Coordination and Agreement

Web Applications and Services

Spring Term

Naercio Magaia

UNIVERSITY OF SUSSEX

Contents

- Distributed Mutual Exclusion
- Consensus

Distributed Mutual Exclusion

- Distributed processes need to coordinate their activities
- Mutual exclusion is required, when processes share resources, to prevent interference and ensure consistency
- Shared resources may reside on a server, or a collection of peer processes must coordinate their accesses amongst themselves
- An asynchronous distributed system of N processes $p_i, i = 1, 2, \ldots N$ that do not share variables – communicate through messaging

UNIVERSITY OF SUSSEX

Distributed Mutual Exclusion

- Application-level protocol for executing a critical section (CS)
  - `enter()` (may block), `resourceAccess()`, `exit()`
- Essential requirements for mutual exclusion

1. *Safety*: at most one process may execute in the CS at a time
2. *Liveness*: requests to enter and exit the CS <u>eventually succeed</u>
3. *Happened-before ordering:* if one request to enter the CS happened-before another, then entry to the CS is granted in that order

UNIVERSITY OF SUSSEX

Distributed Mutual Exclusion

- Requirement 2 (i.e., *liveness*) implies freedom from deadlock and starvation
- Requirement 3 is about fairness: if all requests are related by happened-before ordering, then <u>it is not possible</u> for a process to enter the CS more than once while other waits to enter
- Evaluation criteria
  - number of messages sent for *enter* and *exit* operation
  - synchronisation delay (p_1 exiting and p_2 entering)

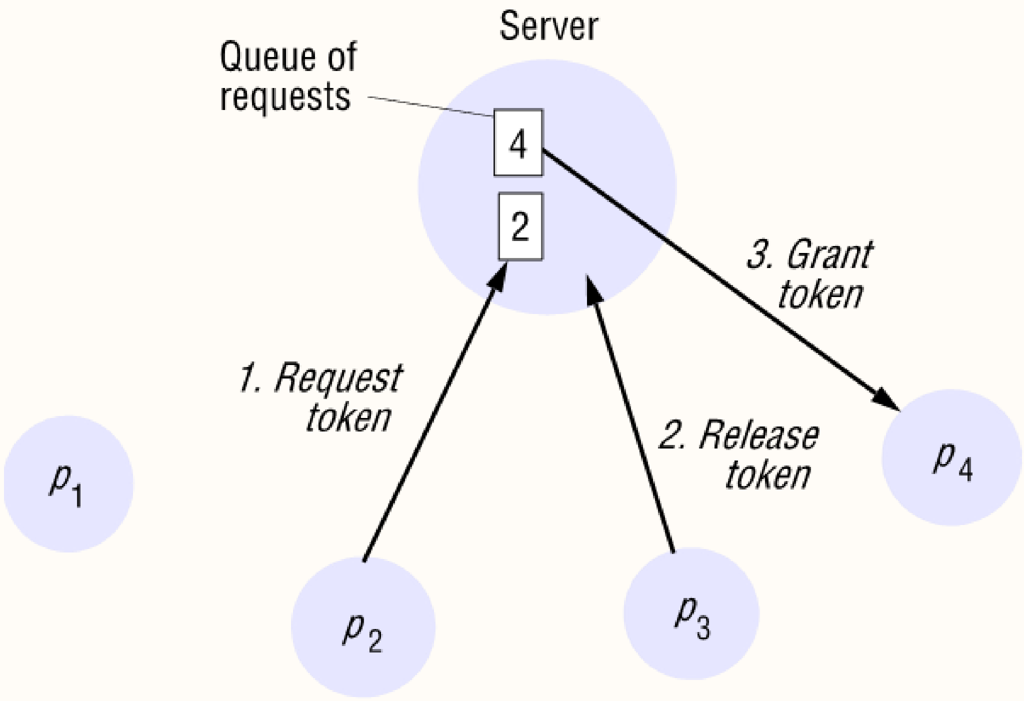
Central Server Algorithm

- A process sends a request message to the server and waits for a permission token (i.e., the reply)
- The server replies immediately, if no other process has the token at the time of the request, or queues the request if token is given
- When a process exits the CS, it sends the token back to the server
- If the queue of waiting processes is not empty, the server chooses the oldest entry in the queue, removes it and replies to the corresponding process

UNIVERSITY OF SUSSEX

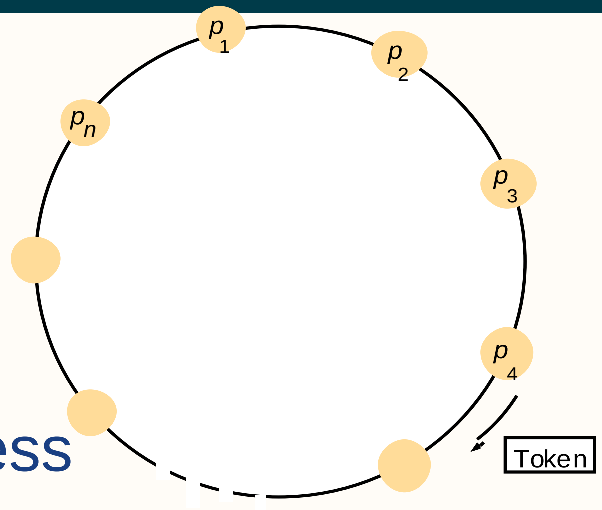
Central Server Algorithm

- *safety* and *liveness* requirements are met
- *happened-before ordering* requirement isn't
- Entering the CS takes two messages
- Exiting the CS takes one message
- Synchronization delay is a round-trip
- Failures?
- How to (s)**elect** the server?



