06-06798 Distributed Systems

Lecture 11: Co-ordination algorithms

20 February, 2002 1

Overview

- Co-ordination in distributed systems
  - why needed, sources of problems
- Mutual exclusion
  - ring-based
  - multicast & logical clocks [Ricart and Agrawala]
- Leader election
  - ring-based [Chang&Roberts]
  - randomised [Itai&Rodeh]
  - tree identify [FireWire IEEE 1394]

20 February, 2002 2

Co-ordination algorithms

are fundamental in distributed systems:

- for resource sharing: concurrent updates of
  - records in a database (record locking)
  - files (file locks in stateless file servers)
  - a shared bulletin board
- to agree on actions: whether to
  - commit/abort database transaction
  - agree on a readings from a group of sensors
- to dynamically re-assign the role of master
  - choose primary time server after crash
  - choose co-ordinator after network reconfiguration

20 February, 2002 3

Why difficult?

- Centralised solutions not appropriate
  - communications bottleneck
- Fixed master-slave arrangements not appropriate
  - process crashes
- Varying network topologies
  - ring, tree, arbitrary; connectivity problems
- Failures must be tolerated if possible
  - link failures
  - process crashes
- Impossibility results
  - in presence of failures, esp asynchronous model

20 February, 2002 4

Co-ordination problems

See [Lynch 1996] for more details.

- Mutual exclusion
  - distributed form of critical section problems
  - must use message passing
- Leader elections
  - after crash failure has occurred
  - after network reconfiguration
- Consensus (also called Agreement): next lecture
  - similar to coordinated attack
  - some based on multicast communication
  - variants depending on type of failure, network, etc

20 February, 2002 5

Failure assumptions

Assume reliable links, but possible process crashes.

- Failure detection service
  - provides query if a process has failed
  - how?
    - processes send ‘P is here’ messages every T secs
    - failure detector records replies
  - unreliable, especially in asynchronous systems
- Observations of failures:
  - Suspected: no recent communication, but could be slow
  - Unsuspected: but no guarantee it has not failed since
  - Failed: crash has been determined

20 February, 2002 6

Distributed mutual exclusion

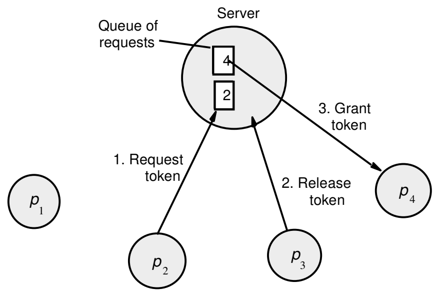
- The problem:
  - N asynchronous processes, for simplicity no failures
  - guaranteed message delivery (reliable links)
  - to execute critical section (CS), each process calls:
    - **enter**()
    - **resourceAccess**()
    - **exit**()
- Requirements

  (**MC1**) At most one process is in CS at the same time.

  (**MC2**) Requests to **enter** and **exit** are eventually granted.

  (**MC3** - Optional, stronger) Requests to **enter** granted according to causality order.

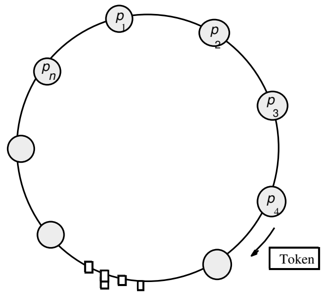
Centralised mutual exclusion



Centralised service?

- Single server implements imaginary token:
  - only process holding the token can be in CS
  - server receives **request** for token
  - replies **grant**ing access if CS free; otherwise, request queued
  - when a process **release**s token, oldest request from queue granted
- It works though...
  - does not respect causality order of requests (MC3) - why?
- but
  - server is performance bottleneck!
  - what if server crashes?

Ring-based algorithm



Arrange processes in a logical ring, let them pass token.

Ring-based algorithm

- No server bottleneck, no master
- Processes:
  - continually pass token around the ring, in one direction
  - if do not require access to CS, pass on to neighbour
  - otherwise, wait for token and retain it while in CS
  - to exit, pass to neighbour
- How it works
  - continuous use of network bandwith
  - delay to enter depends on the size of ring
  - causality order of requests not respected (MC3) - why?

Ricart&Agrawala algorithm

- Based on multicast communication
  - N inter-connected asynchronous processes, each with
    - unique id
    - Lamport's logical clock
  - processes multicast request to enter:
    - timestamped with Lamport's clock and process id
  - entry granted
    - when all other processes replied
    - simultaneous requests resolved with the timestamp
- How it works
  - satisfies the stronger property (MC3)
  - if hardware support for multicast, only one message to enter

