Mutual Exclusion

CS425 /ECE428 – DISTRIBUTED SYSTEMS – FALL 2021

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya

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 at most one client 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

Single-process mutex

```
await lock.acquire()
# do critical section
lock.release()
```

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 recipients' input buffer in FIFO order.
- Processes do not fail (why?)

Central server Mutex

```
# client
send(server, "request")
message = await receive(server)
assert message = "granted"
# do critical section
send(server, "release")
```

```
# server
async def process_message(message):
 if message == "request":
  await lock.acquire()
  send(message.sender, "granted")
 else:
  assert message == "release"
  lock.release()
```

Central server Mutex

```
# message loop
while True:
  message = receive_message()
  if message == None:
    break
  asyncio.create_task(
     process_message(message))
```

```
# server
async def process_message(message):
 if message == "request":
   await lock.acquire()
   send(message.sender, "granted")
 else:
   assert message == "release"
   lock.release()
```

Explicit Queue

```
def process_message(message):
  if message == "request:
    if held == False:
      held = True
      send(message.sender,
        "granted")
    else:
      queue.append(message)
else:
  assert message == "release"
    if queue.empty():
      held = False
    else:
      message = queue.pop(0)
      send(message.sender, "granted")
```

Refresher - Mutexes

To synchronize access of multiple threads to common data structures

Allows two operations:

```
lock()
        while true:                    // each iteration atomic
                if lock not in use:
                        label lock in use
                        break
unlock()
        label lock not in use
```

How are mutexes used?

```
mutex L= UNLOCKED;

ATM1:
        lock(L); // enter
                // critical section
        obtain bank amount;
        add in deposit;
        update bank amount;
        unlock(L); // exit
```

```
extern mutex L;

ATM2
        lock(L); // enter
                // critical section
        obtain bank amount;
        add in deposit;
        update bank amount;
        unlock(L); // exit
```

One Use: Mutual Exclusion – Bank ATM example

1. Centralized Control of Mutual Exclusion

Central coordinator arbitrates CS access
- elected (next lecture)

Upon receiving *request* message from *p*
- If not in critical section, send *grant* to *p*
  - Mark self as in CS
- Else, add *p*'s request to the *queue*

Upon receiving *release* message
- Send *grant* message to first process in the *queue*
- If *queue* empty
  - Mark self as not in CS

Client enter CS
- Send *request* message
- Wait for *grant* message

Client exit CS
- Send *release* message

Guarantees
- Safety
- Liveness
- ~~Ordering~~

Bandwidth: 2 msg / enter,
1 msg / exit

Client delay: 1 RTT (request/grant)

Synchronization delay: 2 one-way latencies

Server code:

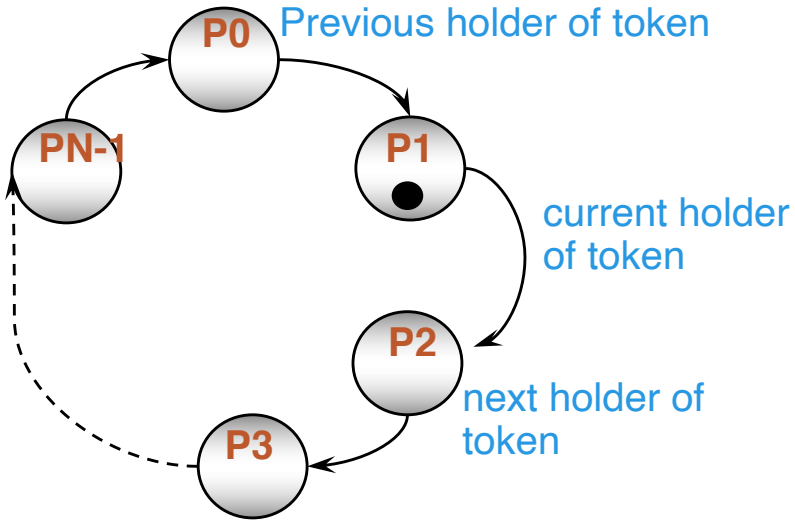
```
Lock L;
handle_message() {
  if (message = request) {
    lock(L);
    send(grant)
  } else
if (message = release) {
  unlock(L);
}
```

2. Token Ring Approach

Processes are organized in a logical ring: pi has a communication channel to
pi+1 mod (n).

