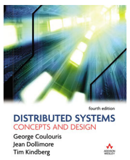
Slides for Chapter 12: Coordination and Agreement



From **Coulouris, Dollimore and Kindberg**
Distributed Systems: Concepts and Design
Edition 4, © Pearson Education 2005

Failure Assumptions and Failure Detectors

- reliable communication channels
- process failures: crashes
- failure detector: object/code in a process that detects failures of other processes
- unreliable failure detector
  - unsuspected or suspected (evidence of possible failures)
  - each process sends ``alive'' message to everyone else
  - not receiving ``alive'' message after timeout
  - most practical systems
- reliable failure detector
  - unsuspected or failure
  - synchronous system
  - few practical systems

Distributed Mutual Exclusion

- provide critical region in a distributed environment
- message passing
- for example, locking files, lockd daemon in UNIX

(NFS is stateless, no file-locking at the NFS level)

Algorithms for mutual exclusion

- *N* processes
- processes don't fail
- message delivery is reliable
- critical region: enter(), resourceAccesses(), exit()
- Properties:
  - [ME1] safety: only one process at a time
  - [ME2] liveness: eventually enter or exit
  - [ME3] happened-before ordering: ordering of enter() is the same as HB ordering

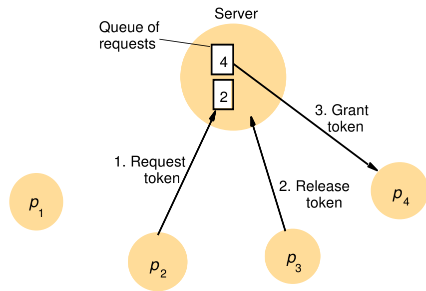
Algorithms for mutual exclusion

- Performance evaluation:
  - overhead and bandwidth consumption: # of messages sent
  - client delay incurred by a process at entry and exit
  - throughput measured by synchronization delay: delay between one's exit and next's entry

A central server algorithm

- server keeps track of a token---permission to enter critical region
- a process requests the server for the token
- the server grants the token if it has the token
- a process can enter if it gets the token, otherwise waits
- when done, a process sends release and exits

Server managing a mutual exclusion token for a set of processes



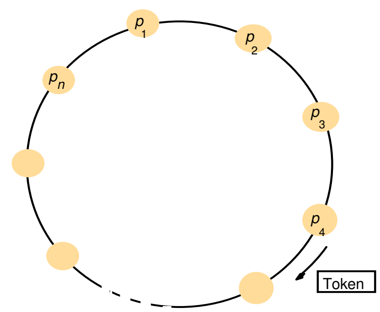
A central server algorithm

- Properties:
  - safety, why?
  - liveness, why?
  - HB ordering not guaranteed, why? [VC in processes vs. server]
- Performance:
  - enter overhead: two messages (request and grant)
  - enter delay: time between request and grant
  - exit overhead: one message (release)
  - exit delay: none
  - synchronization delay: between release and grant
  - centralized server is the bottle neck

A ring-based algorithm

- logical ring, could be unrelated to the physical configuration
- p_i sends messages to $p_{(i+1) \bmod N}$
- when a process holds a token, it can enter, otherwise waits
- when a process releases a token (exit), it sends to its neighbor

A ring of processes transferring a mutual exclusion token



A ring-based algorithm

- properties:
  - safety, why?
  - liveness, why?
  - HB ordering not guaranteed, why?
- Performance:
  - bandwidth consumption: token keeps circulating
  - enter overhead: 0 to *N* messages
  - enter delay: delay for 0 to *N* messages
  - exit overhead: one message
  - exit delay: none
  - synchronization delay: delay for 1 to *N* messages

An algorithm using multicast and logical clocks

- multicast a request message for the token
- enter only if all the other processes reply
- totally-ordered timestamps: $<T, p_i>$
- each process keeps a *state: RELEASED, HELD, WANTED*
- if all have *state = RELEASED*, all reply, a process can hold the token and enter
- if a process has *state = HELD*, doesn't reply until it exits
- if more than one process has *state = WANTED*, process with the lowest timestamp will get all *N*-1 replies first.

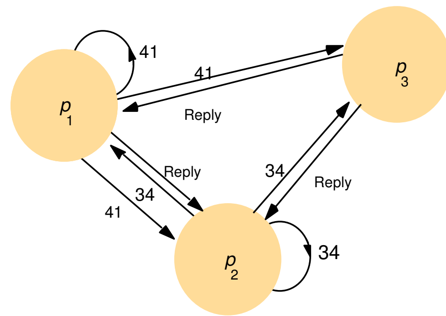
Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the 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 synchronization



An algorithm using multicast and logical locks

- Properties
  - safety, why?
  - liveness, why?
  - HB ordering, why?

