Distributed Mutual Exclusion

Agenda

- What is Mutual exclusion ?
- Mutual exclusion in Distributed operating systems.
- Mutual Exclusion algorithms
  - Centralized Algorithms
  - Distributed algorithms
    - Contention based solutions
    - Token based (Controlled) solutions
- Summary and Conclusions

What is Mutual exclusion?

- Mutual exclusion : makes sure that concurrent process access shared resources or data in a serialized way.

  If a process , say P_i , is executing in its critical section, then no other processes can be executing in their critical sections

Example: updating a DB or sending control signals to an IO device

- What is Mutual exclusion ?
- **Mutual exclusion in Distributed operating systems.**
- Mutual Exclusion algorithms
  - Centralized Algorithms
  - Distributed algorithms
    - Contention based solutions
    - Control based solutions
- Research
- Summary and Conclusions

Mutual exclusion in DOS

There are three major communication scenarios:

1. One-way Communication usually does not need synchronization.

2. Client/server communication is for multiple clients making service request to a shared server. If co-ordination is required among the clients, it is handled by the server and there is no explicit interaction among client processes.

Mutual exclusion in Distributed operating systems Contd…

3. Interprocess communication:

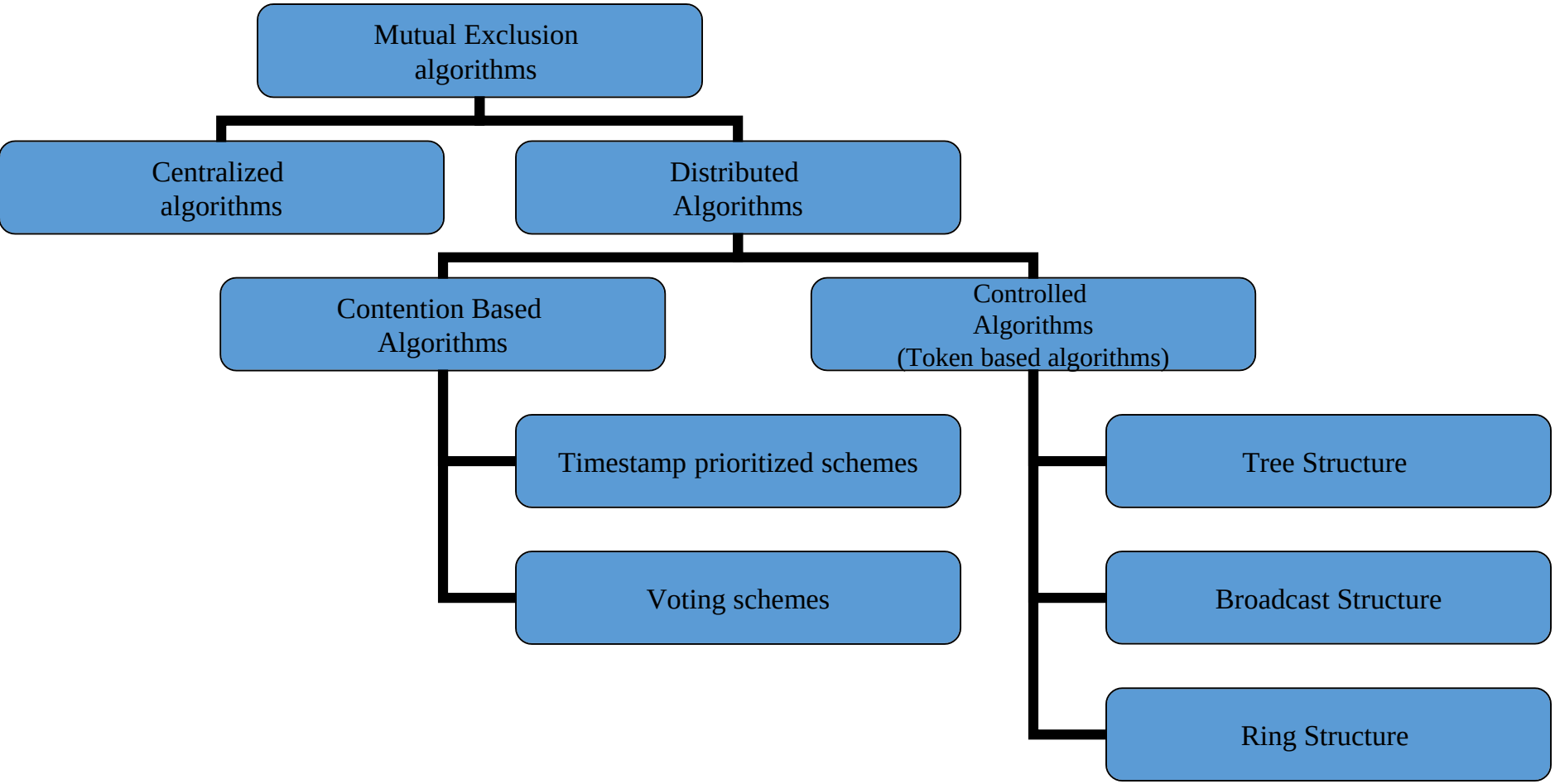
-> not limited to making service requests.