Operations:

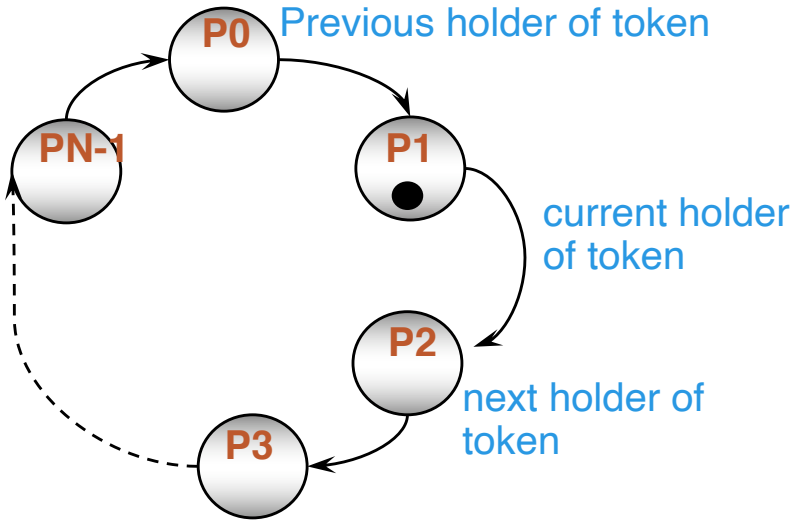
- Only the process holding the token can enter the CS.
- To enter the critical section, wait passively for the token. When in CS, hold on to the token.
- To exit the CS, the process sends the token onto its neighbor.
- If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.



2. Token Ring Approach

Features:

- Safety & liveness are guaranteed, but ordering is not.
- 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.



Multicast-based solution

state = { wanted, held, free }

enter:

- state = wanted
- multicast("request") to all other nodes
- wait for reply from each node
- state = held

exit:

- state = free
- reply to all processes in queue

When receiving a request message from p_i

If state = free

- Send reply to p_i

Else

- Add p_i to queue

Properties

- Safety?
- Liveness?

Ricart-Agrawala

state = { wanted, held, free }

enter:

- state = wanted
- multicast("request",T,p_i) to all other nodes
  - T: Lamport timestamp, used to break cycles
- wait for reply from each node
- state = held

exit:

- state = free
- reply to all processes in queue

When receiving a request message from p_i **with timestamp T**

If state = free **OR**

state = wanted and $T_i,p_i < T_j,p_j$

- Send reply to p_i

else

- Add p_i to queue

Ricart-Agrawala Diagram

Ricart-Agrawala Safety

If P_i and P_j both want to enter critical section with timestamps T_i and T_j

- Assume wolog $T_i, P_i < T_j, P_j$

Case 1: P_j's request arrives at P_i when P_i is in wanted state

- P_i queues request until after critical section

Case 2: P_j's request arrives when P_i is in free state

- Then P_i must have finished CS already
- Otherwise P_j's request happened before (Lamport) P_i's
  - Then $T_i > T_j$ (contradiction)

Ricart-Agrawala Properties

Corollary 1:

- If $T_i < T_j$ then P_i enters critical section before P_j

Corollary 2: Liveness

- When P_i wants to enter with timestamp T_i, it only needs to wait for processes with lower timestamps
- When each process receives request, all future requests will have higher timestamps

Corollary 3: Ordering

- Requests are satisfied in causal order
- ~ best you can do!

Ricart-Agrawala Performance

Bandwidth

- 1 multicast message (request)
- N-1 unicast messages (replies)
- O(N) unicast messages if implemented using B-multicast

Client delay

- O(1) if propagation delay dominates
- O(N) if transmission delay dominates

Synchronization delay

- O(1)

Delay components

- Transmission delay: time to write message to wire
- Propagation delay: time for signals to travel to destination

Transmission delay, 100 byte msg:

- 1 Gbps wired: 800 ns; WiFi: 1-3 us

Propagation delay:

- Same data center: 100-200 us
- Same city: 1-10 ms
- Inter-city: 10-200 ms

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

Maekawa's algorithm

- First solution with a sublinear O(sqrt N) message complexity.