An algorithm using multicast and logical locks

- performance:
  - bandwidth consumption: no token keeps circulating
  - entry overhead: $2(N-1)$, why? [with multicast support: $1 + (N-1) = N$]
  - entry delay: delay between request and getting all replies
  - exit overhead: 0 to N-1 messages
  - exit delay: none
  - synchronization delay: delay for 1 message (one last reply from the previous holder)

Maekawa's Voting Algorithm – Main Idea

- We actually don't need all N – 1 replies
- Consider processes A, B, X
  - A needs replies from "A" and X
  - B needs replies from "B" and X
  - **If** X can only reply to (vote) *one process at a time*
    - A and B cannot have a reply from X at the same time
    - Mutex between A and B--hinges on X
- Processes in overlapping groups
  - members in the overlap "control" mutex

Mutual exclusion for all processes

- A group for every process
- A pair of groups for A and B overlaps
  - => mutex(A, B)
- Every possible pair of groups overlaps
  - => mutex(all possible pairs of processes)
  - => mutex(all processes)

Groups and members

- A group for each process
  - N processes
  - N groups
- Each group has K members
  - numbers of processes to request for permission
- Each process is in M groups ($M > 1$)
  - Allows overlapping => mutex
    - If the groups are disjoint, $M = 1$, no overlapping, no mutex
  - number of processes to grant permission

Parameters and group membership

- N = number of processes
- K = voting group size
- M = number of voting groups each process is in
- Optimal (smallest K, why?)
  - $K = M$ ~= sqrt(N)
  - Non-trivial to construct the groups
- Approximation
  - $K = 2$ * sqrt(N) - 1
    - Put process id's in a sqrt(N) by sqrt(N) table
    - Union the rows and columns where p_i is
  - N groups
  - $M = 2$ * sqrt(N) - 1

Group membership

A	B	C
D	E	F
G	H	I

A	B	C
D	E	F
G	H	I

- Group for A: {A, B, C, D, G}
- Group for B: {A, B, C, E, H}
- …
- Group for I: {G, H, I, C, F}
- Every pair of groups overlap

- N groups
- Each group has $K = 2$ * sqrt(N) – 1 members
- $M = 2$ * sqrt(N) – 1 [# of groups each process are in]

Group membership (alternative?)

A	B	C
D	E	F
G	H	I

A	B	C
D	E	F
G	H	I

- Groups for A, B, C: {A, B, C, D, G}
- Groups for D, E, F: {D, E, F, B, H}
- Groups for G, H, I: {G, H, I, C, F}
- Every pair of groups overlap

- N groups [Sqrt(N) unique groups]
- Each group has $K = 2$ * sqrt(N) – 1 members
- $M = 3$ [sqrt(N)] for A, E, I, but $M = 6$ [2*sqrt(N)] for the rest
  - Undesirable, why?

Sketch of the Voting Algorithm

- A group of processes V_i for each process p_i
  - Each pair of groups overlap
    - X in groups: [A,X] and [B,X]
  - => the groups for any pair of processes overlap
- To enter the critical region, p_i
  - Sends REQUEST's to all processes in V_i
  - Waits for REPLY's (VOTE's) from all processes in V_i
- To exit the critical region, p_i
  - Sends RELEASE's to all processes in V_i
- Three types of messages, not two as in the multicast alg.

Figure 12.6
Maekawa's voting algorithm

On initialization
  state := RELEASED;
  voted := FALSE;

For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;

On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Deadlock?

- Processes: A, B, C
  - Group A: A, B
  - Group B: B, C
  - Group C: C, A
- Deadlock
  - A has A's reply, waiting for B's reply
  - B has B's reply, waiting for C's reply
  - C has C's reply, waiting for A's reply
- Timestamp the requests in HB ordering
  - holding according to the timestamp

Properties

- Safety: No process can reply/vote more than once at any time.
- Liveness: timestamp (HB ordering)
- HB ordering: above

Performance

- Entry overhead [assuming k = sqrt(N)]
  - Sqrt(*N*) requests + Sqrt(*N*) replies
  - 2* Sqrt(*N*)
    - < 2(*N* - 1) [*N* > 4]
- Exit overhead
  - Sqrt(*N*) releases

