msgs
- Processes do not spontaneously start new computation (must first receive msg)
- One process is identified as controller used for deadlock detection & recovery

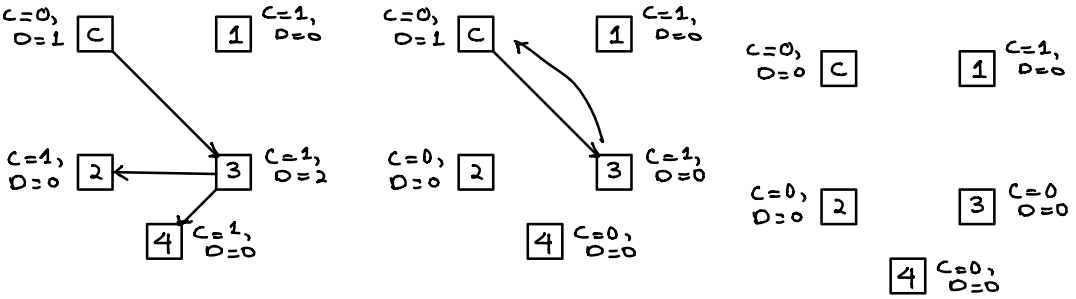
Goal: Determine when all are blocked (global deadlock)
Initially, all processes blocked except controller
Controller sends messages to one or more processes to break deadlock
Computation spreads as processes send messages
Construct a tree of processes that expands as computation spreads, contracts as processes become idle.
Processes in tree → engaged
Processes not in trees → disengaged
A disengaged process, becomes an engaged process when it receives a msg.
An engaged process becomes disengaged when it is a leaf node and it is idle.
If the tree only includes the controller, they are in deadlock.

C → msgs rcvd by process for those signals haven't returned
D → msg sent by process for those signals haven't returned



Initially all are in deadlock,
C: Controller





Signaling Protocol

- When an engaged process receives a msg
  Immediately return a signal to sender indicating
  - msg did not spawn a new node in the tree
- When an disengaged process receives a msg
  Receiving process becomes engaged
  Do not send a signal until it becomes disengaged
- An engaged process becomes disengaged (and sends a signal to its parent in the tree) when
  - It is idle and it is leaf node in the tree
  - Process is a leaf it has rcvd signals for all msgs it has sent

Each process maintains two variables
C = # msgs received for which process hasn't returned a signal
D = # msgs sent for which a signal has not yet been received

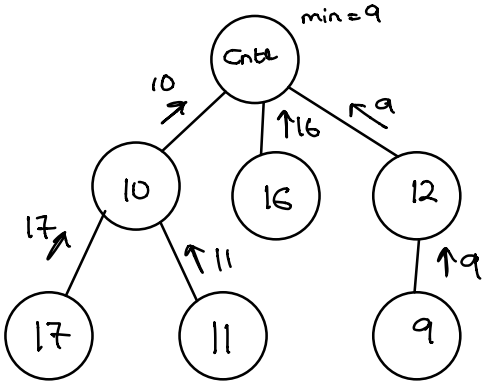
When is a process disengaged?
A process is disengaged when C=D=0

When is a process a leaf node of tree?
if C>0, D=0

When does a process send a signal?

If C=1, D=0 and the process is idle (becomes disengaged) or if it receives a msg and C>0

Deadlock detection → C=D=0 in the controller



- Broadcast msg to every other process in the system
- By constructing a spanning tree:
  - Controller sends request msg to its descendents
  - Upon receiving each forward to its descendents
  - Leaf node return event value to its parent
  - Every parent compute the minimum among
    - Its own min value
    - Received min from descendents
  - Finally controller broadcast the min value to all the processes

Drawbacks

→ Only specific smallest time events as being safe to process

- Deadlock detection and recovery algo is based on entire computation becoming deadlocked
- Literature exist that talks about partial deadlock detection & recv
  - Complexity increases
  - Results show no performance improvement

