Deadlock handling strategies ⇒

* Prevention – inefficient
* Avoidance – impractical
* Detection & Resolution
  1. Progress (no undetected deadlocks) – detected and resolved
  2. Safety (No false deadlocks) – No phantom deadlocks

} Essential conditions for a deadlock detection algorithm

Models of deadlocks ⇒ A process might require a single resource or a combination of resources (for its execution)

1) The single resource model — A process can have one outstanding request for only one unit of resource.

2) AND Model — A process may request more than one resources simultaneously and the request is satisfied only after all the requested resources are granted to the process.

3) OR Model — A process may request more than one resources simultaneously and the request is satisfied if any one of the resources requested is granted.

→ A cycle doesn't imply deadlock in OR Model unlike AND Model

Presence of
→ A Knot indicates a deadlock

⇒ In a WFG, a vertex v is a Knot, if for all u: u is reachable from v : v is reachable from u

No path originating from a Knot have deadends.

1. Each of the processes in the set S is blocked
2. The dependent set for each process in S is a subset of S
3. No grant message is in transit between any two processes in the set S.
   (a process may release a resource)

4) AND – OR Model - In this model, we need x, y, z ...

5) $\binom{p}{q}$ Model ⇒ requires p resources from a pool of q resources

↳ Request to obtain any p available resources from a pool of q resources.

Special case :- $\binom{p}{p}$ Model – corresponds to AND Model

$\binom{1}{p}$ Model – corresponds to OR Model.

6) Unrestricted Model – No assumptions are made regarding the underlying structure of resource requests.

↳ detect a deadlock based on the global state.

Deadlock Detection Algorithm :-

* Path pushing – arrive at a global WFG. (all requirements for all the processes)

* edge-chasing – probe message is sent in the opposite direction of request messages along the edges to detect cycle.

* Diffusion computation – query messages to the adjacent processes, it'll reply if it is in deadlock

→ deadlock detection is diffused.

* global state detection – arrive at global WFG and analyse for cycles or knots.

Path-pushing algorithms –

– maintain an explicit global WFG.
– each process will send WFG through one process.

Edge-chasing algorithms –

– Presence of a cycle in a distributed graph structure is verified by propagating special messages called probes along the edges of the graph.
– If the probe message comes back, then there's a cycle

Diffusion-Computation-based algorithm –

– Deadlock detection computation is diffused through the WFG of the system, makes use of echo algorithm to detect deadlocks. (reply is an echo)

Global state detection based algorithms –

– Taking a snapshot of the system and examining it for the condition of a deadlock (cycles and knots)

* Mitchell and Merritt's algorithm for a single resource model ⟶ Edge chasing algorithm.

• Probes are sent in the opposite direction to the edges of WFG.

1.) Only one process in a cycle detects the deadlock – This can be improved by including priorities and the lowest priority process can be aborted.

2.) In this algorithm, a process that is detected in a deadlock is aborted spontaneously, even though under this assumption, phantom deadlocks cannot be excluded.

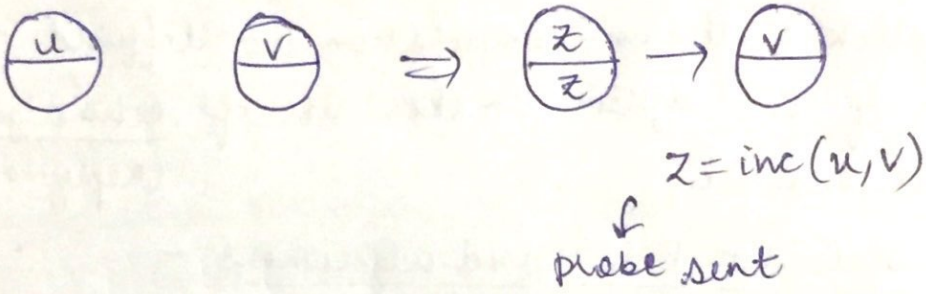
In the absence of spontaneous aborts, genuine deadlocks will be detected.

Each node of WFG, two local variables = labels

a public label — a private label.

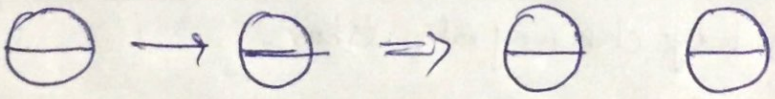
→* Block creates an edge in WFG – Two messages are needed, one resource request and one message back to the blocked process to inform of the public label of the process it is waiting for

Block :-



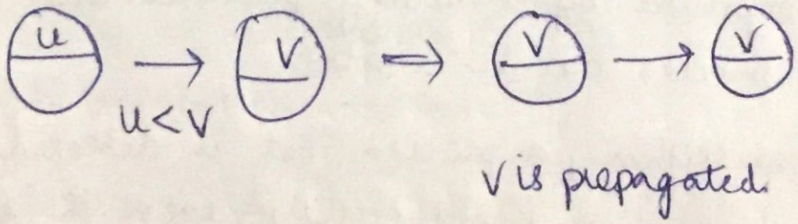
→* Activate denotes that a process has acquired the resource from the process it was waiting for.

Activate :-



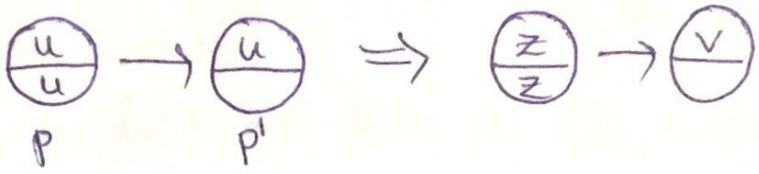
→* Transmit propagates larger labels in the opposite direction to the edges by sending a probe message.

Transmit :-



Detect means that the probe with the private label of some process has returned to it, indicating a deadlock

Detect :



Initially, we have $u = v$ for all processes
The only request that change u & v, Block and activate

→ u has been propagated from p to p'
→ probe transmitted only by a deadlocked process

Chandy-Misra-Haas algorithm for the AND Model ⇒

* Edge-chasing algorithm
* Probe is sent out in the form of triplet (i, j, k)
  [it is sent along the edges unlike ↑ opposite direction]
  - i: process initiating
  - j, k: probe is sent from process j to k across sites.

