Distributed Mutual Exclusion
Last time…

- Synchronizing real, distributed clocks
- Logical time and concurrency
- Lamport clocks and total-order Lamport clocks
- Vector clocks
- Happens-before relation
Goals of distributed mutual exclusion

• Much like regular mutual exclusion
  – Safety: mutual exclusion
  – Liveness: progress
  – Fairness: bounded wait and in-order

• Secondary goals:
  – reduce message traffic
  – minimize synchronization delay
    • i.e., switch quickly between waiting processes
Distributed mutex is different

• Regular mutual exclusion solved using shared state, e.g.
  – atomic test-and-set of a shared variable…
  – shared queue…

• We solve distributed mutual exclusion with message passing
  – Note: we assume the network is reliable but asynchronous…but processes might fail!
Solution 1: A central mutex server

• To enter critical section:
  – send REQUEST to central server, wait for permission

• To leave:
  – send RELEASE to central server
Solution 1: A central mutex server

• Advantages:
  – Simple (we like simple!)
  – Only 3 messages required per entry/exit

• Disadvantages:
  – Central point of failure
  – Central performance bottleneck
  – With an asynchronous network, impossible to achieve in-order fairness
  – Must elect/select central server
Solution 2: A ring-based algorithm

- Pass a token around a ring
  - Can enter critical section only if you hold the token

- Problems:
  - Not in-order
  - Long synchronization delay
    - Need to wait for up to $N-1$ messages, for $N$ processors
  - Very unreliable
    - Any process failure breaks the ring
2’: A fair ring-based algorithm

- Token contains the time \( t \) of the earliest known outstanding request
- To enter critical section:
  - Stamp your request with the current time \( T_r \), wait for token
- When you get token with time \( t \) while waiting with request from time \( T_r \), compare \( T_r \) to \( t \):
  - If \( T_r = t \): hold token, run critical section
  - If \( T_r > t \): pass token
  - If \( t \) not set or \( T_r < t \): set token-time to \( T_r \), pass token, wait for token
- To leave critical section:
  - Set token-time to null (i.e., unset it), pass token
Base case: null token circulates around the system
1/2 Simple case: one request

Request @ time

Critical Section
2/2 Simple case: one request

Node A

Request @ time

Node B

T[●]

Node C

T[null]

Node D

T[null]

Critical Section
1/2 Competing requests: \( \bullet < \bullet \)

Node A: Request @ time, \( T[\bullet] \)

Node B: \( T[\bullet] \)

Node C: \( \bullet < \bullet \)

Node D: \( T[\bullet] \)

Critical Section
2/2 Competing requests: • < •

Node A

Request @ time

T[•]

Node B

T[•]

Request @ time

Node C

T[•] < •

Node D

T[•]

Critical Section

T[null]

T[•]

T[•] = •
1/2 Competing requests: ○ < •

Node A
Request @ time

Node B

Node C
Request @ time
○ < •

Node D

Critical Section
Solution 3: Ricart and Agrawala dist. mutual exclusion alg

- Relies on Lamport totally ordered clocks, having the following properties:
  - For any events e, e’ such that e --> e’ (causality ordering), T(e) < T(e’)
  - For any distinct events e, e’, T(e) != T(e’)

General idea

• When want to enter critical section (C.S.) node i sends time-stamped request to all other nodes. These other nodes reply (eventually).

• When i receives n-1 replies, then can enter C.S.

• Trick: Node j having earlier request doesn’t reply to i until after it has completed its C.S.
Ricart-Agrawala overview

Node A

Request

Request

Response

Critical Section

Node B

Node C

Response

Response
Notation

- $\mathbb{N}_i = \{1, 2, ..., i-1, i+1, ..., n\}$ (n is the number of processes)

- Message types
  - (Request, i, T): Process i requests lock with timestamp T
  - (Reply, j): Process j responds to some request for lock

- For each node i, maintain following values:
  - $T_i()$: Function that returns value of local Lamport clock
  - should_defer: Boolean Set when process i should defer replies to requests
  - $T_r$: Time stamp of pending local request
  - $R$: Subset of Ni. Set of processes from which have received reply
  - $D$: Subset of Ni. Set of processes for which i has deferred the reply to their requests
  - lock(), unlock(): A local mutex lock, to keep the two threads from interfering with each other
Design

- Process i consists of two threads. One servicing the application, and one monitoring the network.

**Application thread:**

```
Request()       // Request global mutex
Wait for Notification // Wait until notified by network thread
Critical Section    // Operate in exclusive mode
Release()         // Release mutex
```
Application functions

Request():
  lock() // Don’t want app/network fns to step on each other
  Tr = Ti() // Get time stamp
  R = {}
  D = {}
  should_defer = true
  Send (Request, i, Tr) to each j in Ni
  unlock()

Release():
  lock()
  should_defer = false
  Send (Reply, i) to each j in D
  unlock()
Network function

while true:
    m = \texttt{Receive}()
    lock()
    if m == (Request, j, T):
        if should\_defer && Tr < T:
            D = D U \{j\} // Defer response to j
        else
            \texttt{Send} (Reply, i) to j
    else if m == (Reply, j):
        R = R U \{j\}
        if R == Ni
            Notify application
    unlock()
Request at logical time 1

Request at logical time 0
1/6 Ricart-Agrawala close-up

Node A

Node B

Node C

Req(1)

D={B}

Req_1

Req_0
2/6 Ricart-Agrawala close-up

Node A

Node B

Req(1)

Req(1)

D={B}

 Req_0

Node C
3/6 Ricart-Agrawala close-up

Node A

Node B

Node C

Req(1)

Req(1)

Req(1)

Req(0)

Req(0)

R={ }, D={B}

R={A}, D={}

Reply
4/6 Ricart-Agrawala close-up

Node A

Node B

Node C

Req(1) → Req(1)

R={A}, D={ } → Req(0)

R={A}, D={B} → Req_0
5/6 Ricart-Agrawala close-up
6/6 Ricart-Agrawala close-up

Node A

Req(1)

Reply

Reply

R={A,B}, D={ }

Critical Section

Node B

Req_1

Req(1)

R={A}, D={ }

R={A}, D={B}

Reply

Reply

R={A}, D={ }

R={A}, D={B}

Node C

Req_0

R={ }, D={B}

R={A,B}, D={B}
Ricart and Agrawala safety

- Suppose request $T_1$ is earlier than $T_2$.
- Consider how the process for $T_2$ collects its reply from process for $T_1$
  - $T_1$ must have already been time-stamped when request $T_2$ was received, otherwise the Lamport clock would have been advanced past time $T_2$
  - But then the process must have delayed reply to $T_2$ until after request $T_1$ exited the critical section. Therefore $T_2$ will not conflict with $T_1$. 
Ricart and Agrawala overview

- **Advantages:**
  - Fair
  - Short synchronization delay

- **Disadvantages**
  - Very unreliable
  - \(2(N-1)\) messages for each entry/exit