Synchronization

Module-3

By

Mrs. Vishakha Shelke

Clock synchronization

- Ordering of events taking place in processes of a distributed system according to their temporal association.

- Required to ensure mutual exclusion to guarantee serialization of concurrent access to shared resources.

Logical Clocks

- **Why Logical Clocks?**

  It is difficult to utilize physical clocks to order events uniquely in distributed systems.

- Logical clock assigns timestamp(sequence number) to events for sequencing them in the order agreed upon by all processes.
- The essence of logical clocks is based on the happened-before relationship presented by **Lamport.**

Happen-Before Relationship

- Happened before relation (casual ordering)
  - If a and b are events in the same process, and a occurs before b then a→ b is true.
  - If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then a → b is also true.
  - If a → b and b → c, then a → c (transitive).
  - Concurrent Events - events a and b are concurrent (a||b) if neither a → b nor b → a is true.

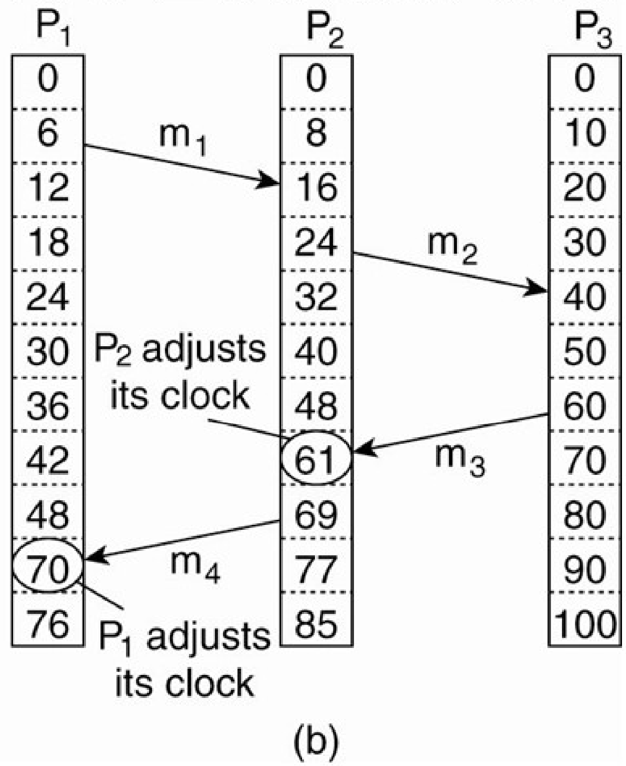
Logical Ordering

- If $T(a)$ is the timestamp for event a, the following relationships must hold in a distributed system utilizing logical ordering.
- If two events, a and b, occurred at the same process, they occurred in the order of which they were observed. That is $T(a) < T(b)$.
- If a sends a message to b, then $T(a) < T(b)$.
- If a happens before b and b happens before c, $T(a) < T(b)$, $T(b) < T(c)$, and $T(a) < T(c)$.

Lamport's Algorithm

- Each process increments its clock counter between every two consecutive events.
- If *a* sends a message to *b*, then the message must include *T(a)*. Upon receiving *a* and *T(a),* the receiving process must set its clock to the greater of [*T(a)+d, Current Clock*]. That is, if the recipient's clock is behind, it must be advanced to preserve the happen-before relationship. Usually *d=1*.

Lamport's Logical Clocks (3)



Event c: P3 sends m3 to P2 at t = 60
Event d: P2 receives m3 at t = 56
Do C(c) and C(d) satisfy the conditions?

Figure Lamport's algorithm corrects the clocks.

Mutual Exclusion

- In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs.
- In distributed systems, since there is no shared memory, these methods cannot be used.

Mutual Exclusion

- A file must not be simultaneously updated by multiple processes.
- Such as printer and tape drives must be restricted to a single process at a time.
- Therefore , Exclusive access to such a shared resource by a process must b ensured.
- This exclusiveness of access is called mutual exclusion between processes.

Performance metrics of mutual exclusion.

- **Synchronization delay:** Time interval between CR exit and new entry by any process.
- **System Throughput:** Rate at which requests for the CR get executed.
- **Message complexity:** Number of messages that are required per CR execution by a process.
- **Response time:** Time interval from a request send to its CR execution completed.

Election Algorithms

- Many distributed algorithms employ a *coordinator* process that performs functions needed by the other processes in the system
  - enforcing mutual exclusion
  - maintaining a global wait-for graph for deadlock detection
  - replacing a lost token
  - controlling an input/output device in the system
- If the coordinator process fails due to the failure of the site at which it resides, a new *coordinator* must be selected through an *election algorithm.*

Solution – an *Election*

- All processes currently involved get together to *choose* a coordinator
- If the coordinator crashes or becomes isolated, elect a new coordinator
- If a previously crashed or isolated process, comes on line, a new election *may* have to be held

- A process begins an election when it notices, through timeouts, that the coordinator has failed.

- Three types of messages
  - An *election* message is sent to announce an election.
  - An *ok* message is sent in response to an election message.
  - A *coordinator* message is sent to announce the identity of the elected process (the “new coordinator”).

The Bully Algorithm (Cont.)

- The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a *coordinator* message to all processes with lower identifiers.

- A process with lower identifier begins an election by sending an *election* message to those processes that have a higher identifier and awaits an *ok* message in response.
  - If none arrives within time T, the process considers itself the coordinator and sends a *coordinator* message to all processes with lower identifiers.
  - If a reply arrives, the process waits a further period *T'* for a *coordinator* message to arrive from the new coordinator. If none arrives, it begins another election.

The Bully Algorithm (Cont.)

- If a process receives an *election* message, it sends back an *ok* message and begins another election (unless it has begun one already).

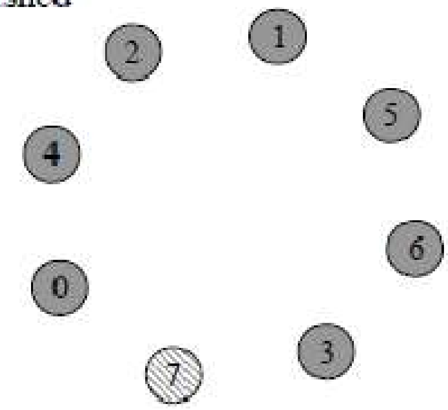
- If a process receives a *coordinator* message, it sets its variable *coordinator-id* to the identifier of the coordinator contained within it.

The Bully Algorithm (Cont.)

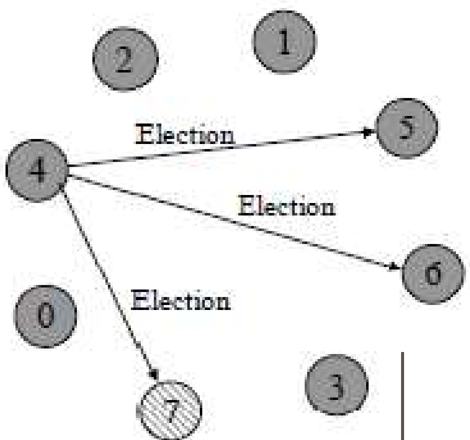
- What happens if a crashed process recovers and immediately initiates an election?

- If it has the highest process identifier (for example P4 in previous slide), then it will decide that it is the coordinator and may choose to announce this to other processes.
  - It will become the coordinator, even though the current coordinator is functioning (hence the name “bully”)
  - This may take place concurrently with the sending of *coordinator* message by another process which has previously detected the crash.

