Unit-4 – Check pointing & Rollback Recovery & Consensus.

In Distributed System check pointing & Rollback recovery are used for fault Tolerance.

→ Fault tolerance increases the performance of a System in terms of Availability & Reliability

Fault tolerance is an ability or capability of a System to continue its operation without interruption, when one or more components of the System fail.

Rollback Recovery - Restore the System back to a consistent state after a failure. It achieve fault tolerance by periodically saving the state of a process during the failure free execution

→ It treats distributed System as a collection of processes that communicate through a network.

- up on failure, a failed process restarts from on of its Saved states. which reduces the amount of lost computation.

→ The Saved state of a process at an instance of a time is called checkpoint

→ In distributed System, all processes Save their local states at certain instants of time. This Saved state is called local checkpoint.

→ The contents of a check point depend upon the application content and the check pointing method.

→ Each process periodically Saves its state on stable storage.

→ The Saved state contains Sufficient information to restart the process Execution.

→ Global checkpoint is a Set of 'N' local check points.

check points & Different types of messages.

The Saved states of a process at a particular instance of time is check point.

All processes save their local states at certain instants of time.

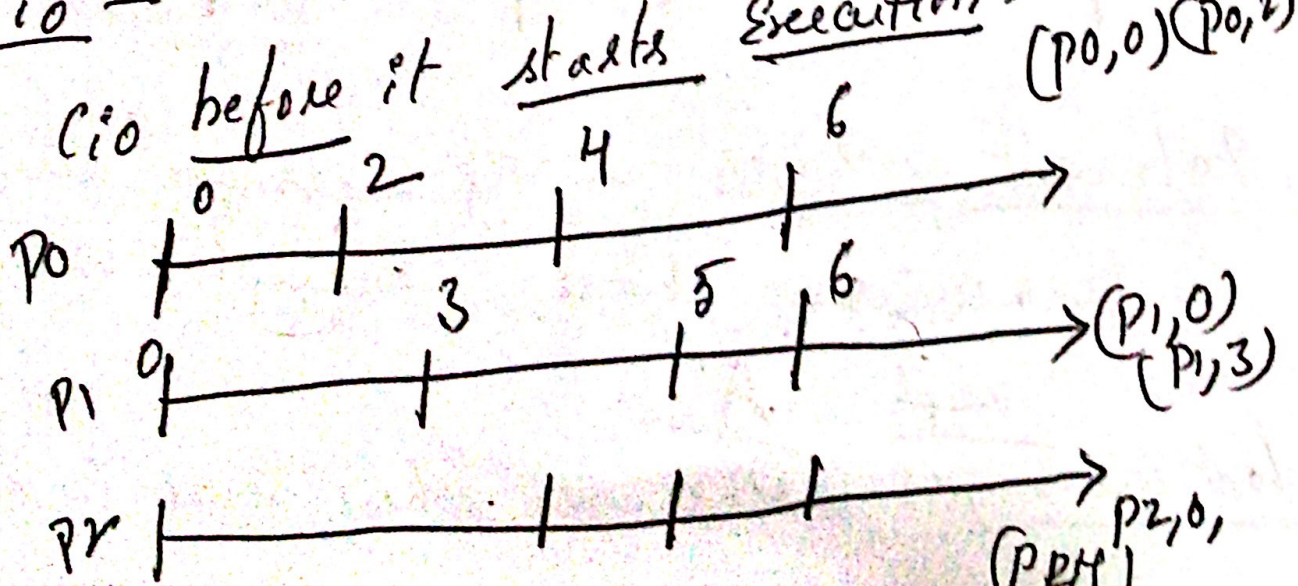
→ A local check point is a snapshot of the state of the process at a given instance.

→ A process stores all local checkpoints on the stable storage.

→ A process is able to roll back to any of its existing local checkpoints.

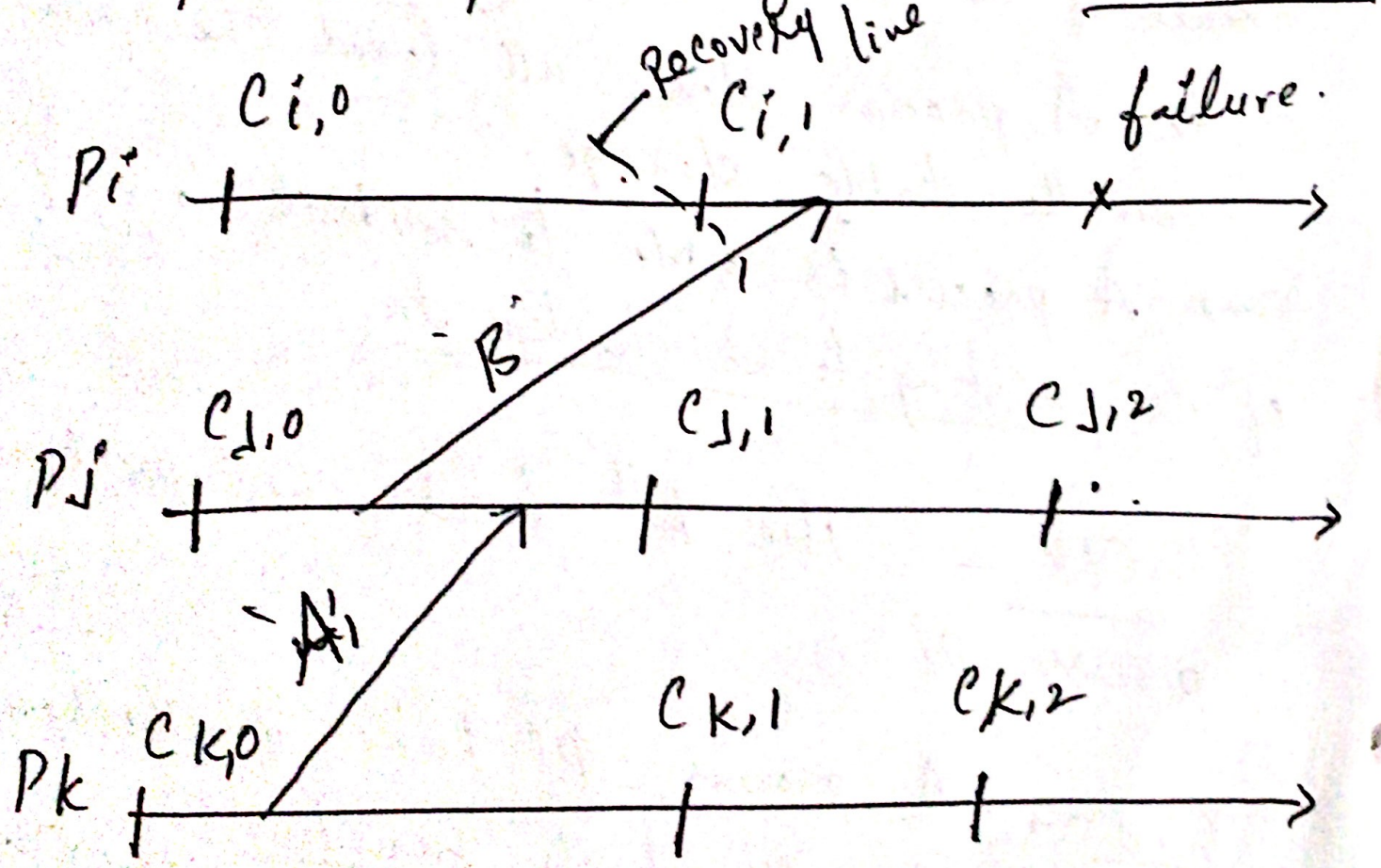
C_{ik} – The k^{th} local check point at process P_i.

C_{io} – A process P_i takes a checkpoint C_{io} before it starts Execution.



Messages considered on failure issues -

processes P_i, P_j, & P_k have taken check points $\{C_{i,0}, C_{i,1}\}$, $C_{j,0}$ $C_{j,1}$ $C_{j,2}\}$ and $\{C_{k,0}, C_{k,1}\}$ respectively. and assume different types of following messages are generated from A to J



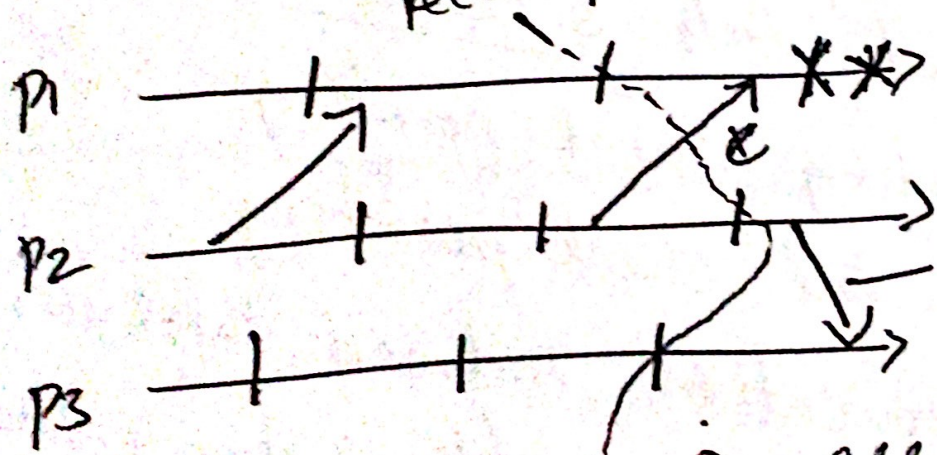
In transit messages :- The messages which are in transmission phase. example message 'A'

lost messages:- Messages whose send is not

but receive is undone due to roll back are called lost messages.

This type of messages occurs when the process rolls back to a check point prior to the reception of the message. Message 'B' is an example.

Delayed messages. Messages whose receive is not recorded, because the receiving process was either down or messages arrived after the roll back.



messages C & D are delayed messages.

Orphan messages. Messages receive is recorded, but sent not recorded are called Orphan messages.

Duplicate messages.

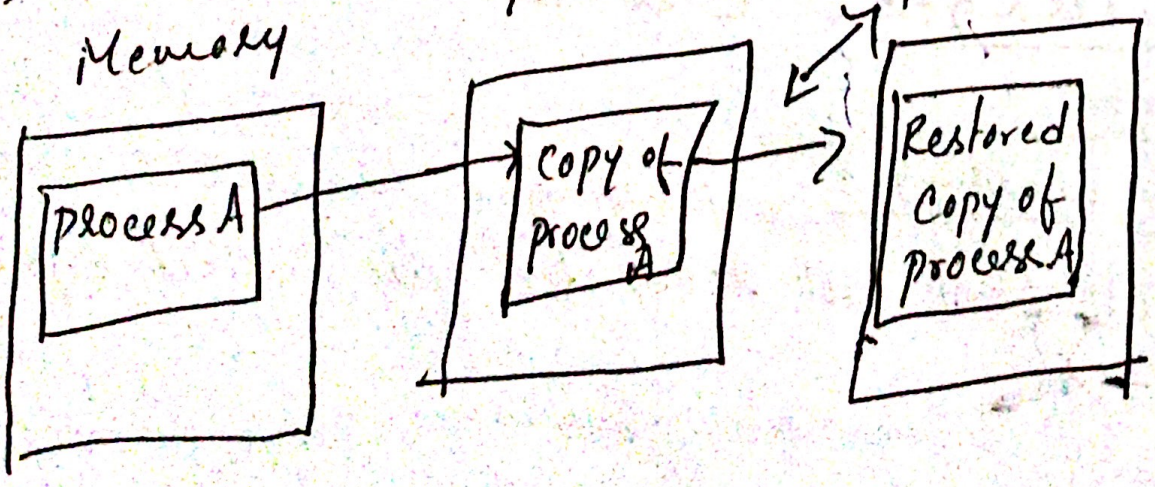
Duplicate messages arise due to message logging & Replaying during process recovery.