→ A deadlock detection initiated for P_i and is being sent by the home site of process P_j to the home site of process P_k.

→ A probe message travels along the edges of the global WFG and a deadlock is detected when a probe message returns to the process that initiated it.

→ A process P_j is said to be dependent on another process P_k if there exists a sequence of processes.

$P_j, P_{i1}, P_{i2}, \cdots P_{im}, P_k$

could be null also
(there might be direct dependency also)

Each process P_i maintains a boolean array $dependent_i$ where $dependent_i(j)$ is true only if P_i knows that P_j is dependent on it.

Initially, $dependent_i(j)$ is false for $\forall i, j$,

If P_i is locally dependent on itself, then declare a deadlock

else for all P_j and P_k such that

(a) P_i is locally dependent on P_j and

(b) P_j is waiting on P_k

(c) P_j and P_k are on different sites.

send a probe (i, j, k) to the home site of P_k.

On the receipt of a probe (i, j, k), the site takes the following actions –

if

(d) P_j is blocked and

(e) $dependent_k(i)$ is false.

(f) P_k has not replaced updating P_j

then

begin

$dependent_k(i)$ is true

if $k = i$ then declare that P_i is deadlocked

else for all P_m and P_n such that

(a) P_k is locally dependent on P_m and

(b) P_m is waiting on P_n

(c) P_m and P_n are on different sites

send a probe (i, m, n) to the home site of P_n

end

- should not require extra additional communication channel.

Chandy-Misra-Haas algorithm for the OR Model →

→ Diffusion computation

→ 2 types of messages ⇒ query (i,j,k) and

P_j sends a query to P_k

reply (i,j,k)

P_j sends a reply to P_k

→ If an active process receives a query, it ignores the message.

Process P_k maintains a boolean variable $wait_k(i)$ that denotes the fact that it has been continuously blocked since it received the last engaging query from process P_i

Each process has a local variable $num_k(i)$

⇒ Initiate a diffusion computation for a blocked process Pi

⇒ Send query(i,i,j) to all processes P_j in the dependent set DS_i of P_i

$$num_i(i) = |DS_i|, \quad wait_i(i) = true.$$

⇒ When a blocked process P_k receives a query (i,j,k),

if this is the engaging query for process P_i

then send query (i,k,m) to all P_m in its dependent set $|DS_k|$

$$num_k(i) = |DS_k|; \quad wait_k(i) = true$$

else if $wait_k(i)$, then send a reply (i,k,j) to P_j

when a process P_k receives a reply (i,j,k)

→ if $wait_k(i)$ then $num_k(i) = num_k(i) - 1$

if $num_k(i) = 0$, then if $i = k$, then

"Declare a deadlock"

else

send reply (i,k,m) to the process P_m (which sent the engaging query)

* Messages are diffent from the underlying computation messages and the message communication is also transparent.

⇒ If every member of the dependent set sends a reply, then "a deadlock is detected".

Termination Detection → not easy to determine.

Halting problem in Turing Machines.

Two distributed computations

1. the underlying computation — basic messages
2. termination detection algorithm — control messages

*A termination detection (TD) algorithm must ensure the following –

1. Execution of TD algorithm cannot ~~independently~~ indefinitely delay the underlying computation, must not

freeze the underlying computation.

(2) The TD algorithm must not require addition of new communication channels between processes.

⇒ A distributed computation has the following characteristics:-

(1) Processes can be only one of two states – active and idle or (busy and passive)

(2) An active process can become idle at any time

(3) An idle process can become active only on the receipt of a message

(4) Only active processes can send messages

(5) A message can be received by a process when the process in either of the two states : active or idle

(6) The sending of a message and receipt of message occur as atomic access. (not affected by any other external means)

* $\left(\forall i :: P_i(t_0) = \text{idle}\right) \wedge \left(\forall ij :: C_{ij}(t_0) = 0\right)$

Termination Detection using Distributed Snapshots -

* Global Snapshot [collection of individual snapshots]
  ↓
  Distributed

The algorithm is defined by the following four rules-
Each process i applies one of the rules when it is applicable.

R1: When process i is active, it may send a basic message to process j at any time by doing -

Send a B(x) to j

↓

basic message sent by the process.

R2: upon receiving a $B(x')$, process i does -

Let $x = x' + 1$

if $(i \equiv \text{idle}) \longrightarrow$ go active [// become active on receiving a message]

R3: When a process i goes idle, it does =>

Let $x = x + 1$

Let $K = i$

Send message $\underbrace{R(x,k)}_{\substack{\text{control} \\ \text{message}}}$ to all other processes.

[with time & process no.]

Take a local snapshot for the request of $R(x,k)$.

R4: Upon receiving message $R(x',k')$, process i does →

$[[(x',k') > (x,k) \wedge (i \text{ is idle}) \rightarrow$

Let $(x,k) = (x',k')$

Take a local snapshot for the request by $R(x',k')$

□ requesting process is earlier to the current process.

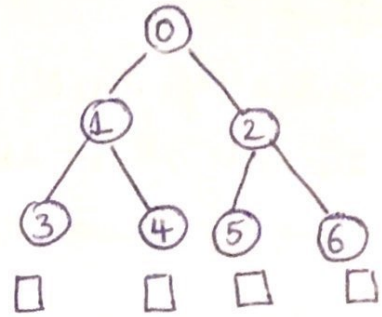
$((x',k') \leq (x,k) \wedge (i \text{ is idle}) \longrightarrow$ do nothing

□

$(i \text{ is active}) \rightarrow \text{let } x = \max(x,x')]$

Just, a tree datastructure
+
colouring scheme
(black + white tokens)

→ associated with leaf nodes



tokens are passed on once all the children nodes are terminated.

- sends a message (white tokens) — turns black
- before terminates — sends a black token to its parent
- ~~An intermediate node~~ root might receive one/more black tokens → infer termination not completed.

which means one of the processes hasn't terminated.

— if the root node receives a black token, it repeats the whole process again

— This process terminates iff the root node receives a white token/s.

Spanning-tree-based termination-detection algorithm :-

1. * Tokens - a contracting wave of signals that move inward from the leaves to the root.

