Distributed Systems (5DV147)

Mutual Exclusion and Elections

Fall 2014

Distributed mutual exclusion

Motivation

- Is needed to coordinate access to a shared resource
  - Concurrent access to a shared resource is serialized

    … but the solution need to be based on message passing
- Three basic approaches
  - Token-based
  - Permission-based (Timestamp-based)
  - Quorum-based

Assumptions

- The system is asynchronous, process do not fail, and message delivery is reliable
- N processes `p`$_i$ `(i=1, 2, …, N)` that do not share variables
  - `p`$_i$ access shared resources in a critical section
  - `p`$_i$`’s` are well behaved, finite time on the critical section

```
enter()
resourceAccesses()
exit()
```

$\}$ Application level protocol for executing a critical section

Essential requirements

Safety: At most 1 process may enter the critical section at a time

Liveness: requests to enter and exit the critical section eventually succeed

- Freedom of *deadlock* and *starvation*

→ ordering: if a request to enter the critical section *happened-before* another, then access is granted according to that order

Fairness

- Absence of starvation
- Maintain the order in which requests are made
  - No global clocks
  - Happened-before ordering:
    - it is not possible for a process to enter the critical section more than once while another waits to enter

Criteria for evaluating algorithms

- Bandwidth consumed
  - Number of messages for entry and exit operations
- Client delay
  - Depends how many processes want access and how (typically) long are those accesses
    - Short and rarely, dominant factor is the algorithm
    - Long and frequent, dominant factor is waiting for everyone to take a turn
- Throughput of the system
  - Synchronization delay, one process exiting and another one entering the critical section

MUTEX Algorithms

Central Server

- ☐ Send request to server, oldest process in queue gets access (a *token*), return token when done
- ☐ No process has token → reply (enter) immediately
- ☐ Otherwise → queue request
- ☐ Oldest process in the queue gets token after released