UNIVERSITY OF SUSSEX

Ring-based Algorithm



- No Central Server
- *N* processes arranged in a logical ring
- Each process has a communication channel to the next process
- A token is passed from process to process in a single direction
- If no access to CS is required, token is passed to next process
- A process that needs access to the CS waits for the token

UNIVERSITY OF SUSSEX

Ring-based Algorithm

- Upon exiting the CS, the token is passed to the next process
- *liveness* and *safety* are met but <u>not</u> the *happened-before ordering*
  - Processes exchange messages independently of the rotation of the token
- Bandwidth is constantly consumed
  - Processes send messages around the ring even when no process requires access
- Delay to enter
  - 0 to N messages

UNIVERSITY OF SUSSEX

Ring-based Algorithm

- Delay to exit
  - Only 1 message
- Synchronization delay
  - 1 to N messages

UNIVERSITY OF SUSSEX

Mutual Exclusion with Multicast and Logical Clocks

- Processes multicast a request message
  - enter only when all the other processes have replied
- Each process keeps a *Lamport Virtual Clock*
  - updated for internal events and when receiving messages
- Messages requesting entry are of the form $< T, p_i >$
- Each process records its state of being outside the CS (*RELEASED*), wanting entry (*WANTED*) or being in the CS (*HELD*) in a variable

UNIVERSITY OF SUSSEX

Mutual Exclusion with Multicast and Logical Clocks

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;

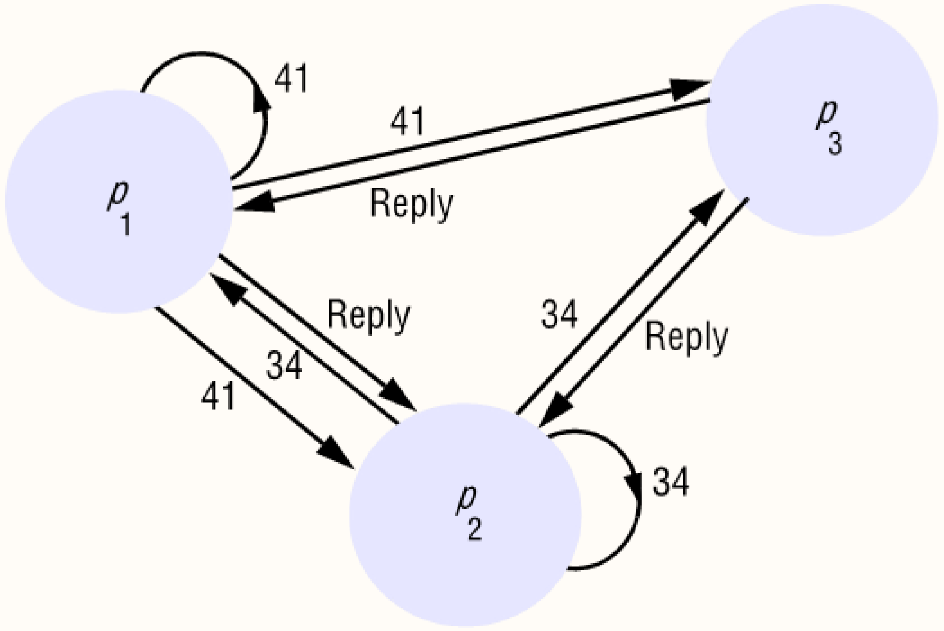
Request processing deferred here

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;

