Ankush Khulia
2014CS50279

Date 12 | 10 | 18

COL 830 - Homework 2

Problem 1 - Distributed Mutual Exclusion

(1) • In Rickart-Agrawala's algorithm, critical sections are accesse according to the increasing order of timestamps. We prove this using contradiction

→ Assume the contrary to above, i.e. suppose process p_1 issues a request to enter critical section at time t_1 & process p_2 does so at time t_2 with $t_1 < t_2$ & p_2 enters CS first.

→ From algorithm, p_2 enters CS ⇒ p_2 received all reply msgs ⇒ p_2 receives reply msg from p_1

⇒ p_1 reply to p_2's CS request → (p_1 not requesting CS) V (t_2 <

- Since p_1 requested CS, (p_1 not req. CS) is false
- From Modus ponens, if assumption is true & assumption → (t_2 < then $t_2 < t_1$. But this is a contradiction! as $t_1 < t_2$ was the assumption (part of it)

∴ ¬(Assumption) is true always & hence CS are accessed in incr. order of timestamps

(2) Modified Richart Agrawala algorithm

→ Each process P_i keep some additional state

$pred_i$ ← process id of immediate predecessor (ie. executes CS just before that / priority of req. is just higher than P_i)

$next_i$ ← process id of imm. successor

→ Requesting CS as P_i

(a) Broadcast (REQ) message to all processes

(b) When P_j recv REQ msg.

- If $t_{pred} < t_i$ & $(t_i < t_j)$ (t_i is timestamp of REQ)

  $t_{pred} = t_i$; $pred_j = i$

- If (P_j req. CS) & $(t_i > t_j)$ & $(t_i < t_{next})$

  $t_{next} = t_i$; $next_j = i$

→ Executing CS for P_i

If [(P_i recv. REPLY from pred) & (pred != null)] OR [(P_i recv. REPLY from all proc.) & (pred = null)]

then execute (CS.i)

→ Releasing CS (for P_i)

send REPLY to $next_j$, reset $t_{next} = INT\_MIN$

Date 12 | 10 | 18

→ Correctness :

(i) '$pred_i$' has process id of process which has requested the last amongst all processes that requested before P_i. Thus $pred_i$ gives process with priority just higher than i's req.

(ii) '$next_i$' has process id of process which has requested the CS after i's request. It is the process with highest priority / least timestamp amongst all such processes. Since $t_i < t_{next}$, i's priority just preceeds $next_i$ & so it will be waiting for reply to request from i only.

(iii) Initially, if no one makes requests & only i does, it will need a reply from all processes, even if someone requests CS, their req. time stamp can be higher, in which case it becomes successor, or lower, then it can become predecessor & i will need a reply.

(iv) Since this only modifies no. of message, the earlier proof of access to CS in increasing order of timestamps, combined with (iii) also works here. Thus fairness is ensured.

(v) Since fairness enforces an order of access of CS & since 2 requests can't have same priority (tie-break using proc. id) & proc can't enter CS simultaneously, ensuring correctness.

(vi) Message complexity is $O(N-1) + 1 \approx O(N)$ (B.cast, to $next_i$)

(3) In Lamport's algorithm, REPLY msg serves to flush all request messages in the channel with smaller timestamps. If P_i has recvd. REPLY from P_j, it has also recvd. all requests made by j of smaller timestamp & can decide which of these requests, or its own have higher priority & execute CS first.

Date 12 | 10 | 18

(a) For each process in rectangular grid P_{ij}, denoting the i^{th} row, j^{th} column (of numCol columns & numRow rows)

$R_{ij} = \{P_{ik}\}_{k=1}^{numCol} \cup \{P_{kj}\}_{k=1}^{numRows}$, clearly $P_{ij} \in R_{ij}$

- $R_{ij} \cap R_{i'j'} \neq \phi$, $= \{P_{ij'}, P_{i'j}\}$
- $\forall\ i,j,i',j'$ $R_{ij} \not\subseteq R_{i'j'}$ since (i,j) uniquely determine a proc
- $|R_{ij}| =$ numRows + numCols $- 1 = K\ \forall\ i,j$
- $\forall\ P_{ij}$, $R^{ij} = \{R_{i'j'} \mid P_{ij} \in R_{i'j'}\}$ $|R^{ij}| =$ numRow + numCol $- 1$ i.e. P_{ij} appears in exactly K req. sets
- Given this new R_i set satisfies properties, we can run Maekawa's algorithm for mutual exclusion as it is, resulting in
- Achieving mutual exclusion - proof by contradiction

if P_{ij} & P_{kl} are in CS → both received reply from R_{ij} & R_{kl} respectively → $r \in R_{ij} \cap R_{kl}$ replied to requests made & since $R_{ij} \cap R_{kl} \neq \phi$, a process replied to 2 processes → A contradiction