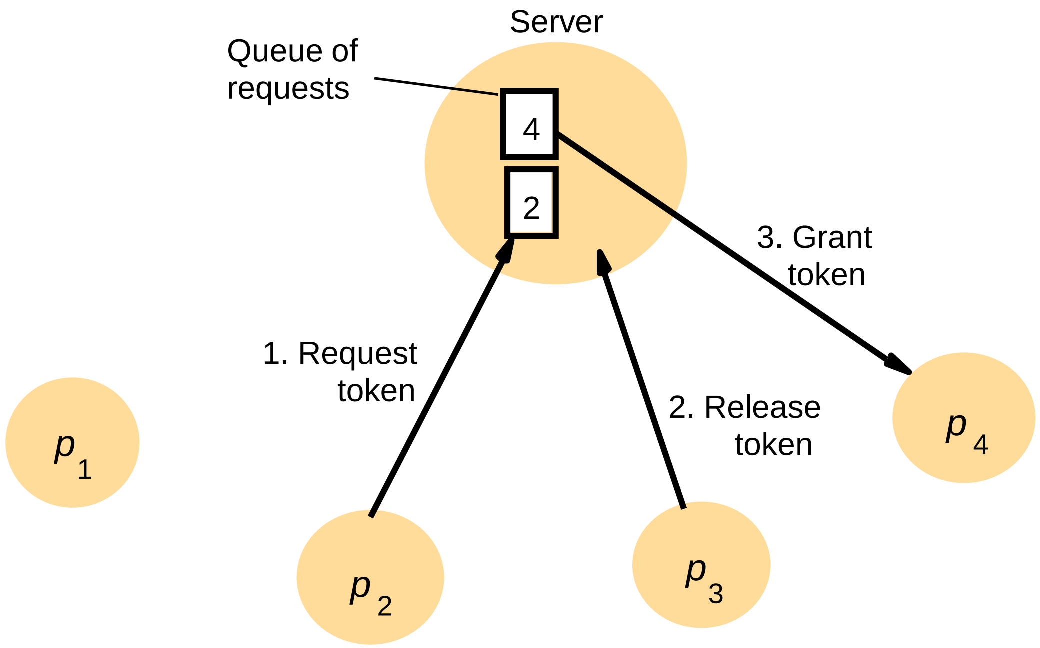


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 15.2**

Properties

- Safety? Yes! (no deadlock and no starvation)
- Liveness? Yes (as long as server does not crash)
- → ordering? No! Why not?
- Performance
  - **Entering** : 2 messages (request + grant)
  - **Exiting** : 1 message
  - **Client delay** : 2 messages (request + grant)
  - **Synch delay** : 2 messages (release + grant)

Performance bottleneck

Single point of failure

Ring-based

- ❑ Token is passed around a ring of processes
  - ➢ Want access? Wait until token comes, and claim it (then pass the token along)
  - ➢ Can’t use the same token twice
- ❑ Can’t estimate when a process will see a token
- ❑ To recover from a process crash
  - ➢ Receipt acknowledgments

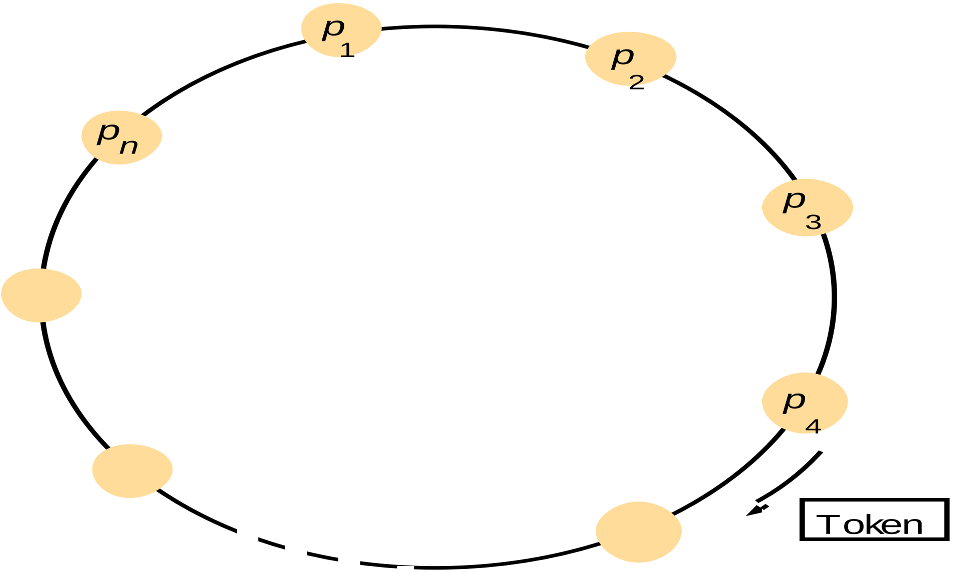


Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 15.3**

Properties

- Safety? Liveness? Yes (assuming no crashes)
- → ordering? Not even close!
- Performance
  - Continuously uses network bandwidth
  - Client delay : between 0 – N messages
  - Exiting : 1 message
  - Synchronization delay : between 1 – N messages

Ricart and Agrawala

- ❑ Distributed algorithm, no central coordinator
  - ➢ Use Lamport’s timestamps to order requests
- ❑ Multicast a request message
  - ➢ Enter critical section only when all other processes have given permission
  - ➢ Processes work cooperatively to provide access in a fair order
- ❑ Use multicast primitive or each process needs a group membership list

Details

- ❑ Each process
  - ➢ Has unique process ID
  - ➢ Has communication channels to the other processes
  - ➢ Maintains a logical (Lamport) clock
  - ➢ Is in a state ∈ `{wanted, held, released}`
- ❑ Requests are multicasted to group
- ➢ (process ID and clock value) `<id, value>`
- ❑ Lowest clock value gets access first
  - ➢ Equal values? Check process ID!