2. * Repeat signal - if a token wave fails to detect the termination, node P_0 initiates another round of termination detection by sending a signal called "Repeat" signal to the leaves.

3. * The nodes which have one or more tokens at any instant form a set S

4. * A node j is said to be outside of set S if j doesnot belong to S and the path (in the trees) from the root to j contains an element of S

5. Note that all nodes outside of S are idle

Algorithm -

1. Initially, each leaf process is provided with a token. The set S is used for book keeping tokens which processes have the token
2. Initially, all processes and tokens are white
3. When a leaf node terminates, it sends the token it holds to its parent process.
4. A parent process will collect the token sent by each of its children. After it has received a token from all of its children and after ~~which~~ it has terminated the process sends a token to its parent.
5. A process turns black, when it sends a message to some other process. 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 the parent node if to indicate that a message passing was involved in the subtree.
8. Tokens are propagated to the root in this fashion. The root after 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 of 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 using the "Repeat" signal.

(11.) The root concludes the termination has occured – if (should satisfy all the conditions)

a) If it is white.

b) If it is idle.

c) If it has received a white token from all of its children

[Sukumarghosh]

*Termination detection for diffusion computation ⇒

Environment node, internal nodes

→ signals, acks.

⇒ For an edge (i,j), the difference between the number of signals sent by i and the number of acks received from j will be called a deficit.

⇒ A process keeps track of two different types of deficit

C = total deficit along its incoming edges

D = total deficit along its outgoing edges

By definition, these deficits are nonnegative integers.

INV1: $(C \geq 0) \wedge (D \geq 0)$

Environment Node ⇒ $C = 0, D = k$

For every other node in the system, the proposed signalling ~~structure~~ scheme processes the following invariant

INV 2: $(C > 0) \vee (D = 0)$

The sending of an ack, reduces the senders deficit C by 1

$$(C - 1 \geq 0) \wedge (C - 1 > 0 \vee D = 0)$$

$$= (C \geq 1) \wedge (C > 1 \vee D = 0)$$

$$= (C > 1) \vee (C \geq 1 \wedge D = 0)$$

$$= (C > 1) \vee (C = 1 \wedge D = 0)$$

Algorithm :- — Sukumar Ghosh
& (some 1980 papers)

Program detect {for an internal node i}
define C, D : integer
m : (signal, ack) {represents the type of message received}
state : (active, passive)

initially C = 0, D = 0, parent(i) = i

do (m = signal) ∧ (C = 0) ⟶ C = 1, state = active, parent = sender

[] m = ack ⟶ D := D − 1

[] (C = 1 ∧ D = 0) ∧ (state = passive) ⟶ send ack to sender.

[] (m = signal) ∧ (C = 1) ⟶ send ack to the sender

The environment node initiates the termination detection algorithm by making D = k (k > 0) and detects termination by the condition D = 0 after it receives k acks.

Leader Election →

Leader ⟹ focus of control = system wide management

It is tempting to compare leader election with the problem of mutual exclusion

* when a leader fails, you have to elect a new leader.

Comparison :- (ME vs Leader election)

1. Failure is not an inherent part of mutual exclusion algorithm.
2. Starvation is an irrelevant issue in leader election
   [A leader, which ceases to fail, remains a leader ... but that leads to starvation in the case of ME]
3. If leader election is viewed from the perspective of mutual exclusion, the exit from the critical section is unnecessary.
   [there's no point of leaving the CS in Leader election]

Let $G_1 = (V, E)$, each process $i \in V$ has a unique identifier,

→ L(i) identifies the leader.

→ ok(i) indicates whether process is active / not faulty.

1. $\forall i, j \in V : ok(i) \wedge ok(j) \Rightarrow L(i) = L(j)$
2. $L(i) \in V$ (Leader also belongs to the ~~same~~ set of vertices.)
3. $ok(L(i)) = true$

Bully Algorithm →

Assumptions

① Communication links are fault-free

② Process can only fail by stopping

③ failures can be correctly detected using some mechanism like timeout

Messages : Election, reply and leader

Algorithm :

Program bully {program for process i}

Define failed : Boolean {set if the failure of the leader is detected}

L : process {identifies the leader}

m : message {election | leader | reply}

state : idle | wait for reply | wait for leader

initially $\forall i \in V$, state = idle, failed = false

do (failure of L(i) is detected) $\wedge$ (failed = false) $\longrightarrow$
failed = true,

[] failed = true $\longrightarrow \forall j > i$; send election to j,
state = wait for reply,
failed = false;

[] state = idle $\wedge$ (m = election) $\longrightarrow$ send reply to sender,
failed = true;

[] (state = wait for reply) $\wedge$ (m = reply) $\longrightarrow$ state = wait for leader

[] (state = wait for reply) $\wedge$ timeout $\longrightarrow$ L(i) = i, state = idle,
send leader(message) to all.

[] (m = leader) $\longrightarrow$ L(i) = sender, state = idle

[] (state = wait for leader) ∧ timeout ⟶ failed = true,
state = idle.

od

Chang-Roberts Algorithm ⇒ Leader election algorithm for a unidirectional ring.

define token : process id, color ∈ {red, black}

Initially all processes are red and i sends a token <i> to its neighbor

(for a red process i)

do token<j> received ∧ (j<i) ⟶ skip {j's token removed, so j quits}

[] token <j> received ∧ (j>i) ⟶ send <j>, color = black {i quits}

[] token <j> received ∧ (j=i) ⟶ L(i) = i {i becomes the leader}

od

(for a black process)

do token <j> received ⟶ send <j>

od

Concensus and agreement algorithm →.

* commit decision in a database system, commit/reboot

⇒ Computers controlling a spaceship's engine — proceed/abort.

Feasibility of designing algorithm to reach agreement under various system models and failure models. →

Assumptions underlying the study of agreement algorithms

Failure models ⇒ among the n processes in the system at utmost f processes can be faulty

The various failure models – fail-stop and omission and receive omission and Byzantine failure.

