Distributed Systems

CS425/ECE428

Feb 20 2023

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta and Nikita Borisov
Logistics

• HW2 released today
  • You can solve the first two questions right away.
  • You can solve the third question by the end of this class.
  • Hopefully, you can solve the fourth question by end of this week, and the 5\textsuperscript{th} and the 6\textsuperscript{th} questions by next Monday.
  • Due on March 6\textsuperscript{th} (Monday)

• MP1 is also due on March 6\textsuperscript{th} (Monday).

• So please start working on your assignments right-away!
Logistics

• Early lecture slides are rough and transient.
Today’s agenda

• Mutual Exclusion
  • Chapter 15.2

• Leader Election (if time)
  • Chapter 15.3
Recap: Problem Statement for mutual exclusion

- **Critical Section Problem:**
  - Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.

- Each process can call three functions
  - `enter()` to enter the critical section (CS)
  - `AccessResource()` to run the critical section code
  - `exit()` to exit the critical section
Recap: Mutual exclusion in distributed systems

• Processes communicating by passing messages.

• Cannot share variables like semaphores!

• How do we support mutual exclusion in a distributed system?
Recap: Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm
Recap: System Model

• Each pair of processes is connected by reliable channels (such as TCP).

• Messages sent on a channel are eventually delivered to recipient, and in FIFO (First In First Out) order.

• Processes do not fail.
  • Fault-tolerant variants exist in literature.
Mutual exclusion in distributed systems

• Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  • Central server algorithm
  • Ring-based algorithm
  • Ricart-Agrawala Algorithm
  • Maekawa Algorithm
Ricart-Agrawala’s Algorithm

- Classical algorithm from 1981
- Invented by Glenn Ricart (NIH) and Ashok Agrawala (U. Maryland)

- No token.
- Uses the notion of causality and multicast.
- Has lower waiting time to enter CS than Ring-Based approach.
Key Idea: Ricart-Agrawala Algorithm

• **enter()** at process \( P_i \)
  - **multicast** a request to all processes
    - Request: \(<T, P_i>, \text{ where } T = \text{ current Lamport timestamp at } P_i>\)
    - Wait until **all** other processes have responded positively to request
  - Requests are granted in order of causality.
  - \(<T, P_i> \) is used lexicographically: \( P_i \) in request \(<T, P_i>\) is used to break ties (since Lamport timestamps are not unique for concurrent events).
Messages in RA Algorithm

• enter() at process Pi
  • set state to Wanted
  • multicast “Request” <Ti, Pi> to all other processes, where Ti = current Lamport timestamp at Pi
  • wait until all other processes send back “Reply”
  • change state to Held and enter the CS

• On receipt of a Request <Tj, j> at Pi (i ≠ j):
  • if (state = Held) or (state = Wanted & (Ti, i) < (Tj, j))
    // lexicographic ordering in (Tj, j), Ti is Lamport timestamp of Pi’s request
    add request to local queue (of waiting requests)
  else send “Reply” to Pj

• exit() at process Pi
  • change state to Released and “Reply” to all requests queued at Pi.
Example: Ricart-Agrawala Algorithm

Request message
<T, Pi> = <102, 32>
Example: Ricart-Agrawala Algorithm

N32 state: Held.
Can now access CS
**Example: Ricart-Agrawala Algorithm**

N12 state: **Wanted**

N3 state: **Held**

N32 state: **Held**

Can now access CS

Request message: <115, 12>

Request message: <110, 80>

N80 state: **Wanted**
Example: Ricart-Agrawala Algorithm

N12 state: **Wanted**

N32 state: **Held.**
Can now access CS

N80 state: **Wanted**

Request message

Reply messages

N32

N5

N6

N3

N12

<115, 12>

<110, 80>
Example: Ricart-Agrawala Algorithm

N12 state: Wanted

N12

Request message

<115, 12>

N3

Reply messages

N6

N32

N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

N80

Request message

<110, 80>

N5

N80 state: Wanted

<110, 80>
Example: Ricart-Agrawala Algorithm

N12 state: Wanted

N12

Request message <115, 12>

N3

Reply messages

N32

N32 state: Held.
Can now access CS
Queue requests: <115, 12>, <110, 80>

N80

N80 state:
Wanted
Queue requests: <115, 12> (since > (110, 80))

