Distributed Systems

CS425/ECE428

02/21/2020

Today's agenda

- Wrap-up Mutual Exclusion
  - Chapter 15.2
  - Analysis of Ricart-Agrawala algorithm
  - Maekawa algorithm

- Leader Elections
  - Chapter 15.3

- Acknowledgement:
  - Materials derived from Prof. Indy Gupta and Prof. Nikita Borisov.

Recap: Mutual Exclusion

- Mutual exclusion important problem in distributed systems.
- Ensure at most one process is executing a piece of code (critical section) at a given point in time.

Mutual exclusion in distributed systems

- Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala algorithm
  - Maekawa algorithm

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.
    - Constantly uses bandwidth, O(N) client and synchronization delay
  - Ricart-Agrawala algorithm
  - Maekawa algorithm

Ricart-Agrawala’s Algorithm

- enter() at process Pi
  - set state to Wanted
  - multicast “Request” <Ti, Pi> to all processes, where Ti = current Lamport timestamp at Pi
  - wait until all 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 queued requests.

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

- Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricarta-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.*

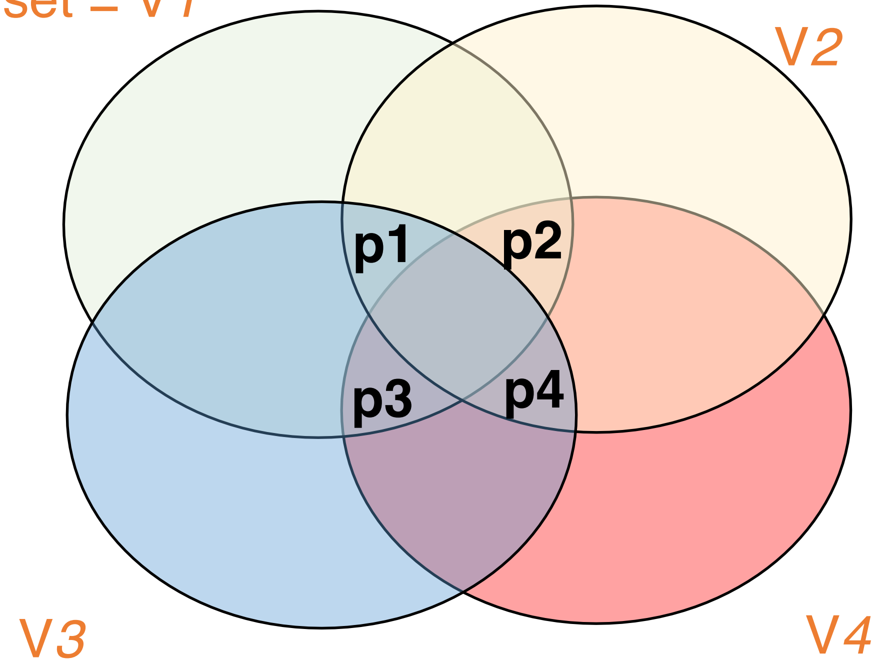
A way to construct voting sets

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

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

p1	p2
p3	p4

P1’s voting set = V1



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 P*i* receives a Request from P*j*:

```
if (state == Held OR voted = true)
        queue Request
else
        send Reply to Pj and set voted = true
```

Actions (contd.)

- When P*i* receives a Release from P*j*:

```
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 $K=M=\sqrt{N}$ works best.

Optional self-study: Why $\sqrt{N}$?

- Each voting set is of size K *and* each process belongs to M other voting sets.
- Total number of voting set members (processes may be repeated) = $K*N$
- But since each process is in M voting sets
  - $K*N = M*N \Rightarrow K = M$ (1)
- Consider a process Pi
  - Total number of voting sets = members present in Pi's voting set and all their voting sets = $(M-1)*K + 1$
  - 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.,
    - $N = (M-1)*K + 1$
    - $N = (K-1)*K + 1$ (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 $K=M=\sqrt{N}$ works best.
- Matrix technique gives a voting set size of $2*\sqrt{N}-1 = O(\sqrt{N})$.

Performance: Maekawa Algorithm

- Bandwidth
  - 2K = 2√*N* messages per enter
  - K = √*N* messages per exit
  - Better than Ricart and Agrawala's (2*(*N-1*) and *N-1* messages)
  - √*N* quite small. *N* ~ 1 million => √*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;
  - *Can still happen depending on which message is received earlier.*
- 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 Px 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.

Handling 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 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;
- P1 will send inquire to itself when it receives P0's request after its own.
- P2 will send fail to P1 when it receives P1's request after P0.
- P2 will send fail to itself when it receives its own request after P0.
- P5 will send inquire to P2 when it receives P1's request.
- P1 will send relinquish to V_1. P1 will set "voted = false" and reply to P0. P5 will remove P1's request from its queue.
- P0 can now enter critical section.
- P2 will send relinquish to V_2. P5 and P4 will set "voted = false".

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(√N) bandwidth, O(1) client and synchronization delay.

Today's agenda

- Wrap-up Mutual Exclusion
  - Chapter 15.2
  - Analysis of Ricart-Agrawala algorithm
  - Maekawa algorithm
- Leader Elections
  - Chapter 15.3
- Acknowledgement:
  - Materials largely derived from Prof. Indy Gupta.

Why Election?

- Example: Your Bank account details are replicated at a few servers, but one of these servers is responsible for receiving all reads and writes, i.e., it is the leader among the replicas
    - What if there are two leaders per customer?
    - What if servers disagree about who the leader is?
    - What if the leader crashes?

Each of the above scenarios leads to inconsistency

More motivating examples

- The root server in a group of NTP servers.
- The master in Berkeley algorithm for clock synchronization.
- In the sequencer-based algorithm for total ordering of multicasts, the “sequencer” = leader.
- The central server in the “central server algorithm” for mutual exclusion.
- Other systems that need leader election: Apache Zookeeper, Google’s Chubby.

Leader Election Problem

- In a group of processes, elect a *Leader* to undertake special tasks
  - And *let everyone know* in the group about this Leader
- What happens when a leader fails (crashes)
  - Some process detects this (using a Failure Detector!)
  - Then what?
- Focus of this lecture: Election algorithm. Its goal:
  1. Elect one leader only among the non-faulty processes
  2. All non-faulty processes agree on who is the leader

Calling for an Election

- Any process can call for an election.
- A process can call for at most one election at a time.
- Multiple processes are allowed to call an election simultaneously.
    - All of them together must yield only a single leader
- The result of an election should not depend on which process calls for it.

Election Problem, Formally

- A run of the election algorithm must always guarantee:
  - **Safety**: For all non-faulty processes *p*:
    - *p* has elected:
      - (q: a particular non-faulty process with the *best attribute value*)
      - or Null
  - **Liveness**: For all election runs:
    - election run terminates
    - & for all non-faulty processes *p*: *p*'s elected is not Null
- At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
  - Common attribute : leader has highest id
  - Other attribute examples: leader has highest IP address, or fastest cpu, or most disk space, or most number of files, etc.

System Model

- *N* processes.
- Messages are eventually delivered.
- Failures may occur during the election protocol.
- **Each process has a unique id.**
  - Each process has a unique attribute (based on which Leader is elected).
  - If two processes have the same attribute, combine the attribute with the process id to break ties.

Next class: Classical Election Algorithms

- Ring election algorithm
- Bully algorithm