MUTEX: Algorithms

```
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;
```

request processing deferred here

Have access or want access and <id, value> is lower than incoming request?

RELEASED or earlier timestamp

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 15.5**

Example

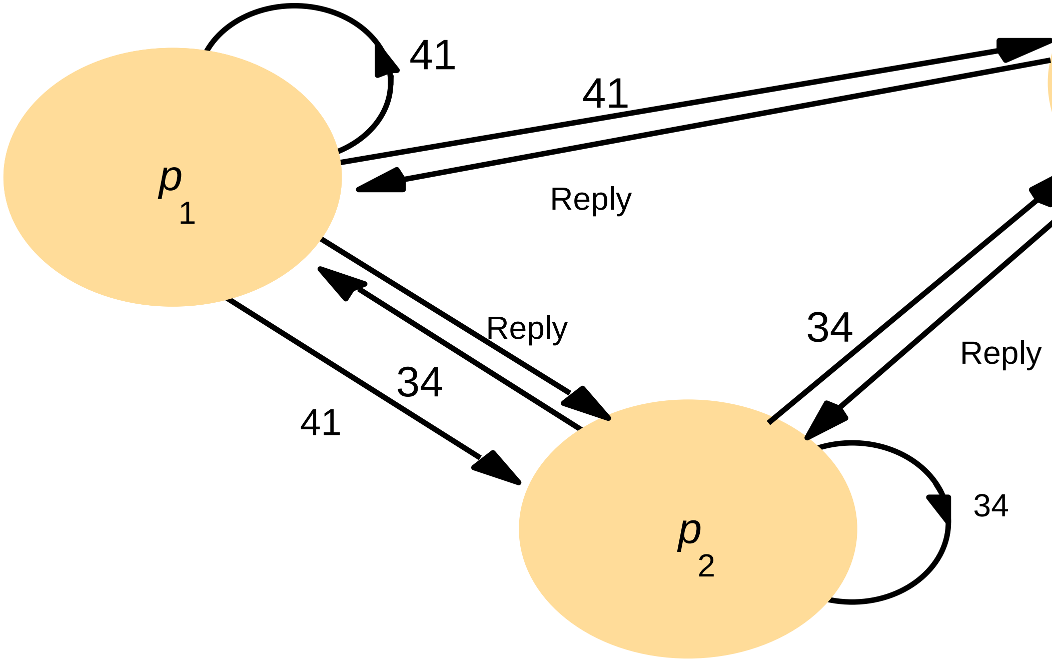


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 15.5**

Properties

- ❑ Safety? Liveness? → ordering? Yes!
  - ➢ …but every node is a point of failure
- ❑ Performance
  - ➢ Entering : 2(n-1) messages

    (n-1) multicast request + (n-1) replies
  - ➢ Client delay : 2(n – 1)
  - ➢ Synchronization delay : 1 message transmission
- ❑ Improved performance
  - If process wants to re-enter critical section, and no new requests have been made, just do it!
  - Grant access using simple majority

Maekawa’s voting

Optimization: need to only ask a subset of processes for entry

- ❑ Key is how to build the subsets
  - ➢ At least one common member in any two voting sets
  - ➢ Every voting set is of the same size
  - ➢ Each process is in as many voting sets as the number of processes in a voting set
  - ➢ Works as long as subsets overlap
  - ➢ Use matrix of $\sqrt{n}$ by $\sqrt{n}$ and voting sets are the union of rows and columns
- ❑ Processes can vote only in one election at a time

Details

```
On initialization
    state := RELEASED;
    voted := FALSE;
For $p_i$ to enter the critical section
    state := WANTED;
    Multicast request to all processes in $V_i$;
    Wait until (number of replies received = $K$);
    state := HELD;
On receipt of a request from $p_i$ at $p_j$
    if (state = HELD or voted = TRUE)
    then
        queue request from $p_i$ without replying;
    else
        send reply to $p_i$;
        voted := TRUE;
    end if
```

```
For $p_i$ to exit the critical section
    state := RELEASED;
    Multicast release to all processes in $V_i$;
On receipt of a release from $p_i$ at $p_j$
    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
```