* Concensus and Agreement algorithms →

Assumptions underlying the study of agreement algorithms :-

* Failure Models – n processes f-processes.
  ↓
  Failstop, send omission, Receive omission and byzantine failure.
* Synchronous / Asynchronous communication –
  → Impossibility of reaching agreement in asynchronous system in any failure model
  → In a synchronous system, the scenario in which a message has not been sent can be recognised by the intended recepient.
  ↓
  it'll know.
* Network Connectivity –
  The system has full logical connectivity
* Sender identification – A process that receives a message always knows the identity of a sender message.

* channel reliability — The channels are reliable and only the processes may fail (under one of the various failure models).

⇒ Two different kinds of messages — authenticated & non-authenticated messages

Authenticated Vs Non-authenticated messages —

⇒ [we'll be dealing with only non-authenticated messages]

~~Messages~~ with unauthenticated messages, when a healthy process relays messages to the processes

(i) it can forge the message and claim that it was received from another process and

(ii) it can tamper with the contents of a received message ~~and~~ before relaying it, ~~whereas~~ when a process receives a message, it has no way to verify its authenticity.

Unauthenticated message is refered to as — Oral / unsigned message

Agreement variable — may be boolean or multivalued and need not be an integer.

⇒ Byzantine agreement and other problems —

* Byzantine agreement problem ⇒ Requires a designated process called the source process with an initial value to reach an agreement with the other processes about its initial value, subject to the following conditions.

Agreement - All non faulty processes must agree on the same value.

Validity - If the same process in nonfaulty, then the agreed upon value by all the nonfaulty processes must be the same as the initial value of the source.

Termination - Each nonfaulty process must eventually decide on a value.

* Concensus problem → Each process should have an initial value and all the correct processes must agree to a single value

Agreement → All nonfaulty processes must agree on the same (single) value.

Validity → If all the nonfaulty processes have the same initial value, then the agreed upon value by all the nonfaulty processes must be that same value

Termination → Each non faulty process must eventually decide on a value

* The interactive consenting problem - Each process has an initial value and they must agree upon a set of values with one value for each process.

Agreement ⇒ All the non faulty processes must agree on the same array of values $A[v_1, v_2 \ldots v_n]$

Validity ⇒ If process i is non faulty and its initial value is V_i; then all the non faulty processes

agree on v as the i^{th} element of the array A.
If process j is faulty, then the non faulty processes agree on any value ~~in~~ for A[j].

Termination – Each nonfaulty process must eventually decide on the array, A.

Equivalence of the problems and notation :-

⟹ The three problems defined above are equivalent in the sense that a solution to any one of them, can be used as a solution to the other two problems, the equivalence can be shown using a reduction of each problem to the other two problems.

→ If problem A is reduced to problem B, then a solution to problem B can be used as a solution to problem A in conjunction with the reduction.

If P is reducible to P^1 and P^1 is solvable so is P.
If P is reducible to P^1 and P is unsolvable, so is P^1.

– Halting problem (HP) of TM is unsolvable
If follows that every class of P^1 of yes/no problems, if HP is reducible to P^1, then P^1 is unsolvable.

– It is sometimes possible to device a solution to one problem using a solution to another.
– Suppose that there exist solutions to consensus (C), byzantine agreement (BG) and interactive consistency (IC) as follows –

$C_i(v_1, v_2 \ldots v_n)$ returns the decision value of P_i in a run of the solution to consensus problem, where $v_1, v_2, \ldots v_n$ are the values that the processes proposed.

$BG_j(j, V)$ returns the decision value of P_i in a run of the solution to the byzantine problem where P_j, the commander proposes the value V

$j \rightarrow$ commander.

$IC_i(v_1, v_2 \ldots v_n)[j]$ returns the j^{th} value of in the decision vector of P_i in a run of the solution to the interactive consistency problem where $V_1, V_2 \ldots V_n$ are the values that the processes proposed.

IC from BG → We construct a solution to IC from BG by running BG N times, once with each $P_i(i,j)$ acting as commander.

$$IC_i(v_1, v_2, \ldots v_N)[j] = BG_i(j, v_j) \quad (i,j = 1, 2, \ldots N)$$

C from IC → For the case where a majority of processes are correct, we construct a solution to C from IC by running IC to produce a vector of values at each process, then applying an appropriate function on the vector's agreed values to derive a single value.

$$C_i(V_1, \ldots V_N) = \text{majority}\left(IC_i(V_1, \ldots V_N)[1], \ldots IC_i(V_1, \ldots V_N)[N]\right)$$

$\forall i = 1, 2, \ldots N$, where majority is defined as above.
(majority over all ~~values~~ processes)

BG from C – The commander j sends its proposed value V to itself and each of the remaining processes

– All processes run C with the values $V_1, V_2, \ldots V_N$ that they receive (P_j may be faulty).

They derive,

$$BG_j(j, V) = C_i(V_1, \ldots V_N) \quad \forall i = 1, 2, \ldots N$$
for each process.

A solution to one problem $\Longrightarrow$ Solution to other problems.

⇒Agreement in a failure-free system (synchronous/asynchronous system)

In a failure-free system, cancensus can be reached by collecting information from the different processes, assuming at a decision and distributing the decision in the system.

In a synchronous system, in a constant number of rounds.

In an asynchronous system, in a constant number of hops

Agreement in (message passing) synchronous system with failures →

Concensus algorithm for crash failure — Kshemakalyani

↓

other processes need not be knowing, when a process crashes or detected

unlike fail-stop

→ (n) processes in the system, out of which (f) processes may crash ($f < N$)

Concensus variable, x is integer-valued

⇒

(global constants)

integer f ← // maximum no. of crash failures tolerated

(local variables)

integer x ← local variable

(1) Process P_i ($1 \le i \le N$) executes the concensus algorithm for upto f crash failures.

1(a) for round from 1 to f+1 do

1(b) if the constant value of x has not been broadcasted then

1(c) broadcast(x);

1(d) $y_j \leftarrow$ value (if any) received from process j in the round.

1(e) $x \leftarrow \min_{\forall j}(x, y_j)$;

1(f) output x as the concensus value.

The agreement condition is satisfied because in f+1 rounds, there must be atleast one, in which no process failed.

The validity condition is satisfied because processes donot send fictious values in their failure model.

The termination condition is seen to be satisfied.