Deadlock detection & recovery can work for zero lookahead

If lookahead is available, we use to find more safe events

Synchronization

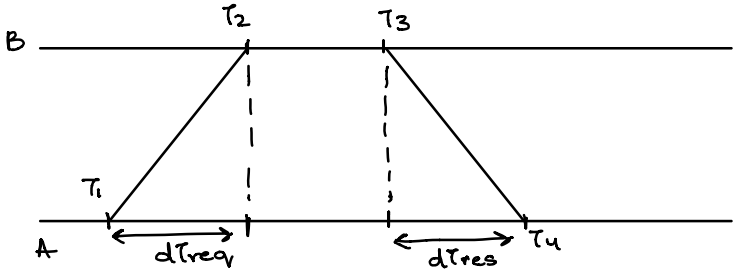
Physical Clock Synchronization — Christian's Algorithm
Berkeley Algorithm

Christian's Algorithm

Server is passive, equipped with accurate clock or receiver

client request the server for time

Problem is message delay outdated the reported time



- A send request to B with timestamp T1
- B will in turn record time of receipt T2
- Return response with T3, piggyback with T2
- A records the time of response arrival T4

- Assume prop. delay is same A-B & B-A

$$\theta = \frac{1}{2}((T_2 - T_1) + (T_4 - T_3))$$

$$\delta = T_3 + \theta - T_4$$

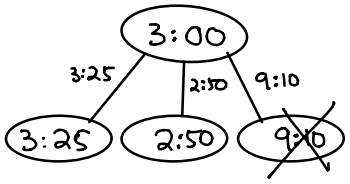
$$T_{local} = T_{local} + \delta$$

Christian's Algorithm - NTP

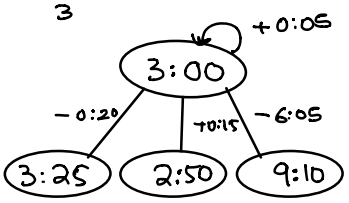
In NTP, B will also probe A for time
The offset is computed
The delay is calculated $\Rightarrow \frac{1}{2}((T_2 - T_1) + (T_4 - T_3))$
8 pair of values are buffered (offset, delay)
Finally taking minimum value as best estimation

Berkeley Algorithm

- Assume that no machine has accurate source of time
- The server process in the Berkeley Algorithm, master, periodically polls the slave processes.
- The master is chosen via election process
- The master polls the slaves who reply back with their time
- The master then averages the time
- Sends back the amount (+ve, -ve) that each slave must adjust its clock. This avoids further uncertainity due to RTT at the slave process.



$\frac{3:25 + 2:50 + 3:00}{3} = 3:05$ (fault tolerant avg)



→ send msg back to adjust

Lamport's Logical Clock

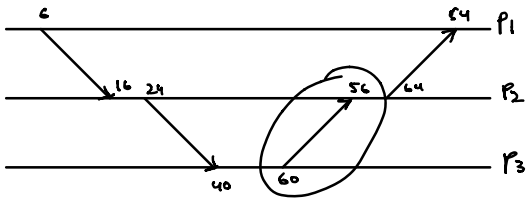
- If two processes do not interact, it is not necessary their clocks be synchronized because lack of synchronization would not be observeable
- It does not matter all processes agree on what time it rather they agree on the order in which event occurs

$a \rightarrow b$ (a happens before b)
observed in two situations:

1) a & b are two events and a occurs before b, then $a \rightarrow b$ is true
2) If a the event of msg being sent by one process and b is the event of msg received by another process then $a \rightarrow b$ is true

$\Rightarrow$ If $a \rightarrow b$ then $C(a) \rightarrow C(b)$
'a' and 'b' occur on the same process

Lamport's Algorithm follows happens-before relation



Implementation

Each process maintain a local counter C_i

1) Before executing any event, P_i executes $C_i = C_i + 1$
2) When P_i send a message m to P_j its set m's timestamp equal to C_j after executing prev. step
3) Upon receipt of msg m, P_j adjusts its own local counter $C_j = \max(C_j, m(ts))$ and execute step 1 and deliver msg to application

