# INTRODUCTION TO CLOUD COMPUTING (MINOR SPECIALIZATION) ASSIGNMENT [IT1841]

**Name: Rishabh** **Reg.No: 201800631**

**Date : 26 May 2021**

### 1. Classical Distributed Algorithms

Distributed algorithms are algorithms designed to run on multiple processors, without tight centralized control. In general, they are harder to design and harder to understand than single-processor sequential algorithms. Distributed algorithms are used in many practical systems, ranging from large computer networks to multiprocessor shared-memory systems.

They are designed to run on computer hardware constructed from interconnected processors. Distributed algorithms are used in different application areas of distributed computing, such as telecommunications, scientific computing, distributed information processing, and real-time process control. Standard problems solved by distributed algorithms include leader election, consensus, distributed search, spanning tree generation, mutual exclusion, and resource allocation.

Distributed algorithms are a sub-type of parallel algorithm, typically executed concurrently, with separate parts of the algorithm being run simultaneously on independent processors, and having limited information about what the other parts of the algorithm are doing. One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behaviour of the independent parts of the algorithm in the face of processor failures and unreliable

communications links. The choice of an appropriate distributed algorithm to solve a given problem depends on both the characteristics of the problem, and characteristics of the system the algorithm will run on such as the type and probability of processor or link failures, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes.

## 2. What is Global Snapshot? -Global Snapshot Algorithm.

• Each distributed application has several processes (leaders) running on a number of physical servers

• These processes communicate with each other via channels (text messaging)

• A snapshot captures the local states of each process (e.g., program variables) along with the state of each communication channel

**Uses of global system snapshot:**

• Checkpointing – can restart distributed system on failure

• Garbage collection of objects – objects at servers that don't have any other objects (at any servers) with references to them

• Deadlock detection – useful in database transaction systems

• Termination of computation – useful in batch computing systems

• Debugging – useful to inspect the global state of the system

**Global Snapshot Algorithm:**

**System model**

• Problem: record a global snapshot (state for each process and channel)

• Model

  – N processes in the system with no failures

  – There are two FIFO unidirectional channels between every process pair (Pi → Pj and Pj → Pi ).

  – All messages arrive, intact, not duplicated.

• Future work relaxes these assumptions

**System requirements**

• Taking a snapshot shouldn't interfere with normal application behavior

  – Don't stop sending messages

  – Don't stop the application!

• Each process can record its own state

• Collect state in a distributed manner

• Any process can initiate a snapshot

**Initiating a snapshot**

• Let's say process Pi initiates the snapshot

• Pi records its own state and prepares a special marker message (distinct from application messages)

• Send the marker message to all other processes (using N-1 outbound channels)

• Start recording all incoming messages from channels Cji for j not equal to i

**Propagating a snapshot**

• For all processes Pj (including the initiator), consider a message on channel Ckj

• If we see marker message for the first time

  – Pj records own state and marks Ckj as empty

  – Send the marker message to all other processes (using N-1 outbound channels)

  – Start recording all incoming messages from channels Clj for l not equal to j or k

• Else add all messages from inbound channels since we began recording to their states

**Terminating a snapshot**

· All processes have received a marker (and recorded their own state)

