Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

October 4, 2012

Lecture 12

Mutual Exclusion

Reading: Sections 15.2

© 2012, I. Gupta, K. Nahrtstedt, S. Mitra, N. Vaidya, M. T. Harandi, J. Hou

Why Mutual Exclusion?

- **Bank's Servers in the Cloud: Think of two simultaneous deposits of $10,000 into your bank account, each from one 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: Think of two simultaneous deposits of $10,000 into your bank account, each from one 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?**
- **The ATMs need *mutually exclusive* access to your account entry at the server (or, to executing the code that modifies the account entry)**

Mutual Exclusion

- ***Critical section*** **problem: Piece of code (at all clients) for which we need to ensure there is <u>at most one client</u> executing it at any point of time.**
- **Solutions:**
  - **Semaphores, mutexes, etc. in single-node operating systems**
  - **Message-passing-based protocols in distributed systems:**
    - **enter() the critical section**
    - **AccessResource() in the critical section**
    - **exit() the critical section**
  - **Distributed mutual exclusion requirements:**
    - **Safety – At most one process may execute in CS at any time**
    - **Liveness – Every request for a CS is eventually granted**
    - **Ordering (desirable) – Requests are granted in the order they were made**

Refresher - Semaphores

- **To synchronize access of multiple threads to common data structures**
- **Semaphore S=1;**

Allows two operations: wait and signal

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

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

Each while loop execution and S++ are each atomic operations

– **how?**

2. signal(S) (or V(S)):

```
S++; // atomic
```

Refresher - Semaphores

- **To synchronize access of multiple threads to common data structures**
- **Semaphore S=1;**

Allows two operations: wait and signal

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

enter()

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

Each while loop execution and S++ are each atomic operations

– **how?**

2. signal(S) (or V(S)):

exit()

```
S++; // atomic
```

How are semaphores used?

One Use: Mutual Exclusion – Bank ATM example

```
semaphore S=1;

ATM1:
   wait(S); // enter
          // critical section
   obtain bank amount;
   add in deposit;
   update bank amount;
   signal(S); // exit
```

```
extern semaphore S;

ATM2
   wait(S); // enter
          // critical section
   obtain bank amount;
   add in deposit;
   update bank amount;
   signal(S); // exit
```

Distributed Mutual Exclusion: Performance Evaluation Criteria

- ***Bandwidth***: **the total number of messages sent in each *entry* and *exit* operation.**
- ***Client delay***: **the delay incurred by a process at each entry and exit operation (when *no* other process is in, or waiting)**

  (We will prefer mostly the entry 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)**

- **These translate into *throughput* -- the rate at which the processes can access the critical section, i.e., x processes per second.**

(these definitions more correct than the ones in the textbook)

Assumptions/System Model

- **For all the algorithms studied, we make the following assumptions:**
  - **Each pair of processes is connected by reliable channels (such as TCP).**
  - **Messages are eventually delivered to recipient in FIFO order.**
  - **Processes do not fail.**

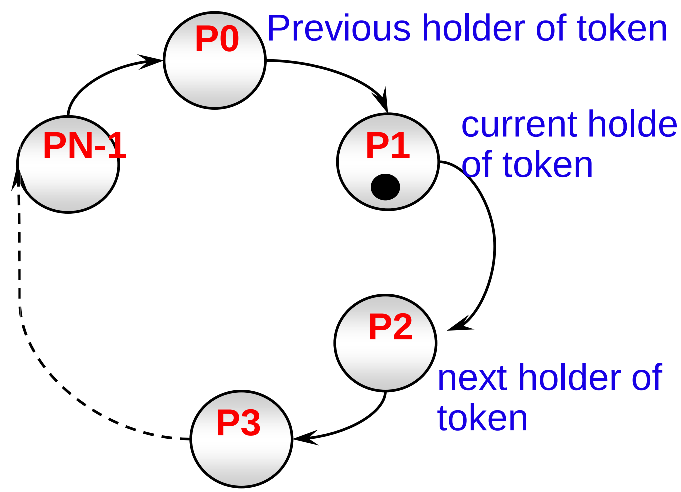
1. Centralized Control of Mutual Exclusion