With Lamport's clock, we can say that for any two events a in P_i and b in P_j, if $a \to b$, $C_i(a) < C_j(b)$

- But if $C_i(a) < C_j(b)$ then we cannot say for sure if $a \to b$ or not
- This is partial ordering

We need totally-ordered multicast - A multicast operation by which all msgs are delivered in same order to each receiver

Totally Ordered Multicast

- Update msg is time stamped with sender's logical time
- Update msg is multicast (including sender itself)
- When msg is recvd
- Put into local queue
- Order according to timestamp
- Multicast acknowledgement

Msg is delivered to application only when:

- It is at head of queue
- It has been acknowledged by all involved processes
- P_i sends an acknowledgement to P_j if
- P_i has not made an update request
- P_i's identifier is greater than P_j's identifier
- P_i's update has been processed
- Lamport's algorithm (extended for total order) ensures total ordering of events

Vector Clocks

Event s happens before event t means the vector clock value of s is smaller than vector clock clock value of t.

Each event has vector of n integers as its vector clock value

- $v1 = v2$ if all n fields same
- $v1 \leq v2$ if every field in v1 is less than or equal to corresponding field in v2

$\Rightarrow$ A vector clock VC(a) assigned to event a has the property that if $VC(a) < VC(b)$ for some event b

→ Then event is known to casually precede event b

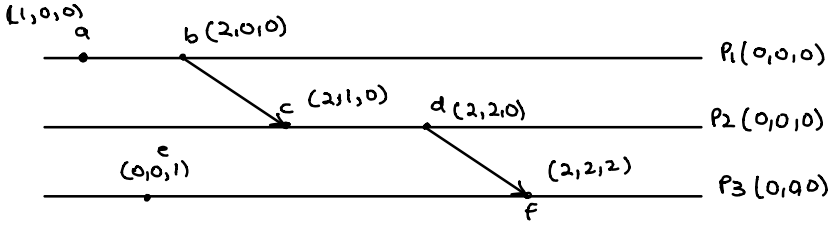
⇒ Vector clock is constructed by letting each process P_i maintain a vector VC_i with two properties

1) $VC_i[i]$ is a number of events that have occurred so far at P_i.
   $VC_i[i]$ is logical clock P_i

2) $VC_i[j] = k$ then P_i knows that k events have occurred at P_j. P_i has a knowledge of local time at P_j

Implementation

- Before executing an event P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$
- when process P_i sends a msg m to P_j, it sets m's timestamp equal to VC_i after having executed the previous step
- Upon receipt of a message m, process P_j adjusts its own vector by setting $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k after which it executes the first step & delivers the msg to the application



Causality
If $a \rightarrow b$ then a can affect event b

concurrency
If neither $a \rightarrow b$ nor $b \rightarrow a$ then one event cannot affect the other

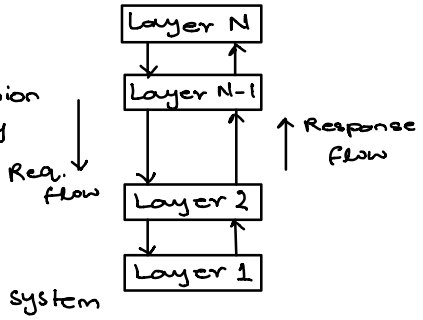
Partial Ordering
Causal events are sequenced

Total Ordering
All events are sequenced

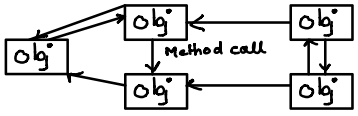
Distributed System Architecture

Software architecture – Logical organization of distributed systems into software components
components connected with each other, exchange data
Jointly configured into a system

