Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner. Only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion.

Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state. Three basic approaches for distributed mutual exclusion:

1. Token based approach
2. Non-token based approach
3. Quorum based approach

M**utual exclusion** is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

Mutual exclusion in single computer system Vs. distributed system:

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

Requirements of Mutual exclusion Algorithm:

- **No Deadlock:**
  Two or more site should not endlessly wait for any message that will never arrive.

- No Starvation:
  Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section

- Fairness:
  Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.
- Fault Tolerance:
  In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Solution to distributed mutual exclusion:

As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

1. Token Based Algorithm:
   - A unique token is shared among all the sites.
   - If a site possesses the unique token, it is allowed to enter its critical section
   - This approach uses sequence number to order requests for the critical section.
   - Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
   - This approach insures Mutual exclusion as the token is unique

Example:

   - Suzuki-Kasami's Broadcast Algorithm

2. Non-token based approach:
   - A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
   - This approach use timestamps instead of sequence number to order requests for the critical section.

- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme
- **Example:** Lamport's algorithm, Ricart–Agrawala algorithm

3. Quorum based approach:
    - Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a quorum.
    - Any two subsets of sites or Quorum contains a common site.
    - This common site is responsible to ensure mutual exclusion
    - Example: Maekawa's Algorithm

Ricart-Agrawala Algorithm
Algorithm:

- **To enter Critical section:**
    - When a site Si wants to enter the critical section, it send a timestamped **REQUEST** message to all other sites.
    - When a site Sj receives a **REQUEST** message from site Si, It sends a **REPLY** message to site Si if and only if
        - Site Sj is neither requesting nor currently executing the critical section.
        - In case Site Sj is requesting, the timestamp of Site Si's request is smaller than its own request.
    - Otherwise the request is deferred by site Sj.
- **To execute the critical section:**
    - Site Si enters the critical section if it has received the **REPLY** message from all other sites.
- **To release the critical section:**
    - Upon exiting site Si sends **REPLY** message to all the deferred requests.

Example :
[https://season-lab.github.io/SC/archive/aniello_sc2_aa1617_teoria__SC2-2017-07__Logical_time,_Ricart-Agrawala_algorithm.pdf]

The condition where several processes tries to access the resources and modify the shared data concurrently and outcome of the process depends on the particular order of execution that leads to data inconsistency, this condition is called Race Condition.This condition can be avoided using the technique called Synchronization or Process Synchronization, in which we allow only one process to enter and manipulates the shared data in Critical Section.

Election Algorithms:

Election algorithms choose a process from group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted.

Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number.Then this number is send to every active process in the distributed system.

We have two election algorithms for two different configurations of distributed system.

1. The Bully Algorithm –
This algorithm applies to system where every process can send a message to every other process in the system.
Algorithm – Suppose process P sends a message to the coordinator.

- If coordinator does not respond to it within a time interval T, then it is assumed that coordinator has failed.
- Now process P sends election message to every process with high priority number.
- It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.
- Then it sends a message to all lower priority number processes that it is elected as their new coordinator.
- However, if an answer is received within time T from any other process Q,

(I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.
(II) If Q doesn't responds within time interval T' then it is assumed to have failed and algorithm is restarted.

2. The Ring Algorithm –
This algorithm applies to systems organized as a ring(logically or physically). In this algorithm we assume that the link between the process are unidirectional and every process can message to the process on its right only. Data structure that this algorithm uses is active list, a list that has priority number of all active processes in the system.

Algorithm –

- If process P1 detects a coordinator failure, it creates new active list which is empty initially. It sends election message to its neighbour on right and adds number 1 to its active list.
- If process P2 receives message elect from processes on left, it responds in 3 ways:
  (I) If message received does not contain 1 in active list then P1 adds 2 to its active list and forwards the message.
  (II) If this is the first election message it has received or sent, P1 creates new active list with numbers 1 and 2. It then sends election message 1 followed by 2.
  (III) If Process P1 receives its own election message 1 then active list for P1 now contains numbers of all the active processes in the system. Now Process P1 detects highest priority number from list and elects it as the new coordinator.

[http://user.it.uu.se/~carle/DS2/Notes/03_ElectionAlgorithms.html]
[https://www.cs.colostate.edu/~cs551/CourseNotes/Synchronization/BullyExample.html]

Multicasting in computer network is a group communication, where a sender(s) send data to multiple receivers simultaneously. It supports one – to – many and many – to – many data transmission across LANs or WANs. Through the process of multicasting, the communication and processing overhead of sending the same data packet or data frame in minimized.

- A node can join a multicast group, and receives all messages sent to that group
- The sender sends only once: to the group address
- The network takes care of delivering to all nodes in the group
- Note: groups are restricted to specific networks such as LANs & WANs
  Multicast in the university network will not reach nodes outside the network

Consensus is the task of getting all processes in a group to agree on some specific value based on the votes of each processes. All processes must agree upon the same value and it must be a value that was submitted by at least one of the processes (i.e., the consensus

algorithm cannot just invent a value). In the most basic case, the value may be binary (0 or 1), which will allow all processes to use it to make a decision on whether to do something or not.

With election algorithms, our goal was to pick a leader. With distributed transactions, we needed to get unanimous agreement on whether to commit. These are forms of consensus. With a consensus algorithm, we need to get unanimous agreement on some value. This is a simple-sounding problem but finds a surprisingly large amount of use in distributed systems. Any algorithm that relies on multiple processes maintaining common state relies on solving the consensus problem. Some examples of places where consensus has come in useful are:

- synchronizing replicated state machines and making sure all replicas have the same (consistent) view of system state.
- electing a leader (e.g., for mutual exclusion)
- distributed, fault-tolerant logging with globally consistent sequencing
- managing group membership
- deciding to commit or abort for distributed transactions

Consensus among processes is easy to achieve in a perfect world. For example, when we examined distributed mutual exclusion algorithms earlier, we visited a form of consensus where everybody reaches the same decision on who can access a resource. The simplest implementation was to assign a system-wide coordinator who is in charge of determining the outcome. The two-phase commit protocol is also an example of a system where we assume that the coordinator and cohorts are alive and communicating — or we can afford to wait for them to restart, indefinitely if necessary. The catch to those algorithms was that all processes had to be functioning and able to communicate with each other. Faults make it difficult. Faults include process failures and communication failures.