N6

N5

Request message <110, 80>
Example: Ricart-Agrawala Algorithm

N12 state: Wanted

Request message: <115, 12>

Reply messages

N3 state:

N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

N6 state:

N80 state:
Wanted
Queue requests: <115, 12> (since > (110, 80))
Example: Ricart-Agrawala Algorithm

N12 state: Wanted
Request message: \langle115, 12\rangle
Reply
N6

N12
Request message: \langle110, 80\rangle
N3

N32 state: Held.
Can now access CS
Queue requests: \langle115, 12\rangle, \langle110, 80\rangle

N80 state: Wanted
Queue requests: \langle115, 12\rangle
Example: Ricart-Agrawala Algorithm

N12 state: Wanted
N6

N12

Request message
<115, 12>

Reply

N3

N32

N32 state: Released.

N80

N80 state:
Wanted
Queue requests: <115, 12>

N5

Request message
<110, 80>
Example: Ricart-Agrawala Algorithm

N12 state: Wanted
N80 state: Wanted
Queue requests: <115, 12>

N32 state: Released.
Multicast Reply to <115, 12>, <110, 80>

Request message <115, 12>
Reply
Request message <110, 80>
Example: Ricart-Agrawala Algorithm

N12 state: **Wanted** (waiting for N80’s reply)

N6

N12

Request message

<115, 12>

Reply messages

N3

N32

N32 state: **Released**.
Multicast Reply to

<115, 12>, <110, 80>

N80

Request message

<110, 80>

N5

N80 state:
**Held.** Can now access CS.
Queue requests: <115, 12>
Analysis: Ricart-Agrawala’s Algorithm

- Safety
  - Two processes $P_i$ and $P_j$ cannot both have access to CS
    - If they did, then both would have sent Reply to each other.
    - Thus, $(T_i, i) < (T_j, j)$ and $(T_j, j) < (T_i, i)$, which are together not possible.
    - What if $(T_i, i) < (T_j, j)$ and $P_i$ replied to $P_j$’s request before it created its own request?
      - But then, causality and Lamport timestamps at $P_i$ implies that $T_i > T_j$, which is a contradiction.
      - So this situation cannot arise.
Analysis: Ricart-Agrawala’s Algorithm

- **Safety**
  - Two processes $P_i$ and $P_j$ cannot both have access to CS.

- **Liveness**
  - Worst-case: wait for all other $(N-1)$ processes to send Reply.

- **Ordering**
  - Requests with lower Lamport timestamps are granted earlier.
Analysis: Ricart-Agrawala’s Algorithm

- **Safety**
  - Two processes $P_i$ and $P_j$ cannot both have access to CS.

- **Liveness**
  - Worst-case: wait for all other $(N-1)$ processes to send Reply.

- **Ordering**
  - Requests with lower Lamport timestamps are granted earlier.
Analysis: Ricart-Agrawala’s Algorithm

• Bandwidth:
  • \(2(N-1)\) messages per enter operation
    • \(N-1\) unicasts for the multicast request + \(N-1\) replies
    • Maybe fewer depending on the multicast mechanism.
  • \(N-1\) unicasts for the multicast release per exit operation
    • Maybe fewer depending on the multicast mechanism.