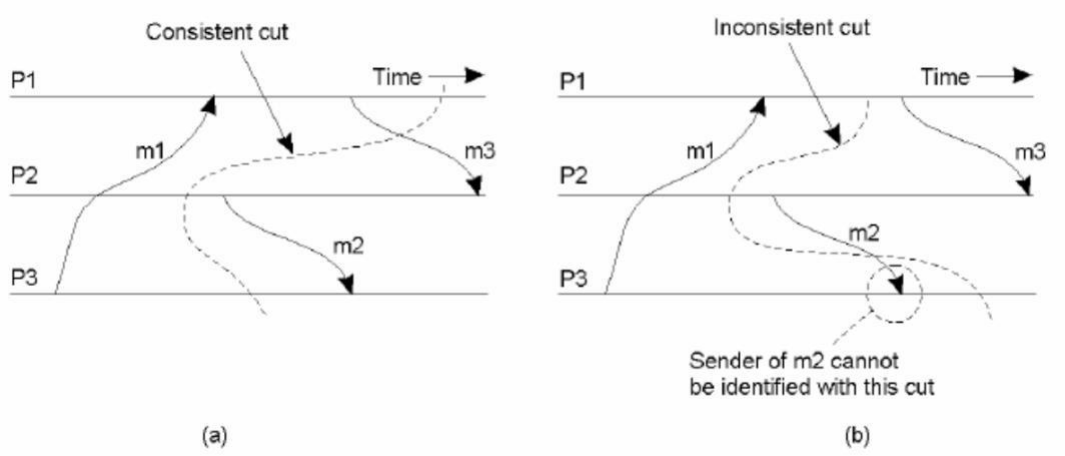
· All processes have received a marker on all the N-1 incoming channels (and recorded their states)

· Later, a central server can gather the partial state to build a global snapshot

## 3. Consistent Cuts

· A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE

· A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut. Such a cut is known as a consistent cut.



## 4. Safety and liveness

Safety and liveness are two important kinds of properties provided by all distributed systems.

Informally, safety guarantees promise that nothing bad happens, while liveness guarantees promise that something good eventually happens.

Every distributed system makes some form of safety and liveness guarantees, and some are stronger than others.

For example, atomic consistency guarantees that operations will appear to happen instantaneously across the system (safety) but operations won't always succeed in the presence of network partitions (liveness, in the form of availability).

## 5. Multicast Ordering

• We want messages delivered in "correct" (intended, consistent etc) order

• FIFO: If a process p performs 2 multicasts, then every working process that delivers these 2 messages deliver in the correct order

• Causal: if p.multicast(g,m) → q.multicast(g,m') then every process which delivers both, deliver m before m'

