Distributed Deadlock Detection

CS60002: Distributed Systems

Pallab Dasgupta
Professor,
Dept. of Computer Sc. & Engg.,
Indian Institute of Technology Kharagpur

Preliminaries

- The System Model
  - The system has only reusable resources
  - Processes are allowed only exclusive access to resources
  - There is only one copy of each resource

- Resource vs. Communication Deadlocks
- A Graph-Theoretic Model
  - Wait-For Graphs

Deadlock Handling Strategies

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

Issues in Deadlock Detection & Resolution

- Detection
  - Progress: No undetected deadlocks
  - Safety: No false deadlocks
- Resolution

Control Organization for Deadlock Detection

- Centralized Control
- Distributed Control
- Hierarchical Control

Centralized Deadlock-Detection Algorithms

- The Completely Centralized Algorithm
- The Ho-Ramamoorthy Algorithms
    - The Two-Phase Algorithm
    - The One-phase Algorithm

Distributed Deadlock-Detection Algorithms

- **A Path-Pushing Algorithm**
  - The site waits for deadlock-related information from other sites
  - The site combines the received information with its local TWF graph to build an updated TWF graph
  - For all cycles 'EX -> T1 -> T2 -> Ex' which contains the node 'Ex', the site transmits them in string form 'Ex, T1, T2, Ex' to all other sites where a sub-transaction of T2 is waiting to receive a message from the sub-transaction of T2 at that site

Chandy et al.'s Edge-Chasing Algorithm

To determine if a blocked process is deadlocked

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 upon P_j, and

(b) P_j is waiting on P_k, and

(c) P_j and P_k are on different sites,

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

Algorithm Contd..

On the receipt of *probe (i, j, k)*, the site takes the foll. actions:

if (a) P_k is blocked, and

(b) *dependent$_k$(i)* is false, and

(c) P_k has not replied to all requests of P_j,

then begin

dependent$_k$(i) = true;

if k = i then declare that P_i is deadlocked

else for all P_m and P_n such that

(i) P_k is locally dependent upon P_m, and

(ii) P_m is waiting on P_n, and

(iii) P_m and P_n are on different sites,

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

end.

Other Edge - Chasing Algorithms

- The Mitchell – Merritt Algorithm

- Sinha – Niranjan Algorithm

Chandy et al.'s Diffusion Computation Based Algo

- **Initiate a diffusion computation for a blocked process P_i:**

  send query (i, i, j) to each process P_j in the

  dependent set DS_i of P_i;

  $num_i(i) := |DS_i|$; $wait_i(i) := true$

- **When a blocked process P_k receives a query (i, j, k):**

  if this is the engaging query for process P_k then

  send *query* (i, k, m) to all P_m in its dependent set DS_k;

  $num_k(i) := |DS_k|$; $wait_k(i) := true$

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

Chandy et al.'s Algo. Contd.

- When a process P_k receives a reply (i, j, k):

  if $wait_k(i)$ then begin $num_k(i) := num_k(i) - 1;$

  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

A Global State Detection Algorithm

***$wait_i$* : boolean (:= false)   /* records the current status */**

***t_i* : integer (:= 0)   /* current time */**

***in (i)* : set of nodes whose requests are outstanding at i**

***out (i)* : set of nodes on which i is waiting**

***p_i* : integer (:= 0) /* number of replies required for unblocking */**

***w_i* : real (:= 1.0) /* weight to detect termination of deadlock detection algorithm */**

A Global State Detection Algorithm

- **REQUEST_SEND (i):**

  /*executed by node i when it blocks on a p_i-out of-q_i request */

  For every node j on which i is blocked do

  out (i) ← *out (i) U {j};* send REQUEST (i) to j;

  set p_i to the number of replies needed; $wait_i$:= true

- **REQEST_RECEIVE (j):**

  /* executed by node i when it receives a request made by j */

  in (i) ← *in (i) U {j};*

- **REPLY_SEND (j):**

  /* executed by node i when it replies to a request by j */

  in (i) ← *in (i) - {j};* send REPLY (i) to j;

A Global State Detection Algorithm (Contd..)

