(I) INTRO Distributed Systems

- Major Comp.
- Advantages
- Harder to design
  - Lack of global state
  - Lack of global time
- Models of systems
  - Topology
  - Communication
  - Synchronous ?
  - Unique IDs ?
- Complexity measures
  - Message
  - Communication
  - Time complexity
  - Space
- Distributed search
  - Model 1: Asynch, FC topology, reliable
  - Model 2; " " " , unreliable
  - Model 3: Synch, " " , reliable
  - Model 4: Arb topology.

(II) TIME

- Physical clocks
  - clock drift : $r < \delta/2\rho$ { δ – app. bound, ρ – drift, r – resync.
  - Cristian's Algo (External Sync)
    - i) Time server, periodically client sends requests, estimates time.
    - ii) Processing time are timestamped
    - iii) Fast clocks are slown down

- Berkeley's Algo (Internal sync)
  i) Time server is active
  ii) Asks client about time and estimates it for all of them.
  iii) Averages & sends adjustment to all client

- NTP (External sync)
  i) Syncing with UTC time
  ii) Organise internet PCs into stratums and request a PC in the above stratum for sync
  iii) Reference clocks in stratum 0
  iv) More servers at high stratums & less accurate

• Logical clocks
  - Lamport's Ordering
    i) $x \to y$ (x & y are local, x is send and y is receive)
       $x \parallel y$ (concurrent)
    ii) Causal ordering - potential dependencies
    iii) Each node has C_i. Each event X increments C_i by 1 ($C_i(x)$). If X is send from 'i' to 'j', then
    $$C_j = \max(C_j, C_i(x)) + 1$$
    iv) If $x \to y$, then $C(x) < C(y)$

  - Vector clocks
    i) C_i - vector of size n (n processes)
    ii) If event happens at 'i' $C_i[i]$++.
    If 'i' sends message to 'j' with timestamp T, on receive
    $$C_j[k] = \max_{C_j[j]\,++}(C_j[k], T[k])\ \forall k$$

iii) 2 ts t_x & t_y :

* $t_x = t_y$ ($t_x[k] = t_y[k]$ $\forall k$)
* $t_x \leq t_y$ ($t_x[k] \leq t_y[k]$ $\forall k$)
* $t_x < t_y$ ($t_x \leq t_y$ & $t_x \neq t_y$)

$t_x < t_y \iff x \to y$

$t_x \not< t_y$ & $t_y \not< t_x \Rightarrow x \parallel y$

- Birman-Schiper-Stephenson (Causal Order Broadcast)

i) sender - i
receiver - j
TS - VT_m

ii) Wait till
* $C_j[i] = VT_m[i] - 1$ (Till all prev. messages have been delivered)
* $C_j[k] \geq VT_m[k]$ $\forall k \neq i$

causal ← (Till all prev messages of k that i knows of have been delivered)

iii) When delivering update according to VC rule

(iii) Global Snapshot

→ collect both node & channel states
→ LS_i - local state of node i
m_{ij} - message from i to j

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

$inconsistent(LS_i, LS_j) = \{m_{ij} \mid S(m_{ij}) \notin LS_i \wedge R(m_{ij}) \in LS_j\}$

- Chandy-Lamport's Algo

Model: Async, FIFO, arbitrary connected

i) Use marker message, initiator starts. Sends through all outgoing channels

i)

ii) If node rec. marker first time, record LS; & send marker through all outgo. channels. Record $C = \emptyset$ (channel)

iii) If not 1st marker, record C = {m | m is a message after 1st & before arrivin by C}

iv) Build a spanning tree on the way to collect the snapshot at the initiator.

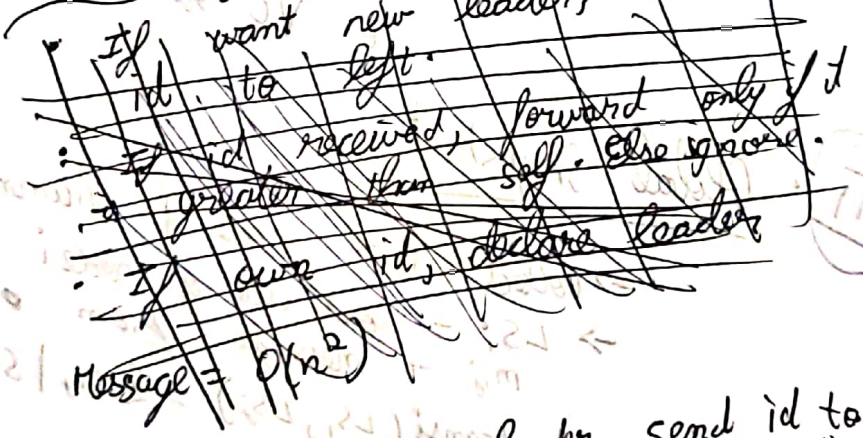
IV. Leader Election

* One process elected as leader. The process knows it is the leader. Others know they are not.