• Total: All working processes deliver messages in the same order

• Causal implies FIFO

• Total ordering

– Requires messages are delivered same order by each process

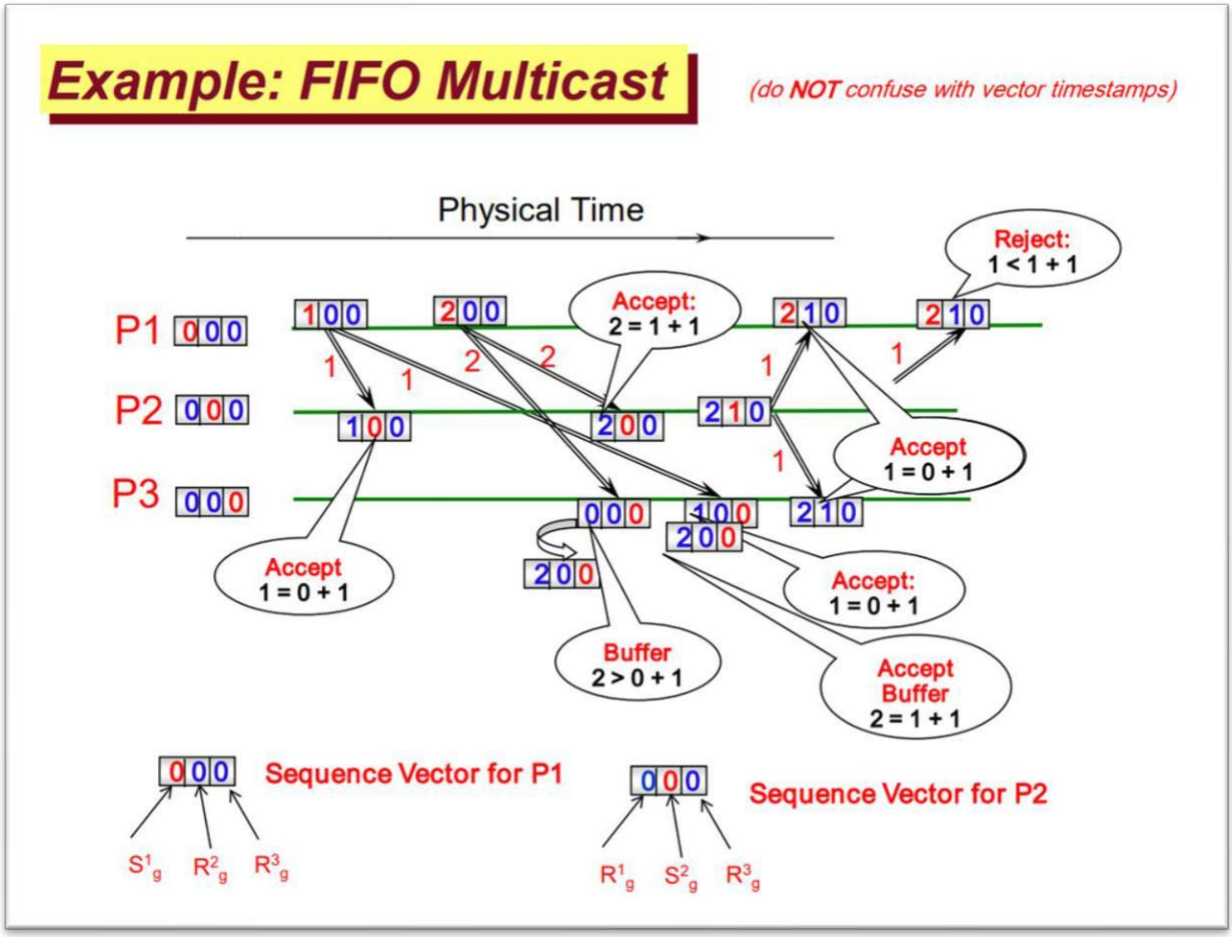
– But this order may have no relation to causality or message sending order

– Can be modified to be FIFO-total or Causal total orders

## 6. Implementing Multicast Ordering

Implementing FIFO Ordering (FIFO-ordered multicast)

• Sp g: count of messages p has sent to g.
• Rq g: the recorded sequence number of the latest message that p has delivered from q to the group g.
• For p to FO-multicast m to g
  – p increments Sp g by 1
  – p "piggy-backs" the value Sp g onto the message.
  – p B-multicasts m to g.
• At process p, upon receipt of m from q with sequence number S:
  – p checks whether S= Rq g+1. If so, p FO-delivers m and increments Rq g
  – If S > Rq g+1, p places the message in the hold-back queue until the intervening messages have been delivered and S= Rq g+1.
  – If S < Rq g+1, then drop the message (we have already seen the message)



**Causal Multicast**

• Let us focus on multicast group g

• Each process i∈g maintains a vector Vg i of length |g| where

– Vg i [j] counts the number of group g messages from j to i

• Messages multicast by process i are tagged with the vector timestamp Vg i

• Recall rule for receiving vector timestamps Max(Vreceiver[j] , Vmessage[j]), if j is not self Vreceiver[j] + 1 otherwise