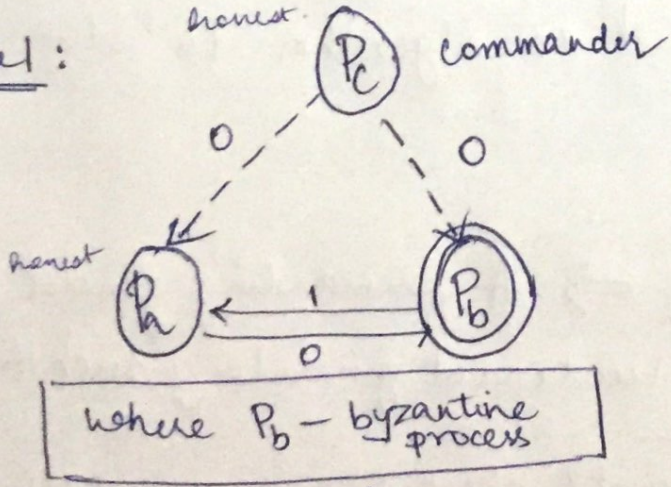
Concensus algorithm for Byzantine failures [Synchronous systems]

upperbound in Byzantine processes =

In a system of n processes, the byzantine agreement problem can be solved in a synchronous system only if the number of byzantine processes f is such that $f \leq \lfloor \frac{n-1}{3} \rfloor$

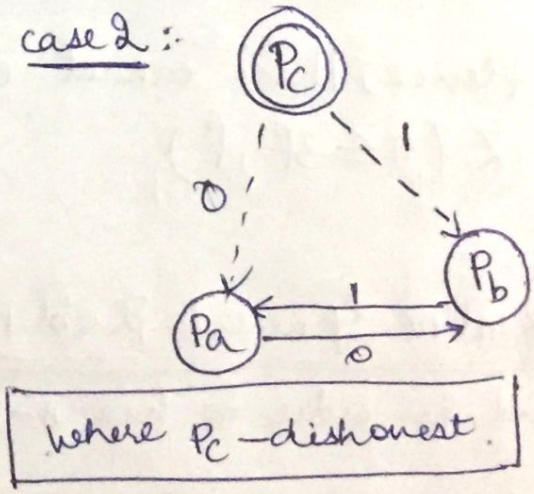
* with $n = 3$ processes, the byzantine agreement problem cannot be solved if the number of byzantine processes, $f = 1$.

case 1:



where P_b – byzantine process

case 2:



where P_c – dishonest.

* P_a receives 0 from P_c, 1 from P_b ⟷ P_a receives 0 from P_c, 1 from P_b

↓

cant decide who is disloyal, because P_a gets the same numbers in both the cases.

BG(3,1) → not solvable, considering the two situations.

↓

can be extended to n processes.

⇓

with n processes and $f > n/3$ processes, the Byzantine agreement cannot be solved, the correctness agreement of this result can be shown using reduction.

Let $Z(3,1)$ denote the byzantine agreement problem for parameters $n=3$ & $f=1$, let $Z(n \leq 3f, f)$ denote the Byzantine agreement problem for parameters $n(\leq 3f)$ and f

A reduction from $Z(3,1)$ to $Z(n \leq 3f, f)$ need to be shown.

→ n processes divided into 3 groups, one group with all the faulty processes.

→ with this reduction in place, if there exists an ~~solution~~ algorithm to solve $Z(n \leq 3f, f)$ i.e to satisfy validity, agreement and termination conditions, then there also exists an algorithm to solve $Z(3,1)$ which has been seen to be unsolvable.

Hence, there cannot exist an algorithm to solve $Z(n \leq 3f, f)$

Byzantine Generals Problem ⇒ A commanding general must send an order to his n−1 lieutenant generals, such that ⇒

IC1: All loyal lieutenants obey the same order

IC2: If the commanding general is loyal, then

every loyal lieutenant obey the order he sends.

IC1, IC2 → interactive consistency condition.

Concensus with oral messages ⇒ The oral messages model satisfies the following conditions ⇒

1. Messages are not corrupted in transit
2. Messages can be lost, but the absence of messages can be detected.
3. When a message is received (or its absence is detected) the receiver knows the identity of the sender (or the defaulter)

Let m be the number of traitors.
An algorithm that satisfies the IC1 and IC2 message model of communication in the presence of at most m iterations and be called an OM(m) algorithm.
The OM(0) algorithm is based on direct communication only.

⇒ OM(m) Algorithm

In order to reach a concensus in the presence of m traitors, $n \geq 3m+1$ generals are modelled. This algo is recursive ⇒

OM(m) → OM(m−1) → OM(m−2)

Algorithm = OM(0) → 0 traitors

1. Commander i sends out a value $v \in \{0,1\}$ to every lieutenant $j \neq i$.
2. Each lieutenant j accepts the value from i as the order from commander i.

Algorithm = OM(m)

1. Commander i sends out a value $v \in \{0,1\}$ to every lieutenant $j \neq i$.
2. If $m > 0$, then each lieutenant j, after receiving a value from the commander starts a new phase broadcasting it to the remaining lieutenants using OM(m−1). In this phase, j acts as a commander.

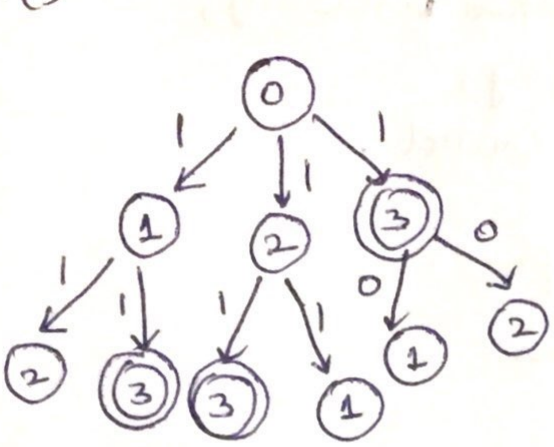
   Each lieutenant thus receives (n−1) values.