-> processes need to exchange information to reach some conclusion about the system or some agreement among the cooperating processes.

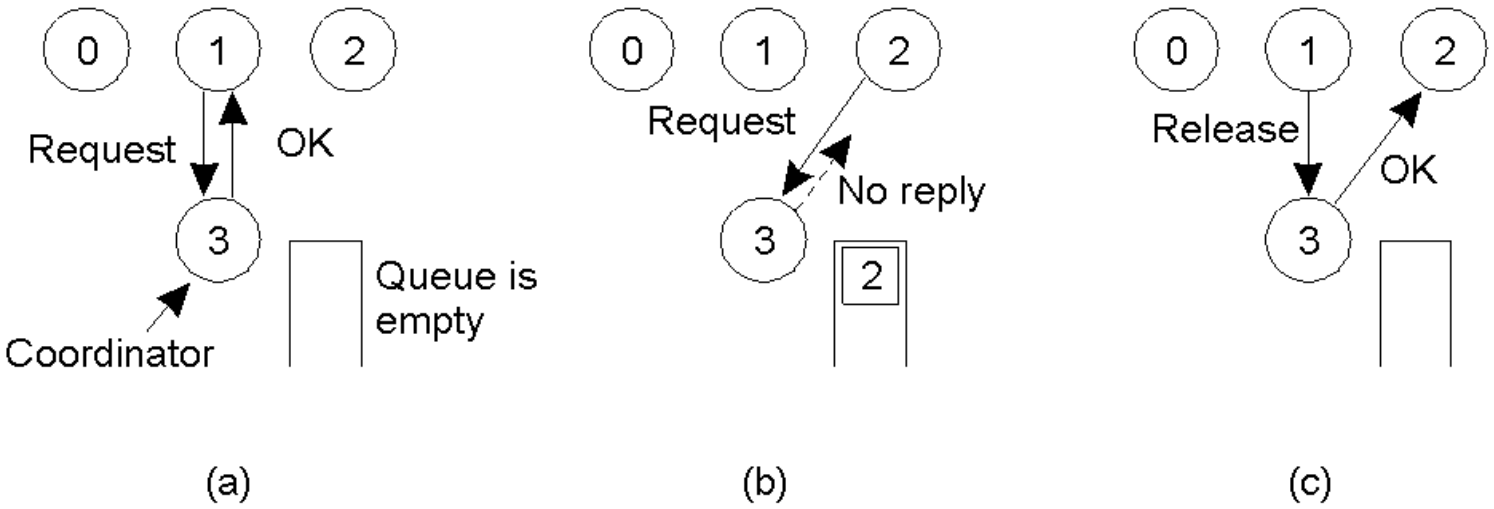
These activities require peer communication; there is no shared object or centralized controller.

- What is Mutual exclusion ?
- Mutual exclusion in Distributed operating systems.
- **Mutual Exclusion algorithms**
  - Centralized Algorithms
  - Distributed algorithms
    - Contention based solutions
    - Control based solutions
- Research
- Summary and Conclusions

Mutual Exclusion Algorithms



Centralized Algorithm



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, which then replies to 2 [2]

Centralized algorithms contd..