- **A central coordinator (master or leader)**
  - Is elected (which algorithm?)
  - Grants permission to enter CS & keeps a queue of requests to enter the CS.
  - Ensures only one process at a time can access the CS
  - Has a special token message, which it can give to any process to access CS.
- **Operations**
  - To enter a CS Send a request to the coord & wait for token.
  - On exiting the CS Send a message to the coord to release the token.
  - Upon receipt of a request, if no other process has the token, the coord replies with the token; otherwise, the coord queues the request.
  - Upon receipt of a release message, the coord removes the oldest entry in the queue (if any) and replies with a token.
- **Features:**
  - Safety, liveness are guaranteed
  - Ordering also guaranteed (what kind?)
  - Requires 2 messages for entry + 1 messages for exit operation.
  - Client delay: one round trip time (request + grant)
  - Synchronization delay: 2 message latencies (release + grant)
  - ☹ The coordinator becomes performance bottleneck and single point of failure.

2. Token Ring Approach

- **Processes are organized in a logical ring: p_i has a communication channel to $p_{(i+1) mod N.}$**
- **Operations:**
  - **Only the process holding the token can enter the CS.**
  - **To enter the critical section, wait passively for the token. When in CS, hold on to the token and don't release it.**
  - **To exit the CS, send the token onto your neighbor.**
  - **If a process does not want to enter the CS when it receives the token, it simply forwards the token to the next neighbor.**

- **Features:**
  - **Safety & liveness are guaranteed**
  - **<u>Ordering</u> is not guaranteed.**
  - **Bandwidth: 1 message per exit**
  - **Client delay: 0 to N message transmissions.**
  - **Synchronization delay between one process's exit from the CS and the next process's entry is between 1 and N-1 message transmissions.**



3. Timestamp Approach: *Ricart & Agrawala*

- Processes requiring entry to critical section multicast a request, and can enter it only when *all* other processes have replied positively.
- Messages requesting entry are of the form $<T,p_i>$, where T is the sender's timestamp (from a *Lamport* clock) and p_i the sender's identity (used to break ties in T).
- To enter the CS
  - set state to wanted
  - multicast "request" to all processes (including timestamp) – use R-multicast
  - wait until all processes send back "reply"
  - change state to held and enter the CS
- On receipt of a request $<T_i, p_i>$ at p_j:
  - if (state = held) or (state = wanted & $(T_j, p_j)<(T_i,p_i)$), // lexicographic ordering

