Why Mutual Exclusion?

• Bank’s Servers in the Cloud: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate ATM.
  • Both ATMs read initial amount of $1000 concurrently from the bank’s cloud server
  • Both ATMs add $10,000 to this amount (locally at the ATM)
  • Both write the final amount to the server
  • What’s wrong?
Why Mutual Exclusion?

- **Bank’s Servers in the Cloud**: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of $1000 concurrently from the bank’s cloud server
  - Both ATMs add $10,000 to this amount (locally at the ATM)
  - Both write the final amount to the server
  - You lost $10,000!
- The ATMs need *mutually exclusive* access to your account entry at the server
  - or, mutually exclusive access to executing the code that modifies the account entry
More Uses of Mutual Exclusion

- Distributed File systems
  - Locking of files and directories
- Accessing objects in a safe and consistent way
  - Ensure at most one server has access to object at any point of time
- Server coordination
  - Work partitioned across servers
  - Servers coordinate using locks
- In industry
  - Chubby is Google’s locking service
  - Many cloud stacks use Apache Zookeeper for coordination among servers
**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
ATM1:
    enter(S);
    // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    exit(S); // exit

ATM2:
    enter(S);
    // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    exit(S); // exit
**Approaches to Solve Mutual Exclusion**

- **Single OS:**
  - If all processes are running in one OS on a machine (or VM), then
  - Semaphores, mutexes, condition variables, monitors, etc.
Approaches to Solve Mutual Exclusion (2)

• Distributed system:
  • Processes communicating by passing messages

Need to guarantee 3 properties:
• Safety (essential) – At most one process executes in CS (Critical Section) at any time
• Liveness (essential) – Every request for a CS is granted eventually
• Ordering (desirable) – Requests are granted in the order they were made
**Processes Sharing an OS: Semaphores**

- Semaphore == an integer that can only be accessed via two special functions
- Semaphore S=1; // Max number of allowed accessors

1. **wait(S) (or P(S) or down(S))**:

```c
while(1) {// each execution of the while loop is atomic
    if (S > 0) {
        S--; //atomic
        break;
    }
}
```

Each while loop execution and S++ are each **atomic** operations – supported via hardware instructions such as compare-and-swap, test-and-set, etc.

2. **signal(S) (or V(S) or up(s))**:

```c
S++; // atomic
```
Our Bank Example Using Semaphores

Semaphore S=1; // shared

ATM1:
    wait(S);
    // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    signal(S); // exit

Semaphore S=1; // shared

ATM2:
    wait(S);
    // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    signal(S); // exit
• In a distributed system, cannot share variables like semaphores
• So how do we support mutual exclusion in a distributed system?
Before solving any problem, specify its System Model:

- Each pair of processes is connected by reliable channels (such as TCP).
- Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
- Processes do not fail.
  - Fault-tolerant variants exist in literature.
Central Solution

- Elect a central master (or leader)
  - Use one of our election algorithms!
- Master keeps
  - A queue of waiting requests from processes who wish to access the CS
  - A special token which allows its holder to access CS
- Actions of any process in group:
  - `enter()`
    - Send a request to master
    - Wait for token from master
  - `exit()`
    - Send back token to master
Central Solution

• Master Actions:
  • On receiving a request from process $P_i$
    if (master has token)
      Send token to $P_i$
    else
      Add $P_i$ to queue
  • On receiving a token from process $P_i$
    if (queue is not empty)
      Dequeue head of queue (say $P_j$), send that process the token
    else
      Retain token
**Analysis of Central Algorithm**

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With $N$ processes in system, queue has at most $N$ processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- FIFO Ordering is guaranteed, in order of requests received at master
Efficient mutual exclusion algorithms use fewer messages, and make processes wait for shorter durations to access resources. Three metrics:

- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.

- **Client delay**: the delay incurred by a process at each enter and exit operation (when no other process is in, or waiting)
  
  (We will prefer mostly the enter operation.)

- **Synchronization delay**: the time interval between one process exiting the critical section and the next process entering it (when there is only one process waiting)
**Analysis of Central Algorithm**

- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.
  - 2 messages for enter
  - 1 message for exit

- **Client delay**: the delay incurred by a process at each *enter* and *exit* operation (when *no* other process is in, or waiting)
  - 2 message latencies (request + grant)

- **Synchronization delay**: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)
  - 2 message latencies (release + grant)
But...