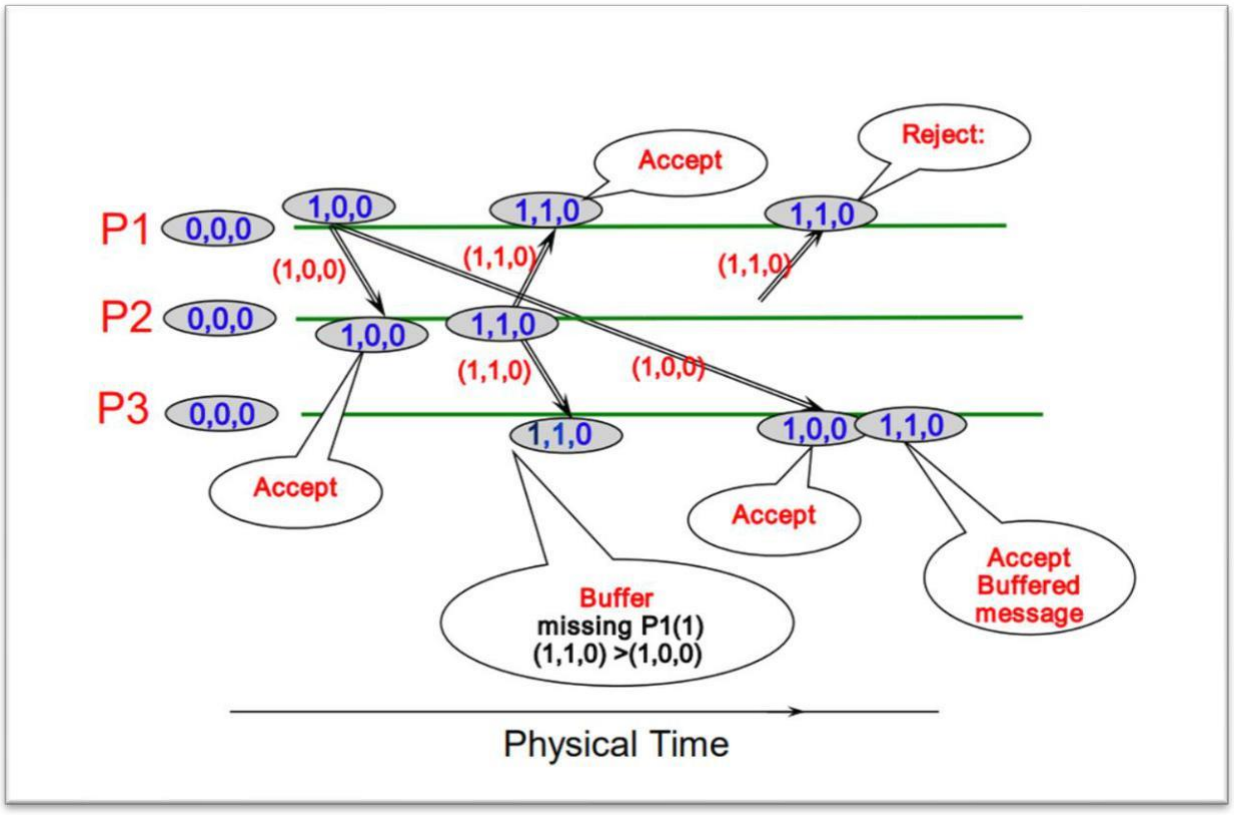
• i.e. when process i receives a from j, then

– Vg i [k] = max(Vg i [k], Vg j [k]) if k ≠ i

– Vg i [k] = Vg i [k] + 1 if k = i

• Remember V(a) < V(b) iff a happens before b

**Causal Ordering using vector time stamps.**



## 7. Reliable Multicast

A **reliable multicast protocol** allows a group of processes to agree on a set of messages received by the group. Each message should be received by all members of the group or by none. The order of these messages may be important for some applications. A reliable multicast protocol is not concerned with message ordering, only message delivery guarantees. **Ordered delivery protocols** can be implemented on top of a reliable multicast service.

A reliable multicast protocol imposes no restriction on the order in which messages are delivered to group processes. Given that multicasts may be in progress by a number of originators simultaneously, the messages may arrive at different processes in a group in different orders. Also, a single originator may have a number of simultaneous multicasts in progress or may have issued a sequence of multicast messages whose ordering we might like preserved at the recipients. Ideally, multicast messages should be delivered instantaneously in the real-time order they were sent, but this is unrealistic as there is no global time and message transmission has a possibly significant and variable latency.