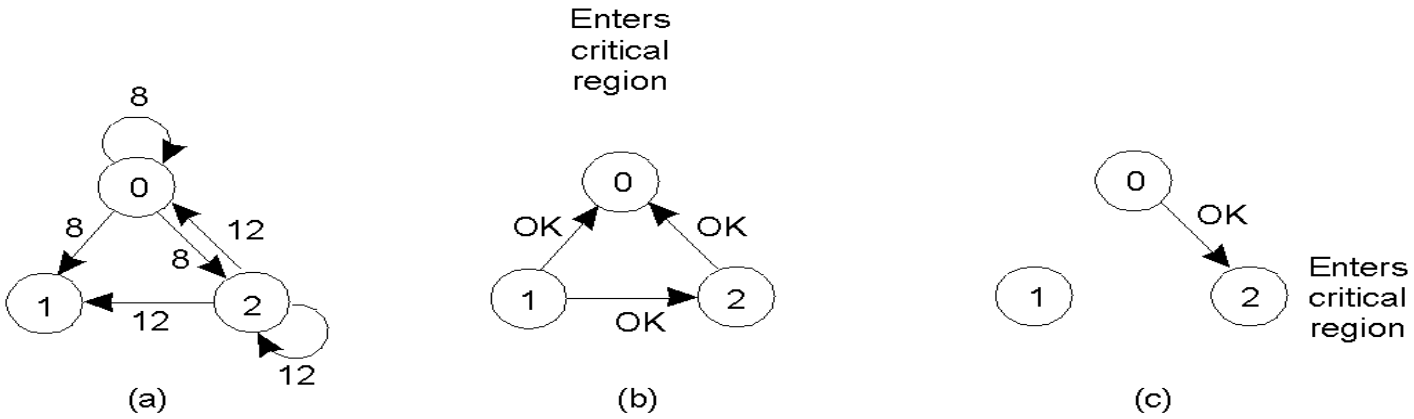
- Advantages
  - Fair algorithm, grants in the order of requests
  - The scheme is easy to implement
  - Scheme can be used for general resource allocation

  Critical Question: When there is no reply, does this mean that the coordinator is “dead” or just busy?
- Shortcomings
  - Single point of failure. No fault tolerance
  - Confusion between No-reply and permission denied
  - Performance bottleneck of single coordinator in a large system

Distributed Algorithms

- Distributed Mutual Exclusion
  - Contention-based Mutual Exclusion
    - Timestamp Prioritized Schemes
    - Voting Schemes
  - Token-based Mutual Exclusion
    - Ring Structure
    - Tree Structure
    - Broadcast Structure

Timestamp Prioritized Schemes



A> Two processes want to enter the same critical region. Each send a message to all the others with their id and timestamp (+ critical section id)

B> Process 0 has the lowest timestamp, so it wins.

C> When process 0 is done, it sends an OK also, so 2 can now enter the critical region[2]

- No deadlock, starvation (because of timestamps)
- N points of fail
- ures !

Timestamp Prioritized Schemes

Lamport's "logical clock" is used to generate timestamps. These are the general properties for the **method**:

- The general mechanism is that a process **P[i]** has to send a **REQUEST** (with ID and time stamp) to **all** other processes.

- When a process **P[j]** receives such a **request**, it sends a **REPLY** back.

When responses are received from all processes, then **P[i]** can enter its Critical Section.

- When **P[i]** exits its critical section, the process sends **RELEASE** messages to all its deferred requests.

  The total message count is 3*(N-1), where N is the number of cooperating processes.

Ricart and Agrawala algorithm

Requesting Site:

- A requesting site *Pi* sends a message *request*(*ts*,*i*) to all sites.

Receiving Site:

- Upon reception of a *request*(*ts*,*i*) message, the receiving site *Pj* will immediately send a timestamped *reply*(*ts*,*j*) message if and only if:
  - *Pj* is not requesting or executing the critical section OR
  - *Pj* is requesting the critical section but sent a request with a higher timestamp than the timestamp of *Pi*
- Otherwise, *Pj* will defer the *reply* message.

Ricart and Agrawala algorithm Contd..

Performance

- Number of network messages; 2*(N-1)
- Synchronization Delays: One message propagation delay

- Mutual exclusion: Site *Pi* enters its critical section only after receiving all *reply* messages.
- Progress: Upon exiting the critical section, *Pi* sends all deferred *reply* messages.

Ricart and Agrawala algorithm Contd..

Disadvantage of Ricart and Agarwala Algorithm:

- Failure of a node – May result in starvation.
- Solution: This problem can be solved by detecting failure of nodes after some timeout.

Voting schemes

Requestor:

- Send a request to all other processes.
- Enter critical section once REPLY from a **majority** is received
- Broadcast RELEASE upon exit from the critical section.

Other processes:

- REPLY to a request if no REPLY has been sent. Otherwise, hold the request in a queue.
- If a REPLY has been sent, do not send another REPLY till the RELEASE is received. [1]

Possibility of a Deadlock

Consider a situation when each candidate wins one-third of votes…..