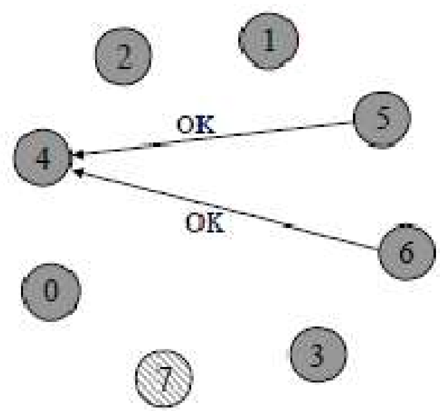
- Example: processes 0-7, 4 detects that 7 has crashed



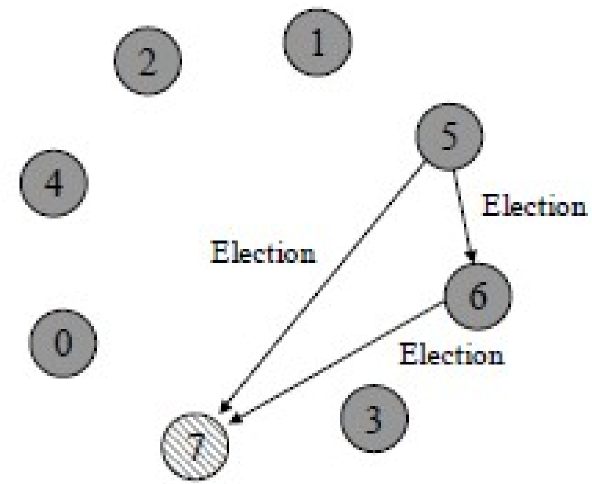
- Example: process 4 holds an election



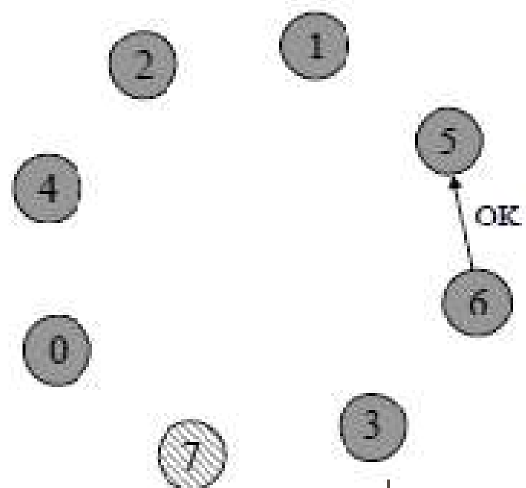
- Example: processes 5 and 6 respond with OK



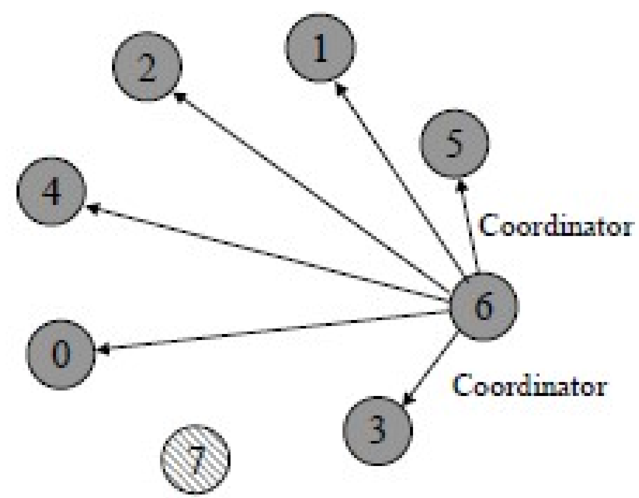
- Example: Processes 5 and 6 hold elections



- Example: process 6 sends OK



- Example: process 6 is the new Coordinator



Distributed Mutual Exclusion (DME)

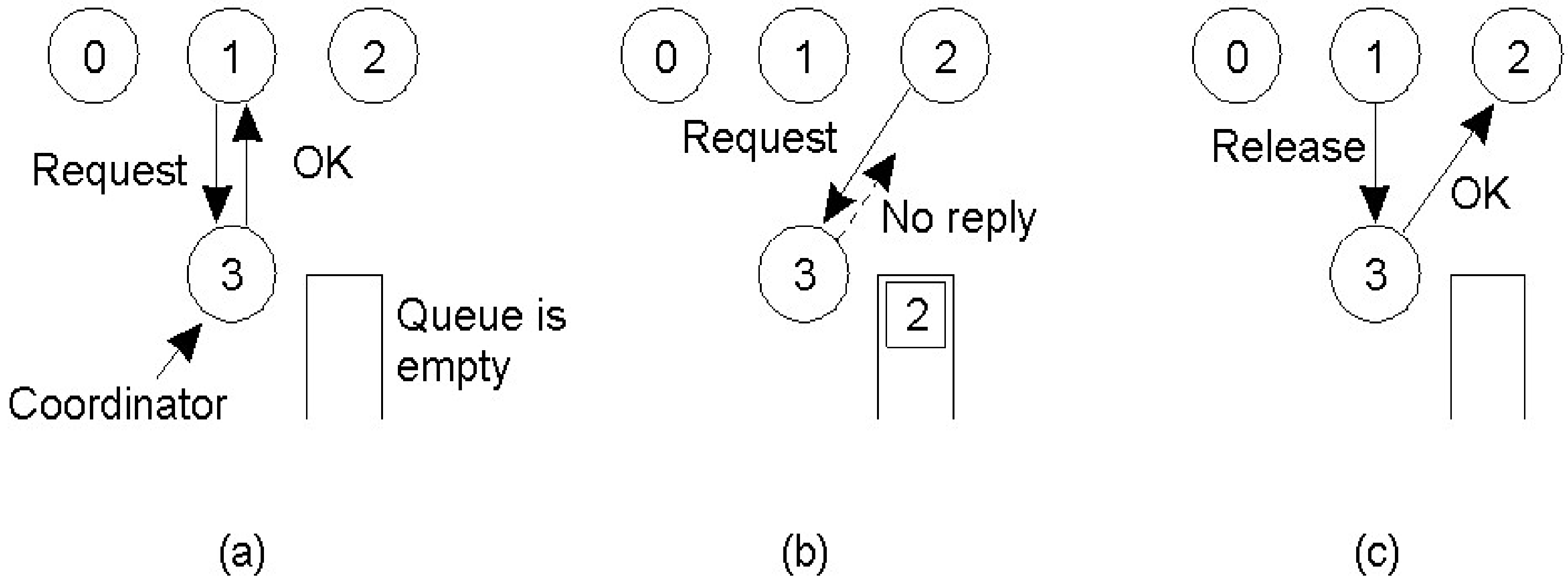
- Assumptions
  - The system consists of *n* processes; each process P_i resides at a different processor.

- The application-level protocol for executing a critical section proceeds as follows:
  - *Enter() :* enter critical section (CS/CR)
  - *ResourceAccess():* access shared resources in CS
  - *Exit():* Leave CS – other processes may now enter.

DME: The Centralized Server Algorithm

- One of the processes in the system is chosen to coordinate the entry to the critical section.
- A process that wants to enter its critical section sends a *request* message to the coordinator.
- The coordinator decides which process can enter the critical section next, and it sends that process a *reply* message.
- When the process receives a *reply* message from the coordinator, it enters its critical section.
- After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.