- 3K messages - K REQ, K REPLY & K RELEASE with K = numRows + numCols $- 1$
- Problem of Deadlocks - consider P_{ij} & P_{kl} req. CS & P_{il} replies to P_{ij} & P_{kj} replies to P_{kl} ($\{P_{il}, P_{kj}\} = R_{ij} \cap R_{kl}$) Clearly there is a deadlock
- Deadlocks can be handled like by Maekawa's extensions by using additional msg. types FAILED, INQUIRE & YIELD

So msg. complexity ~ 5 $3k - 6K$

Problem 2 - Global Snapshots

(1) We can modify Chandy-Lamport's ~~snapshot~~ snapshotting algo. as follows

→ Marker sending Rule → same as original

→ Marker Receiving Rule → for process j P_j

On receiving marker along channel C

If j hasn't recorded its state then

- Record the state of C = {ϕ}
- Follow 'Marker sending Rule'

Else

let older snapshot = OS_j

- Make a new snapshot by combining OS_j + current state where current state is achieved after receiving all the messages, since last snapshot (OS_j)
- Record ~~then~~ state of C = {ϕ}

→ Termination - ~~If~~ P_i if it has seen marker along each incoming channel, it can exchange local snapshot w/ neighbours to aggregate or send it to its parent, which will do that.

(2) Lai-Yang's algorithm, white ≤ red. Now consider events s, t

⇒ s ⟶ t. Possible cases are

→ s, t on same process, so s can be white/red & t also white/red but if t is white, s can't be red ⇒ s.color ≤ t.color

→ s = send t = matching Recv If t is red s.color ≤ t.color

If t is white & s is white, no issue s.color = t.color

~~If~~ t is white & s is red isn't possible as t's process turns red on receipt of red msg (s) & so t.color = red

- $\exists u \Rightarrow s \rightarrow u \rightarrow t$. From above 2 cases

  $s.color \leq u.color$ & $u.color \leq t.color \Rightarrow s.color \leq t.color$

- $\exists u_1, u_2 \ldots u_n \Rightarrow s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_n \rightarrow t$

  By an inductive argument & transitivity of $\leq$, $s.color \leq t.color$

Problem 3 - Termination

(i) Assuming a controlling agent monitors computation & sends only control messages to processes, we can adapt the weight throwing algorithm to become message-loss tolerant as follows -

- Rule 1 - same as given in Kshemkalyani (controller throwing a part of its weight to a process via basic msg). - P.T.O.
- Rule 2 - on Receipt of message basic(DW) at proc j from i

  If i has already sent basic(DW) i.e. j received same timestamp on a basic message from i w/ same weight

  then send ACK(timestamp(basic(DW))) to i

  Else

  $W_j := W_j + DW$

  If P_j is idle, P_j becomes active

  send ACK(timestamp(basic(DW))) to i
- Rule 3 - Process i becomes idle from active state by sending control message (DW = W_i) to controlling agent

  Wait(ACK(C(DW)));

  If (timeout) resend C(DW) to controller & wait();

  If Recv(ACK(C(DW))) from controller

  $W_i = 0$, change state to idle
- Rule 4 - on Recv C(DW), controller $W_c := W_c + DW$

  If $W = 1$, termination

Date 12 | 10 | 18

To expand Rule 1 →

Send (b(DW)) to a process P_i

Wait (ACK (b(DW))).

If (timeout), resend b(DW) to P_i & wait

If Recv (ACK (b(DW))) then

$W_c := W_c - DW$

(2) We can modify Huang's weight throwing algorithm to use counte

→ For each processe & controller, we keep one counter

- Decrement counter when (basic) message is sent to process
- Increment counter when (control) message is received from proces

→ If $\forall i$ min (counter(P_i), $\underbrace{\text{controller}}_{\text{always min}}$) $= -k$, then the algorith is essentialy the weight throwing algorithm, with $w = k$

→ We can argue termination in a similar manner, if at a process P_i, counter $= 0$, then either process has not started yet (since on receipt of messages, it becomes active & counter > 0), or has moved its work to its children (+ve counter reduced to zero after sends) or has terminated & given it to controller

If the controller process' counter $= 0$, then algo may have not started yet, or has finished (equivalent to $w = 1$)