Voting scheme ,improvement

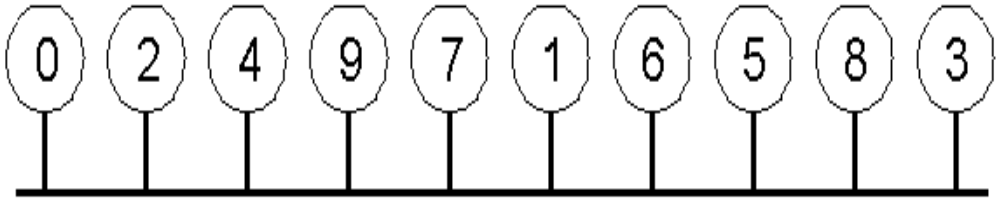
- One of the possible solutions would be :

  Any process retrieves its REPLY message by sending an INQUIRY if the requestor is not currently executing in the critical section. The Requestor has to return the vote through a RELINQUISH message.

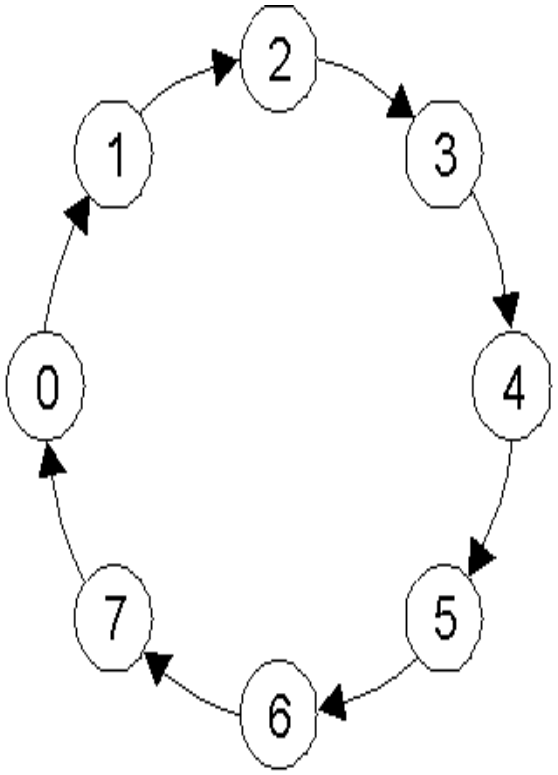
Token-based Mutual Exclusion

Although contention-based distributed mutual exclusion algorithms can have attractive properties, **their messaging overhead is high.** An alternative to contention-based algorithms is to use an explicit control token, possession of which grants access to the critical section.

Ring Structure



(a)



(b)

Ring structure contd..

In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in the previous Fig.

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.