Check-point based Rollback Recovery.

up on a failure, check point based roll back recovery restores the System state to the most recent Consistent Set of check points.

check point based recovery techniques are classified into 3 categories. They are

1) Uncoording check pointing.
2) Coordinated check pointing.
3) Communication - Induced checkpointing.



Any consistent global checkpoint can be used to restart the process execution upon a failure.

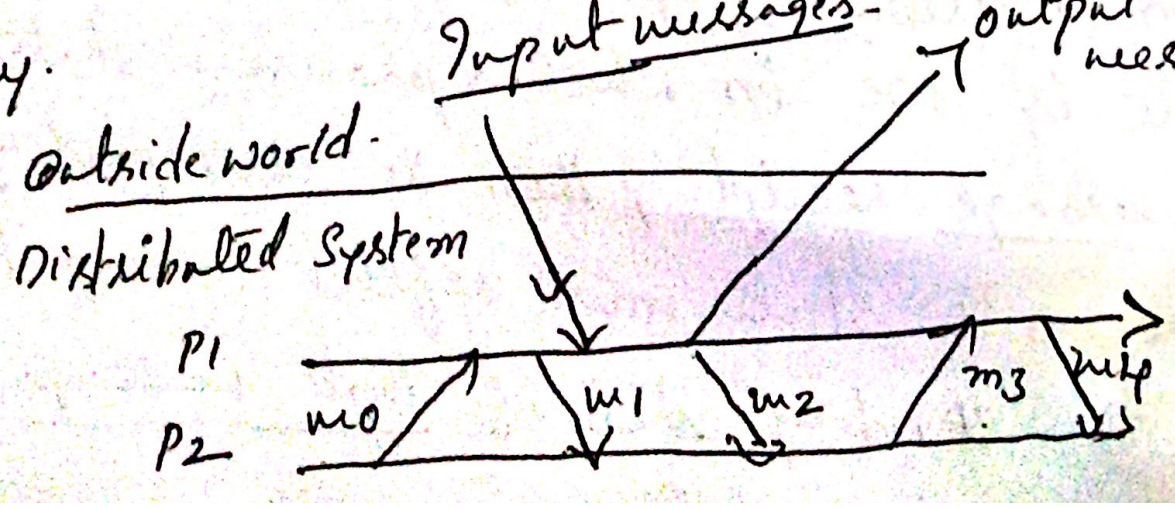
→ The most recent consistent global checkpoint is termed as the recovery line.

→ The fundamental goal of the recovery roll back protocol is to bring the system into a consistent state whenever inconsistencies occur because of a failure.

Background: System model: / Issues in Failures.

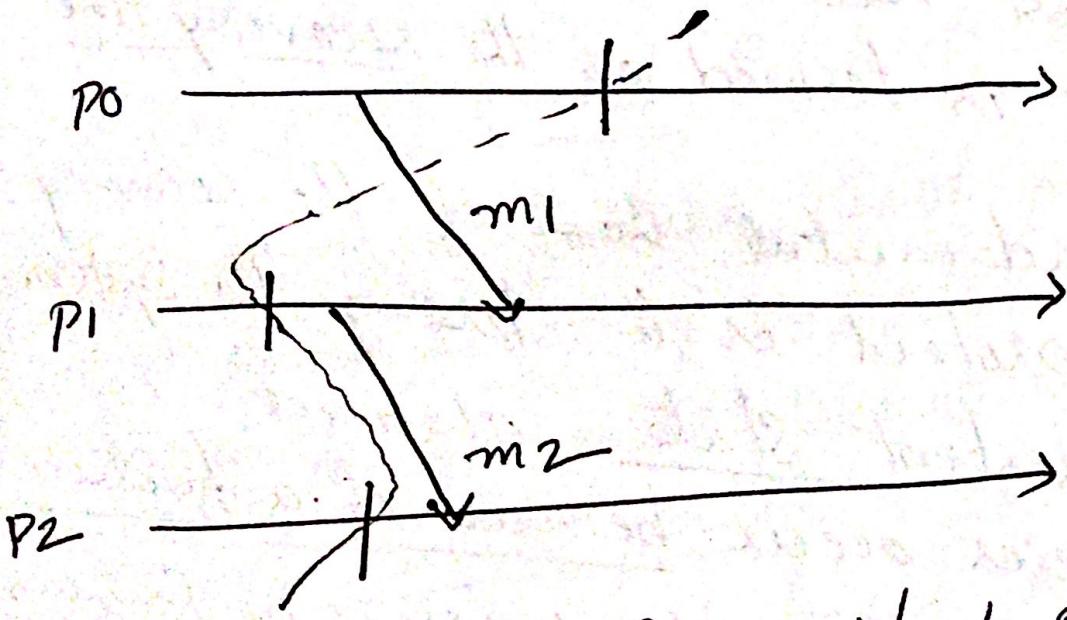
A distributed system consists of a fixed no. of processes $P_1, P_2, \cdots P_N$, which communicate through messages.

processes cooperate with each other as well as to the outside world through messages only.

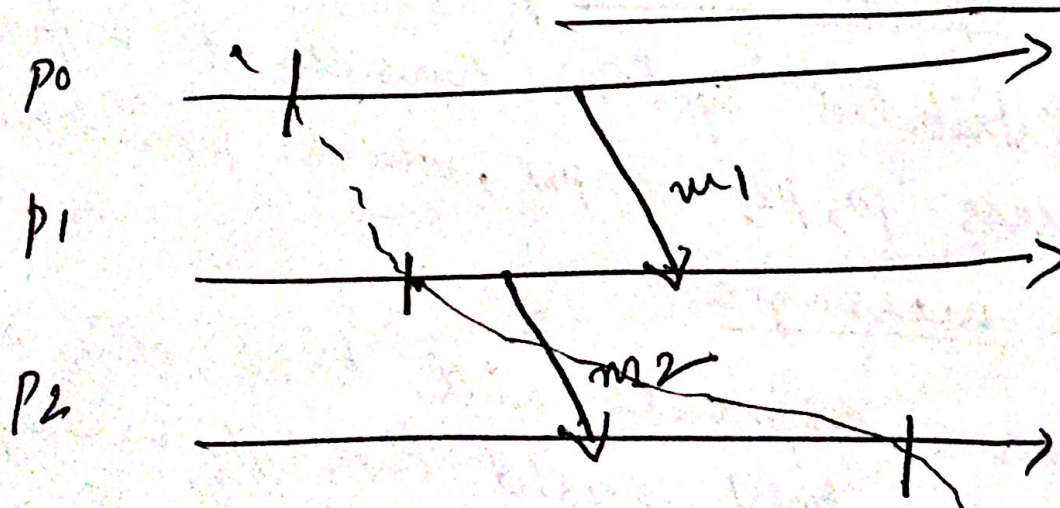


Always consistent state is needed for failure free execution in distributed System.

consistent state



Inconsistent state



In distributed System, a consistent System state is one in which a process's state is a message receipt, then the corresponding Sender process's state must reflect ~~corresp~~ the sending state of that message.

Check point based Roll back Recovery

Continuation from previous -

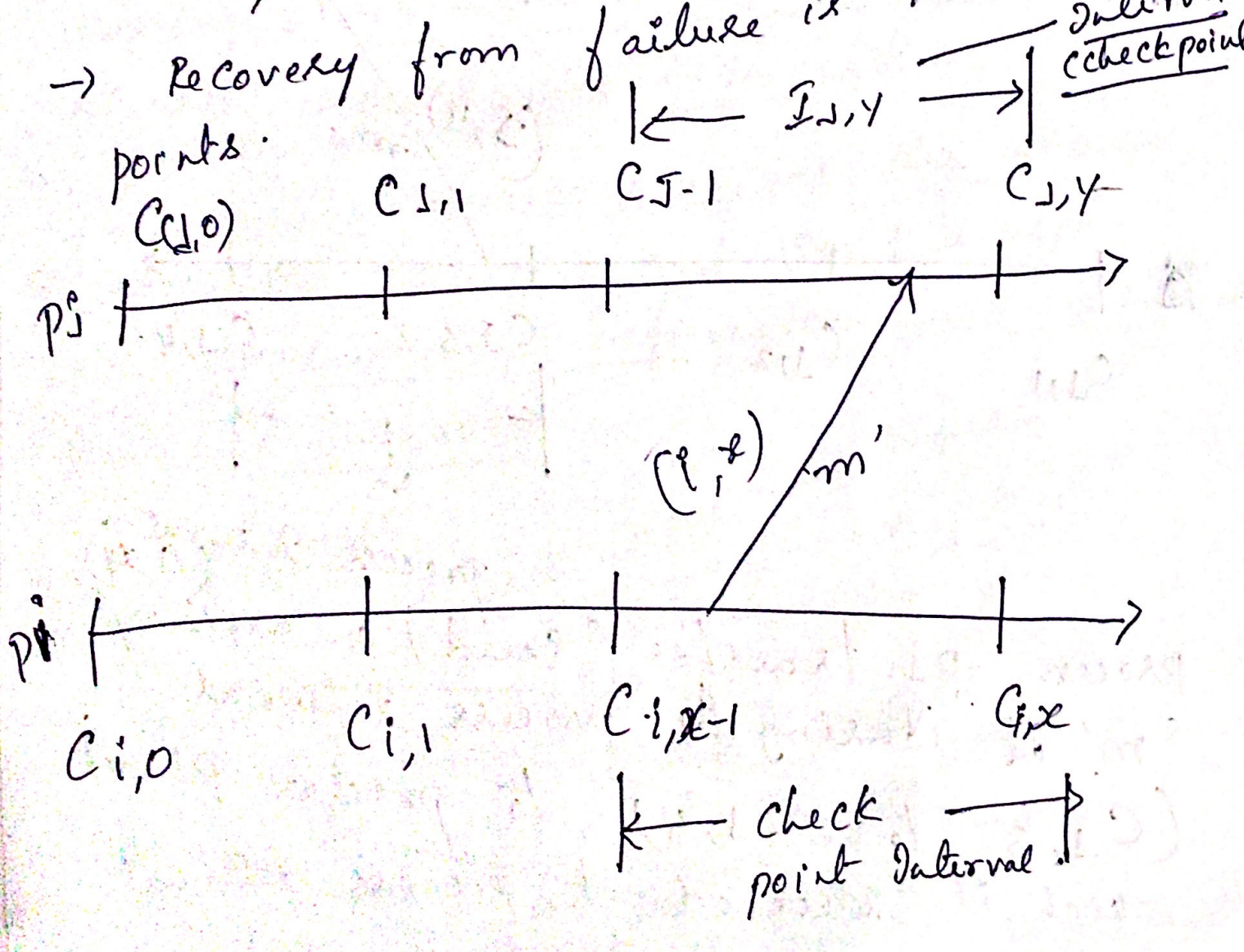
1. Uncoordinated check pointing.