Mutual Exclusion:

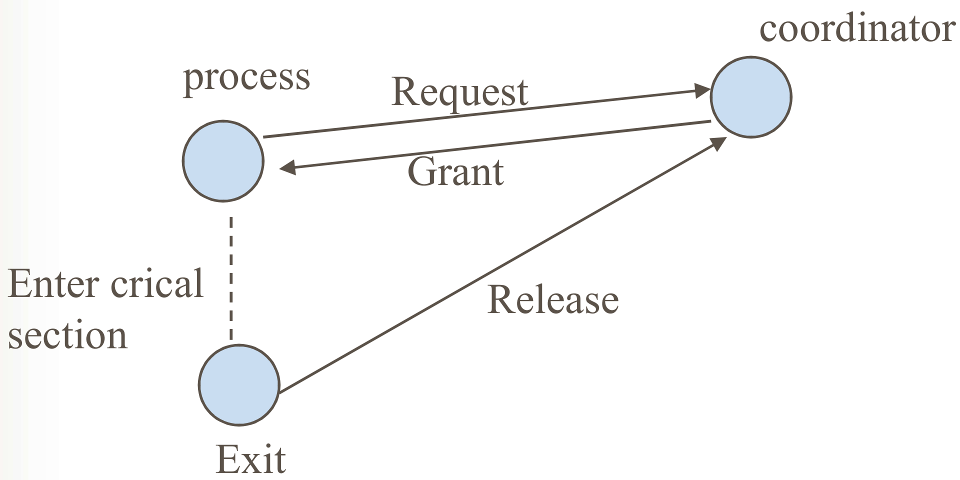


a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

A Centralized Algorithm



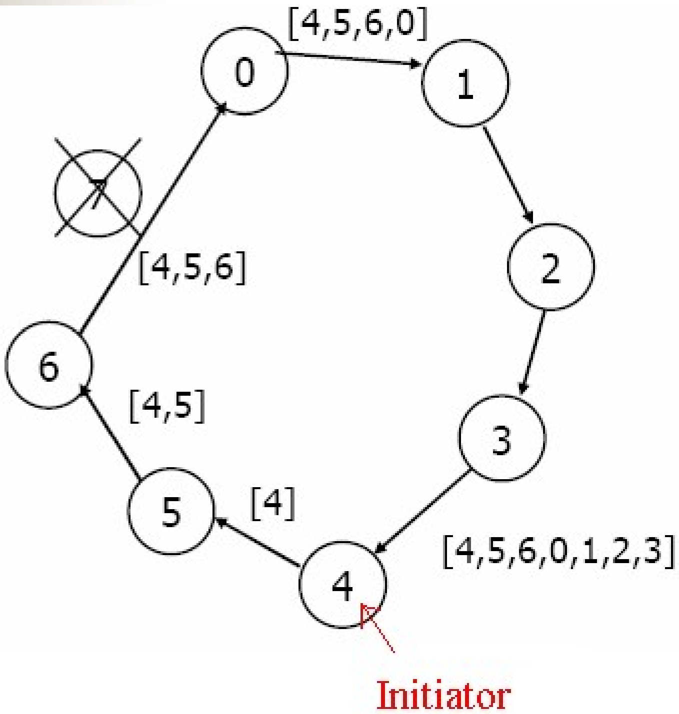
- Advantages: It is fair, easy to implement, and requires only three messages per use of a critical region (request, grant, release).
- Disadvantages: single point of failure.

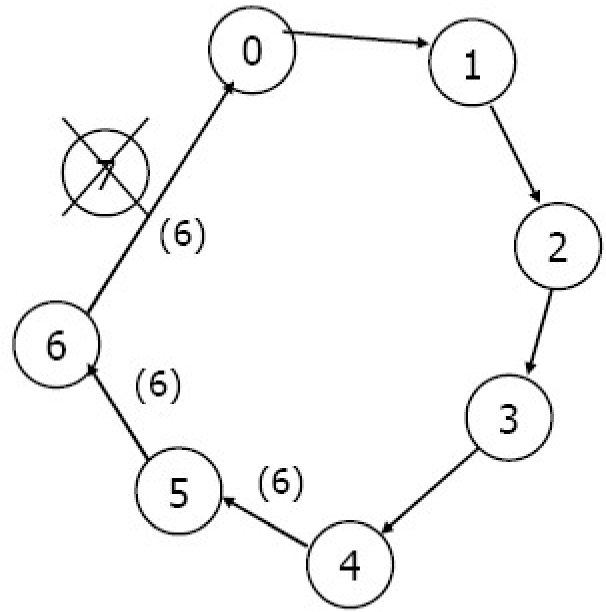
Ring Algorithm

- All processed organized in ring
  - Independent of process number
- Suppose *P* notices no coordinator
  - Sends *election message* to successor with own process number in body of message
  - (If successor is down, skip to next process, etc.)
- Suppose *Q* receives an election message
  - Adds own process number to list in message body
- …

Ring Election Algorithms

- Suppose *P* receives an election message with its own process number in body
  - Changes message to *coordinator* message, preserving body
  - All processes recognize *highest numbered process* as new coordinator
- If multiple messages circulate …
  - …they will all contain same list of processes (eventually)
- If process comes back on-line
  - Calls new election





Initiation:

1.Process 4 sends an ELECTION message to its successor (or next alive process) with its ID

2.Each process adds its own ID and forwards the ELECTION message

Leader Election:

3. Message comes back to initiator, here the initiator is 4.

4. Initiator announces the winner by sending another message around the ring

Ring Algorithm

- Suppose *P* receives an election message with its own process number in body
  - Changes message to *coordinator* message, preserving body
  - All processes recognize *highest numbered process* as new coordinator
- If multiple messages circulate …
  - …they will all contain same list of processes (eventually)
- If process comes back on-line
  - Calls new election

Distributed Mutual Exclusion Algorithms

- Non-token based:
  - A site/process can enter a critical section when an assertion (condition) becomes true.
  - Algorithm should ensure that the assertion will be true in only one site/process.
- Token based:
  - A unique token (a known, unique message) is shared among cooperating sites/processes.
  - Possessor of the token has access to critical section.
  - Need to take care of conditions such as loss of token, crash of token holder, possibility of multiple tokens, etc.

Non-token Based Algorithms

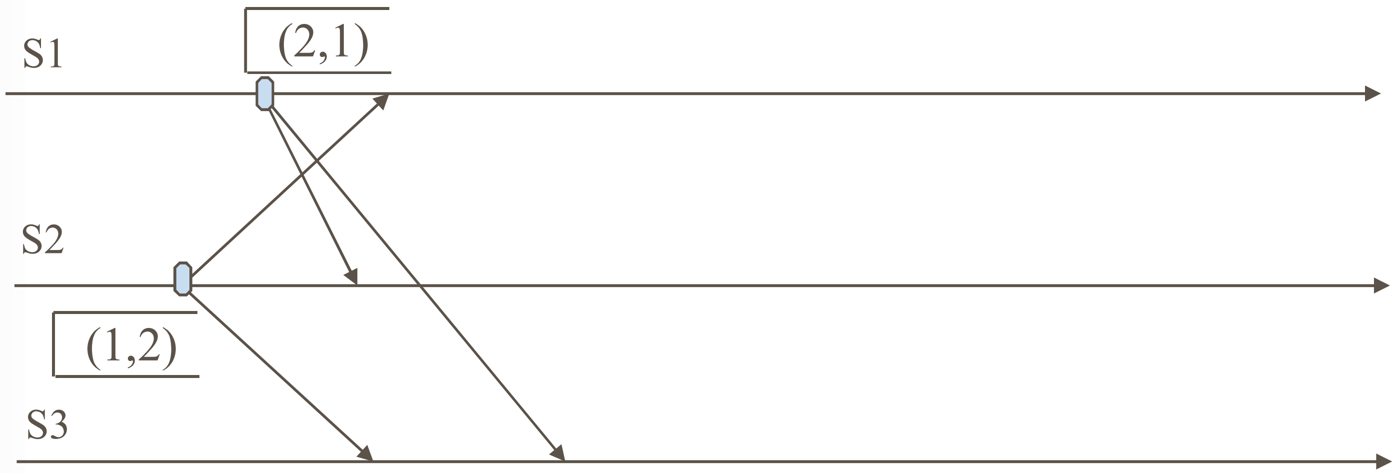
- Notations:
  - Si: SiteI/ Nodes
  - Ri: Request set, containing the ids of all Sis from which permission must be received before accessing CS.
  - Non-token based approaches use time stamps to order requests for CS.
  - Smaller time stamps get priority over larger ones.
- Lamport’s Algorithm
  - Ri = {S1, S2, …, Sn}, i.e., all sites. Or nodes
  - Request queue: maintained at each Si. Ordered by time stamps.
  - Assumption: message delivered in FIFO.