Properties

- ❑ Safety? Yes
- ❑ Liveness? No, deadlocks can happen! $V_1=\{p_1,p_2\}$, $V_2=\{p_2,p_3\}$, $V_3=\{p_3,p_1\}$
- ❑ → ordering? No, but can
  - ➢ Add Lamport's clocks
  - ➢ Retrieve votes if an earlier request arrives to a processor that has already voted
- ❑ Performance
  - ➢ Bandwidth: $3\sqrt{N}$, $2\sqrt{N}$ messages for entering and $\sqrt{N}$ messages to exit
  - ➢ Client delay : 1 round-trip
  - ➢ Synchronization delay : 1 round-trip

Comparison of algorithms

- Central server:
  - Simple and error-prone!
  - …but otherwise good performance!!
- Ring-based algorithm:
  - Also simple, but not single point of failure
  - Not fair at all!
- Ricart and Agrawala:
  - Completely distributed and decentralized
  - Slower, more expensive, and less robust
  - … but fair!
- Maekawa's voting algorithm:
  - Only a subset of processes grant access: works if subsets are overlapping

… more comparison

- ❑ Message loss?
  - ➢ None of the algorithms handle this
- ❑ Crashing processes?
  - ➢ Ring? No! others? depends
    - ▪ Central – not server nor process holding or having requested token
    - ▪ Ricart & Agrawala – no
    - ▪ Maekawa’s – only if crashed process is not in voting set

Summary

- ❑ Control access to shared resources
- ❑ Algorithms
  - ➢ Central server
  - ➢ Ring-based
  - ➢ Ricart and Agrawala
  - ➢ Maekawa's voting algorithm

Election algorithms

Motivation

- How to choose a process to play a particular role in the system
- Start with all process in same state
  - One process will reach state *leader*
  - Other processes will reach state *lost*
- Each process requires a unique identifier (totally ordered)
- Every process knows the id(s) of other (all) processes

Details

- Any process can call an election but can only call one election at a time
- Each process has the same local algorithm
- The elected process is the one with the largest identifier
- The election must always produce a unique winner

Essential requirements

Safety:

A participant has `elected`$_i$ `= False` or `elected`$_i$ `= P`, where `P` is chosen as the non-crashed process with the highest identifier

Liveness:

All processes participate and eventually set `elected`$_i$ to not `=False` *or* crash

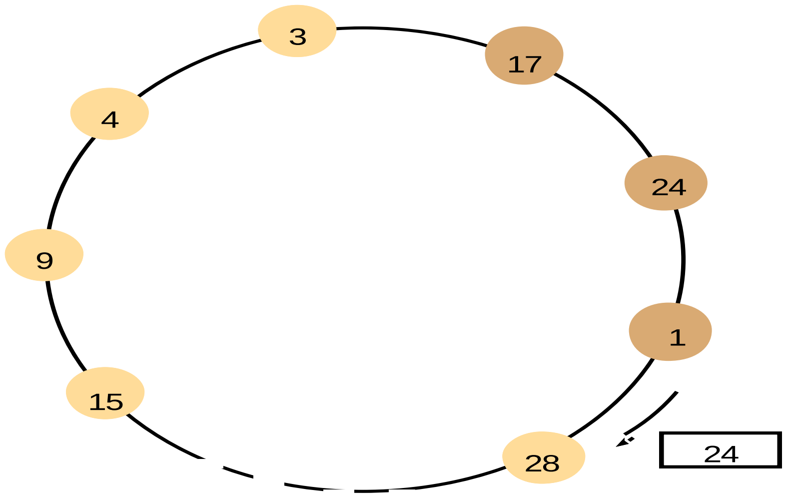
Election Algorithms

Ring-based algorithm

- Goal is to elect a single process – the *coordinator*
  - process with the largest identifier
- During election, pass `max(own ID, incoming ID)` to next process
  - If a process receives own `ID`, it must have been highest and may send that it has been elected