- Ring Topology

No deterministic algo for synchronous, non-uniform, anonymous leader elections

1) Lelann-Chang-Roberts Algo (Unidirectional)



- If want new leader, send id to left.
- Initially all are potential candidates.
- If received id is greater, quit abdio and forward id.
- If received id is less, ignore.
- If received same id, declare leader

Message = $O(n^2)$

2) Hirschberg - Sinclair Algorithm (Bidirectional)

- Operates in phases. In k^{th} phase ($k \geq 0$), each node, sends a Elect message to 2^k nodes to the left & right.
- If receive Elect message from node id < current id, ignore. Else, forward to destination and quit from race.
- If dest. received Elect message, send back reply to source.
- If a node receives reply from both sides it goes to next phase (k+1)
- If receive own Elect message, declare leader.

Message = $O(n \log n)$

→ each phase k, distance between 2 alive nodes are separated by 2^k nodes. Therefore, $\left\lceil \frac{n}{2^k+1} \right\rceil$ nodes alive.

→ Each node sends at most $\theta(2^k)$.

→ $1 \leq k \leq O(\log n)$

3) Peterson's Algorithm (Unidirectional)

⊕ Synchronous

- A process communicates using alias ids. Initially all processes are red (in the race) and $A_i = ID_i$. A process turns black eventually (quits) until only 1 red remains (leader). Black just forwards message
- Red pred of i = $N(i)$
  Red pred+pred of i = $NN(i)$
- Algorithm runs in phases. Node i receives messages from both $N(i)$ & $NN(i)$

- If $A_{N(i)}$ is the largest, process continues with $A_i = A_{N(i)}$, else turns black.

Message = $O(n \log n)$

→ Each round, of either i & N(i) one of them turns blak
→ log n phases.

4) Variable time algo (Sync, Unidir)

Round = Maximum message delay for 1 circulation

- After node id (ID_i) round time, if no message received, declare leader.

Message = $O(n)$
Time = $O(\min ID)$

- Smallest ID - leader

– Arbitrary Topology

1) FloodMax (Synchronous)

- Happens in rounds
- Each nodes stores the largest id it has seen till now. Initially $L(i) = i$.
- At each round, it exchanges messages with all neighbours and compares and replaces $L(i)$
- After diameter D rounds, each $L(i)$ th is the same the max ID
- Declare that as leader

Message = $O(d \cdot m)$

Mutual Exclusion

Every node has a common section that should be treated as CS.

Necessary:
- At most 1 process in CS
- No deadlock when multiple process requests
- Bounded wait
- Requests are granted in some order.
- Complexity :- Messages / entry, Synchronisation delay

- Permission Based
  * Nodes take permission from all / subset of nodes

1) Lamport's Algo
   - permission from all
   - Model: FC topology, FIFO channels
   - Uses Logical clock systems

→ All nodes have request queue's q_i which keeps request sorted by timestamps.

→ To enter CS, request with timestamp to all other nodes and put it in the queue as well

→ When received any message greater than timestamp of REQ from all other nodes and its REQ is in the topmost, enter CS.

→ Receiving REQ, send ACK & put it in q_i.

→ After CS, send release to all nodes.

Obs. when sending a reply, the node informs that it has no more earlier requests.

Message = 3(n-1) / entry
Sync. delay = one transmission

2) Ricart – Agarwala Algo

- Improvement over Lamport. Does away with the queue.

→ Here, a node sends REQs to all other nodes if it wants to enter CS. If receiving ACKs from all of them, enter CS

→ On receiving REQ, if its timestamp is greater than its REQ or currently in CS, defer ACK. After exiting or its lowest req

→ Think of nodes as vertices & requests as edges from higher ts to lower. As a node finishes CS, it sends reply to all incoming edges

- Message = 2(n-1) / entry
Sync. delay = one transmission

3) Maekawa's Algo

- Hinges on getting permission only from a subset of nodes (quorum) – R_i $\forall i$
- i) $R_i \cap R_j \neq \phi$ $\forall i,j$ (safety)
ii) $\forall i$, $i \in R_i$
iii) $|R_i| = K$
iv) a node is present in exactly D quorums

→ To enter CS, i sends REQ to all nodes in R_i & waits for ACK from them.