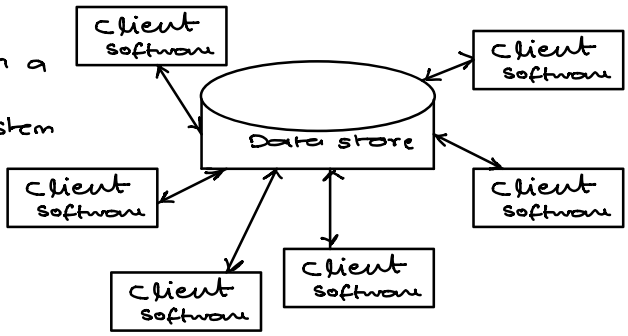
1) Layered Architecture
   - Components are organized in layered fashion
   - widely adopted in networking community



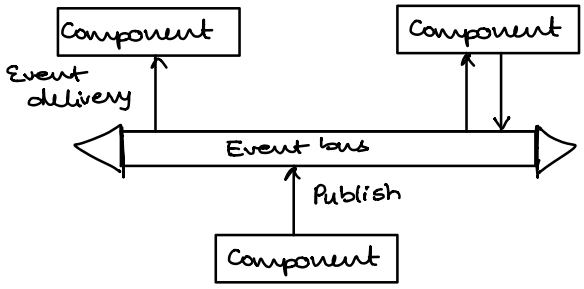
2) Object-based Architecture
   - Connected through a well-defined interface
   - Architectures matches with client server system



3) Data-centred Architecture
- Process communicates through a common repository
- Web based distributed system
- Content servers



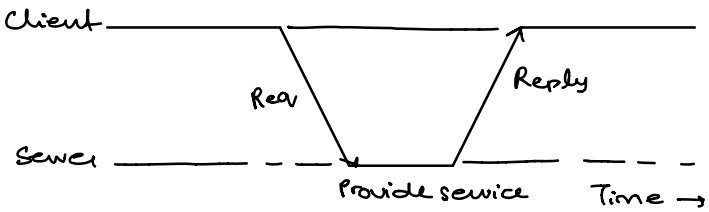
4) Event based Architecture
- Processes communicate through propagation of events
- Publish/subscribe system
- Event-based architecture combined with data centric architecture - shared data spaces



Organizing Systems

Centralized Architectures

Client server architecture → client request services from server
- ↳ In basic client server, process are divided into two groups
- ↳ server is a process implementing different services i.e. file system, database, and etc
- ↳ Client is a process that request service from the server by sending a request msg and subsequent wait reply



- connectionless protocol

Idempotent ops → The operations that can be repeated without any harm

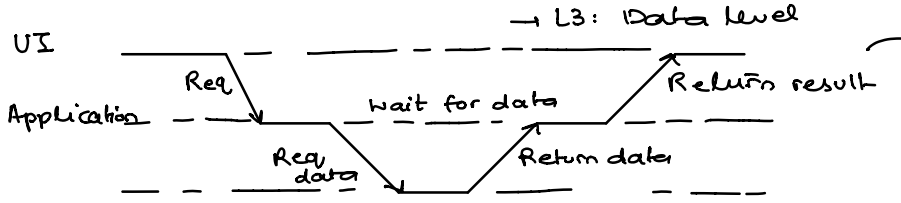
connection-oriented protocol → not in LANs, works fine in WAN

At-most once: Delivery means that for each msg handed to mechanism, that msg is delivered zero or one times, in simpler terms, it means that msgs may be lost

At-least once: Delivery means that for each msg, multiple attempts are made such that at least one succeeds; again in more casual terms this means that msgs may be duplicated but not lost

Exactly once: Exactly one delivery is made to recipient; the msg can neither be lost nor duplicated

Centralized Architecture → L1: user interface level
→ L2: Processing level
→ L3: Data level