Details

- ☐ Safety? Liveness? Yes!
- ☐ Tolerates no failures (limited use)



Worst case, N-1 messages until reaching peer with largest identifier

\+

N messages to complete another circuit

\+

N messages advertising the election

3N-1 messages

- ❖ The election was started by process 17
- ❖ The highest process identifier encountered so far is 24.
- ❖ Participant processes are shown in a darker color

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 15.7**

Bully algorithm

- Requires:
  - Synchronous system
  - All processes know of each other (which ones have higher ids)
  - Reliable failure detectors
  - Reliable message delivery
- Allows
  - Crashing processes

Details

- Process P discovers that leader has crashed
    - P sends an Election message to all processes with higher numbers
    - If no one responds, P wins the election and becomes coordinator
    - If one of the higher ups answers, it takes over, P's job is done
- Upon receiving an Election message, the receiving process respond to sender and initiates an election

Example

- ☐ Safety? No
  - ➢ if process IDs can be reused!
- ☐ Liveness? Yes
  - ➢ if message delivery is reliable

The election of coordinator p_2, after the failure of p_4 and then p_3

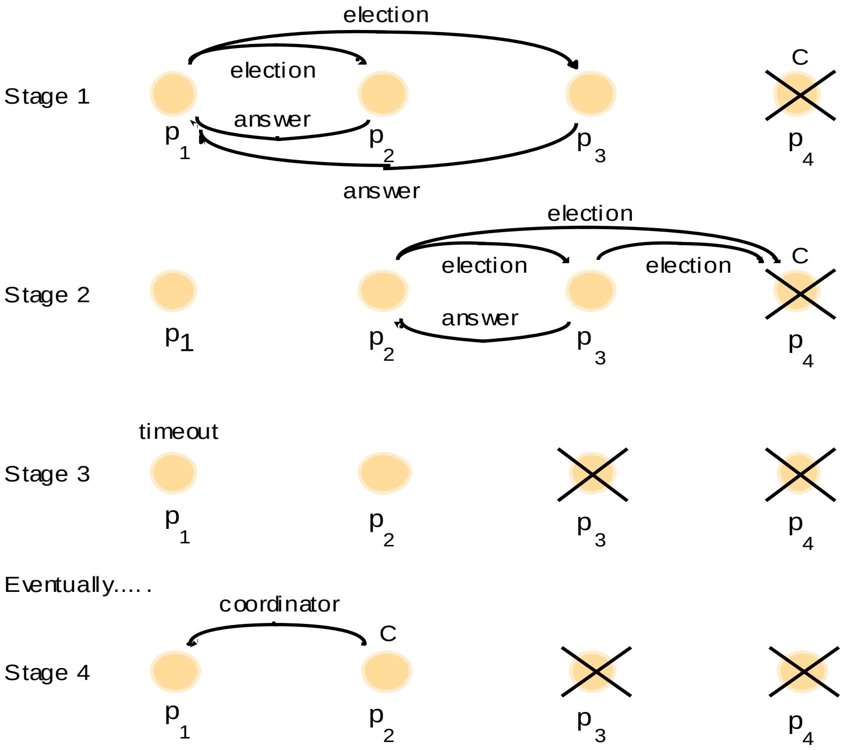


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – **Figure 15.8**

Summary

- Election algorithms
  - Seems like a simple problem, but non-trivial solutions are... non-trivial
  - Ring and Bully algorithms
- Want to read more about non-trivial election algorithms?
  - http://www.sics.se/~ali/teaching/dalg/l06.ppt

Next Lecture

Group Communication