Ordered Reliable Multicasts

A **FIFO ordered** protocol guarantees that messages by the same sender are delivered in the order that they were sent. That is, if a process multicast a message m before it multicasts a message m', then no correct process receives m' unless it has previously received m. To implement this, messages can be assigned sequence numbers which define an ordering on messages from a single source. Some applications may require the context of previously multicast messages from an originator before interpreting the originator's latest message correctly.

## 8. Virtual Synchrony

Virtual synchrony gives us a model for managing a group of replicated processes (aka state machines) and coordinating communication with that group. With virtual synchrony, a process within a group (one of the replicated servers) can join or leave a group – or be evicted from the group. Any process can send a messag to the group and the virtual synchrony software will implement an atomic multicast to the group. Recall that an atomic multicast is the strongest form of reliable delivery, ensuring that a message is either delivered to *all* processes in the group or to *none*. This all-or-nothing property is central to keeping all members of the group synchronized. Message ordering requirements can often be specified by the programmer but we nominally expect causal ordering of any messages that effect changes so that we can ensure consistency among the replicas.

Virtual synchrony provides a highly efficient way to send group messages with atomic delivery, ensuring that all group members are consistently replicated. It is not a transactional system, which would require resource locking and one-message-at-a-time processing. Message ordering policies can be configured in the framework and are generally causal within a view, thus ensuring that related events are consistently ordered at all replicas. A view change acts as a barrier so that all messages that were sent in an earlier view will be delivered within that view.

A group membership service (GMS) provides a consistent view of group membership. If any process suspect a failed process (e.g., because of a timeout), it informs the GMS. If any process wants to join a group, it contacts the GMS service as well. Whenever the group membership changes, the GMS initiates a *view change*.

Every process sends messages to all group members via a reliable multicast (using TCP and looping through a list of group members). Each successive message indicates that the previous message has been received by all group members. A message that has been received by all group members is considered *stable* and can be delivered to the application. If a sending process died partway through a multicast, any messages that it has sent are *unstable*. During a view change, each process sends unstable messages to all group members and waits for acknowledgements. Any messages that a process receives that are not duplicates are considered stable and are delivered to the application. Finally, each process sends a *flush* message to the group. A group member acknowledges

the *flush* when it has delivered all messages to the application. When all flushes are acknowledged, the view change is complete.

## 9. The Consensus Problem

In the context of distributed systems design, a consensus is often loosely used to mean some form of agreement. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final. Typical consensus algorithms make progress when any majority of their servers is available; for example, a cluster of 5 servers can continue to operate even if 2 servers fail. If more servers fail, they stop making progress (but will never return an incorrect result).

i.e 2f+1 nodes to survive f failed nodes

There are a few properties we expect from a solution to consensus:

Agreement: Every correct process must agree on the same value.

Validity: If all processes propose the same value v, then all correct processes decide v

Termination: Every correct process decides some value. If the protocol never terminates, then the processes are vacuously agreeing on the same thing, which is not deciding.

To summarize, fundamentally, the goal of consensus is not that of the negotiation of an optimal value of some kind, but just the collective agreement on some value that was previously proposed by one of the participating servers in that round of the consensus algorithm. With the help of consensus, the distributed system is made to act as though it were a single entity.

### An example scenario:

For the purpose of simplicity, let’s assume a distributed storage system with 2f+1 nodes participating to form a cluster and these participants act at their own speed, may fail at any time and rejoin after recovering from the failure. And these nodes are connected via a network which transmits messages asynchronously at an arbitrary speed. In short, everything can fail at any time; after failure, participants can recover and rejoin the

system. Yes, we are looking at a fault tolerant storage system. As these nodes can fail at various stages, it's important to have more than one copy of our data. For now, let's assume all the data is replicated across all the cluster nodes (but in reality it may affect overall performance)
And we have a client which is not part of the cluster, requesting for some operation from our distributed storage, like a write or read to a data file. Read operation can be served by any node in our cluster without any issues, but write has to be agreed upon by all cluster members before the write can be committed. If two or more nodes recieve write request at the same time for the same value, how to determine which request to process in a distributed setup? This is an example of consensus problem in distributed systems.