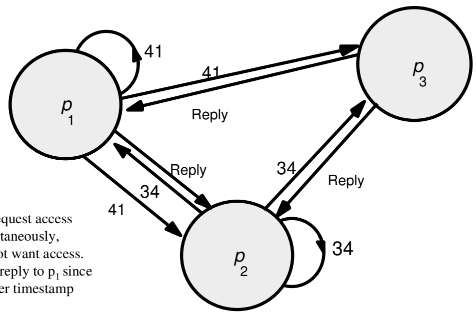
Ricart&Agrawala algorithm

On initialization
state := RELEASED;
*To **enter the critical section***
state := WANTED;
Multicast *request* to all processes;
T := request's timestamp;
Wait until (number of replies received = $(N - 1)$);
state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
if (*state* = HELD) or ((*state* = WANTED) *and* $((T, p_j) < (T_i, p_i))$)
then
queue *request* from p_i without replying;
else
reply immediately to p_i;
end if
*To **exit the critical section***
state := RELEASED;
reply to any queued requests;

Multicast mutual exclusion



p_1, p_2 request access simultaneously,
p_3 does not want access.
p_2 does not reply to p_1 since has lower timestamp

Mutual exclusion summary

- Performance
  - one request-reply enough to enter
  - relatively high usage of network bandwith
  - client delay depends on the frequency of access and size of network
- Fault tolerance
  - usually assume reliable links
  - some can be adapted to deal with crashes
- Other solutions
  - sufficient to obtain agreement from certain overlapping subsets of processes voting set (Maekawa algorithm)

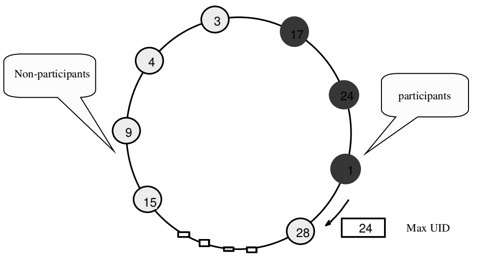
Leader election algorithms

- The problem
  - N processes, may or may not have unique ids (UIDs)
  - for simplicity assume no crashes
  - must choose unique master co-ordinator amongst processes
  - election called after failure has occurred
  - one or more processes can call election simultaneously
- Requirements

  (**LE1**) Every process knows P, identity of leader, where P is unique process id (usually maximum) or is yet undefined.

  (**LE2**) All processes participate and eventually discover the identity of the leader (cannot be undefined).

Chang&Roberts algorithm



Leader election in a ring: asynchronous model, UIDs known.

Chang&Roberts algorithm

- Assumptions
  - unidirectional ring, asynchronous, each process has UID
- Election
  - initially each process non-participant
  - determine leader (*election* message):
    - initiator becomes participant and passes own UID on to neighbour
    - when non-participant receives *election* message, forwards maximum of own and the received UID and becomes participant
    - participant does not forward the *election* message
  - announce winner (*elected* message):
    - when participant receives *election* message with own UID, becomes leader and non-participant, and forwards UID in *elected* message
    - otherwise, records the leader's UID, becomes non-participant and forwards it

Chang&Roberts algorithm

- How it works
  - if UIDs, then identity of leader unique
  - two exchanges around the ring: election, announce winner
  - if one process starts election
    - in worst case 3 round-trips needed - explain?
- but
  - does not tolerate failures (need reliable failure detector)
    - see bully algorithm (synchronous model)
    - works if more than one process simultaneously start election
  - what if no UIDs?
    - nodes on a re-configurable network, 'hot-pluggable'

20 February, 2002 19

Itai&Rodeh algorithm

- Assumptions
  - N processes, unidirectional ring, synchronous
  - processes do not have UIDs
- Election
  - each process selects ID at random from set {1,..K}
    - non-unique! but fast
  - process pass all IDs around the ring
  - after one round, if there exists a unique ID then elect maximum unique ID
  - otherwise, repeat
- Question
  - how does the loop terminate?

Probabilistically!

20 February, 2002 20

Itai&Rodeh algorithm

- How do know the algorithm terminates?
  - from probabilities: if you keep flipping a fair coin then after several *head*s you must get *tails*
- How many rounds does it take?
  - the larger the probability of unique ID, the faster the algo
  - expected time: N=4, K=16, expected 1.01 rounds
- Why use randomisation?
  - symmetry breaker
  - fast solution
- but only probabilistic (with probability 1) guarantees of termination

20 February, 2002 21

FireWire IEEE 1394 tree identify

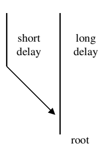
- Assumptions
  - not a ring: acyclic network
  - hot-pluggable, so no UIDs during tree identify
  - must choose a coordinator (to act as bus manager) after each node addition/removal
- How does it work?
  - acyclic network - almost a tree
  - starts with leaves: send 'be my parent' message to neighbours
  - nodes that receive N-1 requests (N is the number of neighbours), send 'be my parent' to the remaining node
- Question: will it determine the root?

NO - root contention!

20 February, 2002 22

Root contention idea

- Problem
  - two nodes may simultaneously send 'be my parent' messages to each other!
- Solution
  - each contender flips a coin, then chooses long or short delay
  - after delay, if it has received 'be my parent' it becomes root
  - otherwise, resends 'be my parent'
  - if contention arises again, process repeated



20 February, 2002 23

Summary

- Distributed algorithms
  - more susceptible to crashes
  - based on message passing
    - consume network bandwith
- Randomisation
  - useful as a symmetry breaker
  - yields simpler, faster algorithms
  - but probabilistic termination guarantees

20 February, 2002 24