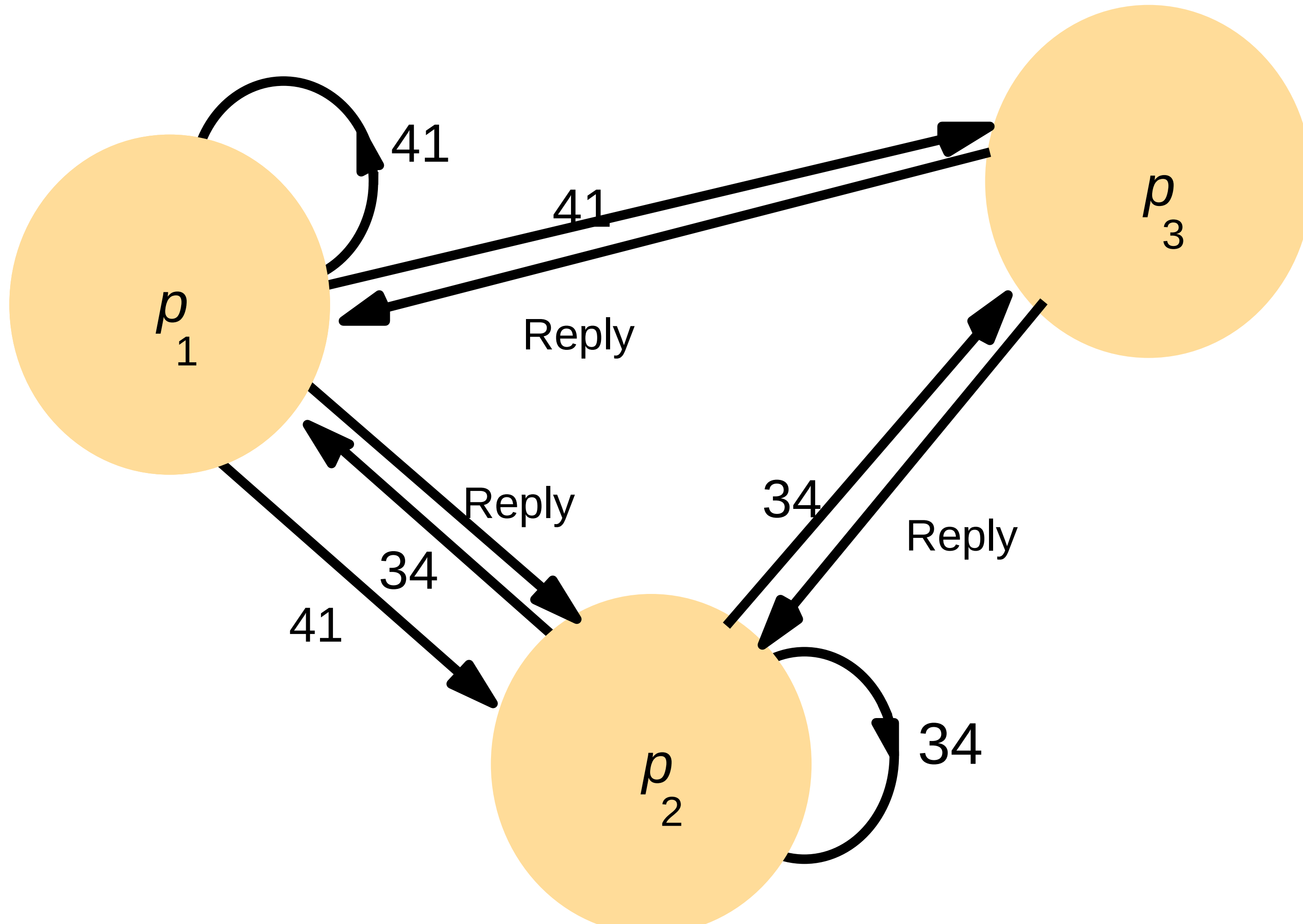
    enqueue request
  - else "reply" to p_i
- On exiting the CS
  - change state to release and "reply" to *all* queued requests.

Ricart & Agrawala's Algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (N -- 1));
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Ricart & Agrawala's Algorithm



Analysis: Ricart & Agrawala

- **Safety, liveness, and ordering (causal) are guaranteed**
  - **Why?**
- **Bandwidth: 2(N-1) messages per entry operation**
  - **N-1 unicasts for the multicast request + N-1 replies**
  - **N messages if the underlying network supports multicast**
  - **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**