- **REPLY_RECEIVE (j):**

  /*executed by node i when it receives a reply from j to its request

  if valid reply for the current request then begin

  out (i) $\leftarrow$ *out (i) – {j}*; $p_i \leftarrow p_i - 1$;

  if $p_i = 0 \rightarrow$

  { $wait_i \leftarrow false$;

  For all k $\in$ *out (i)*, send *CANCEL (i) to k*;

  out (i) $\leftarrow \Phi$ }

  end

- **CANCEL_RECEIVE (j):**

  /* executed by node i when it receives a cancel from j */

  if $j \in in(i)$ then $in(i) \leftarrow in(i) - \{j\}$;

The Algorithm

- FLOOD, ECHO and SHORT control messages use weights (for termination detection).
- Data structures:
  - LS: array [1..N] of record consisting of:
  - LS[init].out /* nodes on which *i* is waiting in snapshot */
  - LS[init].in /* nodes waiting on *i* in the snapshot */
  - LS[init].t /* time when *init* initiated snapshot */
  - LS[init].s /* local blocked state as seen by snapshot */
  - LS[init].p /* value of p_i as seen in snapshot */

The Algorithm

- The distributed WFG is recorded using FLOOD messages in the outward sweep and is examined for deadlocks using ECHO messages in the inward sweep
  - Blocked nodes propagate the FLOOD
  - Active nodes initiate reduction with ECHO messages
- A node is reduced if it receives ECHOs along p_i out of its q_i outgoing edges
- When an ECHO arriving at a node does not unblock the node, its weight is sent directly to the initiator using a SHORT message
- If initiator is not reduced but termination is detected, then we have a deadlock

The Algorithm

- **SNAPSHOT INITIATE**

/* Executed by node *i* to detect whether it is deadlocked */

init ← *i* ;

w_i ← *0;*

LS[init].out ← out(i) ;

LS[init].in ← 0;

LS[init].t ← t_i ;

LS[init].s ← *true* ;

LS[init].p ← p_i ;

send FLOOD(*i, i,* t_i, *1* / |*out(i)*|) to each *j* in *out(i)*.

The Algorithm

FLOOD_RECEIVE(*j, init, t_init, w*)

/* Executed by node *i* on receiving a FLOOD message from *j* */

LS[init].t < t_init $\wedge$ j $\in$ *in(i)* → /* valid FLOOD, new snapshot */

LS[init].out ← out(i) ; LS[init].in ← { *j* };

LS[init].t ← *t_init* ; LS[init].s ← $wait_i$;

$wait_i$ = *true* →

LS[init].p ← p_i ;

send FLOOD(*i, init, t_init, w* / |*out(i)*|) to each *k* in *out(i)*.

$wait_i$ = *false* →

LS[init].p ← *0* ;

send ECHO(*i, init, t_init, w*) to *j*.

LS[init].in ← LS[init].in – { *j* }

The Algorithm

FLOOD_RECEIVE(*j, init, t_init, w*) /* Contd. */

LS[init].t < t_init ∧ j ∉ *in(i)* → /* invalid FLOOD, new snapshot */

send ECHO(*i, init, t_init, w*) to *j*.

LS[init].t = t_init ∧ j ∉ *in(i)* → /* invalid FLOOD, curr snapshot */

send ECHO(*i, init, t_init, w*) to *j*.

LS[init].t = t_init ∧ j ∈ *in(i)* → /* valid FLOOD, current snapshot */

LS[init].s = *false* →

send ECHO(*i, init, t_init, w*) to *j* ;

LS[init].s = *true* →

LS[init].in ← LS[init].in U { *j* } ;

send SHORT(*init, t_init, w*) to *init.*

The Algorithm

ECHO_RECEIVE(*j, init, t_init, w*)

LS[init].t > t_init → discard the ECHO message

LS[init].t < t_init → cannot happen – echo for unseen snapshot

LS[init].t = t_init → /* ECHO for current snapshot */

LS[init].out ← LS[init].out – { *j* } ;

LS[init].s = *false* → send SHORT(*i, init, t_init, w*) to *init* ;

LS[init].s = *true* →

LS[init].p ← LS[init].p – 1 ;

LS[init].p = 0 →

LS[init].s ← false ;

init = i → declare not deadlocked; exit;

send ECHO(*i, init, t_init, w* / |*LS*[init].in|) to *k* $\in$ LS[init].in

LS[init].p $\neq$ 0 → send SHORT(*i, init, t_init, w*) to *init* ;

The Algorithm

SHORT_RECEIVE(*init, t_init, w*)

$t\_init < t\_block_i$ → discard the message (outdated)

$t\_init > t\_block_i$ → not possible

$t\_init = t\_block_i \wedge LS[init].s = false$ → discard

$t\_init = t\_block_i \wedge LS[init].s = true$ →

$w_i \leftarrow w_i + w$;

$w_i = 1$ → declare deadlock and abort.