• Client delay:
  • one round-trip time

• Synchronization delay:
  • one message transmission time

• Client and synchronization delays have gone down to \(O(1)\).
• Bandwidth usage is still high. Can we bring it down further?
Mutual exclusion in distributed systems

• Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  • Central server algorithm
  • Ring-based algorithm
  • Ricart-Agrawala Algorithm
  • Maekawa Algorithm
Maekawa’s Algorithm: Key Idea

- Ricart-Agrawala requires replies from all processes in group.
- Instead, get replies from only some processes in group.
- But ensure that only one process is given access to CS (Critical Section) at a time.
Maekawa’s Voting Sets

- Each process \( P_i \) is associated with a voting set \( V_i \) (subset of processes).
- Each process belongs to its own voting set.
- The intersection of any two voting sets must be non-empty.
A way to construct voting sets

One way of doing this is to put $N$ processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix and for each $P_i$, its voting set $V_i = \text{row containing } P_i + \text{column containing } P_i$.

Size of voting set $= 2^{\sqrt{N}-1}$. 

$P_1$'s voting set $= V_1$
Maekawa: Key Differences From Ricart-Agrawala

• Each process requests permission from only its voting set members.
  • Not from all

• Each process (in a voting set) gives permission to at most one process at a time.
  • Not to all
Actions

• state = **Released**, voted = false

• **enter()** at process $P_i$:
  • state = **Wanted**
  • Multicast **Request** message to all processes in $V_i$
  • Wait for **Reply (vote)** messages from all processes in $V_i$ (including vote from self)
  • state = **Held**

• **exit()** at process $P_i$:
  • state = **Released**
  • Multicast **Release** to all processes in $V_i$
Actions (contd.)

• When Pi receives a Request from Pj:
  if (state == Held OR voted = true)
    queue Request
  else
    send Reply to Pj and set voted = true

• When Pi receives a Release from Pj:
  if (queue empty)
    voted = false
  else
    dequeue head of queue, say Pk
    Send Reply only to Pk
    voted = true
Size of Voting Sets

• Each voting set is of size $K$.
• Each process belongs to $M$ other voting sets.
• Maekawa showed that $K=M=\text{approx. } \sqrt{N}$ works best.
Optional self-study: Why $\sqrt{N}$ ?

- Let each voting set be of size $K$ and each process belongs to $M$ other voting sets.
- Total number of voting set members (processes may be repeated) = $K \times N$
- But since each process is in $M$ voting sets
  - $K \times N = M \times N \Rightarrow K = M$ (1)
- Consider a process $P_i$
  - Total number of voting sets = members present in $P_i$’s voting set and all their voting sets = $(M-1) \times K + 1$
  - All processes in group must be in above
  - To minimize the overhead at each process ($K$), need each of the above members to be unique, i.e.,
    - $N = (M-1) \times K + 1$
    - $N = (K-1) \times K + 1$ (due to (1))
    - $K \sim \sqrt{N}$
Size of Voting Sets

- Each voting set is of size $K$.
- Each process belongs to $M$ other voting sets.
- Maekawa showed that $K=M=approx. \sqrt{N}$ works best.
- Matrix technique gives a voting set size of $2*\sqrt{N}-1 = O(\sqrt{N})$. 
Performance: Maekawa Algorithm

- **Bandwidth**
  - $2K = 2\sqrt{N}$ messages per enter
  - $K = \sqrt{N}$ messages per exit
  - Better than Ricart and Agrawala’s $(2*(N-1)$ and $N-1$ messages)
  - $\sqrt{N}$ quite small. $N \sim 1$ million $\Rightarrow \sqrt{N} = 1K$

- **Client delay:**
  - One round trip time

- **Synchronization delay:**
  - 2 message transmission times
Safety

• When a process $P_i$ receives replies from all its voting set $V_i$ members, no other process $P_j$ could have received replies from all its voting set members $V_j$.
  • $V_i$ and $V_j$ intersect in at least one process say $P_k$.
  • But $P_k$ sends only one Reply (vote) at a time, so it could not have voted for both $P_i$ and $P_j$. 
Liveness

• Does not guarantee liveness, since can have a deadlock.