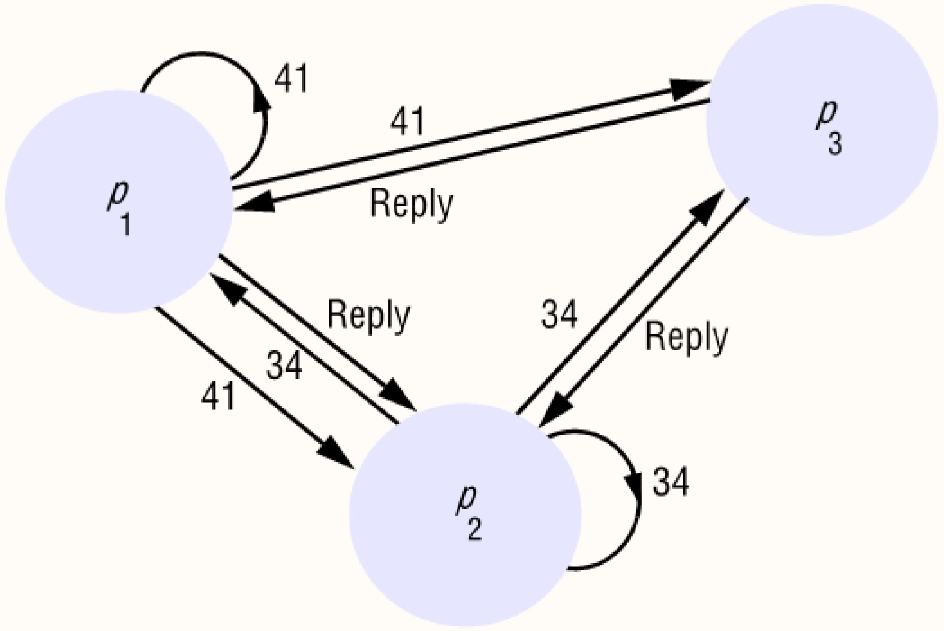
Mutual Exclusion with Multicast and Logical Clocks

- *Safety*
  - If it were possible for two processes to enter the CS at the same time, both of those processes would have to have replied to the other ($< T_i, p_i >$ are causally ordered)
- *Liveness*
  - when exiting the CS, a process replies to all pending requests
- *Happened-before ordering*
  - Lamport Clocks



Mutual Exclusion with Multicast and Logical Clocks

- Delay to enter
  - *2*(N – 1)* messages or *N* if native multicast is supported
- Delay to exit
  - up to *N - 1*
- Synchronization delay
  - 1 message (i.e., one process waits for only 1 message)



UNIVERSITY OF SUSSEX

Consensus

- A collection of processes p_i $(i = 1, 2, \ldots N)$ communicating by message passing
- Requirement: consensus can be reached even in the presence of faults
  - communication is reliable
  - processes may fail (crash or Byzantine (i.e., arbitrary) process failures)
    - up to some number f of the N processes are faulty

UNIVERSITY OF SUSSEX

Consensus

- To reach consensus, every process p_i begins in the <u>undecided</u> state and proposes a single value v_i, drawn from a set D
- Processes communicate by exchanging values
- In the end, each process sets the value of a decision variable d_i and enters the <u>decided</u> state

Consensus

- The requirements of a consensus algorithm is that the following conditions should hold for its every execution
  - Termination
    - Eventually each correct process sets its decision variable
  - Agreement
    - The decision value of all correct processes is the same
  - Integrity
    - If all correct processes proposed the same value, then any correct process in the decided state has chosen that value

Consensus

- Consensus in the absence of failures is straightforward!
  - Each process reliably multicasts a proposed value and waits for all other values
  - Selects the decided value using an appropriate method (majority, min or max)

UNIVERSITY
OF SUSSEX

Consensus in a Synchronous System

- Up to f of the N processes may exhibit crash failures. Timeouts are used to detect.

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f+1)$

B-multicast(g, $Values_i^r - Values_i^{r-1}$); // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f+1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

Consensus in a Synchronous System

- All correct processes will have all proposed values to decide in the end
- If a process failed while multicasting its value either one or none of the processes will circulate this value
- Termination is obvious, i.e., the system is synchronous
- To check the correctness of the algorithm, we must show that each process arrives at the same set of values at the end of the final round

Consensus in a Synchronous System

- Agreement and integrity will then follow
- Assume that two processes differ in their final set of values
- Some correct process p_i possesses a value v that another correct process p_j doesn't
- <u>Only possible explanation</u>: some other process p_k sent v to p_i and then crashed

UNIVERSITY OF SUSSEX

Consensus in a Synchronous System

- But why p_k possesses v but not p_i? Another process crashed in the previous round
- We assumed f crash failures and we have $f + 1$ rounds – a contradiction

UNIVERSITY OF SUSSEX

Impossibility in asynchronous systems

- The assumption above was that message exchanges take place in rounds
    - …processes are entitled to timeout and assume that a faulty process has not sent them a message within the round, because the maximum delay has been exceeded
- No algorithm can <u>guarantee</u> to reach consensus in an asynchronous system, even with one process crash failure
  - The proof is beyond the scope of this module

Impossibility in asynchronous systems

- In an asynchronous system, processes can respond to messages at arbitrary times, so a crashed process is indistinguishable from a slow one
- Note the word <u>guarantee</u>!
- Masking faults
- Consensus using failure detectors
  - Ignore processes that are considered failed (even if they are not)
  - Large timeouts

Next Lecture ...

- ✓ Introduction
- ✓ HTTP, Caching, and CDNs
- ✓ Views
- ✓ Templates
- ✓ Forms
- ✓ Models
- ✓ Security
- ✓ Transactions
- ✓ Remote Procedure Call
- ✓ Web Services
- ✓ Time
- ✓ Elections and Group Communication
- ➢ **Coordination and Agreement**

UNIVERSITY OF SUSSEX