→ In uncoordinated check pointing there is no need of coordination b/w processes.

→ Each process has autonomy to take the check points as per their convenience.

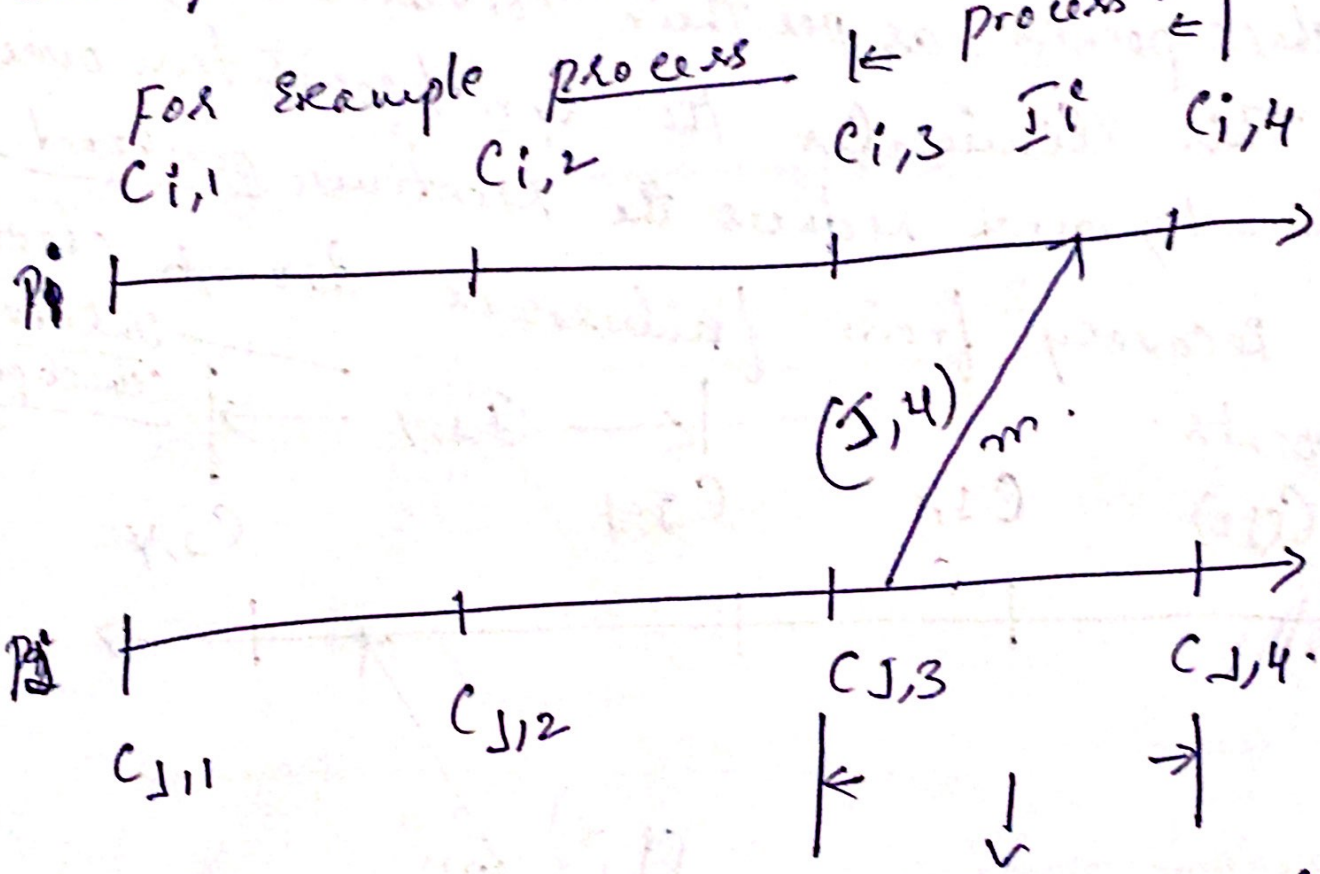
→ This Eliminates the Synchronization overhead, which reduces the runtime overhead.

→ Recovery from failure is through check points.



→ Recovery through this method is using a consistent global check point.

→ A consistent global check point can be determined by recording the dependencies among the check points, caused during message ~~exchange~~ exchange during normal.



process Pi transfer (send) a message 'm' at during the process interval (Cj,3 to Cj,4) to process Pi. which is received at process interval

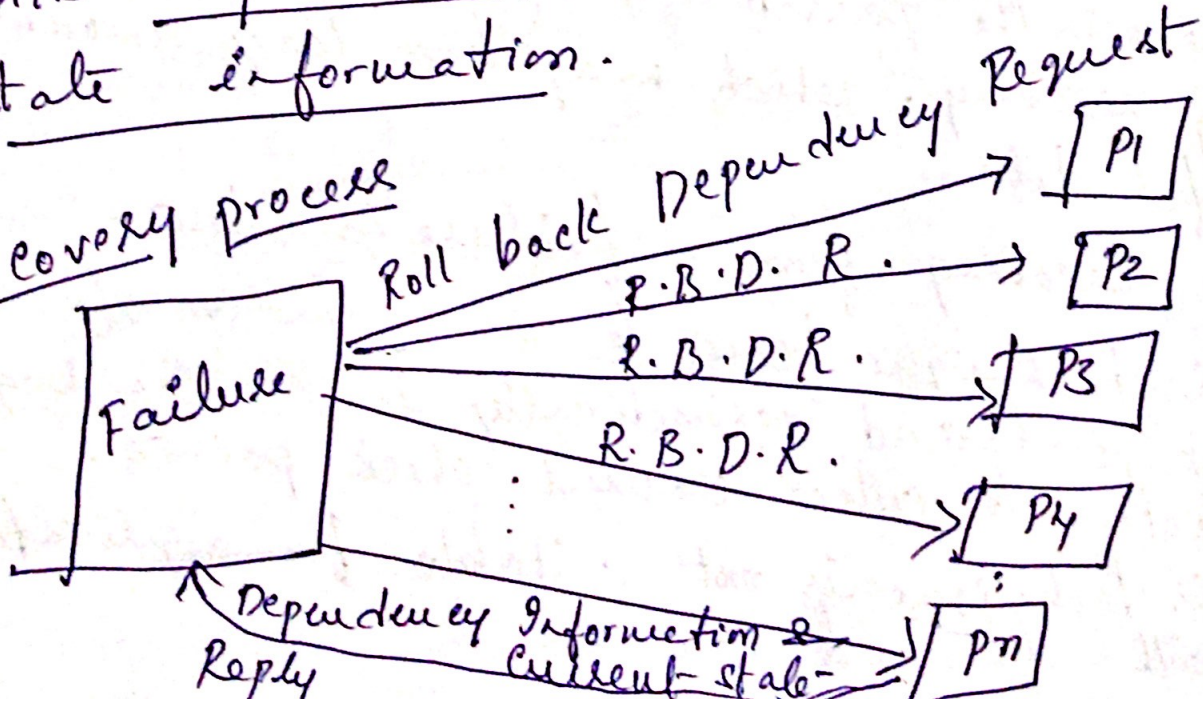
interval (I, 4) i.e between $C_{i,3}$ & $C_{i,4}$.

Along with the message 'm', its dependency (I,4) also 'piggybacks' from processor Pj to Pi.

These dependencies are also saved on stable storage when process pi takes check point (Ci, 4).

→ When ever a failure occurs, the recovering process initiates a rollback by broadcasting a dependency request message to all all dependencies information.

When a process receives this message, it stops its execution and send a reply with dependency information and its current state information.



→ The recovery process then calculates the recovery line based upon the dependencies information from all processes (Global Dependency information).

→ After that it broadcasts a Roll back request message with the recovery line information.

→ Upon receiving this ~~information~~ message, a process whose current state belongs to the recovery line resumes its execution or roll backs to an earlier checkpoint as indicated by the recovery line.

Disadvantages

1) There is the possibility of Domino effect during recovery, which may lose large amounts of useful data.
2) The recovery from a failure is slow.
3) This technique forces to use more no of check points and periodically invokes Garbage Collector to collect unused check points.
4. This technique is not suitable for applications with frequent commits.

Coordinated check pointing - In Coordinated check pointing all processes Coordinate their check points information to form a Global Consistent State.

→ Coordinated check pointing Simplifies recovery and it is not Susceptible to 'Domino effect'

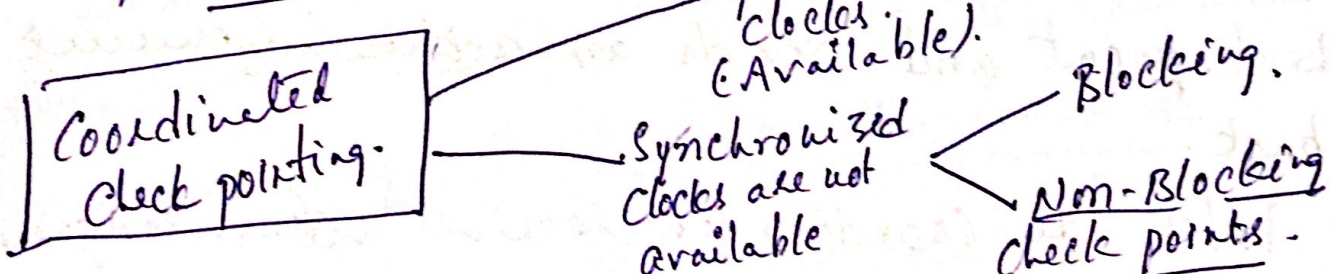
→ Coordinated check pointing technique maintains only one check point on the stable storage.

→ This technique reduces the storage overhead and there is no need for Garbage collection.

→ Large latency is required in this technique.