- "Close to" Ricart-Agrawala's solution, but each process is required to obtain permission from only a subset of peers

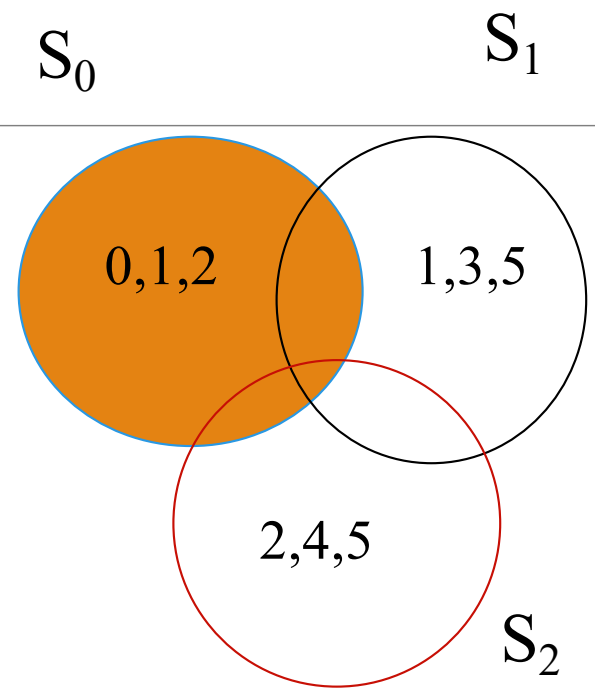
Maekawa's algorithm

With each process i, associate a subset Si. Divide the set of processes into subsets that satisfy the following two conditions:

$i \in Si$

$\forall i,j : 0 \leq i,j \leq n-1 \mid Si \cap Sj \neq \emptyset$

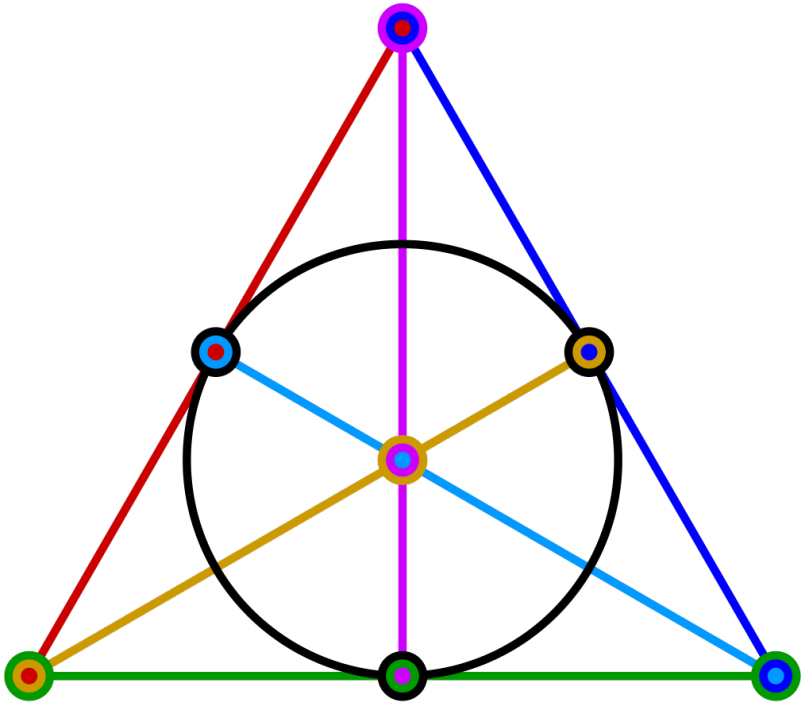
Main idea. Each process i is required to receive permission from Si only. Correctness requires that multiple processes will never receive permission from all members of their respective subsets.



Maekawa's algorithm

Example. Let there be seven processes 0, 1, 2, 3, 4, 5, 6

S0	=	{0, 1, 2}
S1	=	{1, 3, 5}
S2	=	{2, 4, 5}
S3	=	{0, 3, 4}
S4	=	{1, 4, 6}
S5	=	{0, 5, 6}
S6	=	{2, 3, 6}



Maekawa's algorithm

Version 1 {Life of process I}

1. Send timestamped request to each process in Si.

2. Request received → send ack to process with the lowest timestamp. Thereafter, "lock" (i.e. commit) yourself to that process, and keep others waiting.

3. Enter CS if you receive an ack from each member in Si.

4. To exit CS, send release to every process in Si.

5. Release received → unlock yourself. Then send ack to the next process with the lowest timestamp.

S0	=	{0, 1, 2}
S1	=	{1, 3, 5}
S2	=	{2, 4, 5}
S3	=	{0, 3, 4}
S4	=	{1, 4, 6}
S5	=	{0, 5, 6}
S6	=	{2, 3, 6}

Maekawa's algorithm-version 1

Safety. At most one process can enter its critical section at any time.

Let i and j attempt to enter their Critical Sections

$S_i \cap S_j \neq \emptyset$ implies there is a process $k \in S_i \cap S_j$

Process k will never send ack to both.