Lamport's Algorithm for mutual exclusion

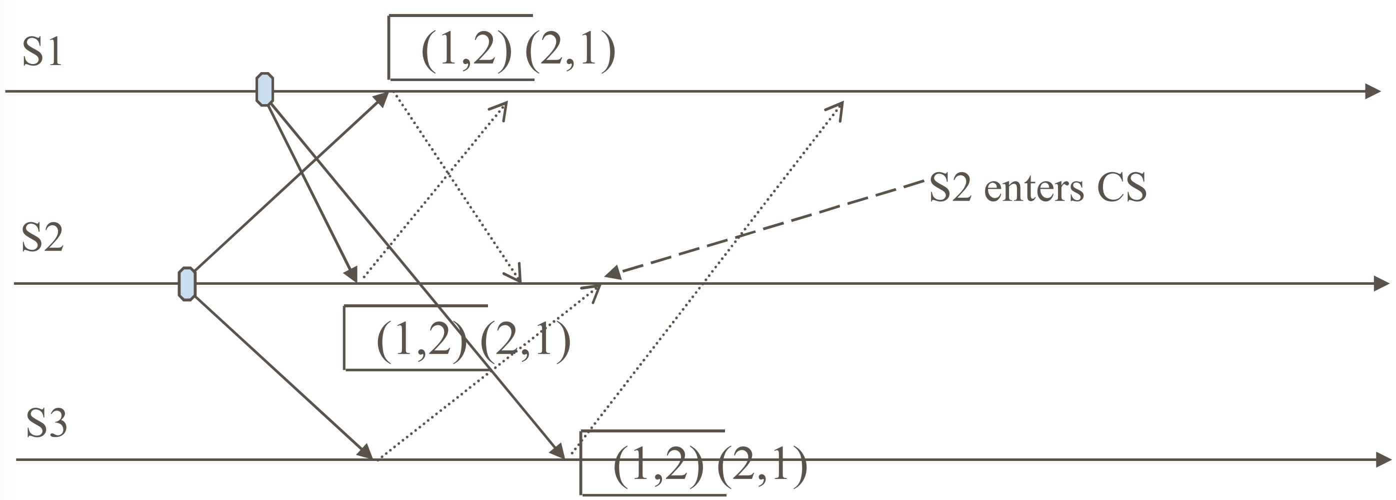
- Requesting CS:
  - Send REQUEST(tsi, i). (tsi,i): Request time stamp. Place REQUEST in *request_queuei*.
  - On receiving the message; sj sends time-stamped REPLY message to si. Si's request placed in *request_queuej*.
- Executing CS:
  - Si has received a message with time stamp larger than (tsi,i) from all other sites.
  - Si's request is the top most one in *request_queuei*.
- Releasing CS:
  - Exiting CS: send a time stamped RELEASE message to all sites in its request set.
  - Receiving RELEASE message: Sj removes Si's request from its queue.

Lamport's Algorithm: Example

Step 1:



Step 2:

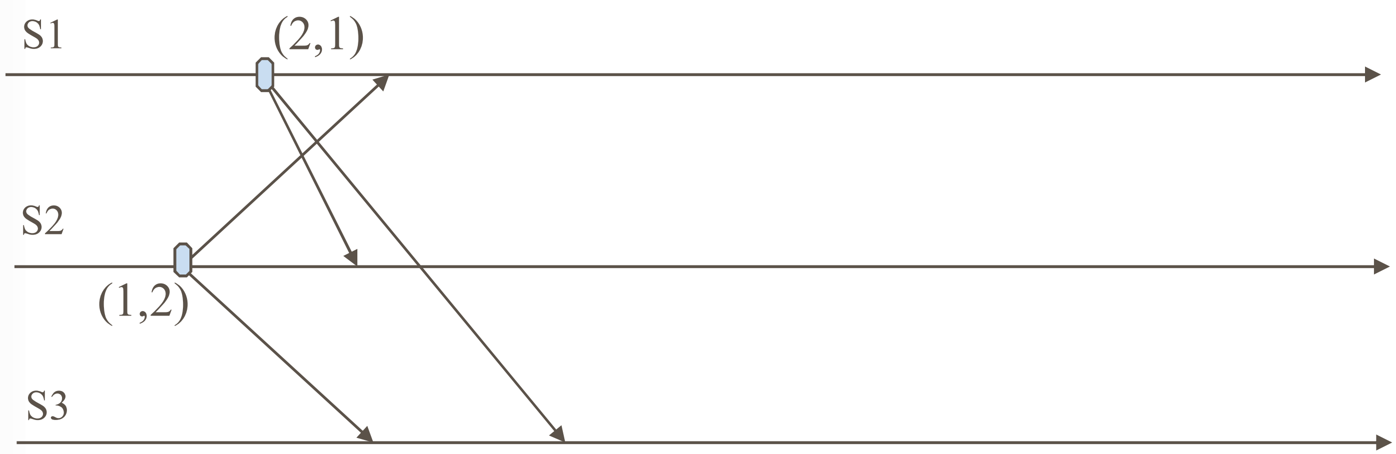


Ricart-Agrawala Algorithm

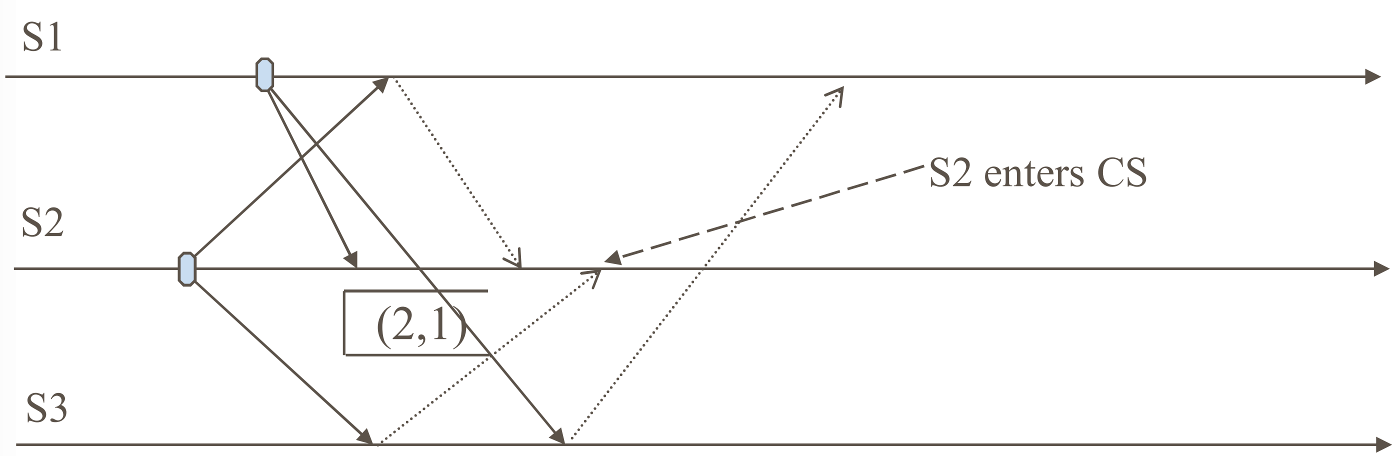
- Requesting critical section
  - Si sends time stamped REQUEST message
  - Sj sends REPLY to Si, if
    - Sj is not requesting nor executing CS
    - If Sj is requesting CS and Si's time stamp is smaller than its own request.
    - Request is deferred otherwise.
- Executing CS: after it has received REPLY from all sites in its request set.
- Releasing CS: Send REPLY to all deferred requests.