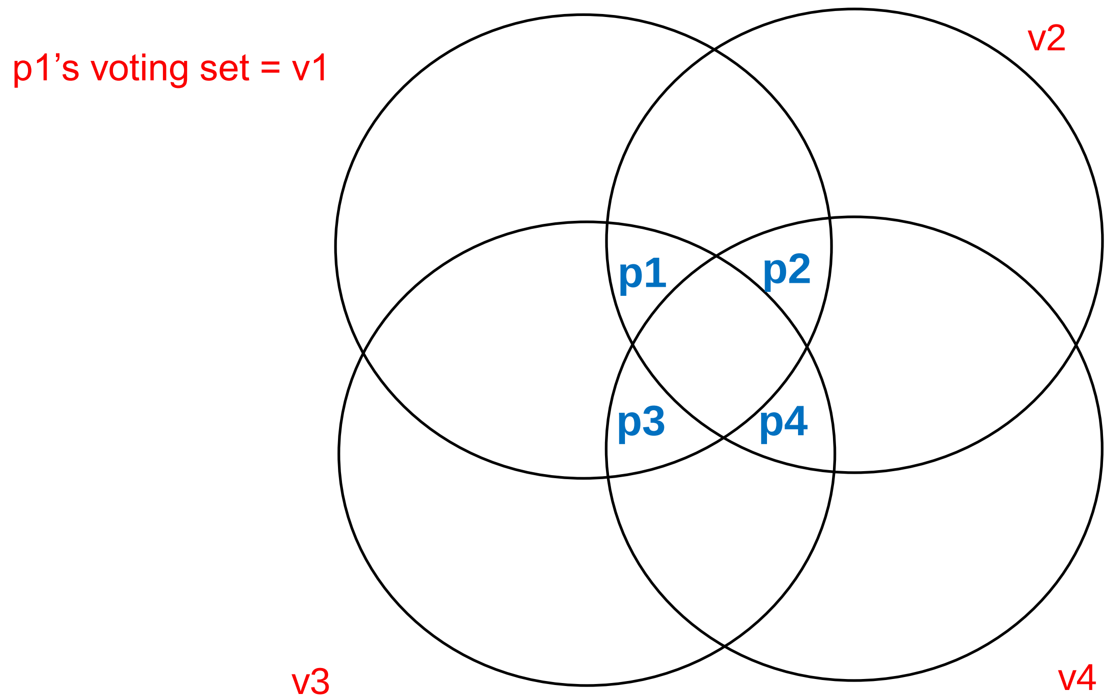
4. Timestamp Approach: Maekawa's Algorithm

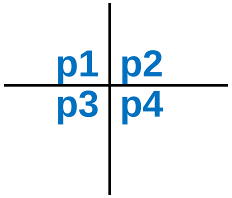
❖ Setup

- ❑ 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 is non-empty*
- ❑ 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, v_i = row + column containing p_i

Maekawa Voting Set with N=4





Timestamp Approach: Maekawa's Algorithm

- **Protocol**
  - Each process p_i is associated with a ***voting set*** v_i (of processes)
  - To access a critical section, p_i requests permission from all other processes in its own voting set v_i
  - Voting set member gives permission to only one requestor at a time, and queues all other requests
  - Guarantees safety
  - May not guarantee liveness (may deadlock)

Maekawa's Algorithm – Part 1

On initialization

state := RELEASED;

voted := FALSE;

For p_i to enter the critical section

state := WANTED;

Multicast *request* to all processes in $V_i \cancel{- \{p_i\}}$;

Wait until (number of replies received = $(K \cancel{- 1})$);

state := HELD;

On receipt of a request from p_i at p_j $(i \cancel{\neq j})$

if (*state* = HELD *or voted* = TRUE)

then

queue *request* from p_i without replying;

else

send *reply* to p_i;

voted := TRUE;

end if

Continues on next slide

Maekawa's Algorithm – Part 2

For p_i to exit the critical section

  state := RELEASED;

  Multicast *release* to all processes in $V_i \cancel{-} \{\cancel{p_i}\}$;

On receipt of a release from p_i at p_j $(\cancel{i \neq j})$

  if (queue of requests is non-empty)

  then

    remove head of queue – from p_k, say;

    send *reply* to p_k;

    voted := TRUE;

  else

    voted := FALSE;

  end if

Maekawa's Algorithm – Analysis

- 2√N messages per entry, √N messages per exit
  - Better than Ricart and Agrawala's (2(N-1) and N-1 messages)
- Client delay: One round trip time
- Synchronization delay: 2 message transmission times

Summary

- **Mutual exclusion**
  - Semaphores review
  - Coordinator-based token
  - Token ring
  - Ricart and Agrawala's timestamp algo.
  - Maekawa's algo.
- **MP2 due this Sunday midnight**
  - By now you should have a fully working system, and be taking measurements
- **Demos next Monday 2-6 pm**
  - Watch Piazza for Signup sheet