So it will act as the arbitrator and establishes ME1

S0	=	{0, 1, 2}
S1	=	{1, 3, 5}
S2	=	{2, 4, 5}
S3	=	{0, 3, 4}
S4	=	{1, 4, 6}
S5	=	{0, 5, 6}
S6	=	{2, 3, 6}

Maekawa's algorithm-version 1

Liveness. Unfortunately deadlock is possible! Assume 0, 1, 2 want to enter their critical sections.

From **S_0= {0,1,2}**, **0,2** send *ack* to **0**, but **1** sends *ack* to **1**;

From **S_1= {1,3,5}**, **1,3** send *ack* to **1**, but **5** sends *ack* to **2**;

From **S_2= {2,4,5}**, **4,5** send *ack* to **2**, but **2** sends *ack* to **0**;

Now, 0 waits for 1 **(to send a release),** 1 waits for 2 **(to send a release),** , and 2 waits for 0 **(to send a release),** . So deadlock **is** possible!

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

$S_6 = \{2, 3, 6\}$

Maekawa's algorithm-Version 2

Avoiding deadlock

If processes always receive messages **in increasing order of timestamp**, then deadlock "could be" avoided. But this is too strong an assumption.

Version 2 uses three ***additional*** messages:

- ***failed***
- ***inquire***
- ***relinquish***