Ricart-Agrawala: Example

Step 1:

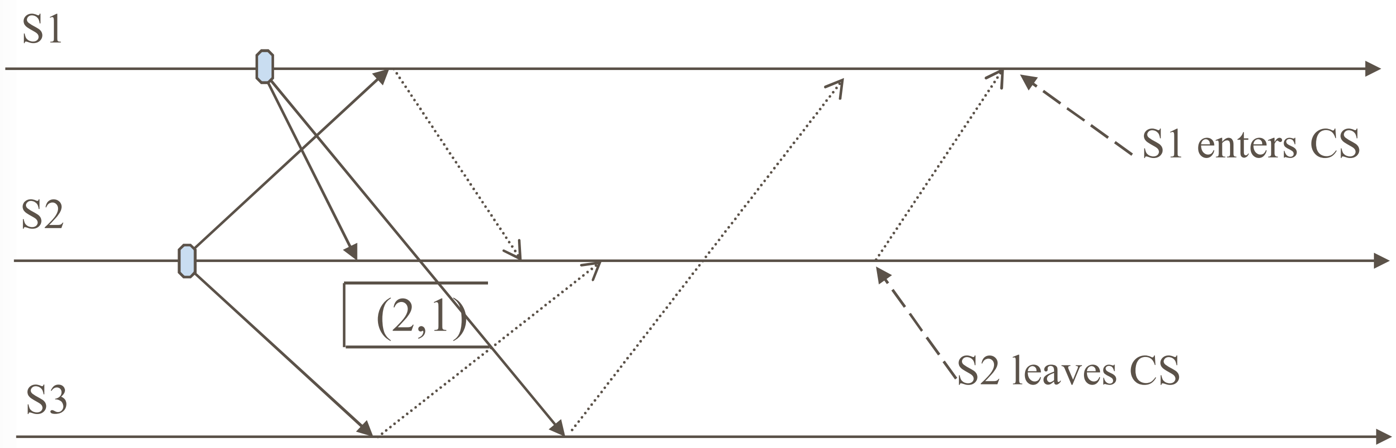


Step 2:



Ricart-Agrawala: Example…

Step 3:



Raymond's Algorithm

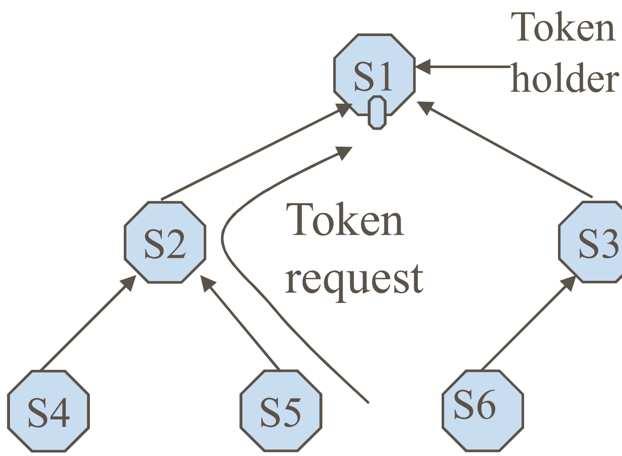
- Sites are arranged in a logical directed tree. Root: token holder. Edges: directed towards root.
- Every site has a variable *holder* that points to an immediate neighbor node, on the directed path towards root. (Root's holder point to itself).
- Requesting CS
  - If Si does not hold token and request CS, sends REQUEST *upwards* provided its *request_q* is empty. It then adds its request to *request_q*.
  - Non-empty *request_q* -> REQUEST message for top entry in q (if not done before).
  - Site on path to root receiving REQUEST -> propagate it up, if its *request_q* is empty. Add request to *request_q*.
  - Root on receiving REQUEST -> send token to the site that forwarded the message. Set *holder* to that forwarding site.
  - Any Si receiving token -> delete top entry from *request_q*, send token to that site, set *holder* to point to it. If request_q is non-empty now, send REQUEST message to the *holder* site.

Raymond's Algorithm …

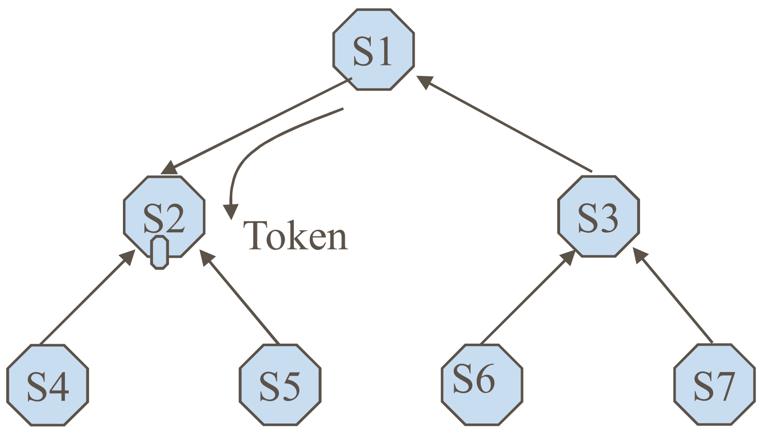
- Executing CS: getting token with the site at the top of *request_q*. Delete top of *request_q*, enter CS.
- Releasing CS
  - If *request_q* is non-empty, delete top entry from q, send token to that site, set *holder* to that site.
  - If *request_q* is non-empty now, send REQUEST message to the *holder* site.

Raymond's Algorithm: Example

Step 1:

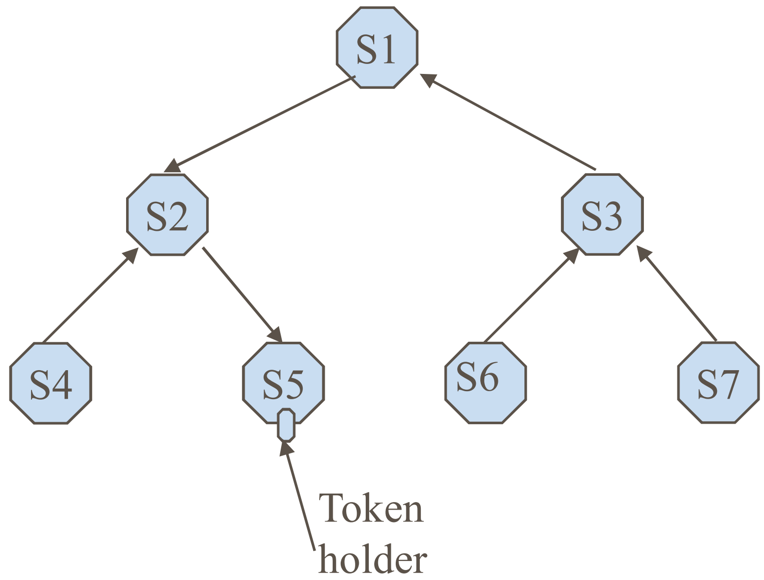


Step 2:



Raymond's Algm.: Example…

Step 3:



Suzuki-Kasami Algorithm

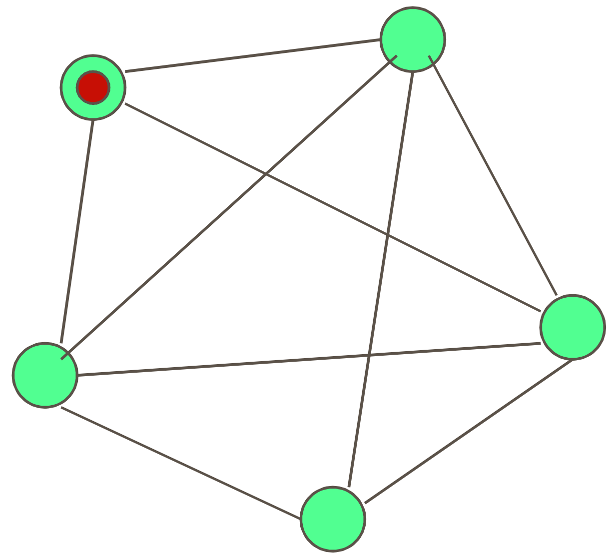
- If a site without a token needs to enter a CS, broadcast a REQUEST for token message to all other sites.
- Token: (a) Queue of request sites (b) Array LN[1..N], the sequence number of the most recent execution by a site j.
- Token holder sends token to requestor, if it is not inside CS. Otherwise, sends after exiting CS.
- Token holder can make multiple CS accesses.
- Design issues:
  - Distinguishing outdated REQUEST messages.
    - Format: REQUEST(j,n) -> jth site making nth request.
    - Each site has RNi[1..N] -> RNi[j] is the largest sequence number of request from j.
  - Determining which site has an outstanding token request.
    - If LN[j] = RNi[j] - 1, then Sj has an outstanding request.