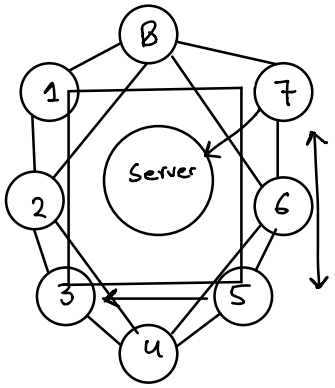
→ 3-tiered architecture

Decentralized Architecture

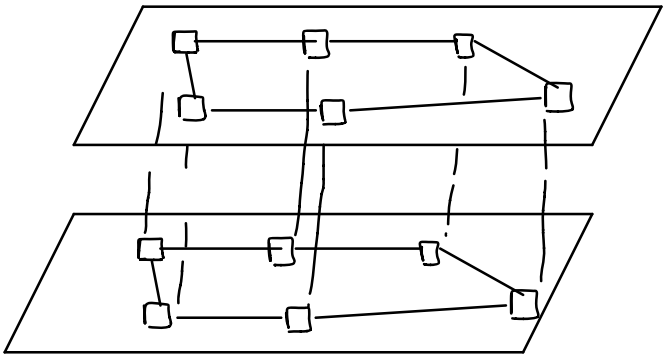
- Vertical distribution
  - Typical multi-tiered arch
- Vertical fragmentation
  - Placing diff components on diff machines
- Horizontal dist
  - All systems are equal - Client/server at same time
  - Peer-to-peer system

Overlay Network
- A computer network build on top of another network
- Nodes of an overlay can be thought of connected by virtual or logical links
- Two types of overlay networks
  → Structured & unstructured (Random connections)

P2P



Overlay Network



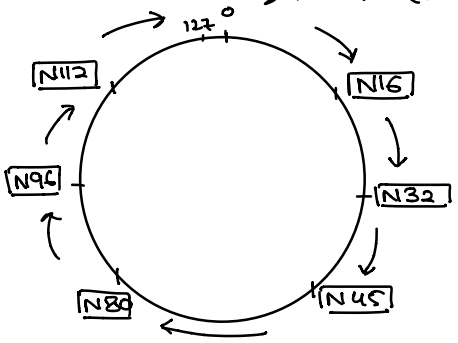
CHORD

- A hashtable allows you to insert, lookup and delete objects with keys
- A DHT allows you to do the same in a distributed setting (objs= files)

CHORD → structured peer-to-peer

⇒ O(Log (N))

Uses consistent hashing on node's (peer's) address
- SHA-1 (ip_addr, port) → 160 bit string
- Truncate to m-bits
- called peer id (number b/w 0 and $2^m - 1$)
- Can then map peers to one of 2^m logical points



- Every node knows its immediate clockwise successor in the ring.
- Each node knows its immediate predecessor.
- For most purposes, CHORD only uses successors

Every node has a fingertable

- ith entry at peer with id n is first peer with id >= $n + 2^i \pmod{2^m}$

Ex. N80 finger table

i	ft[i]
0	96
1	96
2	96
3	96
4	96
5	112
6	16

$= n + 2^i$ → $80 + 2^0 = 80$ so 96
$n + 2^i$ → $80 + 2^1 = 82$ so 96
$80 + 2^2 = 84$ so 96
$80 + 2^3 = 88$ so 96
$80 + 2^4 = 96$
$80 + 2^5 = 112$
$80 + 2^6 = 144$
since m=7
so, $144 \bmod 128 = 16$

To add file

- SHA-1(filename) → 160 bit str
- file is stored at first peer with id greater than or equal to its key (mod 2^m)

Search

At node n, send query for key k to longest successor / finger entry <= k if none exists send query to successor(n)

applied to every node

Churn

- Node join, leave & failure
- Due to churn behavior of chord - high network band width is consumed
- One possible solution → keep pointers to original location

Availability & Resilience → Replicate blocks at r successors

Distance b/w peers in CHORD algo ⇒

$$\text{Distance}(b,a) = (b - a + 2^m) \bmod 2^m$$

Unstructured Peer-to-Peer Arch

- Relies on random algorithms for constructing an overlay network
- Each node maintain a list of neighbors - list constructed in random way
- Data items are randomly placed on data nodes
- Locate data item, flood the network

Super Peers

- Nodes such as those maintaining an index or acting as a broker are generally referred to as super peers
- Super peers hold index of information from its associated peer