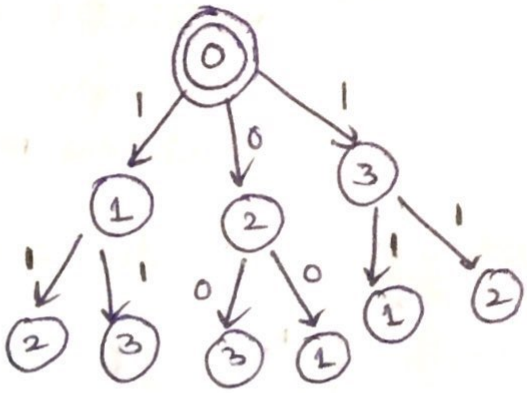
   (a) a value directly received from the commander i of OM(m)

   (b) n−2 values indirectly received from the (n−2) lieutenants resulting from their broadcast of OM(m−1)

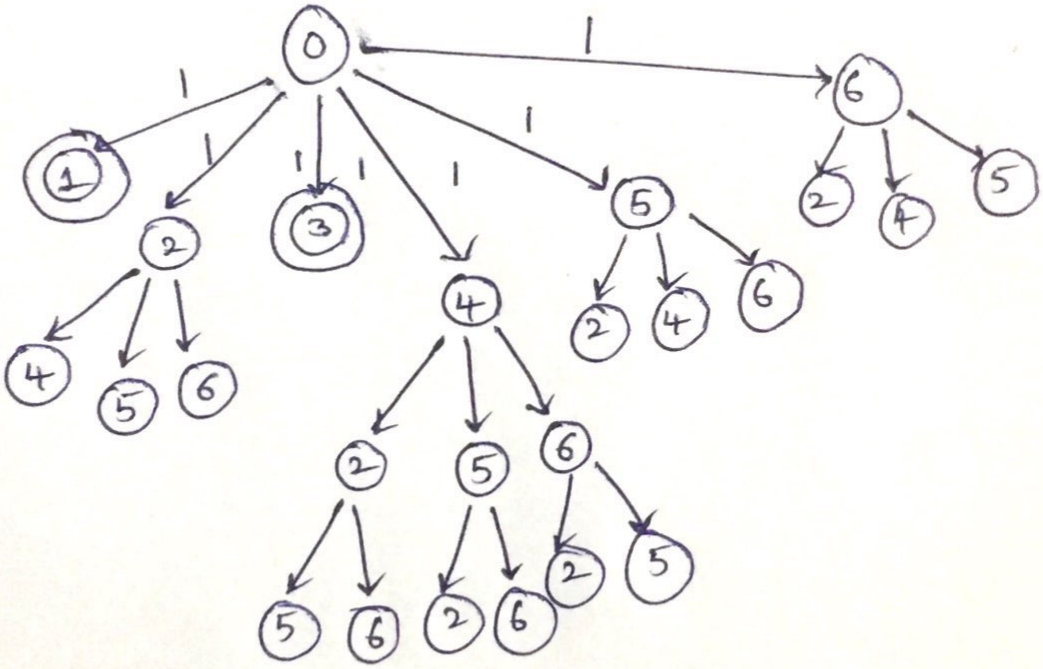
   If a value is not received, then it is substituted by a default value.
3. Each lieutenant chooses the majority of the (n−1) values received by it as the order from the commander.

OM (1)





OM (2)



Distributed Algorithms =

Algorithm → Input / No Input
→ Finiteness
→ Definiteness.

Kepler egraphic processors

→ eglobal State
→ eglobal Time
→ Non-determinism

Application — Telecommunication,
distributed information processing,
scientific computing,
real time process control.

Topology — undirected graph.

3 levels of topology abstraction.
— physical topology
— logical topology
— superimposed topology → for control algorithm execution (ref to as a protocol)

⇒ Distributed algorithm → all processors share computation and communication overhead.

⇒ Centralised algorithm → one or few processes
For ex: client-server systems.

→ Symmetric and asymmetric algorithms

Symmetric ↓
all the processes do similar kind of work.

asymmetric
— centralised algorithm
(few processes do majority of the work compared to other processes)

→ Anonymous algorithm → An anonymous system, in which ~~process/processor code~~ doesn't involve process (IDs) numbers or processor no's to make any execution decisions in the distributed algorithm.

→ structural elegance
→ hard to design

→ An anonymous algorithm is the algorithm which runs as an anonymous system and therefore doesn't use process identifiers or processor identifiers in the code.

Uniform Algorithm → A uniform algorithm does not use n, the number of processes in the system as parameters in its code.

↳ Scalability transparency

Adaptive Algorithm ⟹ Consider the context of problem x, in a system with n nodes, let k ($k \leq n$) be the number of nodes participating in the content of x. If the complexity of the algorithm can be expressed in terms of k rather than, in terms of n, the algorithm is adaptive.

Deterministic vs Non-deterministic Executions. ⟹

Deterministic receive → process knows the sender process.

Ily Nondeterministic receive → process doesnt know the sender process.

Execution inhibition ⇒

- Few protocols may affect the performance [for eg: in sending event delays or receive delays...]
- Blocking communicating primitives, freeze the local execution.
- Non blocking protocols, inhibitory or freezing protocols.

- Globally inhibitory
- Send/Receive inhibitory
- Internal event inhibitory

Synchronous and Asynchronous systems =

↳ Known upper bound on the message communication delay.

→ Known bounded drift rate for the clock

→ Known upper bound to execute a logical step in the execution.

⇒ Asynchronous systems will not satisfy these three conditions.

* Online vs Offline algorithm — depends on data.

↓ unless entire data is available, execution doesnt happen

executed as data is generated.

+ Failure models =

• t-fault tolerant — No. of components failed ≤ t

[processors or communication channels]

Process failure models — Failstop — Processor may stop at a particular point, and this will be known to other processes also

→ Crash failures.
→ Receive omission
→ Send omission
→ General omission

→ Byzantine failure or malicious failure with authentication

⇒ receiving process knows the senders signature and can verify

signature — cannot be manipulated

→ Byzantine or malicious failures.

↳ in the case of manipulated messages, without authentication
or cooked up — we cannot do anything

→ Communication failure models = crash failures
omission failures
byzantine failures.

* Wait
[redacted]-free algorithms → each process has an upper bound

— resilient (continue to execute inspite of other processes)

— fault-tolerant systems.

* Communication channels — FIFO & NON FIFO.