• System of 6 processes \{0, 1, 2, 3, 4, 5\}. 0, 1, 2 want to enter critical section:
  
  • \( V_0 = \{0, 1, 2\} \):
    
    • 0, 2 send reply to 0, but 1 sends reply to 1;
  
  • \( V_1 = \{1, 3, 5\} \):
    
    • 1, 3 send reply to 1, but 5 sends reply to 2;
  
  • \( V_2 = \{2, 4, 5\} \):
    
    • 4, 5 send reply to 2, but 2 sends reply to 0;

• Now, 0 waits for 1’s reply, 1 waits for 5’s reply (5 waits for 2 to send a release), and 2 waits for 0 to send a release. Hence, deadlock!
Analysis: Maekawa Algorithm

- **Safety:**
  - When a process $P_i$ receives replies from all its voting set $V_i$ members, no other process $P_j$ could have received replies from all its voting set members $V_j$.

- **Liveness**
  - Not satisfied. Can have deadlock!

- **Ordering:**
  - Not satisfied.
Breaking deadlocks

- Maekawa algorithm can be extended to break deadlocks.
- Compare Lamport timestamps before replying (like Ricart-Agrawala).
- But is that enough?
  - System of 6 processes \{0, 1, 2, 3, 4, 5\}. 0, 1, 2 want to enter critical section:
    - \(V_0 = \{0, 1, 2\}\): 0, 2 send reply to 0, but 1 sends reply to 1;
    - \(V_1 = \{1, 3, 5\}\): 1, 3 send reply to 1, but 5 sends reply to 2;
    - \(V_2 = \{2, 4, 5\}\): 4, 5 send reply to 2, but 2 sends reply to 0;
  - Suppose \((L_1, P_1) < (L_0, P_0) < (L_2, P_2)\).
  - Deadlock can still happen based on when messages are received.
    - P5 receives P2’s request before P1’s, and replies back to P2 first.
  - We need a way to take back the reply.
Breaking deadlocks

• Say Pi’s request has a smaller timestamp than Pj.
• If Pk receives Pj’s request after replying to Pi, send fail to Pj.
• If Pk receives Pi’s request after replying to Pj, send inquire to Pj.
• If Pj receives an inquire and at least one fail, it sends a relinquish to release locks, and deadlock breaks.
Breaking deadlocks

- System of 6 processes \{0, 1, 2, 3, 4, 5\}. 0, 1, 2 want to enter critical section:
  - \(V_0 = \{0, 1, 2\}\): 0, 2 send \textit{reply} to 0, but 1 sends \textit{reply} to 1;
  - \(V_1 = \{1, 3, 5\}\): 1, 3 send \textit{reply} to 1, but 5 sends \textit{reply} to 2;
  - \(V_2 = \{2, 4, 5\}\): 4, 5 send \textit{reply} to 2, but 2 sends \textit{reply} to 0;
- Suppose \((L1, P1) < (L0, P0) < (L2, P2)\).
- P2 will send \textit{fail} to itself when it receives its own request after P0.
- P5 will send \textit{inquire} to P2 when it receives P1’s request.
- P2 will send \textit{relinquish} to \(V_2\). P5 and P4 will set “voted = false”. P5 will reply to P1.
- P1 can now enter CS, followed by P0, and then P2.
Mutual exclusion in distributed systems

- Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
    - Satisfies safety, liveness, but not ordering.
    - $O(1)$ bandwidth, and $O(1)$ client and synchronization delay.
    - Central server is scalability bottleneck.
  - Ring-based algorithm
    - Satisfies safety, liveness, but not ordering.
    - Constant bandwidth usage, $O(N)$ client and synchronization delay.
  - Ricart-Agrawala algorithm
    - Satisfies safety, liveness, and ordering.
    - $O(N)$ bandwidth, $O(1)$ client and synchronization delay.
  - Maekawa algorithm
    - Satisfies safety, but not liveness and ordering.
    - $O(\sqrt{N})$ bandwidth, $O(1)$ client and synchronization delay.