Suzuki-Kasami Algorithm ...

- Passing the token
  - After finishing CS
  - (assuming Si has token), LN[i] := RNi[i]
  - Token consists of Q and LN. Q is a queue of requesting sites.
  - Token holder checks if RNi[j] = LN[j] + 1. If so, place j in Q.
  - Send token to the site at head of Q.

- **Suzuki-Kasami algorithm**

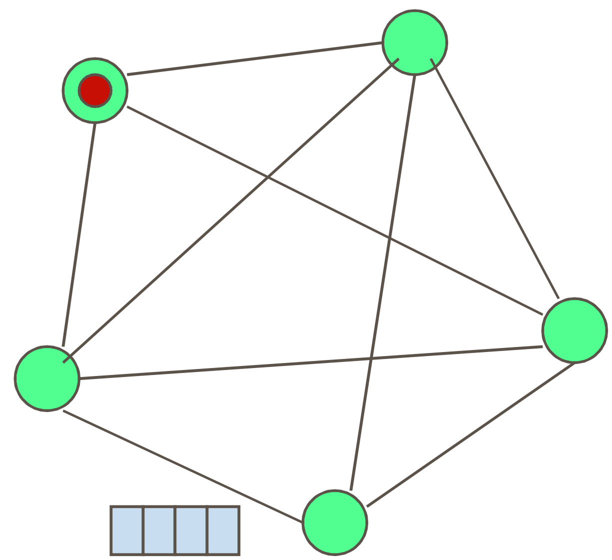
- **Completely connected** network of processes
- There is **one token** in the network. The owner of the token has the permission to enter CS.
- Token will move from one process to another based on demand.



Suzuki-Kasami Algorithm

- When a process **i** receives a request **(i, num)** from process **k,** it sets **req[k] to max(req[k], num)** and enqueues the request in its **Q**

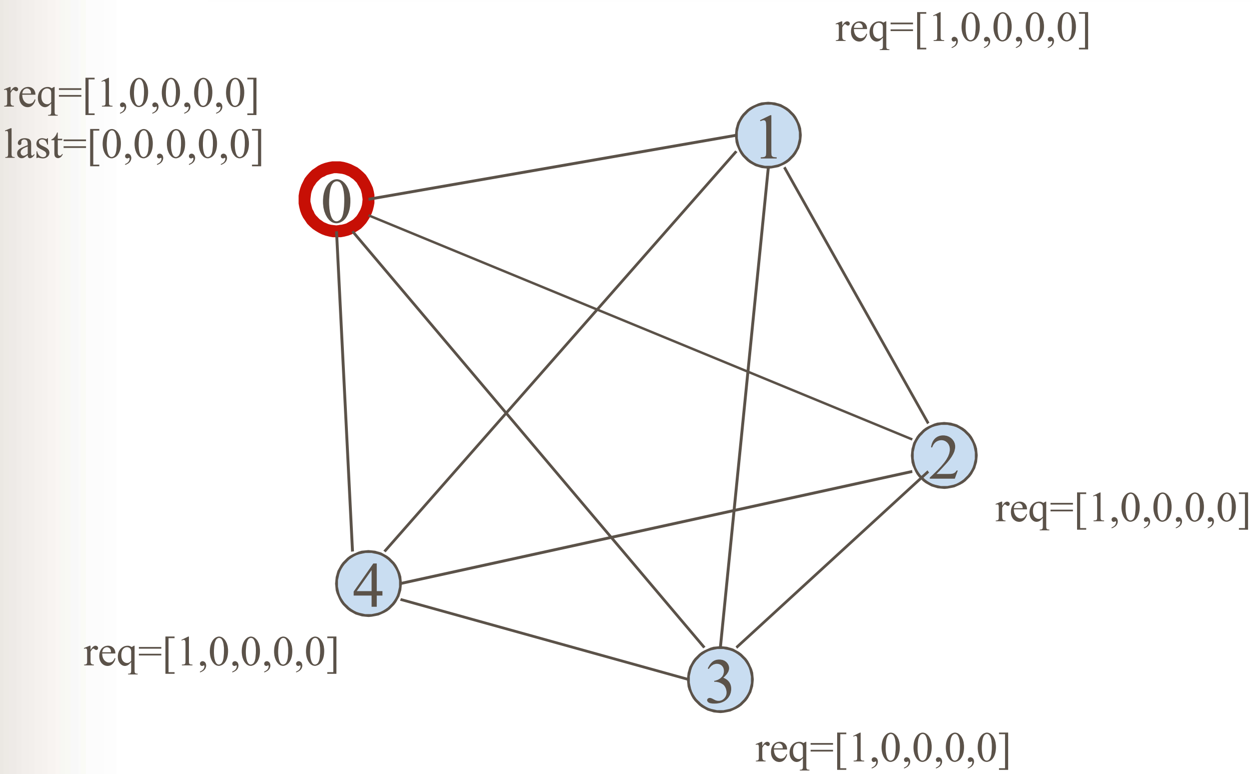
- When process **i** sends a token to the ***head of Q***,
- it sets **last[i]** := its own **num,** and passes the array **last,** as well as the ***tail of Q***,