• The master is the performance bottleneck and SPoF (single point of failure)
Ring-based Mutual Exclusion

Currently holds token, can access CS

Token: ✪
Ring-based Mutual Exclusion

Cannot access CS anymore

Here’s the token!

Token: •
Ring-based Mutual Exclusion

Currently holds token, can access CS

Token: ●
Ring-based Mutual Exclusion

- $N$ Processes organized in a virtual ring
- Each process can send message to its successor in ring
- Exactly 1 token
- `enter()`
  - Wait until you get token
- `exit() // already have token`
  - Pass on token to ring successor
- If receive token, and not currently in `enter()`, just pass on token to ring successor
Analysis of Ring-based Mutual Exclusion

- **Safety**
  - Exactly one token
- **Liveness**
  - Token eventually loops around ring and reaches requesting process (no failures)
- **Bandwidth**
  - Per enter(), 1 message by requesting process but up to $N$ messages throughout system
  - 1 message sent per exit()
Analysis of Ring-Based Mutual Exclusion (2)

- Client delay: 0 to $N$ message transmissions after entering `enter()`
  - Best case: already have token
  - Worst case: just sent token to neighbor
- Synchronization delay between one process’ `exit()` from the CS and the next process’ `enter()`:
  - Between 1 and $(N-1)$ message transmissions.
  - **Best case:** process in `enter()` is successor of process in `exit()`
  - **Worst case:** process in `enter()` is predecessor of process in `exit()`
• Client/Synchronization delay to access CS still $O(N)$ in Ring-Based approach.
• Can we make this faster?
Before solving any problem, specify its System Model:

- Each pair of processes is connected by reliable channels (such as TCP).
- Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
- Processes do not fail.
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 `Pi`
  - *multicast* a request to all processes
    - Request: `<T, Pi>`, where `T` = current Lamport timestamp at `Pi`
    - Wait until *all* other processes have responded positively to request
  
- Requests are granted in order of causality
  
- `<T, Pi>` is used lexicographically: `Pi` in request `<T, Pi>` 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 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, Pj> at Pi (i ≠ j):**
  - if (state = **Held**) or (state = **Wanted** & (Ti, i) < (Tj, j))
    // lexicographic ordering in (Tj, Pj)
    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.
Example: Ricart-Agrawala Algorithm

Request message
\(<T, P_i> = <102, 32>\)
Example: Ricart-Agrawala Algorithm

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

N12 state: Wanted

N32 state: Held.
Can now access CS

N80 state: Wanted

Request message <115, 12>

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

- **N12 state:** Wanted
- **N3 state:**
- **N6 state:**
- **N80 state:** Wanted
- **N32 state:** Held.
  
  Can now access CS

Request message

\(<115, 12>\)

Reply messages

\(<110, 80>\)
**Example: Ricart-Agrawala Algorithm**

N12 state: **Wanted**

N12 → N3

Request message: \(<115, 12>\)

N3 → N32

Reply messages

N32 state: **Held**.

Can now access CS

Queue requests:

\(<115, 12>, <110, 80>\)

N32 → N80

Request message: \(<110, 80>\)

N80 state: **Wanted**

N80 → N5

\(<110, 80>\)

N5 → N6

N6
**Example: Ricart-Agrawala Algorithm**

N12 state: **Wanted**

Request message: `<115, 12>`

N6

N80 state: **Wanted**

Queue requests: `<115, 12>` (since > (110, 80))

N32 state: **Held**

Can now access CS

Queue requests: `<115, 12>, <110, 80>`

Reply messages

N3
EXAMPLE: RICART-AGRAWALA ALGORITHM

N12 state: Wanted

N3

Request message
<115, 12>

Reply messages

N6

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

N80

Request message
<110, 80>

N5

N80 state: Wanted
Queue requests: <115, 12>
Example: Ricart-Agrawala Algorithm

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

N12 state: Request message <115, 12>

N3 state: Reply messages

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

N80 state: Held. Can now access CS. Queue requests: <115, 12>

N5 state:
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?
      - Then it seems like both $P_i$ and $P_j$ would approve each others’ requests
      - 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 (2)

- Liveness
  - Worst-case: wait for all other \((N-1)\) processes to send Reply
- Ordering
  - Requests with lower Lamport timestamps are granted earlier
**Performance: Ricart-Agrawala's Algorithm**

