Algorithms for Distributed Systems Maekawa's Voting Algorithm for Mutual Exclusion

Christian Mittermaier Thomas Prokosch
Igor Rents

February 17, 1999

The I/O Automaton

MaekawaVoting$_i$ automaton

Global:

$M = n^2$	the number of processors in the mesh
$\forall\ 1 \leq j \leq n:\ \mathcal{S}_j$	the voting district of processor j
LTS	the global Lamport timestamp

Signature:

Input:
- $receive(ack)_{j,i}$ $ack \in \{\mathbf{YES}, \mathbf{NO}\}$
- $receive((\mathbf{REQUEST}, request_TS))_{j,i}$ $request_TS \in \mathbf{N}_0$
- $receive(\mathbf{RELINQUISH})_{j,i}$
- $receive(\mathbf{RELEASE})_{j,i}$
- $receive((\mathbf{INQUIRE}, inquire_TS))_{j,i}$ $inquire_TS \in \mathbf{N}_0$

Output:
- $voting_entry()_i$
- $enter_cr()_i$

States:

my_TS	the time stamp for the current CR request, initially 0
yes_votes	the number of procs that voted "yes", initially 0
$have_voted$	**T** if the processor has already cast its vote, initially **F**
$candidate$	the candidate the processor voted for
$candidate_TS$	timestamp of the candidate proc i voted for, initially 0
$have_inquired$	**T** if proc has tried to recall a vote, initially **F**
$WaitingQ$	set of vote requests proc i is deferring, initially $\emptyset$
cr_entry	**T** if proc wants to enter the CR, initially **F**
$\forall j \in \mathcal{S}_i:\ send(j)$	a FIFO queue of messages, initially empty

Transitions:

$voting_entry()_i$
  Precondition:
    $cr_entry = \mathbf{F}$
  Effect:
    $cr_entry := \mathbf{T}$;
    $my_TS := LTS$;
    $\forall j \in \mathcal{S}_i$: add $(\mathbf{REQUEST}, my_TS)$ to $send(j)$;

$receive(ack)_{j,i}$, $ack \in \{\mathbf{YES}, \mathbf{NO}\}$
  Effect:
    if $ack = \mathbf{YES}$ **then**
      $yes_votes := yes_votes + 1$;

$receive((\mathbf{INQUIRE}, inquire_TS))_{j,i}$ $inquire_TS \in \mathbf{N}_0$
  Effect:
    if $my_TS = inquire_TS$ **then**
      add **RELINQUISH** to $send(j)$;

$receive((\mathbf{REQUEST}, request_TS))_{j,i}$ $request_TS \in \mathbf{N}_0$
  Effect:
    if $have_voted = \mathbf{F}$ **then**
      add **YES** to $send(j)$;
      $candidate_TS := request_TS$;
      $candidate := j$;
      $have_voted := \mathbf{T}$;
    else
      add $(j, request_TS)$ to $WaitingQ$;
      add **NO** to $send(j)$;
      if $request_TS < candidate_TS \wedge \neg have_inquired$ **then**
        add $(\mathbf{INQUIRE}, candidate_TS)$ to $send(candidate)$;
        $have_inquired := \mathbf{T}$;

$receive(\mathbf{RELINQUISH})_{j,i}$
  Effect:
    add $(candidate, candidate_TS)$ to $WaitingQ$;
    remove (s, rts) from $WaitingQ$ s.t. rts is minimal;
    add **YES** to $send(s)$;
    $candidate_TS := rts$;
    $candidate := s$;
    $have_inquired := \mathbf{F}$;

$receive(\textbf{RELEASE})_{j,i}$
  Effect:
    if $WaitingQ \neq \emptyset$ **then**
      remove (s, rts) from $WaitingQ$ s.t. rts is minimal;
      add **YES** to $send(s)$;
      $candidate_TS := rts$;
      $candidate := s$;
    else
      $have_voted := \mathbf{F}$;
    $have_inquired := false$;

$enter_cr()_i$
  Precondition:
    $yes_votes = |\mathcal{S}_i|$
  Effect:
    do what has to be done in the CR;
    $yes_votes := 0$;
    $cr_entry := \mathbf{F}$;
    $\forall j \in \mathcal{S}_i$: add **RELEASE** to $send(j)$;

$send(message)_{i,j}$ $message$ a Message
  Precondition:
    $message$ is first on $send(j)$
  Effect:
    remove $message$ from $send(j)$;

Tasks:
for a fixed i : $\{voting_entry()_i, enter_cr()_i\}$
for every $j \in \mathcal{S}_i$: $\{send(message)_{i,j}\}$

Comments on the Automaton

- A processor i that wants to enter the critical region, has to perform a $voting_entry()$ action first. By this he asks all processors in his voting district for their vote and informs them about his current timestamp.
- Any processor that receives a **REQUEST** - message either responds with a **YES** - vote, if it has not voted yet, or stores the request in a queue sorted by the timestamps. If the current request has a timestamp less than the candidate the processor actually voted for, it tries to withdraw the vote in order to serve the currently waiting processor first.
- Having withdrawn a vote, the processor in the queue with minimal timestamp is given a **YES** - vote and the original processor's request is stored in the queue.
- As soon as a processor has received all votes from his coterie, he enters the critical region and respond with a **RELEASE** message to all the members of the voting district when finished.

- Unless the $WaitingQ$ is not empty and the processor receives a **RELEASE** message, the queue entry with minimal timestamp is extracted and served next.

The protocol is not susceptible to deadlock. Using Lamport timestamps, processors prefer to vote for the earliest candidate, so if processor i has cast its vote for processor j and after that processor k asks for i's vote, where k has an earlier timestamp, i tries to retrieve its vote from j. If j has not got all votes of its voting district yet, it will relinquish i's vote which can be given to k then.

So, eventually the candidate with the lowest timestamp will get all votes an can enter the critical region or it is blocked since another candidate enters the critical region. But anyway, some candidate enters the CR and a deadlock is avoided.