→ If no ACK has been sent since last receiving a RELEASE message, send ACK after REQ

→ Send RELEASE after CS to all nodes in R_i

Message $\approx 3\sqrt{N}$ $\{K \approx \sqrt{N}\}$

Sync. delay = 2 × msg transmission

Obs:-

i) Deadlock possible when 2 nodes are requested by 2 other nodes to enter CS, and these nodes send ACK for one of them each

ii) Have more message types possible - lock, failed, inquire, relinquish. Lower ts REQs make a locked node send 'inquire' message to the given permission node. Higher timestamp makes the locked node send a 'failed' message. If received 'failed', then send 'relinquish'.

- Token Based

- Single token is present, enter CS when have token

1) Suzuki-Kasami

Model: FC topology, only 1 CS request afloat.

→ To enter CS, broadcast request to all nodes & enters CS if received token.

Req(i, K) & RN[i]++

If already have token and idle, enter CS.

→ On receiving R(i, k) on node j,
$RN_j[i] = max(RN[i], k)$.

{$RN_j[i]$ - latest request from i}.

If token is idle & $RN_j[i] = LN[i]+1$, send token to i

{LN[i] - last served request no.}

→ On exiting CS at i,
set $LN[i] = RN[i]$
Add j to Q in token ∀ $LN[j]+1 = RN_i[j]$ & $j \notin Q$

{Q - FIFO}

Send token to the head of the queue after deleting it.

Message = $\lceil n \rceil$
Sync. delay = Max transmission

2) Raymond's Algo

- On a tree topology. Each node has parent P_i & FIFO queue Q_i. Initially all parents point to route & all Q_i's empty.

→ When want to enter CS, put node in Q_i. If not already waiting for token, send req to P_i.

→ When receiving req, put the requestor Q_i. If not already waiting, send req to P_i

→ When node finishes CS, it sends token to the head of Q_i & changes P_i. The destination becomes root. Head is removed from Q_i. If Q_i not empty

sends in another request to P_i

→ When head of Q_i is i, enter CS.

VI. Spanning Tree

1) Arbitrary – Flooding

2) Shortest Path tree (shortest distances from root)

→ Bellman-Ford

i) Initially all distances are ∞

ii) Neighbors distance are updated.

iii) Every step neighbours exchange their distances, & relax/update existing ones. Then path is updated (through which neighbor)

$$D_i[k] > D_j[k] + D_i[j] \quad \{\text{Relaxation condition}\}$$

$$P_i[k] = j$$

$$D_i[k] = D_j[k] + D_i[j]$$

→ $\forall i, j, k, j \in N_i$

3) DFS trees

→ Just like normal DFS

→ Order neighbours

→ Recursively carry out expansion.

Once a subtree is done, return <parent>

→ If already visited node, return <already>

⊕ If root not given?

→ Start from everywhere

→ Then, when recursively visiting, already visited, if new visitor is

< Ignore

= return <already>

> overwrite <parent>

④ BFS Tree

→ Synchronous : Easy just parallelise sequential

→ Asynchronous :- Wait for ACK from root before start of each layer. After finishing each layer, send DONE back to root.

⑤ Topology control

→ XTC algorithm

Input :- A graph

Output :- i) A graph without any cycle of length 3

ii) Connected, if original is connected.

VII. Fault tolerance

Fault in component → Errors → system failure

① Dependability

i) Reliability

→ How often system fail

→ MTTF, MTTR, MTBF (= MTTF + MTTR)

ii) Availability

→ How available the system is

→ Uptime (Running / Total)

iii) Safety

→ How safe a system is even if it fails

⊕ Highly reliable ⇒ Highly available

⊕ Dependability at different levels

(2) Fault tolerance :- Ability to deliver service despite component failures

→ Full service (i.e. a primary backup)
→ Degraded service (slower response time)

i) Classification

Depends on find behaviour of system

→ Crash - stops
→ Failstop - stops with condition
→ Omission - ignores some steps
→ Byzantine - behaves arbitrarily
→ Timing - violates timing

ii) Specifications

→ Fault Model (class of faults)
→ Fault tolerance (Each class tolerance)

iii) Handling faults

⊕ Redundancy
→ Hardware (extra components)
→ Software (extra code)
→ Time (Rerun program)
→ Information (Error recovery codes)

⊕ Recovery
→ Forward error recovery
→ Backward " " "

iv) Examples

→ Storage
⊕ RAID - Redundant Array of Inexpensive Disks
(Data in multiple disks)

