Distributed systems – Theory
8. Distributed mutual exclusion

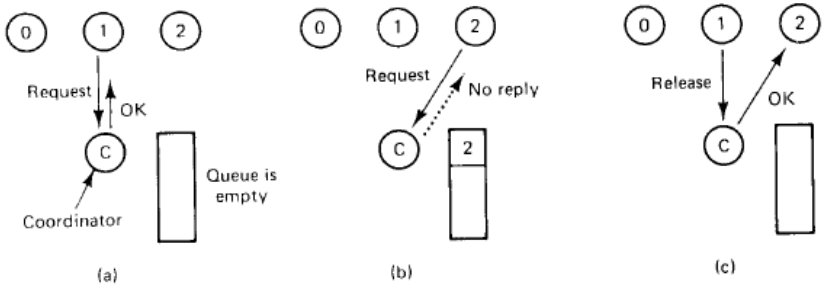
The problem

- Remember:
  - when a process has to read or update certain shared data structures, it enters a critical region to achieve mutual exclusion & ensure that no other process will use the shared data structures at the same time.
- In single-processor systems:
  - critical regions are protected using semaphores, monitors, and similar constructs.
- How critical regions and mutual exclusion can be implemented in distributed systems?

Centralized Algorithm

- Simulate how it is done in a one-processor system.
- One process is elected as the coordinator
  - e.g. the one running on the machine with the highest network address
- Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission.
- If no other process is currently in that critical region, the coordinator sends back a reply granting permission
- When the reply arrives, the requesting process enters the critical region.

Centralized alg. - example



- Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.
- Another process 2 asks for permission to enter the same critical region.
- The coordinator knows that a different process is already in the critical region, so it cannot grant permission.
- The exact method used to deny permission is system dependent.
  - (b): the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply.
  - Alternatively, it could send a reply saying "permission denied."
  - Either way, it queues the request from 2 for the time being.
- When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access - (c)
- The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.
  - If the process was still blocked (i.e. this is the first message to it), it unblocks and enters the critical region.
  - If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic, or block later.
  - Either way, when it sees the grant, it can enter the critical region.

Centralized Algorithm – pro and cons

- Guarantees:
  - mutual exclusion: the coordinator only lets 1 process at a time into each critical reg.
  - It is also fair, since requests are granted in the order in which they are received.
  - No process ever waits forever (no starvation).
- The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release).
- It can also be used for more general *resource allocation* rather than just managing critical regions.
- Shortcomings:
  - The coordinator is a single point of failure, so if it crashes, the entire system may go down.
  - If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back.
  - In a large system, a single coordinator can become a performance bottleneck.

Distr. Alg. of Ricart & Agrawala

- Requires that there be a total ordering of all events in the system:
  - For any pair of events, such as messages, it must be unambiguous which one happened first.
  - Lamport's algorithm is one way to achieve this ordering and can be used to provide time-stamps for distributed mutual exclusion.
- When a process wants to enter a critical region,

1. it builds a message containing the name of the critical region it wants to enter, its process number, and the current time.
2. It then sends the message to all other processes, conceptually including itself.
   - The sending of messages is assumed to be reliable; that is, every message is acknowledged.
   - Reliable group communication if available, can be used instead of individual messages.

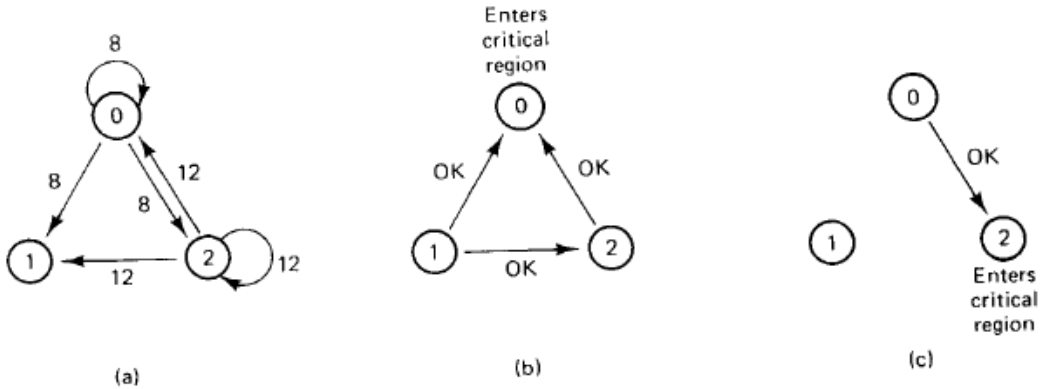
Distr. Alg. of Ricart & Agrawala

- When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message:

1. If the receiver is not in the critical region & does not want to enter it => sends back an OK message to the sender.
2. If the receiver is already in the critical region => it does not reply + queues the request.
3. If the receiver wants to enter the critical region but has not yet done so =>
   - it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone.
   - The lowest one wins:
     - If the incoming message is lower, the receiver sends back an OK message.
     - If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

- After sending out requests asking permission to enter a critical region,
  => process waits until everyone else has given permission.
- As soon as all the permissions are in,
  => process may enter the critical region.
- When it exits the critical region,
  => process sends OK messages to all processes on its queue
  & deletes them all from the queue.

Example



- If there is no conflict, it clearly works.
- Suppose that two processes try to enter the same critical region simultaneously:
  - process 0 sends everyone a request with timestamp 8, while at the same time,
  - process 2 sends everyone a request with timestamp 12.
- Process 1 is not interested in entering the critical region, so it sends OK to both senders.
- Processes 0 and 2 both see the conflict and compare timestamps.
  - Process 2 sees that it has lost, so it grants permission to by sending OK.
  - Process 0
    - queues the request from 2 for later processing and
    - enters the critical region – (b)
    - when it is finished, it removes the request from 2 from its queue
    - sends an OK message to process 2, allowing the latter to enter its critical region - (c).