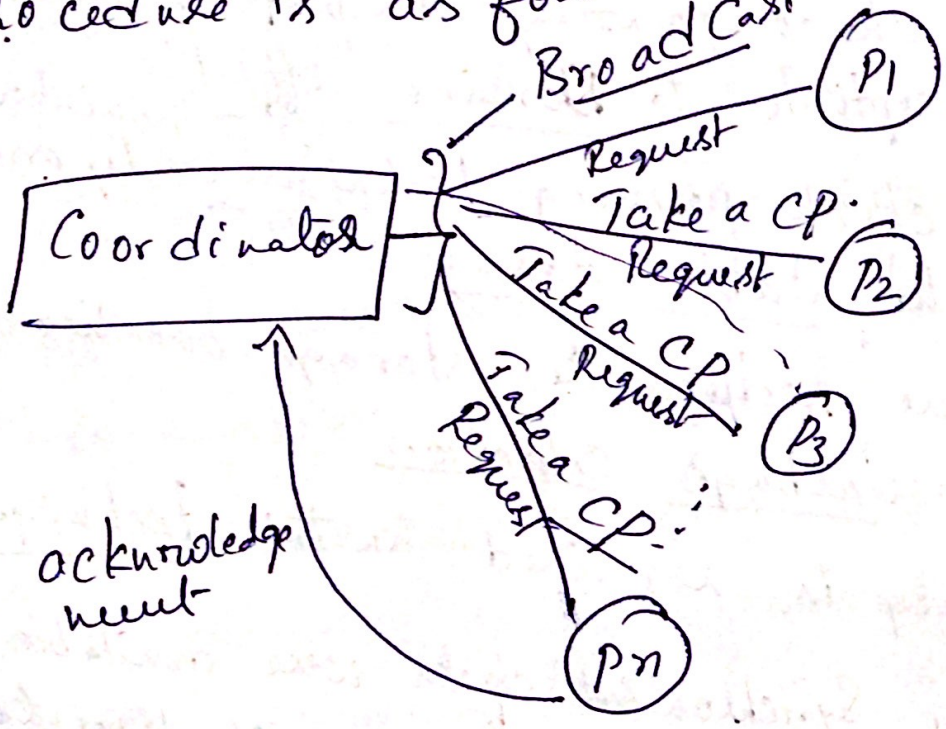
- If perfectly Synchronized clocks were available at processes, All processes agree to take checkpoints at particular instance of time.

→ If perfectly Synchronized clocks are not available 2 types of techniques are used. They are a) Blocking & (b) Non-Blocking.



Blocking coordinated check pointing.

In this techniques, communications are blocked while check pointing protocol executes. The procedure is as follows...

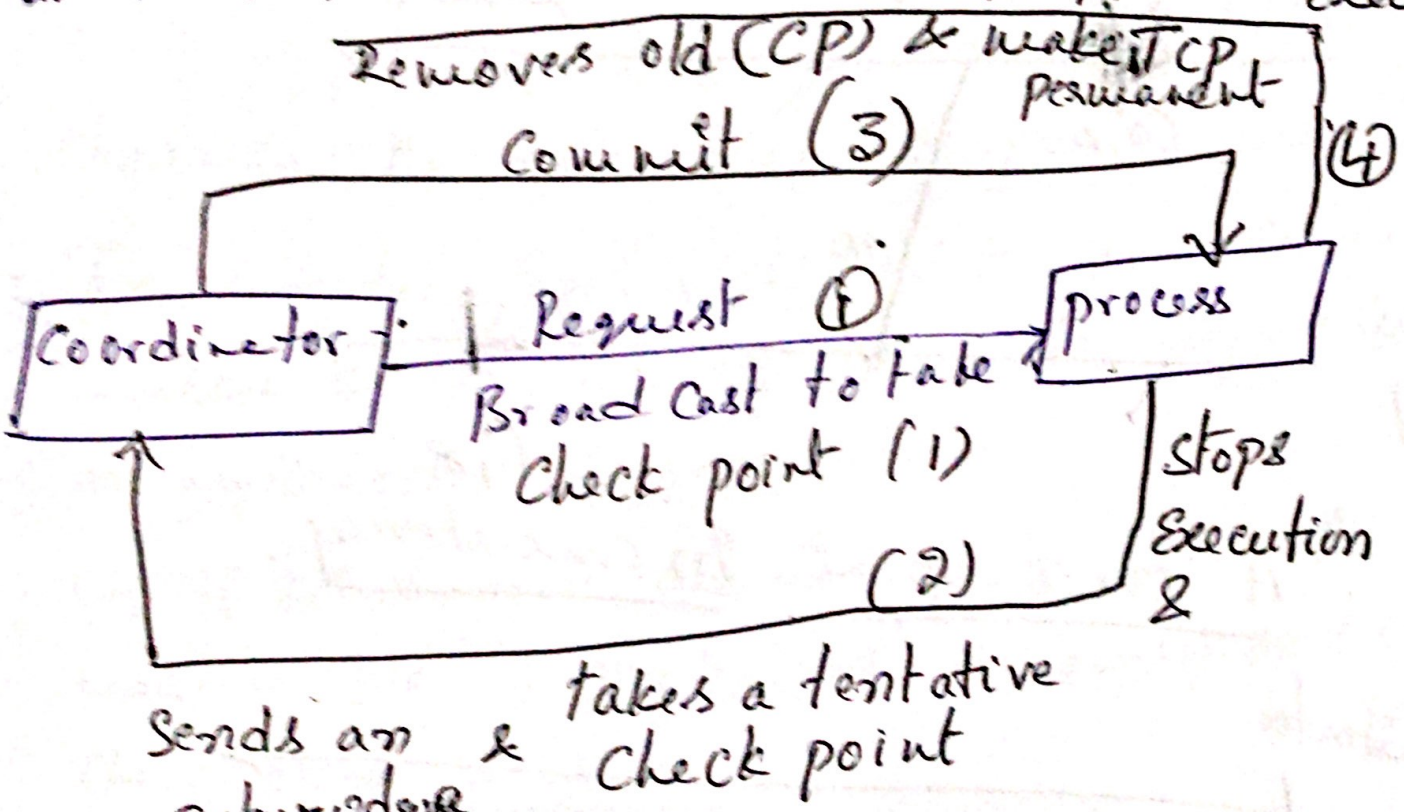


a) The coordinator takes a check point and broadcasts a request message to all processes asking them to take a check point.

b) When a process receives this message, it stops execution and takes a tentative check point and sends an acknowledgement back.

c) After, the coordinator receives acknowledgments from all processes, it broadcasts a commit message.

d) After receiving the commit message the process removes old check point and makes the tentative check point permanent & then resumes its execution.

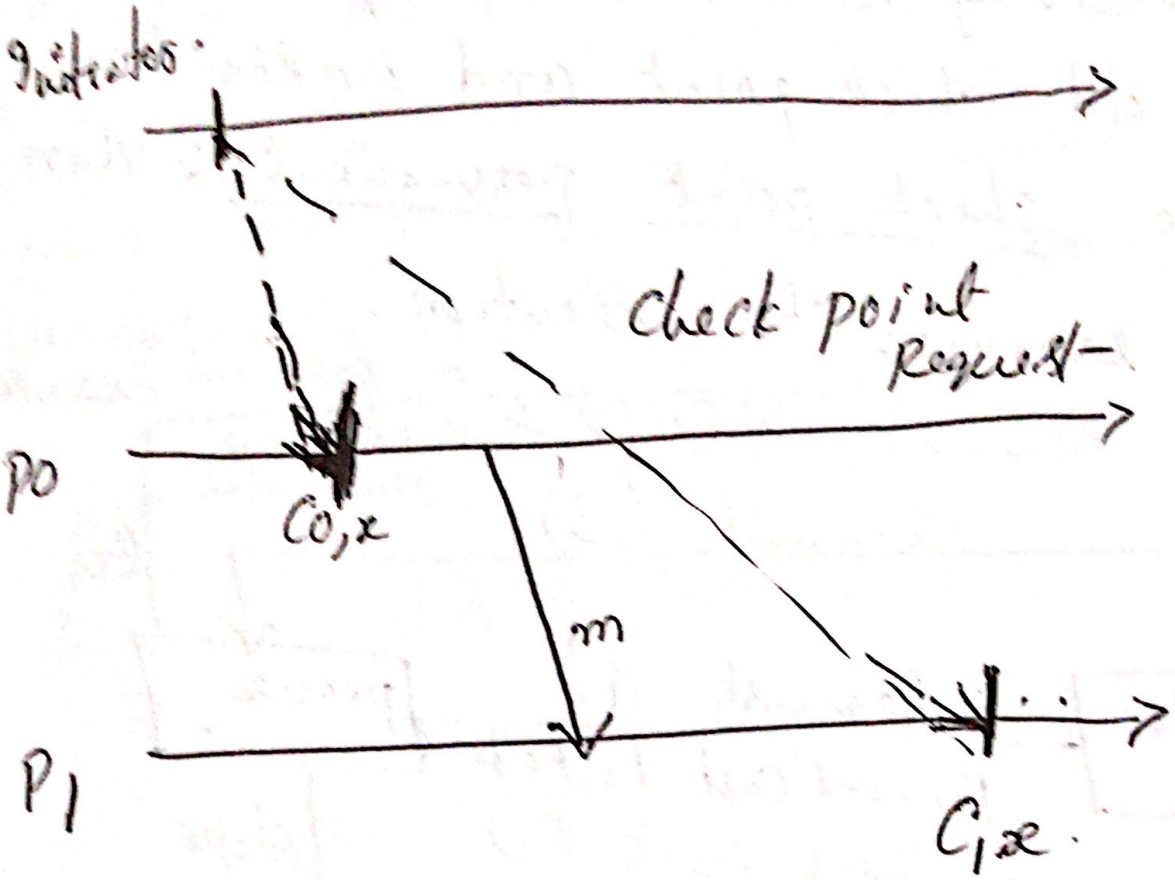


(Same for all processes)

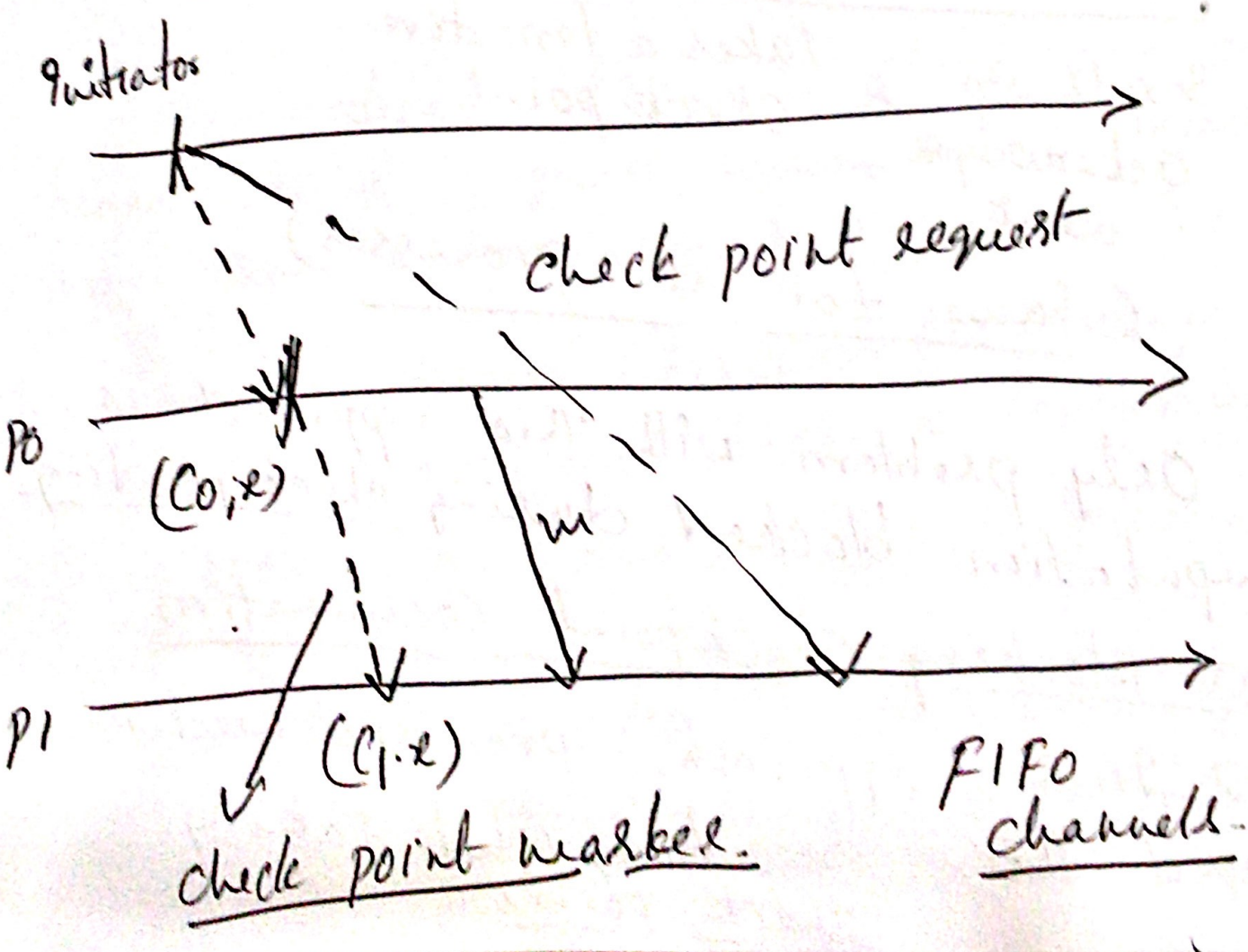
→ Only problem with this approach is Computation blocked during check pointing.