⊕ Shared Network Storage
(Easy to handle instead of individual storage)

→ communication (reliable) from unreliable links

③ Agreement Problems

→ n processes, at most m faulty
→ Synchronous, reliable
→ Identify sender

① ⊕ Byzantine Agreement

→ source node with initial value
→ All non-faulty process agrees on the same value
→ If the source is non-faulty, then the agreed value by non-faulty processes is initial value

⊕ Consensus

→ Each process has initial value
→ If all non-faulty processes have same initial value, then agreed value must be it
→ All non-faulty process must come to an agreement

⑤ Interactive consistency

→ All non-faulty processes must agree on the same array $A[v_1 \ldots v_n]$
→ If i is non-faulty & has v_i as initial value, $A[i] = v_i$

Agreement → IC { Trivial. n process runs agreement }

Consensus → Agreement { Broadcast value to all. Run consensus }

IC → Consensus { Apply majority on the received value }

i) Fault-free Consensus

-> completely connected
-> Broadcast initial value
-> A deterministic function on the received values.

ii) Consensus with crash faults

-> Above algorithm cannot work because of failure in between broadcast

```
for r from 1 to f+1:
    if x is new:
        broadcast(x)
    y_r ← if any value received
    x ← min(x, y_r)
output x
```

Agreement :- 1 round where no sys crash. All nodes received what all the others have. This means everyone has the same min value.

Valid :- Since only crash faults occur, no erroneous values are broadcast. If all non-faulty have same initial value, only that is circulated)

iii) Byzantine Agreement

Let $Z(n, m)$ -> Byzantine agreement with n nodes m faults

- $Z(3, 1)$ -> not solvable
- $Z(n \leq 3m, m)$ -> not solvable

{Divide n into 3 sets of n_1, n_2, n_3. Let the 3 sets correspond to P_1, P_2, P_3 of $Z(3, 1)$. $\exists$ a set with $n_i \leq 3f_i$ treat that as a recursion of this problem where base case is $Z(3, 1)$}

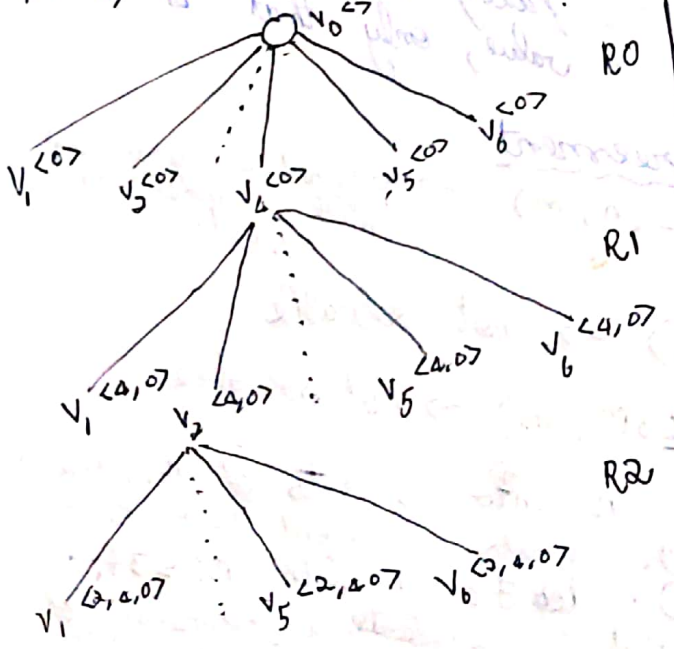
→ Algo

* Commander P_0 initiates OM(f) & se... its values to all other nodes. The values received are $v_0^{<>}$ w.r.t each process.
* Each process i on receiving $v_0^{<>}$, sends $v_0^{<>}$ to all processes j. The received value by j is $v_i^{<0>}$. This is OM(f-1)
* Again, each process j other than 0, sends its $v_i^{<0>}$ to all nodes other than i & 0, as $v_j^{<i,0>}$. OM(f-2)
* ~~Once each process~~ Goes on till OM(f-f).
* In this round, take majority for process i of received messages from the parent & children of parent & estimate the consensus value of the parent for that level by node i.
* Do this until commander level.