Hybrid Architecture → combination of centralized & decentralized architecture

Edge server systems → servers are deployed on internet where services are placed at the edge
ISP is residing at the edge of the internet

Collaborative DS → A hybrid structure is deployed in collaborative distributed systems

CAN: Content Addressable Network

Based on N-dimensional cartesian coordinate space
one hash-function for each of N.
Entire space is partitioned among all nodes, each node owns a zone in overall space

CAN → can store data at points in space
↳ can route from one point to another
Point = Node that owns the zone in which the point (coordinates) is located
Order in which nodes join is important

Insertion

- Discover some node "I" already in CAN
- New node picks its coordinates in space
- I routes to (p,q) and discovers that node J owns (p,q)
- Split J's zone in half. New owns one half

node I: insert(K,V)
a = hx(K)
b = hy(K)
route(K,V) → (a,b)
(a,b) stores (K,V)

Retrieve
node J: retrieve(K)
a = hx(K)
b = hy(K)
route retrieve(K) to (a,b)

CAN Improvements

- Possible to increase number of dimensions d
  → Small increase in routing table size
- Routing weighted by round-trip times
- Uniform partitioning → New node splits the largest zone in the neighborhood instead of zone of responsible node
- Nodes join CAN in different areas, depending on distance to landmarks

- State information → $O(d)$
- Routing → $O(dn^{2/d})$ hops

CAN → provides scalability, distribution, efficiency & fault tolerance, Balanced load
Zones can have different size, but they must have rectangular shape
→ Every node owns one distinct zone
→ In order to provide user's queries, node has to forward user queries to neighbours

→ The set of immediate neighbours serve as a routing table that enable routing b/w points in this space

→ A CAN node maintains a routing table that holds IPaddrs and virtual coordinates of its immediate neighbours

→ Simple greedy forwarding is used.

Maximal path length in each dimension: $d\sqrt[d]{n}/2$

Maximal path length CAN = $d \times d\sqrt[d]{n}/2$

→ If node fails, check first available node and forward from where greedy forwarding can resume

→ A bootstrap node maintains a list of partial list of CAN nodes which are currently in the system

→ Bootstrap node is responsible for establishing connection b/w user & available CAN node.

Leaving Node → finds a neighbor which can be merged with it and forms a proper zone – rectangular shaped.

→ If such a node doesnot exist, can choose any node and then additional algo fixes the inconsistent state of CAN

→ If node crashes, data lost and takeover mechanism initiated by its neighbor node.

→ Several neighbors can start takeover mechanism independently

Takeover mechanism

- Node initializes a timer in proportion to its volume
- If a timer is expired, send a takeover msg to all failed node's neighbors which contain volume of its sender's zone.
- The node which gets msg, compares own volume with sender's, if smaller a new takeover mechanism
- Else node take's zone of deputed zone

MapReduce → Reduce processes and merges all intermediate values associated per key

↓

parallely process individual records to generate immediate key/value pairs

Shuffling → movement of intermediate data from mappers to reducers
Same key, same reducer

YARN → NM, RM, AMaster

↓ Scheduling

↘ Container negotiation with RM & NM

Daemon & server-specific fns

Slow tasks → Stragglers

Speculative Execution → A task considered done when its first replica is complete (delete rest)

Definitions

- A DS is a collection of entities, each of which is autonomous, programable, asynchronous & failure-prone and which communicates through an unreliable medium.

Issues

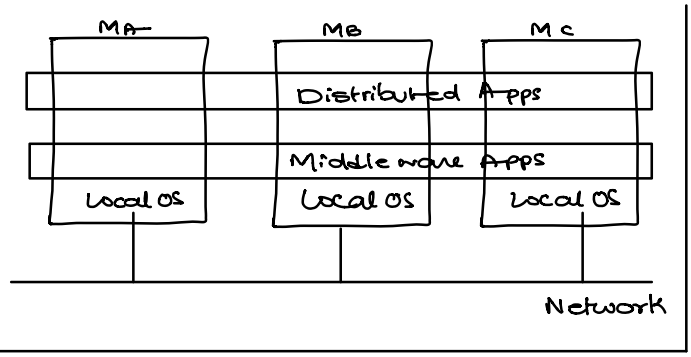
- No global clock
- High variable bandwidth
- failure-prone, unreliable
- Large & variable latency
- large # of costs