## 10. Consensus In Synchronous Systems

The consensus problem may be considered in the case of asynchronous or synchronous systems. While real world communications are often inherently asynchronous, it is more practical and often easier to model synchronous systems, given that asynchronous systems naturally involve more issues than synchronous ones.

In synchronous systems, it is assumed that all communications proceed in rounds. In one round, a process may send all the messages it requires, while receiving all messages from other processes. In this manner, no message from one round may influence any messages sent within the same round.

The FLP impossibility result for asynchronous deterministic consensus In a fully asynchronous message-passing distributed system, in which at least one process may have a crash failure, it has been proven in the famous FLP impossibility result that a deterministic algorithm for achieving consensus is impossible.This impossibility result derives from worst-case scheduling scenarios, which are unlikely to occur in practice except in adversarial situations such as an intelligent denial-of-service attacker in the network. In most normal situations, process scheduling has a degree of natural randomness.

In an asynchronous model, some forms of failures can be handled by a synchronous consensus protocol. For instance, the loss of a communication link may be modeled as a process which has suffered a Byzantine failure.

> Randomized consensus algorithms can circumvent the FLP impossibility result by achieving both safety and liveness with overwhelming probability, even under worst-case scheduling scenarios such as an intelligent denial-of-service attacker in the network.

### 11. Paxos

Paxos is a family of protocols for solving consensus in a network of unreliable or fallible processors. Consensus is the process of agreeing on one result among a group of participants. This problem becomes difficult when the participants or their communications may experience failures.

Consensus protocols are the basis for the state machine replication approach to distributed computing, as suggested by Leslie Lamport and surveyed by Fred Schneider. State machine replication is a technique for converting an algorithm into a fault-tolerant, distributed implementation. Ad-hoc techniques may leave important cases of failures unresolved. The principled approach proposed by Lamport et al. ensures all cases are handled safely.

The Paxos protocol was first submitted in 1989 and named after a fictional legislative consensus system used on the Paxos island in Greece, where Lamport wrote that the parliament had to function "even though legislators continually wandered in and out of the parliamentary Chamber". It was later published as a journal article in 1998.

The Paxos family of protocols includes a spectrum of trade-offs between the number of processors, number of message delays before learning the agreed value, the activity level of individual participants, number of messages sent, and types of failures. Although no deterministic fault-tolerant consensus protocol can guarantee progress in an asynchronous network (a result proved in a paper by Fischer, Lynch and Paterson), Paxos guarantees safety (consistency), and the conditions that could prevent it from making progress are difficult to provoke.

Paxos is usually used where durability is required (for example, to replicate a file or a database), in which the amount of durable state could be large. The protocol attempts to make progress even during periods when some bounded number of replicas are unresponsive. There is also a mechanism to drop a permanently failed replica or to add a new replica.

## 12. The FLP Proof

The **FLP** theorem states that in an asynchronous network where messages may be delayed but not lost, there is no consensus algorithm that is guaranteed to terminate in every execution for all starting conditions, if at least one node may fail-stop.

The FLP impossibility result for asynchronous deterministic consensus

In a fully asynchronous message-passing distributed system, in which at least one process may have a crash failure, it has been proven in the famous FLP impossibility result that a deterministic algorithm for achieving consensus is impossible.This impossibility result derives from worst-case scheduling scenarios, which are unlikely to occur in practice except in adversarial situations such as an intelligent denial-of-service attacker in the network. In most normal situations, process scheduling has a degree of natural randomness.

In an asynchronous model, some forms of failures can be handled by a synchronous consensus protocol. For instance, the loss of a communication link may be modeled as a process which has suffered a Byzantine failure.

Randomized consensus algorithms can circumvent the FLP impossibility result by achieving both safety and liveness with overwhelming probability, even under worst-case scheduling scenarios such as an intelligent denial-of-service attacker in the network.