Complexity :- $O(n^f)$

Refer Kshemkalyani Pg. 522

n=7, f=2, f+1 rounds Tree



Perspective of P_3

P_3 estimates v_0 as majority($v_0^{<>}$, $v_1^{<0>}$, $v_2^{<0>}$, $v_4^{<0>}$, $v_5^{<0>}$, $v_6^{<0>}$)

P_3 estimates v_4 as majority($v_4^{<0>}$, $v_1^{<4,0>}$, $v_2^{<4,0>}$, $v_5^{<4,0>}$, $v_6^{<4,0>}$)

P_3 estimates $v_2^{<4,0>}$ as $<2,4,0>$. Similarly v_3 for others

④ Atomic Commits

When all parts of a ~~task~~ action are carried out simultaneously or none at all.

i) Two-Phase Commit (2PC)

⊕ 1 coordinator, others are cohorts
⊕ write-ahead log to recover from local crashes
⊕ Happens in 2 phases
⊕ Blocking protocol

Phase-1 (Prepare)

→ Coordinator sends REQ to all cohorts
→ Waits for reply
→ On receiving REQ, a cohort
* Send OK, if local commit is success & write logs of prev. state.
* If problem, send NO

Phase-2 (commit)

→ If all cohorts reply OK, co-ord writes 'commit' record in logs & sends COMMIT to all cohorts.
→ If there is at least 1 NO (or timeout) co-ord writes 'abort' record in logs & sends ABORT to all.
→ If ACK not received, send again. If all ACKS, write 'complete' record
→ On receiving COMMIT, cohort sends ACK & release resources
→ On receiving ABORT, cohort sends ACK, release resources & redoes the changes from logs

⑤ check what happens due to failures at each part of the algo

ii) Three-phase commit (3PC)

→ Non-blocking
→ Less than K failure
→ Heavy overload
→ Hardly used in practice

(5) Checkpointing & Rollback Recovery

- It is a backward error recovery (previously known fault free state)
- checkpoint:- Local state of a process stored in a stable storage for rollback on fault
- Usually checkpoints are consistent (If receive is marked, so is send)
- Outside world interactions cannot be rolled back, so have to handle with care
- Different types of messages have to be handled
  - In transit
  - Lost (send is not undone, but receive is)
  - Delayed (not recording receive)
  - Orphan (send undone, but not receive)
  - Duplicate (undone, and redoing a message)

i) Async / Uncoordinated
- Process takes indep. local checkpoints
- To recover, all nodes coordinate to find consistent checkpoints from all possible checkpoints
- Possible that no such case exist (have to restart)

- Good for systems with high comp. & low communication

ii) Communication Induced
- Synchronize checkpointing with send / receive message

iii) Message logging
- log all messages received since last checkpoint
- On rollback, after restarting, replays message log
- Handles outside interaction

iv) Sync / Coordinated
- Coordinate to take a consistent checkpoint & then recover.
- Recovery is just rolling back to last checkpoint

⊕ Koo - Toueg's
-> FIFO, able to cope with message losses

I. Checkpointing

$C_p = \{q \mid$ all q that has sent msg. to p since last checkpoint$\}$

$lmr_p[q]$ = last message from q to p

$fms_q[p]$ = 1st message since last checkpoint from q to p.

-> P_i initiates by taking tentative CP & sending REQ to all C_p. If YES from all, make CP permanent.

-> On receiving REQ, if not willing to take CP, send NO. Else if $1 \le fms_q[p] \le lmr_p[q]$ take CP. Then recurse with q as initiator.

-> If repeat REQ, ignore.

→ P initiates by taking tent. CP & sending REQ($lmr_p[q]$) to all $q \in C_p$. If received YES from all make CP permanent & inform the rest as well. Else, discard & inform.

→ On receiving REQ(), if not willing to CP, send NO. Else take tent. CP if $lmr_r[q] \geq fms_q[p] > -1$. Now, recurse with q as initiator. If receive YES from all, send YES to P, else NO.

(2 Phase Atomic)

→ During the process, do after taking tent. CP do not send other messages, and ignore if repeat REQ is recvd. (simply send YES).

→ If received CONF from source, make tent. CP permanent & send CONF to all child. If recvd DISC, discard CP & inform children.

II Rollback

→ P initiates tent. rollback & sends REC($lms_p[q]$) to all q. If receives YES from all send CONF to all. Else send, DISC.

(2 Phase Atomic)

→ On receiving REC(), if

If not willing send NO { $lms_p[q] < lmr_q[p]$ } tent. rollback. & q acts as initiator. If receives YES from all, send YES to P, else NO.

Prevents unnecessary rollback

→ If receive CONF from source, make rollback permanent & send CONF to all children. Sim for DISC except you discard rollback.

→ Do not send new message after tent. rollback & ignore repeat REC() & simply send YES