Elections

- choosing a unique process for a particular role
- for example, server in dist. mutex
- each process can call only one election
- multiple concurrent elections can be called by different processes
- participant: engages in an election
- process with the largest id wins
- each process p_i has variable $elected_i$ = ? (don't know) initially

Elections

- Properties:
  - [E1] $elected_i$ of a ``participant'' process must be P_{max} (elected process---largest id) or *?*
  - [E2] liveness: all processes participate and eventually set $elected_i$!= *?* (or crash)
- Performance:
  - overhead (bandwidth consumption): # of messages
  - turnaround time: # of messages to complete an election

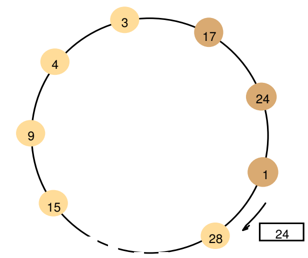
A ring-based algorithm

- logical ring, could be unrelated to the physical configuration
- p_i sends messages to $p_{(i+1) \bmod N}$
- no failures
- elect the coordinator with the largest id
- initially, every process is a non-participant
- any process can call an election:
  - marks itself as participant
  - places its id in an *election* message
  - sends the message to its neighbor

Ring-based algorithm

- receiving an election message:
  - if *id* > *myid*, forward the msg, mark participant
  - if *id* < *myid*
    - non-participant: replace *id* with *myid*: forward the msg, mark participant
    - participant: stop forwarding (why? Later, multiple elections)
  - if *id* = *myid*, coordinator found, mark non-participant, $elected_i$:= *id*, send *elected* message with *myid*
- receiving an elected message:
  - *id* != *myid*, mark non-participant, $elected_i$:= *id* forward the msg
  - if *id* = *myid*, stop forwarding

A ring-based election in progress



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

Ring-based algorithm

- Properties:
  - safety: only the process with the largest id can send an *elected* message
  - liveness: every process in the ring eventually participates in the election; extra elections are stopped

Ring-based algorithm

- Performance:
  - one election, best case, when?
    - *N election* messages
    - *N elected* messages
    - turnaround: 2*N* messages
  - one election, worst case, when?
    - 2*N* - 1 election messages
    - *N elected* messages
    - turnaround: 3*N* - 1 messages
  - can't tolerate failures, not very practical

The bully algorithm

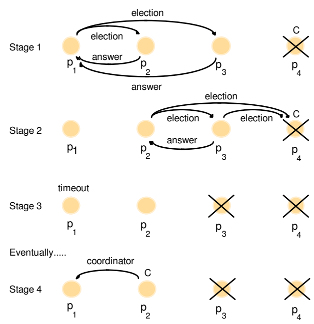
- processes can crash and can be detected by other processes
- timeout $T = 2T_{transmitting} + T_{processing}$
- each process knows all the other processes and can communicate with them
- Messages: election, answer, coordinator

The bully algorithm

- start an election
  - detects the coordinator has failed
  - sends an election message to all processes with higher id's and waits for answers (except the failed coordinator/process)
  - if no answers in time *T*,
    - it is the coordinator
    - sends coordinator message (with its id) to all processes with lower id's
  - else
    - waits for a coordinator message
    - starts an election if timeout

The bully algorithm

The election of coordinator p_2, after the failure of p_4 and then p_3



The bully algorithm

- receiving an election message
  - sends an answer message back
  - starts an election if it hasn't started one—send election messages to all higher-id processes (including the "failed" coordinator—the coordinator might be up by now)
- receiving a coordinator message
  - set $elected_i$ to the new coordinator
- to be a coordinator, it has to start an election
- when a crashed process is replaced
  - the new process starts an election and
  - can replace the current coordinator (hence ``bully'')

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

The bully algorithm

- properties:
  - safety:
    - a lower-id process always yields to a higher-id process
    - However, during an election, if a failed process is replaced
      - the low-id processes might have two different coordinators: the newly elected coordinator and the new process, why?
    - failure detection might be unreliable
  - liveness: all processes participate and know the coordinator at the end

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

The bully algorithm

- Performance
  - best case: when?
    - overhead: N-2 *coordinator* messages
    - turnaround delay: no *election/answer* messages
  - worst case: when?
    - overhead:
    - 1+ 2 + ...+ (N-2) + (N-2)= (N-1)(N-2)/2 + (N-2) *election* messages,
    - 1+...+ (N-2) *answer* messages,
    - N-2 *coordinator* messages,
    - total: (N-1)(N-2) + 2(N-2) = (N+1)(N-2) = O(N^2)
  - turnaround delay: delay of election and answer messages

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005