- **Bandwidth:** $2(N-1)$ messages per `enter()` operation
  - $N-1$ unicasts for the multicast request + $N-1$ replies
  - $N$ messages if the underlying network supports multicast (1 multicast + $N-1$ unicast replies)
  - $N-1$ unicast messages per `exit` operation
    - 1 multicast if the underlying network supports multicast

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

- **Synchronization delay:** one message transmission time
Compared to Ring-Based approach, in Ricart-Agrawala approach
  - Client/synchronization delay has now gone down to $O(1)$
  - But bandwidth has gone up to $O(N)$
  - Can we get both down?
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 process one is given access to CS (Critical Section) at a time
Maekawa's Voting Sets

- Each process $P_i$ is associated with a voting set $V_i$ (of processes)
- Each process belongs to its own voting set
- The intersection of any two voting sets must be non-empty
  - Same concept as Quorums!
- Each voting set is of size $K$
- Each process belongs to $M$ other voting sets
- Maekawa showed that $K=M=\sqrt{N}$ works best
- 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 $V_i =$ row containing $P_i$ + column containing $P_i$. Size of voting set = $2*\sqrt{N}-1$
Example: Voting Sets with N=4

P1's voting set = V1

V1

V2

V3

V4

p1  p2

p3  p4
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 Pi:
  - state = Wanted
  - Multicast Request message to all processes in Vi
  - Wait for Reply (vote) messages from all processes in Vi (including vote from self)
  - state = Held
- exit() at process Pi:
  - state = Released
  - Multicast Release to all processes in Vi
When $P_i$ receives a Request from $P_j$:
\[
\text{if (state == Held OR voted = true)}
\]
\[
\quad \text{queue Request}
\]
\[
\text{else}
\]
\[
\quad \text{send Reply to Pj and set voted = true}
\]

When $P_i$ receives a Release from $P_j$:
\[
\text{if (queue empty)}
\]
\[
\quad \text{voted = false}
\]
\[
\text{else}
\]
\[
\quad \text{dequeue head of queue, say Pk}
\]
\[
\quad \text{Send Reply only to Pk}
\]
\[
\quad \text{voted = true}
\]
Safety

When a process Pi receives replies from all its voting set Vi members, no other process Pj could have received replies from all its voting set members Vj.

• Vi and Vj intersect in at least one process say Pk
• But Pk sends only one Reply (vote) at a time, so it could not have voted for both Pi and Pj
Liveness

• A process needs to wait for at most \((N-1)\) other processes to finish CS
• But does not guarantee liveness
• Since can have a deadlock
• Example: all 4 processes need access
  • P1 is waiting for P3
  • P3 is waiting for P4
  • P4 is waiting for P2
  • P2 is waiting for P1
  • No progress in the system!
• There are deadlock-free versions

\[ P_1 \text{’s voting set} = V_1 \]
Performance

- Bandwidth
  - $2\sqrt{N}$ messages per enter()
  - $\sqrt{N}$ 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
Why $\sqrt{N}$?

- Each voting set is of size $K$
- 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 $P_i$
  - Total number of voting sets = members present in $P_i$’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))
    - $K \sim \sqrt{N}$
Failures?

• There are fault-tolerant versions of the algorithms we’ve discussed
  • E.g., Maekawa

• One other way to handle failures: Use Paxos-like approaches!
Chubby

- Google’s system for locking
- Used underneath Google’s systems like BigTable, Megastore, etc.
- Not open-sourced but published
- Chubby provides *Advisory* locks only
  - Doesn’t guarantee mutual exclusion unless every client checks lock before accessing resource

Reference: http://research.google.com/archive/chubby.html
Chubby (2)

- Can use not only for locking but also writing small configuration files
- Relies on Paxos-like (consensus) protocol
- Group of servers with one elected as Master
  - All servers replicate same information
- Clients send read requests to Master, which serves it locally
- Clients send write requests to Master, which sends it to all servers, gets majority (quorum) among servers, and then responds to client
- On master failure, run election protocol
- On replica failure, just replace it and have it catch up

Server A

Server B

Server C

Server D

Server E

Master
Summary

- Mutual exclusion important problem in cloud computing systems
- Classical algorithms
  - Central
  - Ring-based
  - Ricart-Agrawala
  - Maekawa
- Industry systems
  - Chubby: a coordination service
  - Similarly, Apache Zookeeper for coordination
Announcements

• HW3 and MP3 have been released.
• Due soon. Start early!