Non-Blocking Checkpoint coordination.

→ In this approach, Processes need not stopped their execution, while taking check points.



A check point In consistency-



check point markers.

FIFO channels.

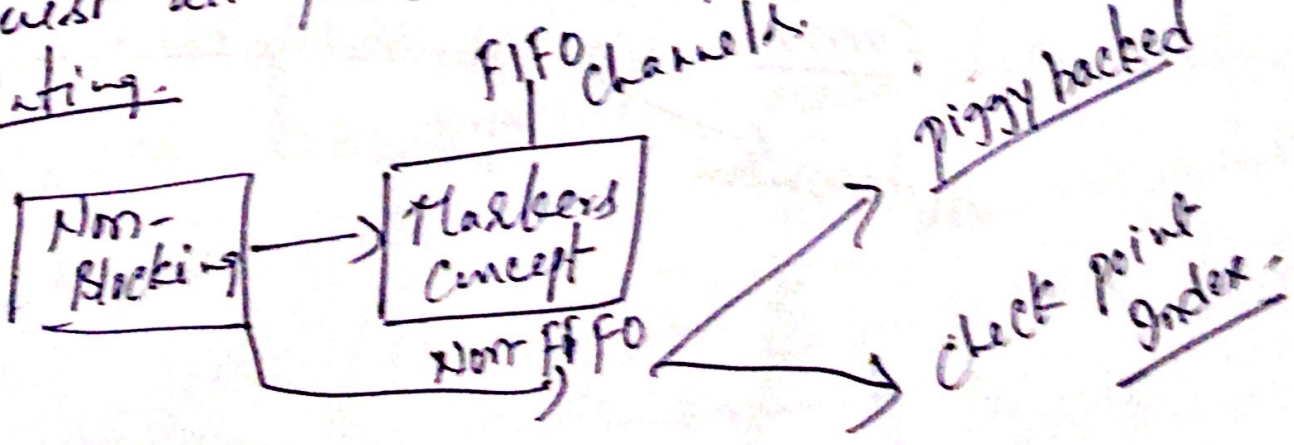
A fundamental problem in Non-blocking Coordinated check point is that because of non blocking of messages, an application message can make checkpoint inconsistent

To avoid this problem, in FIFO channels the concept of "Markers" are used. The initiator takes a checkpoint and sends a marker to all outgoing channels before sending an application message.

→ If the channels are non-FIFO, then the markers are piggybacked on every post-check point message.

(OR)

check point indices can serve this purpose if the receiver's local check point index is lower than the piggybacked check point index, then coordinator request all processes to participate in check pointing.



Communication-induced check pointing Communication induced check pointing (CIC) protocols avoid the domino effect without requiring all checkpoint to be coordinated. **Communication-induced check pointing reduces** or completely eliminates the useless checkpoints.

In communication-induced check pointing, processes take two types of checkpoints, namely **Autonomous and Forced checkpoints.**

The checkpoints that a process takes independently are called ***local*** **checkpoints, / Autonomous check Points ,** while the check points those are taken forcibly are **called *forced* checkpoints.**

The forced checkpoint must be taken before the application may process the contents of the message. It is therefore desirable in these systems to minimize the number of forced checkpoints.

Two types of communication-induced check pointing techniques are used. **They are Model based check pointing and Index-based check pointing.**

In *model-based check pointing,* the system maintains checkpoints and communication structures to prevent the domino effect.

In **Index-based check pointing,** the system uses an indexing scheme for the local and forced checkpoint, such that the checkpoints of the same index at all processes form a consistent state.

Communication Induce check points

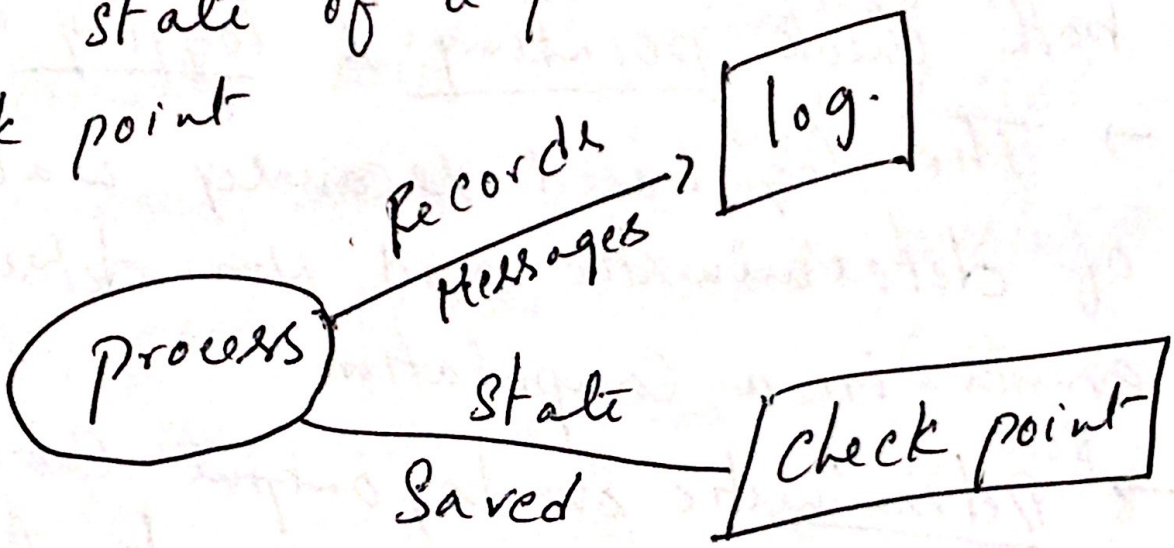
Autonomous Forced.

Log based Rollback recovery.

In general message logging & checkpointing are used for fault tolerance in Distributed Environment.

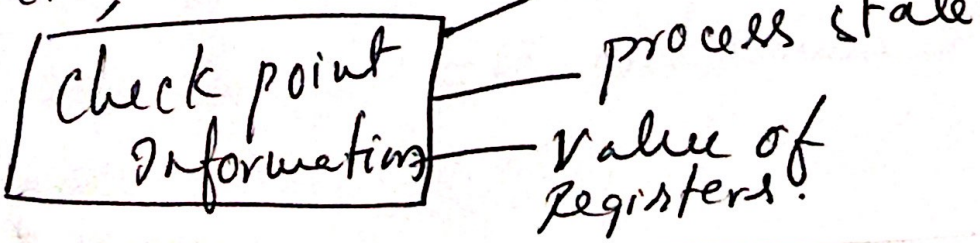
Each message received by a process is recorded in a 'log'.

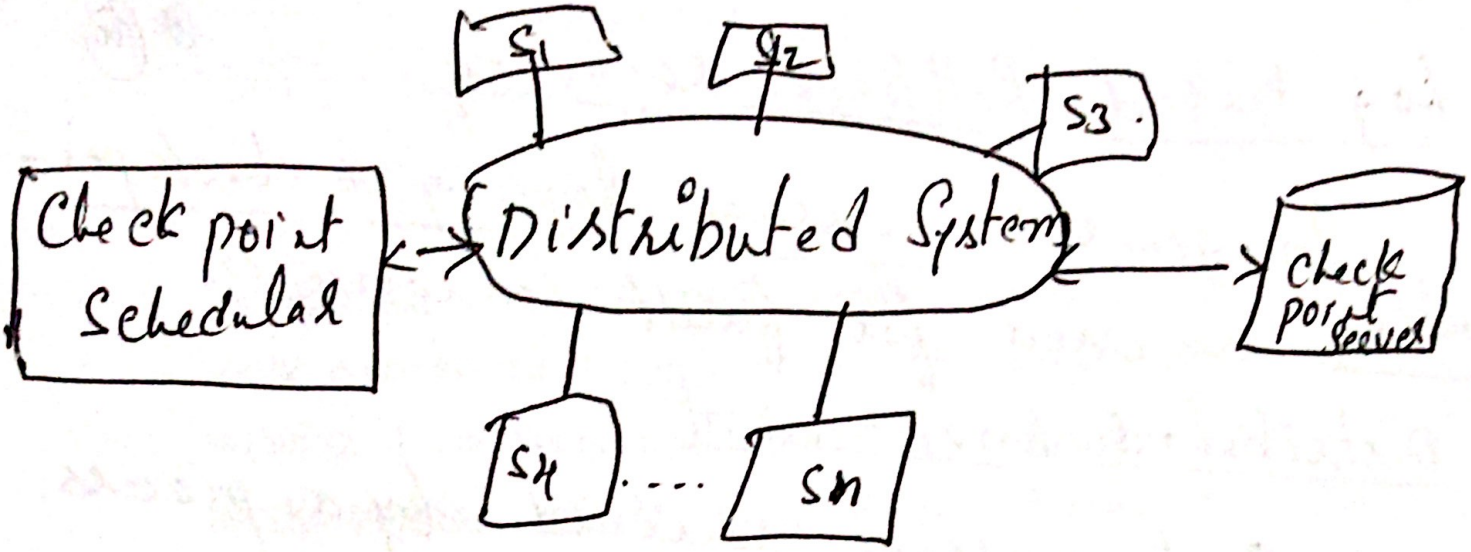
→ The state of a process is saved as check point



logged messages & check points are saved in Memory - which is useful during recovery.