Req: array[0..n-1] of integer

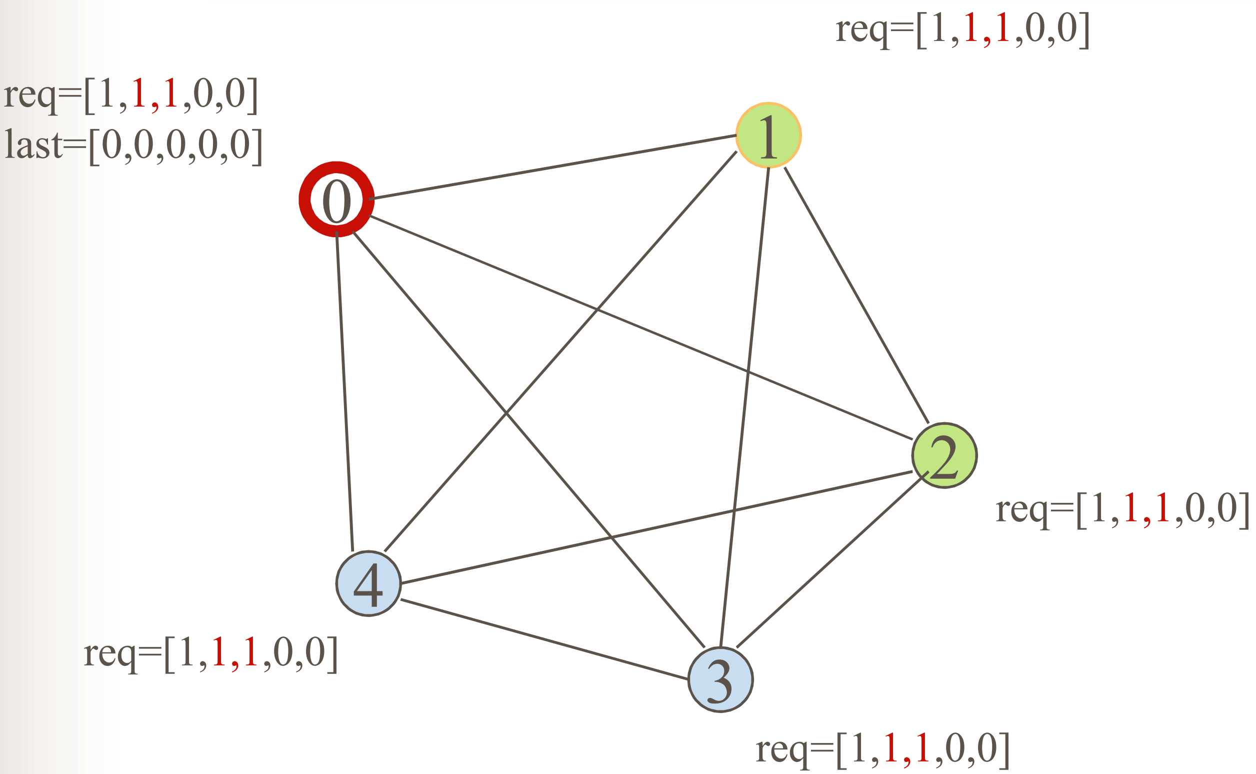
Last: Array [0..n-1] of integer

Example



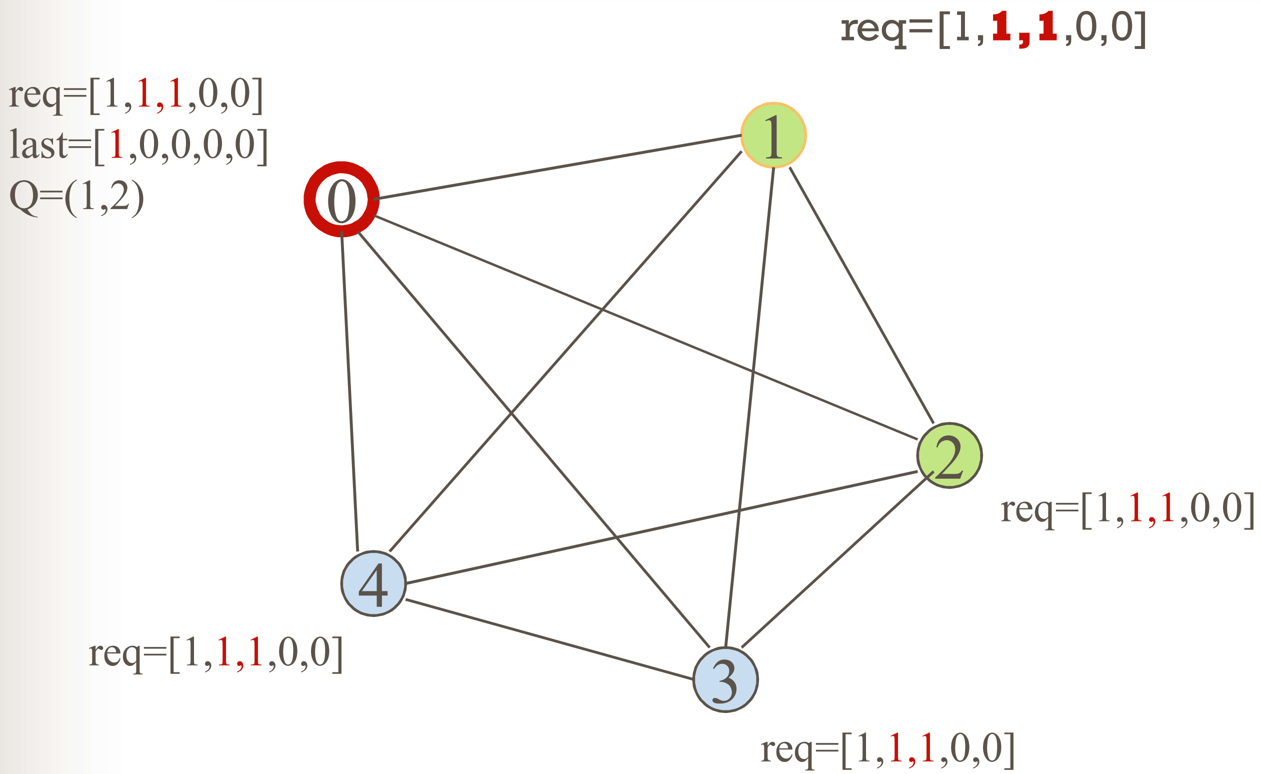
initial state

Example



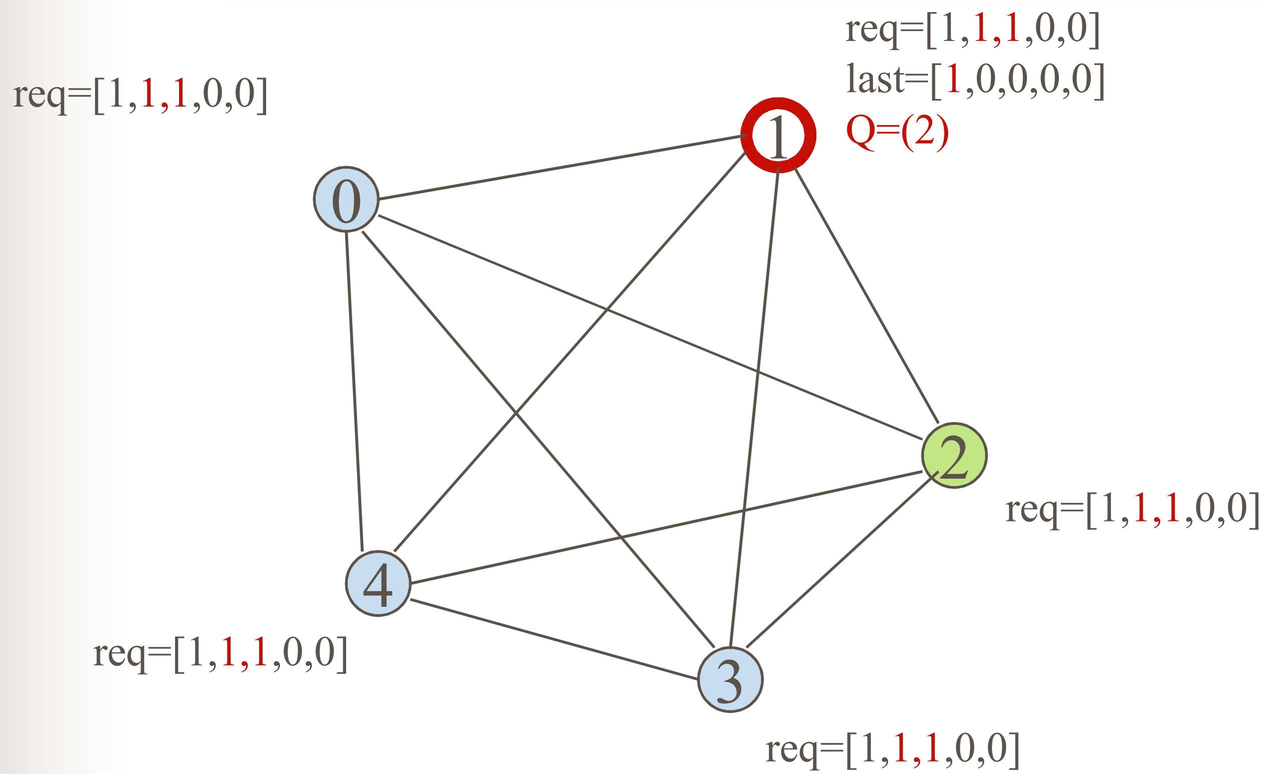
1 & 2 send requests

Example



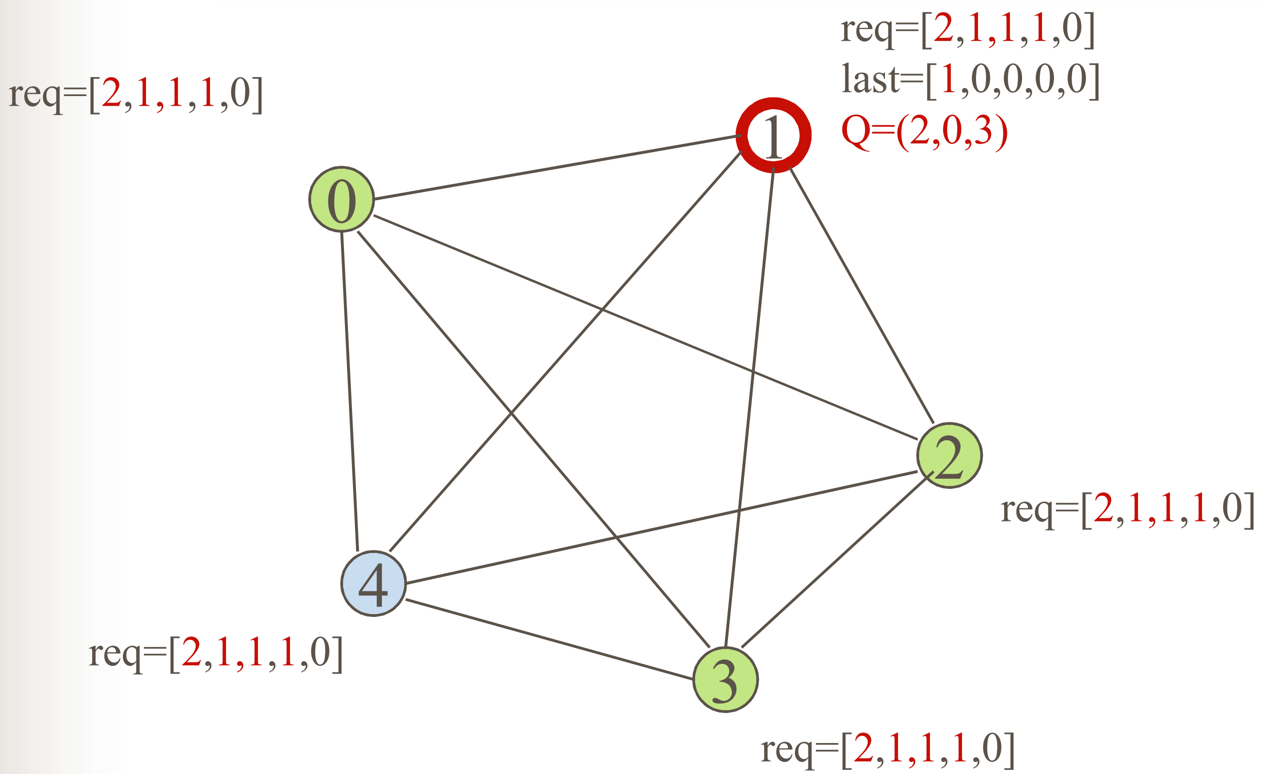
0 prepares to exit CS

Example



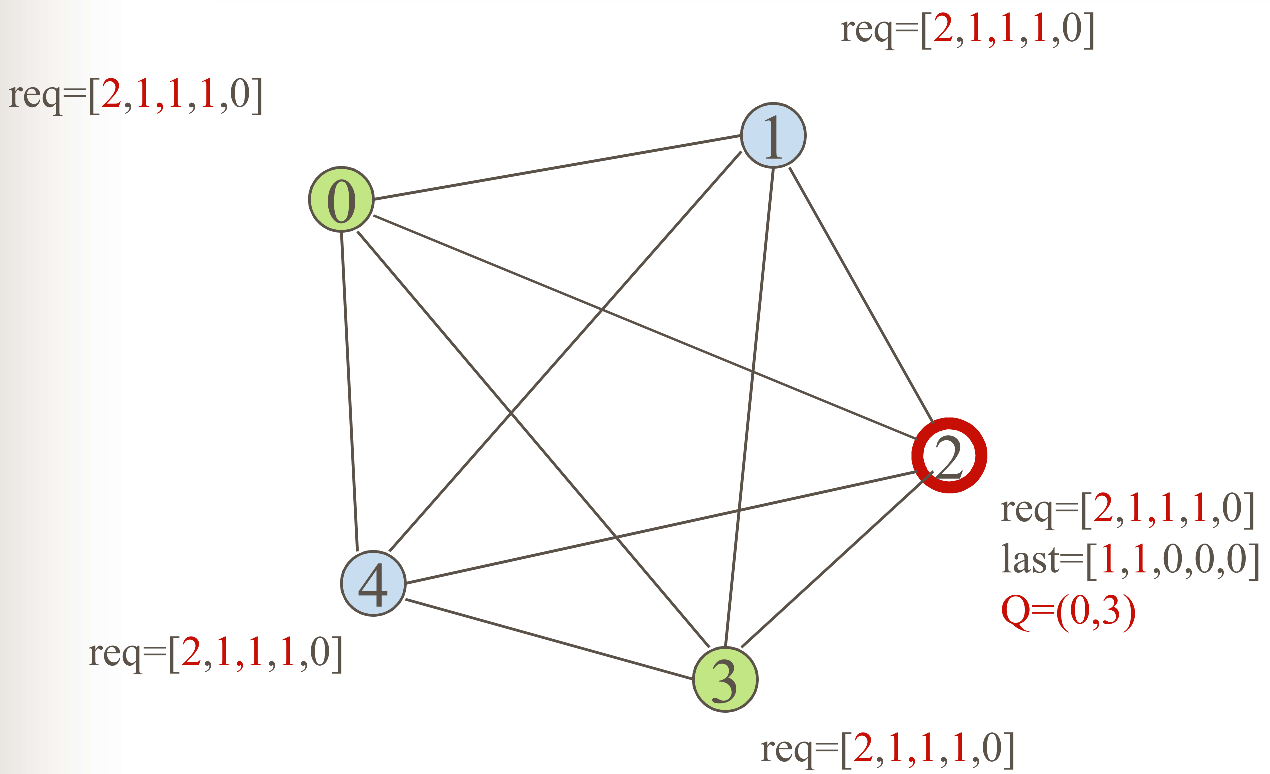
0 passes token (Q and last) to 1

Example



0 and 3 send requests

Example



1 sends token to 2

Singhal's Heuristic token based algorithm

- Instead of broadcast: each site maintains information on other sites, guess the sites likely to have the token.
- Data Structures:
  - Si maintains SVi[1..M] and SNi[1..M] for storing information on other sites: state and highest sequence number.
  - Token contains 2 arrays: TSV[1..M] and TSN[1..M].
  - States of a site
    - R : requesting CS
    - E : executing CS
    - H : Holding token, idle
    - N : None of the above
  - Initialization:
    - SVi[j] := N, for j = M .. i; SVi[j] := R, for j = i-1 .. 1; SNi[j] := 0, j = 1..N. S1 (Site 1) is in state H.