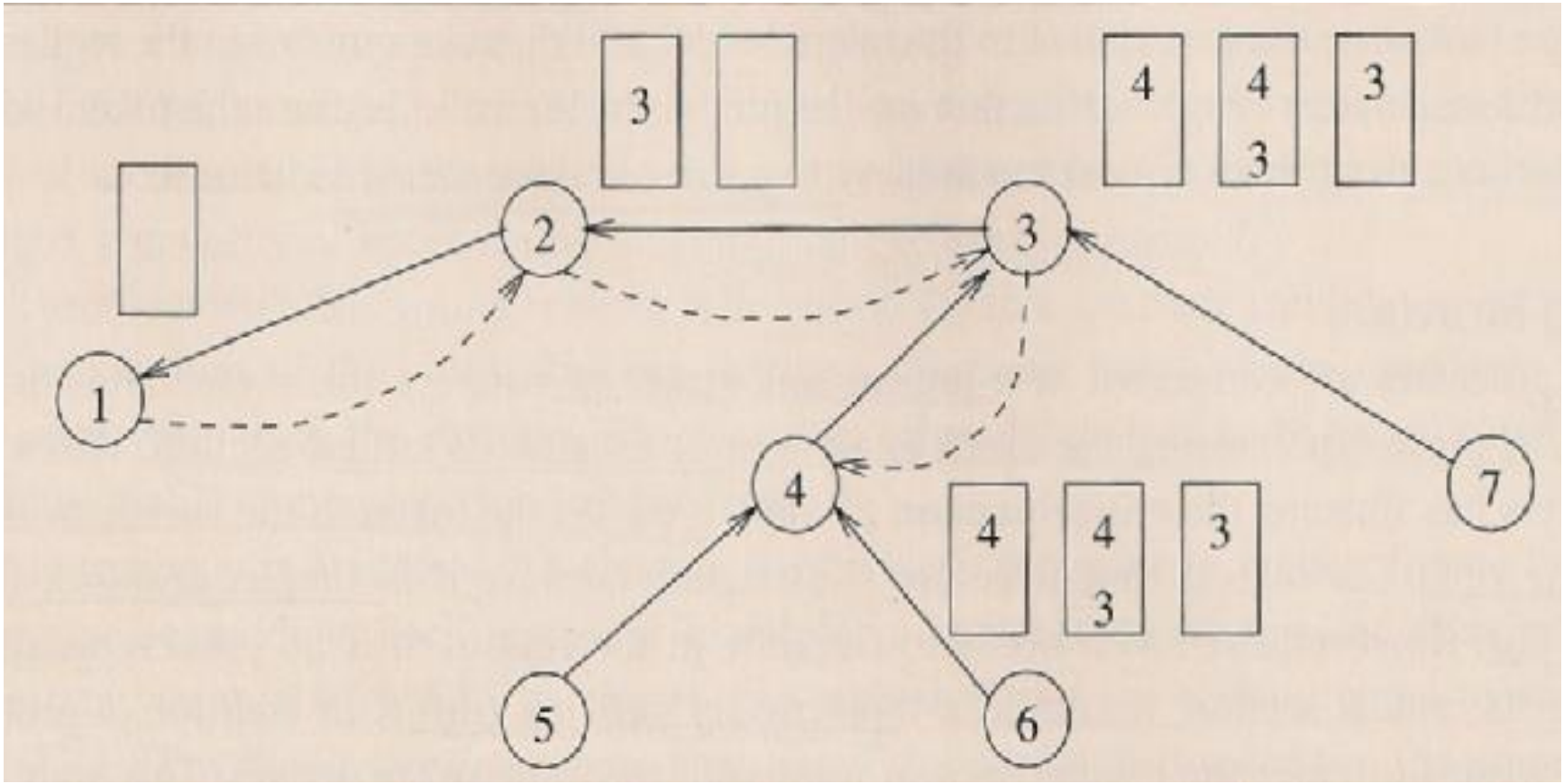
Ring structure contd..

- simple, deadlock-free, fair.
- The token circulates even in the absence of any request (unnecessary traffic).
- Long path (O(N)) – the wait for token may be high.
- Works out when the traffic load is high.
- Token can also carry state information.

Tree structure

- The root of the tree holds the token to start off.
- The processes are organized in a logical tree structure, each node pointing to its parent.
- Further, each node maintains a FIFO list of token requesting neighbors.
- Each node has a variable *Tokenholder* initialized to *false* for everybody except for the first token holder (token generator).

Tree Structured token passing



Tree Structure contd…

The processes are organized in a logical tree structure, each node pointing to its parent. Further, each node maintains a FIFO list of token requesting neighbors. Each node has a variable *Tokenholder* initialized to *false* for everybody except for the first token holder (token generator).

- Entry Condition

If not Tokenholder

If the request queue empty

request token from parent;

put itself in request queue;

block self until Tokenholder is true;

Tree Structure contd…

Exit condition:

If the request queue is not empty

parent = dequeue(request queue);

send token to parent;

set Tokenholder to false;

if the request queue is still not empty, request token from parent;

Tree Structure contd…

Upon receipt of token

Parent **= Dequeue(request queue);**

if self is the parent

Tokenholder = true

else

send token to the parent;

if the queue is not empty

request token from parent;

Broadcast structure(Suzuki/ Kasami's algorithm).

- Imposing a logical topology like a ring or tree is efficient but also complex because the topology has to be implemented and maintained.
- Group communication:
  - Unaware of the topology
  - More transparent and desirable.

Broadcast structure

Data Structure:

The token contains

Token vector *T(.)* – number of completions of the critical section for every process.

Request queue *Q(.)* – queue of requesting processes.

Every process (*i*) maintains the following

seq_no – how many times *i* requested critical section.

Si(.) – the highest sequence number from every process *i* heard of.

Broadcast structure contd…

Entry condition:

Broadcast a REQUEST message stamped with *seq_no.*

Enter critical section after receiving token

Exit Condition:

Update the token vector *T* by setting *T(i)* to $S_i(i)$.

If process *k* is not in request queue *Q* and there are pending requests from *k* ($S_i(k)>T(k)$), append process *k* to *Q.*

If *Q* is non-empty, remove the first entry from *Q* and send the token to the process indicated by the top entry.

Comparison of the Mutual exclusion algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token-Ring	1 to ∞	0 to n – 1	Lost token, process crash

None are perfect – they all have their problems!