* Complexity measures and metrics —

→ time and space complexity. — lower bounds (Ω, ω)
upper bounds (O, o)
exact bound (Θ)

→ message complexity
→ space-complexity per node.
→ system-wide space complexity.
→ Time complexity per node
→ system-wide time complexity.
→ Message complexity — number of messages,
size of messages,
message time complexity

Program structure — Hoore — CSP — concurrent sequential ~~problems~~ proces

proposed by

$$*\,[G_1 \to CL_1 \,\|\, G_2 \to CL_2 \,\|\, \ldots \,\|\, G_k \to CL_k]$$

guard = boolean value — if true, then executed
if all are false, then termination takes places

Synchronous single-initiator spanning tree algorithm using flooding →

Synchronous BFS spanning tree algorithm -

The code shown is for process, P_i — $\forall i \leq n$

(local variables)

int visited, depth ← 0

int parent ← ⊥

set of int neighbors ← set of neighbors.

(message type)

QUERY.

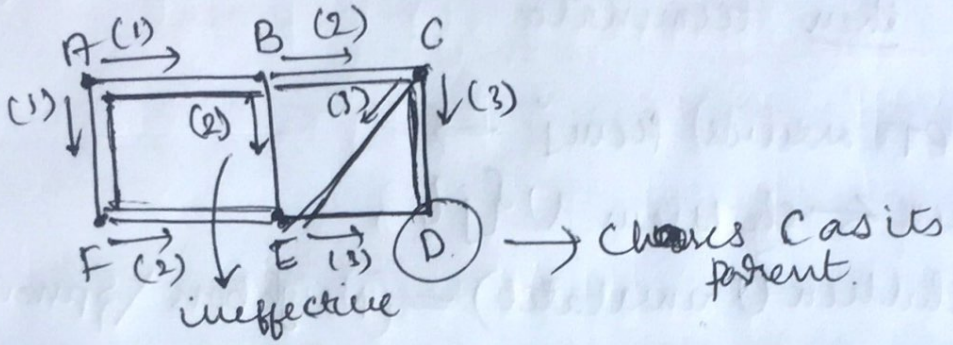
if i = root then
    visited ← 1
    depth = 0
    send QUERY to Neighbors.

for round = 1 to diameter do
    If visited = 0
        if any QUERY message arrives then
            parent ← randomly select a node from which QUERY was received

visited ← 1
depth ← round
send QUERY to Neighbors \ {Senders of QUERY in the round}

delete any QUERY message that is received in the round.

Example :-



→ Asynchronous single-initiator spanning tree algorithm →

Code for process Pi ; $1 \le i \le n$

(local variables)

int parent ← ⊥

Set of int children, unrelated ← ϕ

Set of int Neighbor ← set of neighbors.

(message type)

QUERY, ACCEPT, REJECT

⇒ When the predesignated root node wants to initiate the algorithm.

if (i == root and parent = ⊥) then

send QUERY to all neighbors.

parent = i

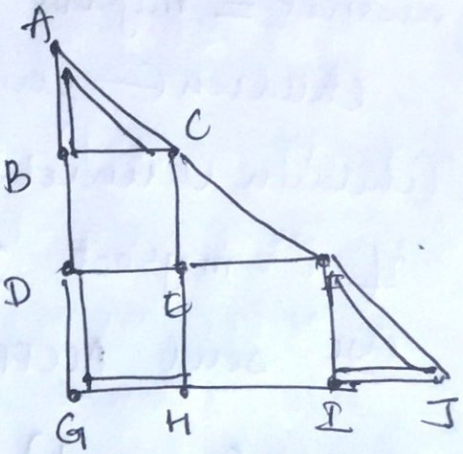
⇒ When QUERY arrives from j :

if parent = ⊥ then
    parent = j
    send ACCEPT to j
    send QUERY to all neighbors except j
    if (children ∪ unrelated) = (neighbors \ {parent})
        ↳ exclude parent
    then terminate

when ACCEPT arrives from j →
    children ← children ∪ {j}
    if (children ∪ unrelated) = (Neighbors \ {parent})
    then terminate.

when REJECT arrives from j ⟹
    unrelated = unrelated ∪ {j}
    if (children ∪ unrelated) = (Neighbors \ {parent})
    then terminate

Asynchronous concurrent - initiator spanning tree algorithm using flooding



(local variables)

int parent, myroot ← ⊥

set of int children, unrelated ← ϕ

set of int Neighbors ← set of Neighbors

(message types)

QUERY, ACCEPT, REJECT

⇒ when the node wants to initiate algorithm as a root

if (parent = ⊥) then send QUERY(i) to all Neighbors.

parent, myroot ← i

⇒ when QUERY (newroot) arrives from j

if myroot < newroot

then // discard partial execution due to its low priority

parent ← j, myroot ← newroot.

children, unrelated ← ϕ

send QUERY (newroot) to all neighbors except j.

if Neighbors = {j} then

send ACCEPT (newroot) to j

terminate ____? ____ node.

else send REJECT (newroot) to j;

// comments -

if newroot = myroot, then parent is already identified

:

When ACCEPT(newroot) arrives from j:

if newroot = myroot then

children ← children ∪ {j}

if (children ∪ unrelated) = Neighbors \ {parent} then

if i = myroot then terminate

else send ACCEPT(myroot) to parent.

When REJECT(newroot) arrives from j:-

if newroot = myroot then

unrelated = unrelated ∪ {j}

if (children ∪ unrelated) = (Neighbors \ {parent})

then i = myroot then terminate

else

send ACCEPT(myroot) to parent.

// if newroot < myroot then ignore the message.

// newroot > myroot → will never occur.

Asynchronous concurrent-initiator DFS spanning tree algorithm -

Algorithm :-

(local variables)

int parent, myroot ← ⊥

set of integers children ← ϕ ;

set of integers Neighbors, unknown ← set of neighbors.

(message types)

QUERY, ACCEPT, REJECT.