The check point Information that is stored includes Environment, process state, value of the Registers etc;





→ log Based recovery techniques combines both check pointing & logging of Events.

→ The log based recovery makes use of deterministic and Non-deterministic events in a Computation.

→ For Deterministic Events, ~~output~~ ~~Can be~~ for a given particular input the computer will always produce the same output.

In non deterministic events - for the same input, different outputs will be generated in different runs.

Pessimistic message logging. (Believe that the worst will happen).

→ pessimistic → the worst will happen).

→ pessimistic message logging protocols log messages Asynchronously

→ These protocols guarantees that any failed processes can be recovered individually without affecting the states of processes, which did not failed.

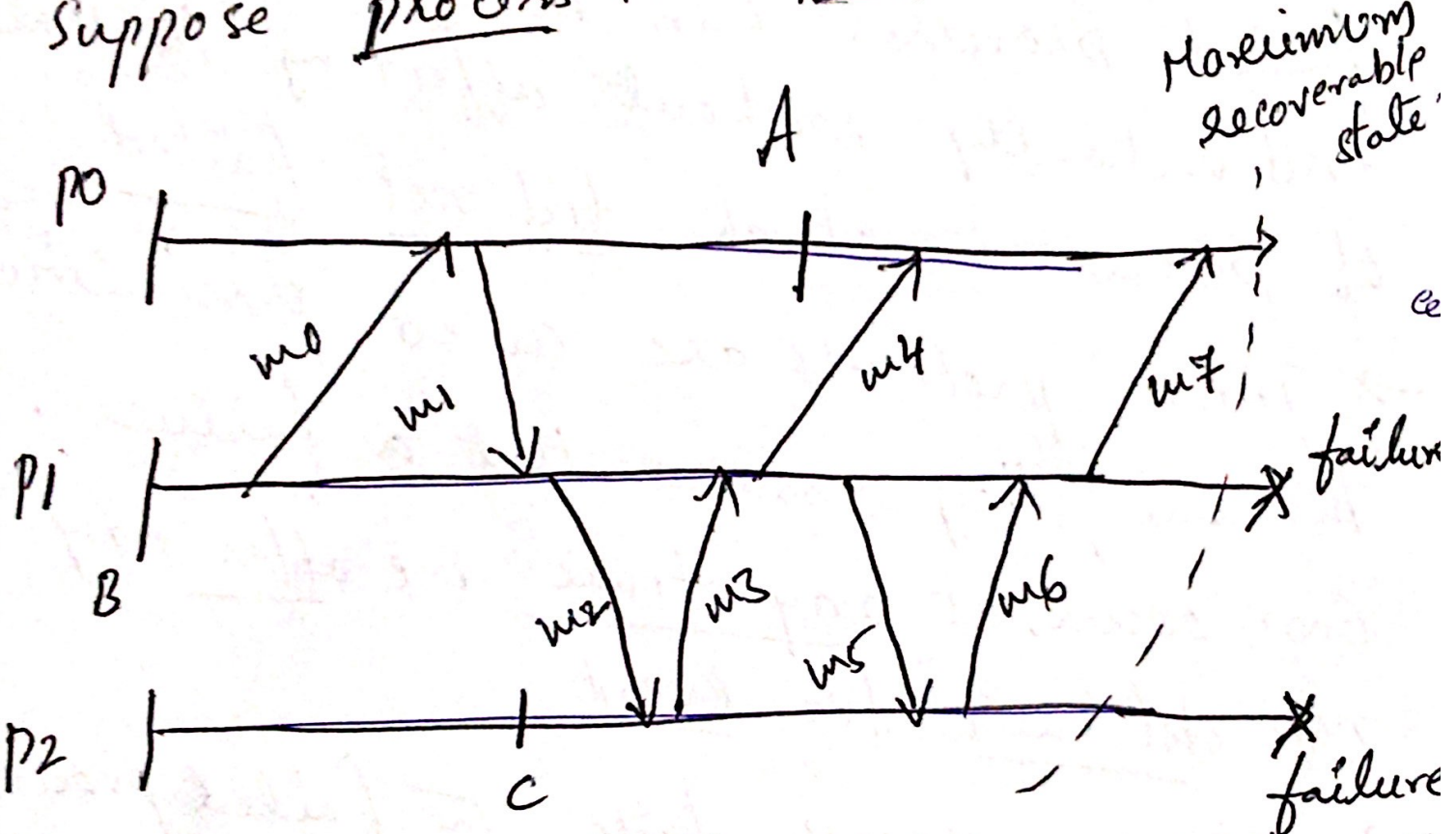
→ These protocols are called pessimistic because, they assume that a failure can occur, at any time i.e after any non deterministic event.

→ When a process fails, the failed process is always restarted from its most recent checkpoint, and all the messages recorded ~~in the~~ in the log after that checkpoint are replayed in the same order

→ Based on this the process deterministically reexecuted.

pessimistic message logging protocols have the advantage of being able to restart the System without affecting the states of other processes which did not fail.

Suppose process P1 & P2 fail as shown



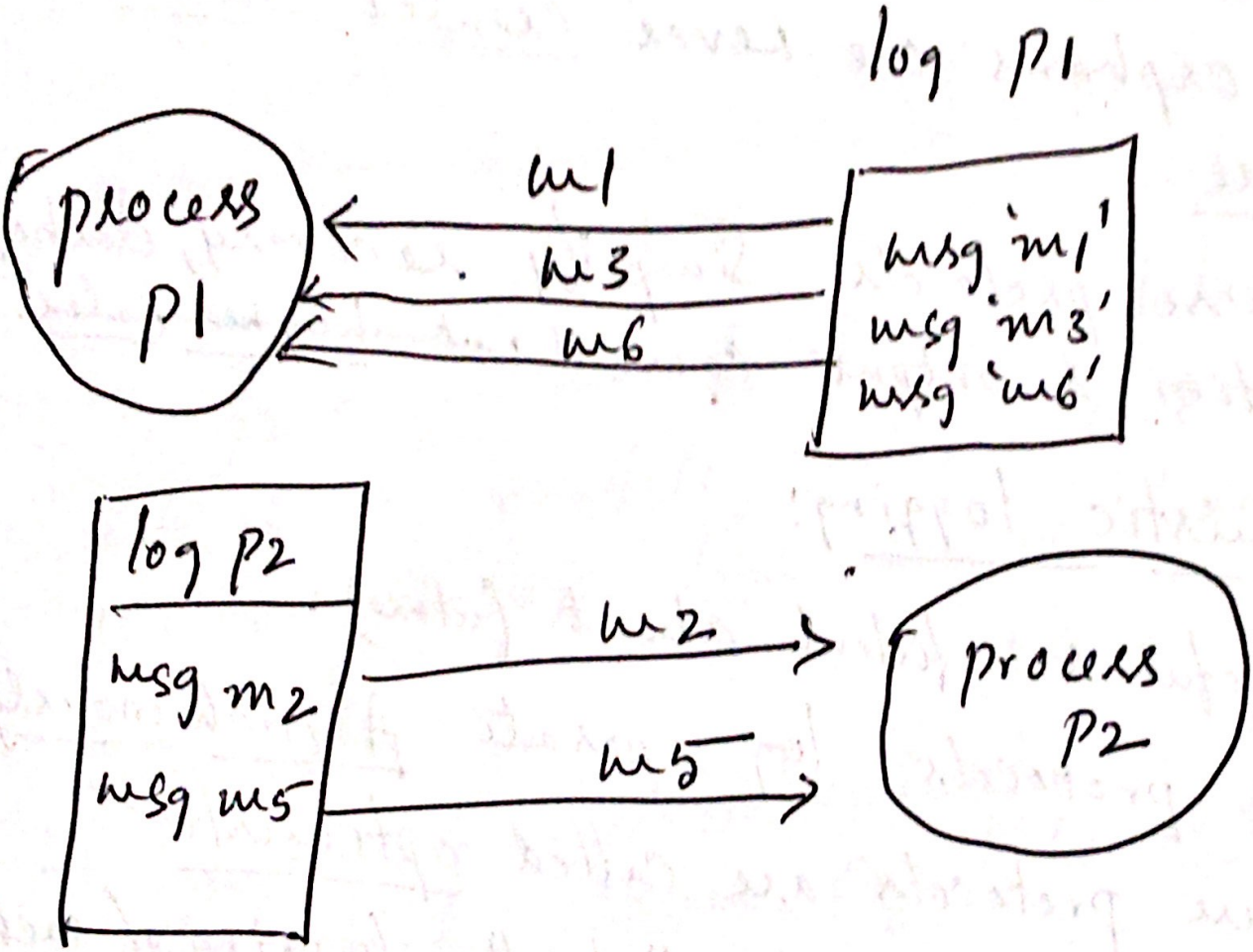
During the failure free operation, the logs of processes P0, P1 & P2 contains determinants needed to restart the System like as follows

P0 — log - {m0, m4, m7}

P1 — {m1, m3, m6}.

P2 — {m2, m5}

If Suppose P1 & P2 fail and restarted from the check points B & C, their concerned determinant logs deliver the same messages in the same sequence as like in pre-failure execution.



This guarantees that P1 & P2 will repeat execution as a part of recovery.

After recovery both the processes will be consistent with the state of of 'P0' that include the receipt of message 'm7' from P1.

In pessimistic logging System, the observable state of each process is always recoverable.

→ Synchronous logging of results high performance.

Pessimistic ~~Optimistic~~ logging - Recovery protocols guarantee that orphans are never created due to failure.

→ These protocols Simplify recovery, Garbage Collection & output commit. ~~at higher rates.~~

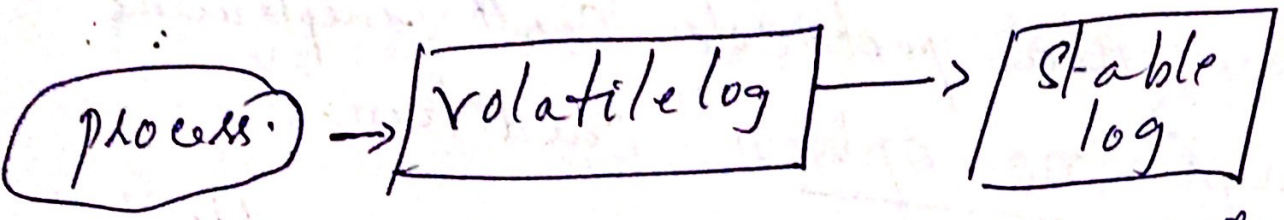
Optimistic logging:-

(Hopeful & confident about future).