Pro and cons

- Mutual exclusion is guaranteed without deadlock or starvation.
- No. messages required per entry is now 2(*n* -1) where the total number of processes in the system is *n*.
- No single point of failure exists – has been replaced by *n* points of failure!
  - If any process crashes, it will fail to respond to requests.
  - This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions.
  - Probability of one of the *n* processes failing is *n* times as large as a single coordinator failing => replacement of a poor alg. with one that is worse & requires much more network traffic
- Patched up:
  - When a request comes in, the receiver always sends a reply, either granting or denying permission.
  - Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead.
  - After a request is denied, the sender should block waiting for a subsequent OK message.
- Another problem:
  - either a group communication primitive must be used, or
  - each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing.

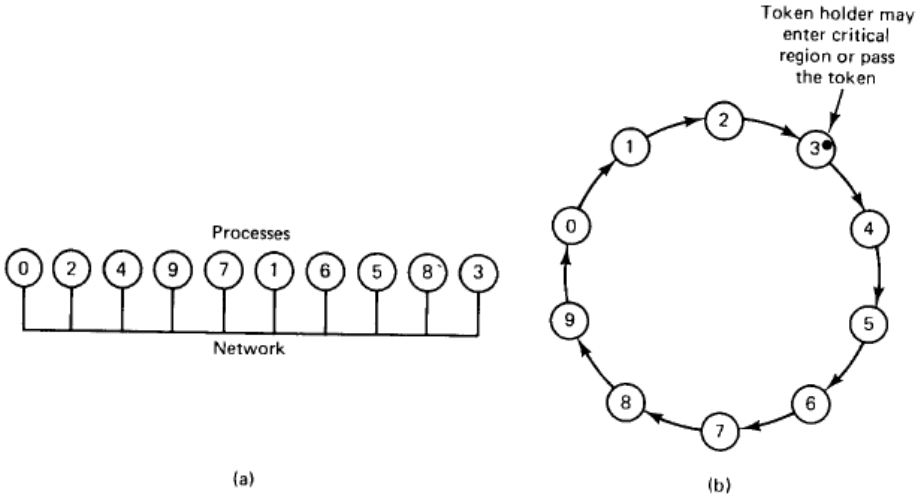
  => the method works best with small groups of processes that never change their group memberships.
- All processes are involved in all decisions concerning entry into critical regions.
  - If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Minor improvements

- Example:
  - getting permission from everyone to enter a critical region is really overkill => ? method to prevent two processes from entering the critical region at the same time.
  - Allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them.
- Conclusions:
  - proposed alg. is slower, more complicated, more expensive, and less robust that the original centralized one.

Token Ring Algorithm

- Assume:
  - bus network, with no inherent ordering of the processes.
  - a logical ring is constructed in which each process is assigned a position in the ring
  - The ring positions may be allocated in numerical order of network addresses or some other means.
  - It does not matter what the ordering is.
  - All that matters is that each process knows who is next in line after itself.



Token Ring Algorithm

- When the ring is initialized, process 0 is given a token.
- The token circulates around the ring.
- It is passed from process k to process k+1 (modulo the ring size) in point-to-point messages.
- When a process acquires the token from its neighbor,
  - It checks to see if it is attempting to enter a critical region.
  - If so,
    1. the process enters the region,
    2. does all the work it needs to,
    3. leaves the region.
    4. After it has exited, it passes the token along the ring.
  - If not,
    1. it just passes it along [=> when no processes want to enter any critical regions, the token just circulates at high speed around the ring]
- It is not permitted to enter a second critical region using the same token.

Pro and cons

- Correctness: only one process has the token at any instant, so only one process can be in a critical region.
- Since the token circulates among the processes in a well-defined order, starvation cannot occur.
- Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.
- Problems:
  - If the token is ever lost, it must be regenerated.
  - Detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded.
  - The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.
- The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases.
  - If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails.
  - At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary.
  - Doing so requires that everyone maintains the current ring configuration.

A Comparison of the Three Algorithms

Algorithm	*Messages*	*Delay before entry*	*Problems*
Centralized	3	2	Coordinator crash
Distributed	2(*n*-1)	2(*n*-1)	Crash of any process
Token ring	1 to infinity	0 to *n* -1	Lost token, process crash

Messages:

- The centralized algorithm is simplest and also most efficient:
  - requires only three messages to enter and leave a critical region: a request and a grant to enter, and a release to exit.
- The distributed algorithm
  - requires n -1 request messages, one to each of the other processes,
  - and an additional *n* -1 grant messages
- With the token ring algorithm, the number is variable.
  - If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered.
  - At the other extreme, the token may sometimes circulate for hours without anyone being interested in it => no. messages per entry into a critical region is unbounded.

A Comparison of the Three Algorithms

- Delay from the moment a process needs to enter a critical region until its actual entry also varies for the 3 algorithms:
  - When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region.
  - When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn.
  - It takes only 2 message times to enter a critical region in the centralized case, and 2(*n* -1) message times in the distrib. case,
    - assuming that the network can handle only one message at a time.
  - For the token ring, the time varies from 0 (token just arrived) to *n*-1 (token just departed).
- Event of crashes.
  - Distributed algorithms are even more sensitive to crashes than the centralized one.
  - In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they are all acceptable.