When the node wants to initiate the algorithm as a root,

if (parent = ⊥) then send QUERY(i) to i (itself)

When QUERY (newroot) arrives from j -

if myroot < newroot then

parent ← j ;

myroot ← newroot

unknown ← set of neighbors.

unknown ← unknown \ {j} // excluding j.

if unknown ≠ 0 then

delete some x from unknown

send QUERY(myroot) to x,

if unknown = 0 then

send ACCEPT (myroot) to j ;

else if myroot = newroot then

Send REJECT to j

// if newroot < myroot, ignore the query

When ACCEPT (newroot) or REJECT (newroot) arrives from j:

if newroot = myroot then

if ACCEPT arrived then

children ← children ∪ {j}

if unknown = 0 then

if parent ≠ i then

send ACCEPT(myroot) to parent.

else if parent = i then

set i as the root

terminate.

delete some x from unknown

send QUERY (myroot) to x.

// if newroot < myroot, ignore the query since

// sending QUERY to j, i has ~~to~~ updated its myroot to j

// will update its myroot to a higher root identifies

// when it a QUERY initiated by it

// newroot > myroot, will never occur.

The Bellman-Ford Shortest path algorithm ⇒

It performs (n-1) times, the relaxation of every edge in the graph.

Input – A weighted directed graph $\vec{G}$ with n vertices and v of $\vec{G}$.

output – A label D[u] for each vertex u of $\vec{G}$ such that D[u] is distance from v to u

```
D[v] ← 0.
for each vertex u ≠ v of G do
        D[u] = +∞.

for i ← 1 to n-1 do                    emerging.
  for each (directed) edge (u,z) from u do
    {perform the relaxation operation on (u,z)}
    if D[u] + w(u,z) < D[z] then
        D[z] ← D[u] + w(u,z).

if there are no edges left with potential relation operation
then return the label D[u] of each vertex u.

else return G contains a negative weight cycle
```

Single Source shortest path algorithm : Syncheonous Belleman-Ford

weighted graph : undirectional links

(local variables)

int length ← ∞

int parent ← ⊥

Set of int Neighbors ← set of neighbors.

Set of int $\{weight_{i,j}, weight_{j,i} \mid j \in Neighbor\}$ ← the known values of the weights of incident links.

(message types)

UPDATE

if $i = i_0$ then length ← 0.

for round $i = 1$ to $n-1$ do

send UPDATE (i, length) to all neighbors.

await UPDATE (j, length) from each $j \in$ neighbors.

for each $j \in$ Neighbors do

if (length > length$_j$ + weight$_{i,j}$) then

length ← length$_j$ + weight$_{j,i}$;

parent = j

→ Source is i_0

→ Code is for process i, P_i $1 \le i \le n$

Single source shortest path algorithm : Asynchronous Bellman-ford

negative cycles → process just goes on.
→ also doesnt allow negative ~~cycles~~ weight cycles.

(local variables)
int length ← ∞
set of int Neighbors ← set of neighbors.
set of int {weight$_{i,j}$, weight$_{j,i}$} ← known values of the weights of incident links.

(message types)
UPDATE.

if $i = i_0$ then length ← 0;
send UPDATE (i_0, 0) to all neighbors;
terminate;

when UPDATE (i_0, length$_j$) arrives from j —
if (length$_i$ > length$_j$ + weight$_{j,i}$) then
length ← length$_j$ + weight$_{j,i}$;
parent ← j
send UPDATE (i_0, length) to all neighbors

→ Depends on the number of hops

algorithm will definitely terminate

Distributed file systems

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network.

[Book - Coulouris]

→ Performance and reliability experienced for access to files stored at a server should be comparable to that of for files stored on local disks.

(emulates unix file systems at different levels of scaling & performance)

→ Sharing of resources ⟹ Key goal for distributed systems.
→ Sharing of stored information → the most important aspect of distributed resource sharing.

→ Design of large scale - wide area read-write file storage systems poses problems of load balancing (how servers might be frequently requested), reliability, availability and security.

Basic distributed file system -

→ Objective: Emulate the functionality of a non distributed file system.

Storage systems and their properties →

~~Sharing, Persistence, Distributed cache/replicas, Consistency maintenance~~

~~Main Memory~~

	Sharing	Persistence	Distributed cache/replicas	Consistency Maintenance	Example
Main Memory	×	×	×	!	RAM
File System	×	✓	×	!	UNIX File System
Distributed File System (address space is not the same)	✓	✓	✓	✓ (slightly weaker)	SUN NFS
Web	✓	✓	✓	× each client may modify - it can be modified multiple times	Web Server
Distributed Shared Memory (address space is the same) - memory is divided into N blocks for N processes	✓	×	✓	✓	

Each processor → one cache. ⟹ (so we need) cache coherent protocols

↓ can be accessed by multiple processes

→ if cache is modified, other processes have to be informed.

Remote objects → RMI/ORB	✓	×	×	1	CORBA
Persistent object store.	✓	✓	×	1	CORBA Persistence State Service
Peer-to-peer storage system	✓	✓	✓	2 (weaker)	

Types of consistency

1 : strict one copy

✓ : slightly weaker guarantees.

2 : considerably weaker guarantees

Characteristics of the file system →

* responsible for the organisation, storage, retrieval, naming, sharing and protection of files.

– They provide a programming interface

File attribute record structure →

→ File length
→ Creation timestamp
→ Read timestamp
→ write timestamp
→ Attribute timestamp
→ Reference count

} managed by file system.

→ Owner
→ file type
→ Access control list

} Managed by user programs.

File System modules : (for nondistributed file system)

- Directory Module — relates file names to file IDs.
- File Module — relates file IDs to particular files.
- Access control module — checks permission for operation requested.
- File Access module — reads or writes file data or file attributes
- Block Module — accesses and allocates disk blocks.
- Device Module — performs disk IO and buffering

Distributed file Systems →

Requirements :-

- Initially, they offered "access transparency" and "location transparency".
- Later performance, scalability, concurrency control, fault tolerance and security were added.

(F) Transparency — The design must balance the flexibility and scalability that derive from transparency against software complexity and performance.

1. Access transparency — client programs should be unaware of the distribution of files.

2. — Location transparency — client programs should see a uniform file namespace (may be located in diff servers, diff places)

3. — Mobility transparency — Neither client programs nor system administration tables in client nodes need not be changed when the files are moved.

4. — Performance transparency — Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

5. — Scaling transparency — The service can be expanded by incremental growth to deal with a wide range of nodes and network sizes.

(2) Concurrent file updates —