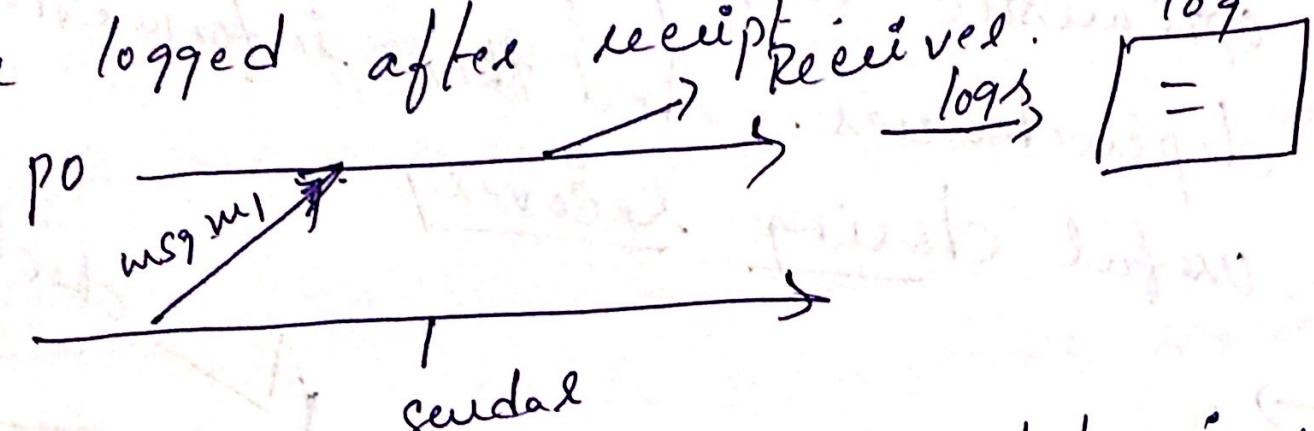
→ These protocols log operate Asynchronously

→ These protocols are called 'optimistic' because, they assume that the logging of each message received by a process will be completed before the process fails.

→ Determinants are first kept in volatile log and are periodically flushed to the stable storage.



In this techniques, the receiver of a message is not Blocked, and messages are logged after receipt.



→ If a process fails, the determinants in its volatile log are lost. So roll back through logs can be done from stable log.

→ The state of non-deterministic events corresponding failed process can't be recovered.

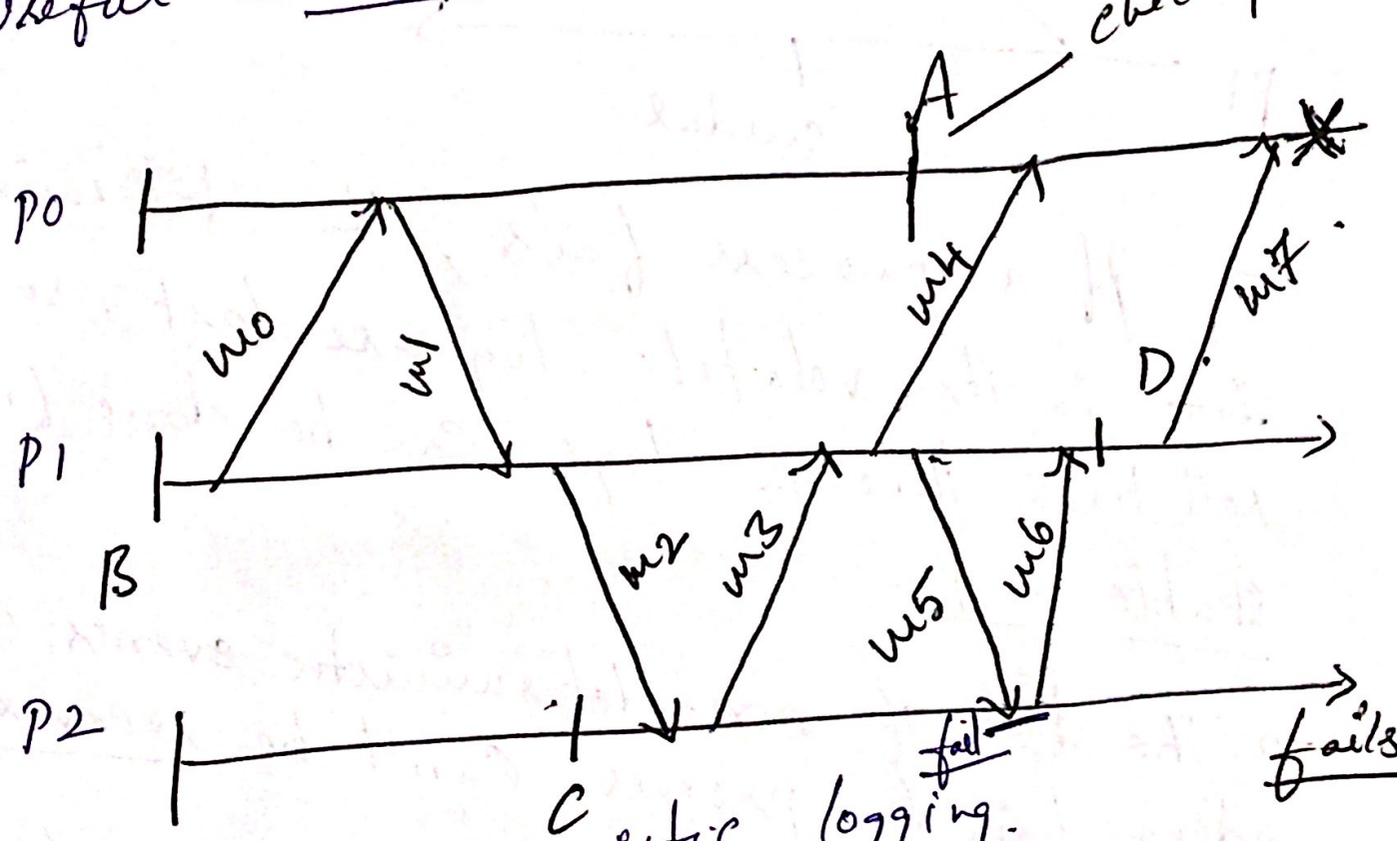
→ If the failed process sent message during state intervals can't be recovered.

→ The receiver of a message becomes an orphan process and it must also roll back to undo the effects of receiving message.

→ Hence, these protocols can't implement 'Always - no - orphan' condition.

→ To perform roll back correctly, optimistic logging protocols use causal dependencies. And this information is useful during recovery.

→



optimistic logging.

Suppose process P2 fails before the determinant for m5 is logged to the stable storage, process 'P1' becomes an 'orphan' process & must roll back to undo the effects of message 'm6'

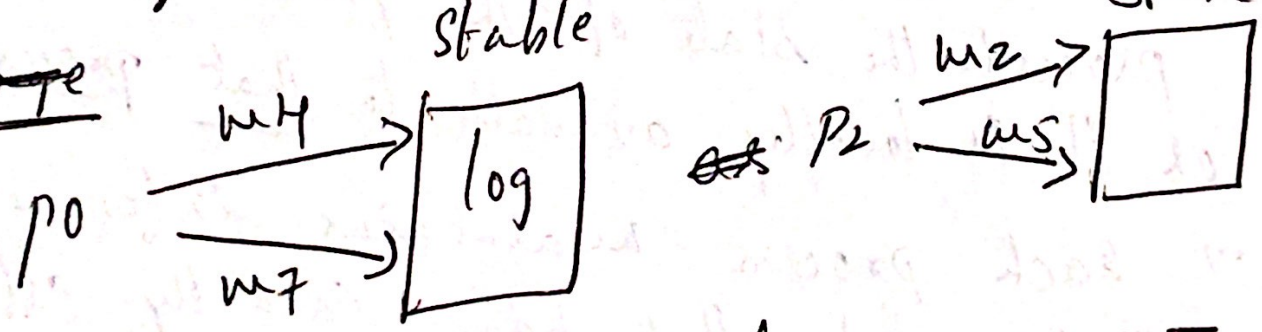
-> The roll back of P_1 further forces P_0 to roll back to undo the effects of receiving message m_7.

-> optimistic protocols need to keep multiple checkpoints for each process.

-> optimistic logging protocols require a "non-trivial garbage collection scheme"

-> Output commit in OL protocols requires a guarantee that no failure scenario can revoke the output.

For example if process 'P0' needs to commit output at state X, it must log messages m_4 & m_7 to the stable storage



In this case if process fails, The computation can be reconstructed up to state X

Storage & ask P2 to log m_2 & m_5.

Casual logging

→ Casual logging combines the advantages of both pessimistic & optimistic.

→ Like optimistic, it does not require synchronous Access to the stable storage except during output Commit.

→ Like pessimistic logging, it allows each process to Commit output independently & never creates orphans.

→ Rollback is limited to the most recent check point on the stable Storage.

→ In this approach, the determinant of each non-deterministic event that casually proceeds the state of the process is either stable or it is locally available to that process.

→ Each process maintains information about all the events that have casually affected its state.

→ This information is useful during failures

The check pointing algorithm

The checkpoint algorithm works on the following assumptions about the distributed system:

- Processes communicate by exchanging messages through communication channels.
- Communication channels are FIFO.
- Communication failures do not partition the network.
- No process fails during the execution of the algorithm.

The checkpoint algorithm takes two kinds of checkpoints on the stable storage, **They are Permanent and Tentative.**

A permanent checkpoint is a local checkpoint at a process and is a part of a consistent global checkpoint.

A tentative checkpoint is a temporary checkpoint

In case of a failure, processes roll back only to their permanent checkpoints for recovery.

First phase

An initiating process Pi takes a tentative checkpoint and requests all other processes to take tentative checkpoints.

After taking the tentative check points, Each process informs to the initiator process Pi whether it succeeded in taking a tentative checkpoint.

A process says "no" to a request if it fails to take a tentative checkpoint.

If Pi learns that all the processes have successfully taken tentative checkpoints, Pi decides that all tentative checkpoints should be made permanent; otherwise, Pi decides that all the tentative checkpoints should be discarded.

Second phase

In second phase, after all check points the initiator process Pi informs to all that we have reached at the end of the first phase. And do not send any messages until there is an instruction / Decision.

[The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the underlying computation until it is informed of Pi's decision.]

Correctness

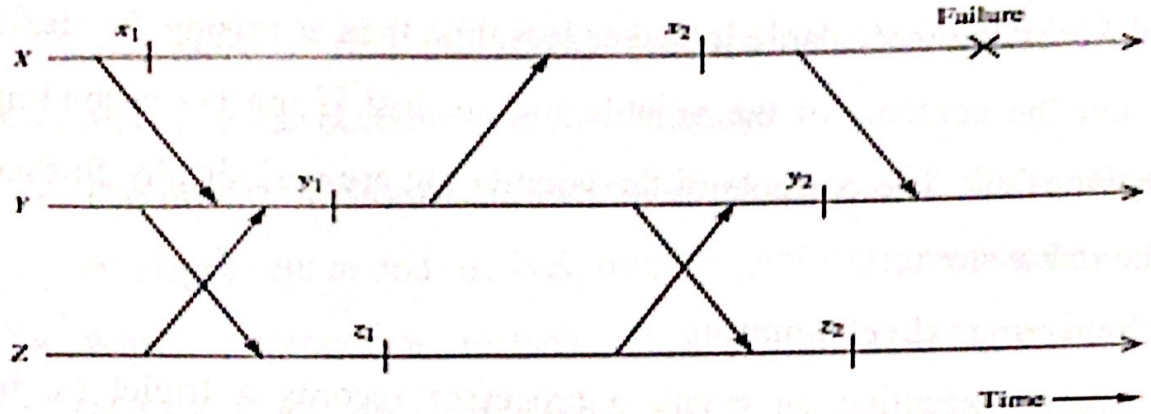
A set of permanent checkpoints taken by this algorithm is consistent because of the following two reasons:

First, Either all or none of the processes take permanent checkpoints, second, no process sends a message after taking a tentative checkpoint until the receipt of the initiating process's decision.