    - Token: TSV[j] := N & TSN[j] := 0, j = 1 .. N.

Singhal's Heuristic token based algorithm

- Requesting CS
  - If Si has no token and requests CS:
    - SVi[i] := R. SNi[i] := SNi[i] + 1.
    - Send REQUEST(i,sn) to sites Sj for which SVi[j] = R. (sn: sequence number, updated value of SNi[i]).
  - Receiving REQUEST(i,sn): if sn <= SNj[i], ignore. Otherwise, update SNj[i] and do:
    - SVj[j] = N -> SVj[i] := R.
    - SVj[j] = R -> If SVj[i] != R, set it to R & send REQUEST(j,SNj[j]) to Si. Else do nothing.
    - SVj[j] = E -> SVj[i] := R.
    - SVj[j] = H -> SVj[i] := R, TSV[i] := R, TSN[i] := sn, SVj[j] = N. Send token to Si.
- Executing CS: after getting token. Set SVi[i] := E.

Singhal's Heuristic token based algorithm

- Releasing CS
  - SVi[i] := N, TSV[i] := N. Then, do:
    - For other Sj: if (SNi[j] > TSN[j]), then {TSV[j] := SVi[j]; TSN[j] := SNi[j]}
    - else {SVi[j] := TSV[j]; SNi[j] := TSN[j]}
  - If SVi[j] = N, for all j, then set SVi[i] := H. Else send token to a site Sj provided SVi[j] = R.

THANK YOU