Common Goals

Heterogeneity → device types
Robustness → robust to crashes & failures
Availability → data & services always there?
Transparency → hide internal workings
Concurrency → can handle multiple clients?
Efficiency → fast enough
Scalability → can handle 100m nodes? more?
Security → can withstand hackers?
Openness → extensible?

Atomic broadcast → same msg, all process, same order

DOS → placed b/w user apps & OS (local)



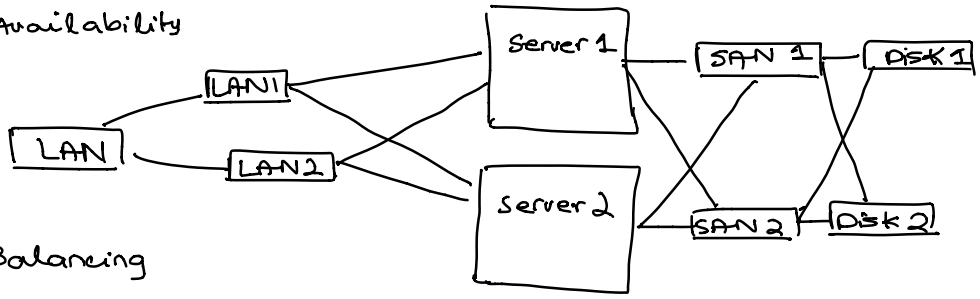
Access Transparency → Clients should be unaware of distribution of files
Location Transparency → Client should be unaware of phy loc. of resources
Mig. Trans → resources can be moved without affecting how they are accessed.
Relocation → relocated while being accessed without user noticing
Replication → replication to increase reliability
concurrency → access shared resource
Failure → user does not notice failure

Openness → Interoperability, Portability
Scalability → Size, Geographic, Administrative → scale along diff administrative domains

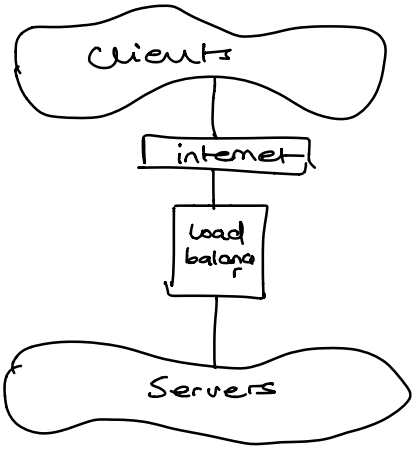
Grid → collection of DC available over LAN or WAN appearing as one large virtual system
↓ control node
↓ middleware

Cluster Computing → Collection of systems working together
↳ similar PCs, same OS

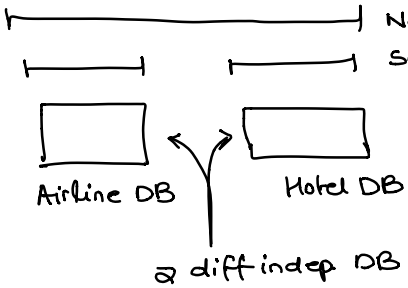
High Availability



Load Balancing



DIS → database, multiple reqs → into one req
↳ Transaction Processing system, Enterprise App. Integration



Dist. Pervasive. C → Mobile and embedded computing devices
↓ → Adhoc, sharing, contextual
Lack administrative control
charges

Cloud features

- Massive Scale . On demand access . Data intensive nature
- New cloud computing paradigms

HaaS → cluster
SaaS → docs, MS Office on demand
IaaS → EC2, S3, OpenStack, GC, Azure
↳ flexible computing & storage
PaaS → computing & storage coupled with platform
↳ Google's App Engine