An optimization : Taking unnecessary check points are avoided.

Algorithm for Asynchronous Check pointing and Recovery

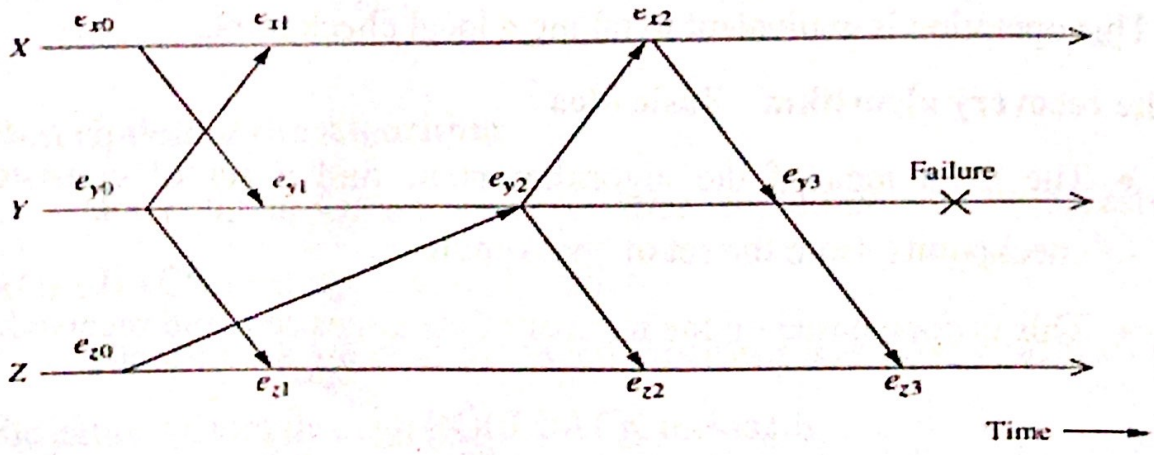
(Juang–Venkatesan algorithm for asynchronous checkpointing and recovery)



System model and assumptions

The algorithm makes the following assumptions about the underlying system:

- The communication channels are reliable, deliver the messages in FIFO order, and have infinite buffers.
- The message transmission delay is arbitrary, but finite.
- The processors directly connected to a processor via communication channels are called its neighbors.
- The underlying computation or application is assumed to be event-driven.



To facilitate recovery after a process failure and restore the system to a consistent state, **two types of log storage are maintained, volatile log and stable log.**

Accessing the volatile log takes less time than accessing the stable log, but the contents of the volatile log are lost if the corresponding processor fails. The contents of the volatile log are periodically flushed to the stable storage.

Asynchronous check pointing

After executing an event, a processor records a triplet (s, m, msgs_sent) in its volatile storage,

where 's' is the state of the processor before the event,

'm' is the message

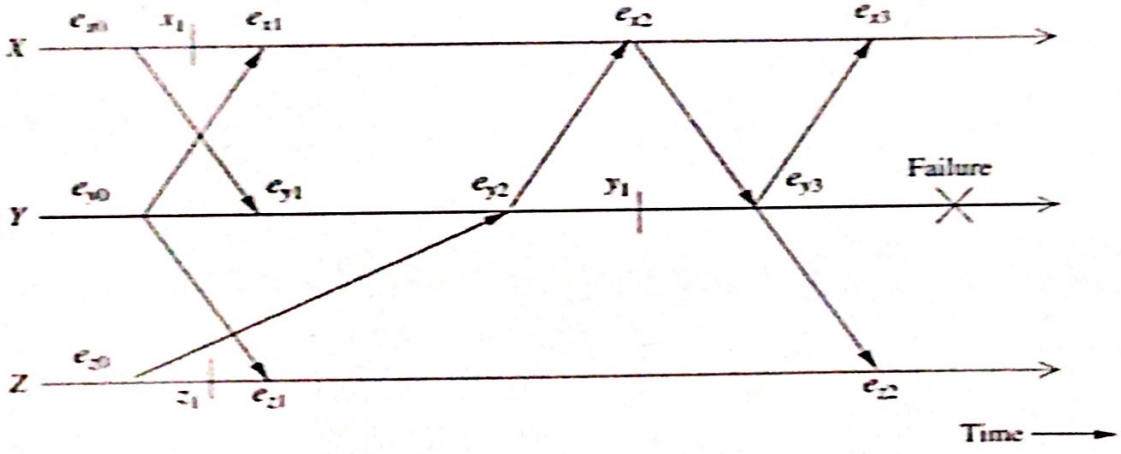
'msqs_sent' is the set of messages that were sent by the processor during the event.

- A local checkpoint at a processor consists of the record of an event occurring at the processor and it is taken without any synchronization with other processors.
- Periodically, a processor independently saves the contents of the volatile log in the stable storage and clears the volatile log.
- This operation is equivalent to taking a local checkpoint.

The recovery algorithm - Basic idea

- The main idea of the algorithm is to find a set of consistent checkpoints, from the set of checkpoints.
- This is done based on the number of messages sent and received.

- Recovery may involve multiple iterations of roll backs by processors.
- Whenever a processor rolls back, it is necessary for all other processors to find out if any message sent by the rolled back processor has become an orphan message.
- The orphan messages are identified by comparing the number of messages sent to and received from neighboring processors.
- When a processor restarts after a failure, it broadcasts a ROLLBACK message that it has failed.
- The recovery algorithm at a processor is initiated when it restarts after a failure or when it learns of a failure at another processor.
- Because of the broadcast of ROLLBACK messages, the recovery algorithm is initiated at all processors.



Description of the Algorithm

When a processor restarts after a failure, it broadcasts a ROLLBACK message that it has failed.

The rollback starts at the failed processor and slowly diffuses into the entire system through ROLLBACK messages.

Consensus & Agreement Algorithm.

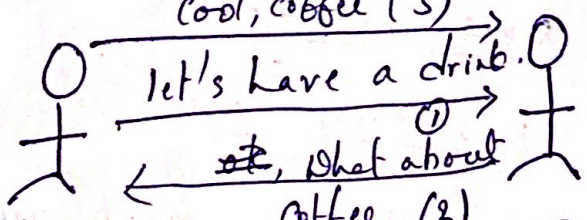
Problem Definition:

The Byzantine General problem is an analogy in Computer Science used to describe the challenge of establishing & maintaining Security on distributed Network. For this all the nodes (majority of nodes) must establish a set of rules & come to an agreement.

Agreement in distributed System among processes is a fundamental ~~problem~~ requirement for a wide range of applications.

Agreement is a common understanding. A classical Example is 'commit' or 'Abort' a transaction must be done with a common Agreement.

For Example, 2 persons wanted to have a drink.



① Alice & Bob wanted to ~~Proposed a~~ have a drink.

② Bod Suggested 'coffee'

③ Alice 'Agreed' to Alice Suggestion.

They both agreed on a 'value' that is Suggested by one of them. This is Consensus.

decide on the ...

In simple words a Consensus is a general agreement on a value.

Hence this is known as Agreement variable or Value. This may be a boolean value or Multivalued and need not be an integer.

The Byzantine & other problems. The Byzantine Agreement problem requires a designated process called "source process" with an initial value to reach an agreement with other processes about its "initial value" with the following conditions.

Agreement: All Non-faulty processes must agree on the same value.

Validity: Validity says that the value proposed by one of the node should be Accepted by all non-faulty nodes. Suggested

Termination: Every non-faulty node will eventually decide some value.

other 2 important problems are

The consensus problem: In this problem, each process has an initial value. And this value is broadcasts to all other processes. This initial value may be different

The following are the conditions.

Agreement: All non-faulty processors agree on the same single value.

validity: If the all the non-faulty processors have the same initial value, it must be agreed by all non-faulty processors.

Termination: Each non-faulty processor must eventually decide on a value.

The iterative consistency problem

In this problem, that each processor has an initial value, and all the correct processes must agree upon a set of values, with one value for each process [20,25].

Agreement: All non-faulty processes must agree on the same array of values $A[v_1, \ldots v_n]$

validity: If process i is non-faulty and its initial value is v_i, then all the non-faulty process agree on v_i, as the i^{th} element of the array A. If process $\perp$ is faulty, then non-faulty processes can agree on any value for $A[j]$

Termination: Each non-faulty process must eventually decide on the array A.

The 3 problems defined are equivalent in the sense. A solution to any of one of them can be used as a solution to other 2 problems.

Problem	Byzantine Agreement	Consensus	Interactive Consistency
Initiation of a value	one processor	All processors	All processors.
Final Agreement	Single value	Single value	A vector of values

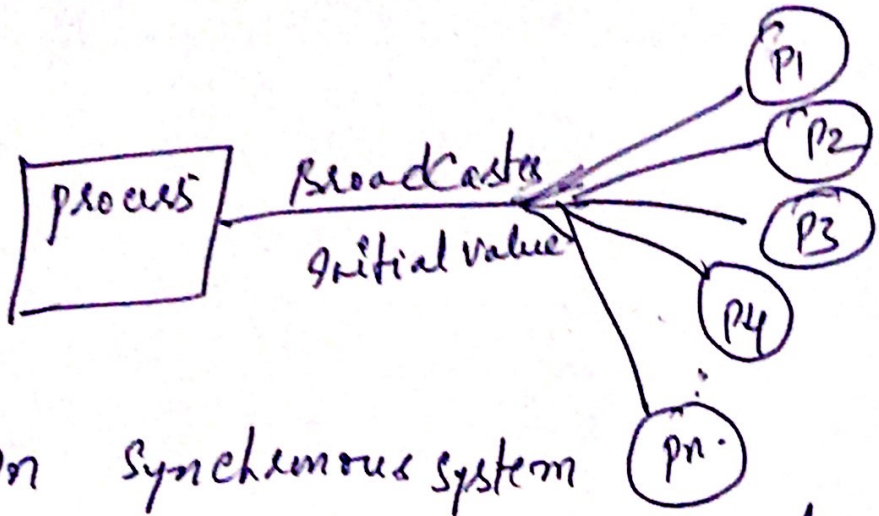
Agreement in a failure free System.

In a failure free system, consensus can be reached by collecting information from the different processes, and performs a decision.

In this each process broadcastes its value to others.

The decision can be reached by using some system functions like 'min', 'max()', majority'

To collect the initial values a token is circulated in logical ring (or) through direct communication.



→ In Synchronous system
* this can be done with simply using constant no. of rounds (Based up on the topology)

→ In another round decision value can be obtained.

→ In an Asynchronous System, Consensus can Similarly be reached in a constant no. of message hops.

From these hops Common decision value is obtained.