$S_0 = \{0, 1, 2\}$

$S_1 = \{1, 3, 5\}$

$S_2 = \{2, 4, 5\}$

$S_3 = \{0, 3, 4\}$

$S_4 = \{1, 4, 6\}$

$S_5 = \{0, 5, 6\}$

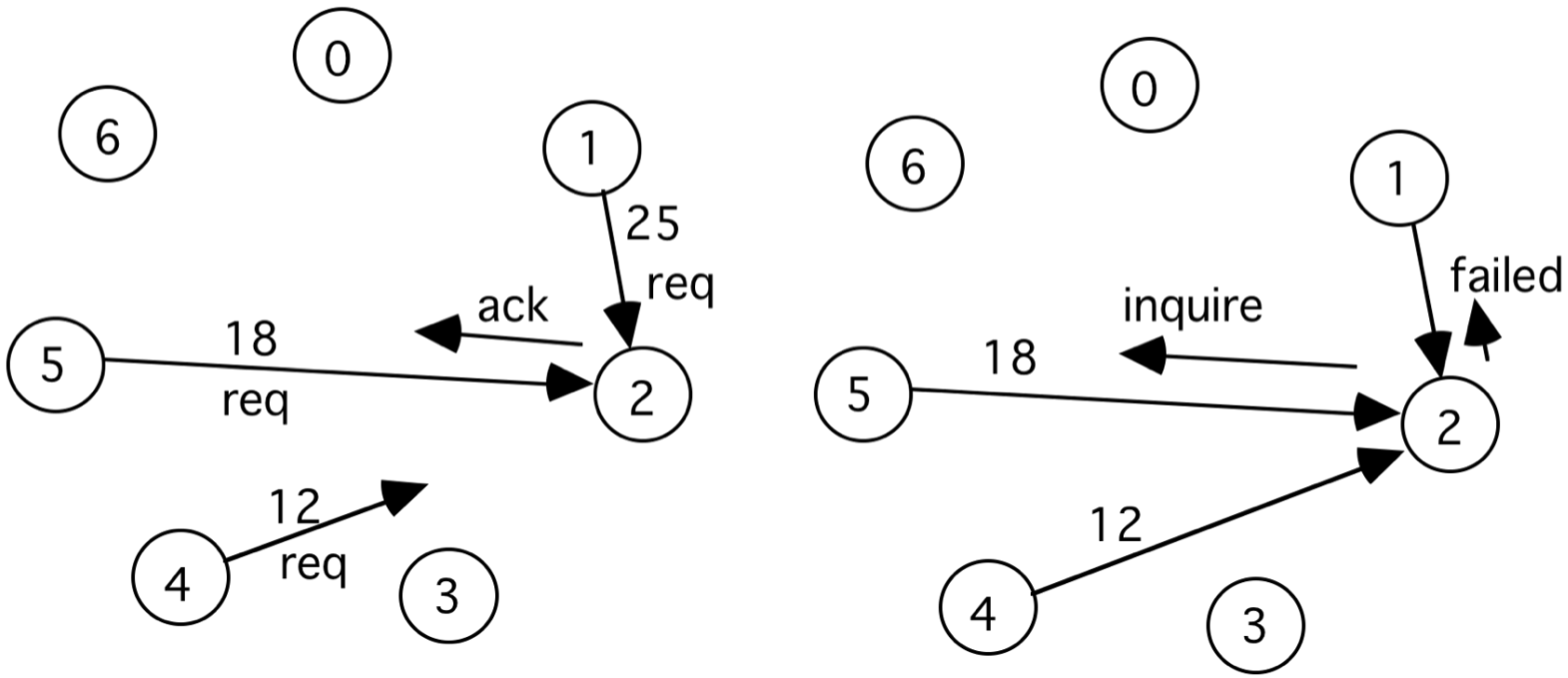
$S_6 = \{2, 3, 6\}$

Maekawa's algorithm-Version 2

New features in version 2

- Send ***ack*** and set **lock** as usual.
- If **lock is set** and a request with a larger timestamp arrives, send ***failed*** *(you have no chance)*. If the incoming request has a lower timestamp, then send ***inquire*** *(are you in CS?) to the locked process*.
- Receive ***inquire*** and at least one ***failed*** message → send ***relinquish***. The recipient resets the lock.

Maekawa's algorithm-Version 2

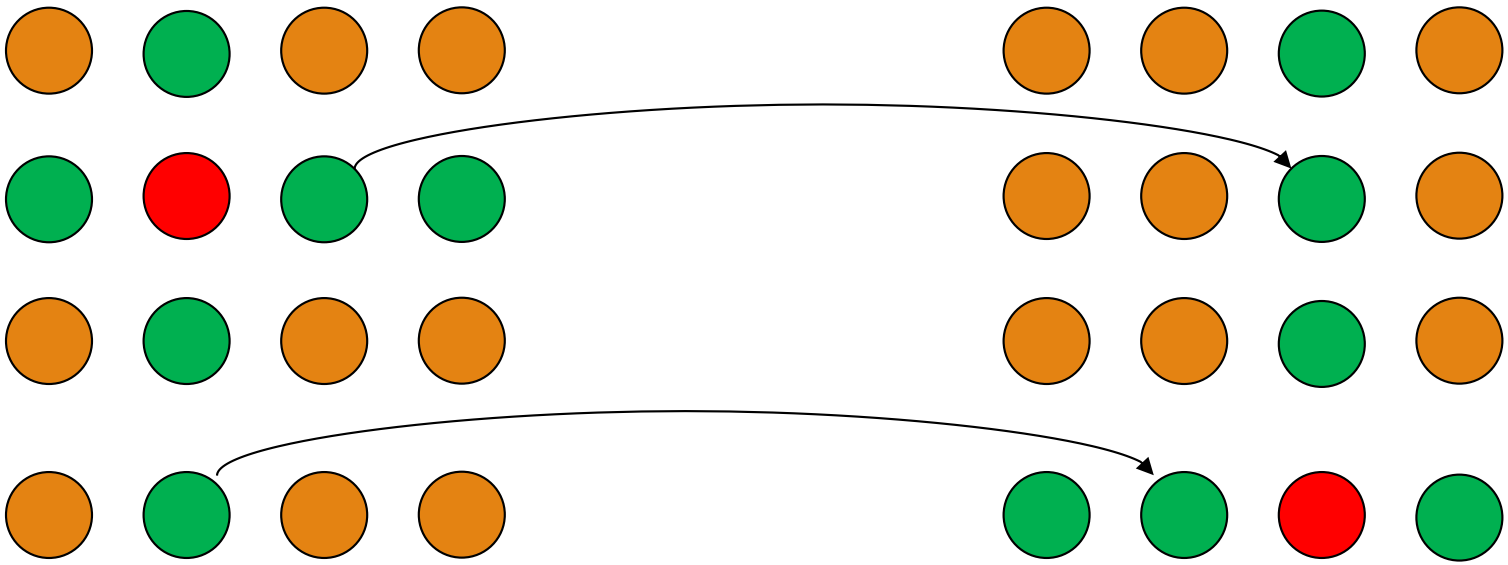


Comments

Let K = |Si|. Let each process be a member of D subsets. When N = 7, K = D = 3. When K=D, N = K(K-1)+1. So K =O(√N)

- The message complexity of Version 1 is 3√N. Maekawa's analysis of Version 2 reveals a complexity of 7√N

Matrix construction



Summary

Mutual exclusion

- Coordinator-based token
- Token ring
- Ricart and Agrawala's timestamp algo.
- Maekawa's algo.