AGREEMENT IN A FAILURE-FREE SYSTEM

- In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a decision, and distributing this decision in the system.
- A distributed mechanism would have each process broadcast its values to others, andeach process computes the same function on the values received.
- The decision can be reached by using an application specific function.
- Algorithms to collect the initial values and then distribute the decision may be based on the token circulation on a logical ring, or the three-phase tree-based broadcast converge cast: broadcast, or direct communication with all nodes.
- In a synchronous system, this can be done simply in a constant number of rounds.
- Further, common knowledge of the decision value can be obtained using an additionalround.

- In an asynchronous system, consensus can similarly be reached in a constant numberof message hops.
- Further, concurrent common knowledge of the consensus value can also be attained.

AGREEMENT IN (MESSAGE-PASSING) SYNCHRONOUS SYSTEMS WITH FAILURES

Consensus algorithm for crash failures (synchronous system)

- Consensus algorithm for crash failures message passing synchronous system.
- The consensus algorithm for n processes where up to f processes where f < n may fail in a fail stop failure model.
- Here the consensus variable x is integer value; each process has initial value xi. If

up to f failures are to be tolerated than algorithm has f+1 rounds, in each round a process i sense the value of its variable xi to all other processes if that value has not been sent before.

- So, of all the values received within that round and its own value xi at that start of the round the process takes minimum and updates xi occur f + 1 rounds the local value xi guaranteed to be the consensus value.
- In one process is faulty, among three processes then f = 1. So the agreement requires f + 1 that is equal to two rounds.
- If it is faulty let us say it will send 0 to 1 process and 1 to another process i, j and k. Now, on receiving one on receiving 0 it will broadcast 0 over here and this particular process on receiving 1 it will broadcast 1 over here.
- So, this will complete one round in this one round and this particular process on receiving 1 it will send 1 over here and this on the receiving 0 it will send 0 over here.

(global constants)

integer: f; // maximum number of crash failures tolerated

(local variables)

Integer: x ← local value;

(1) Process P_i ($1 \leq i \leq n$) execute the consensus algorithm for up to f crash failures:

(1a) **for** round from 1 to f + 1 **do**

(1b) **if** the current value of x has not been broadcast **then**

(1c) **broadcast(x);**

(1d) $y_i \leftarrow$ value (if any) received from process j in this round;

(1e) $x \leftarrow \min_{\forall j} (x, y_j)$;

(1f) **output** x as the consensus value.

Fig : Consensus with up to f fail-stop processes in a system of n processes, n > f

- The agreement condition is satisfied because in the f+ 1 rounds, there must be at least one round in which no process failed.
- In this round, say round r, all the processes that have not failed so far succeed in broadcasting their values, and all these processes take the minimum of the values broadcast and received in that round.
- Thus, the local values at the end of the round are the same, say x_r^i for all non-failed processes.
- In further rounds, only this value may be sent by each process at most once, and no process i will update its value x_r^i.
- The validity condition is satisfied because processes do not send fictitious values inthis failure model.
- For all i, if the initial value is identical, then the only value sent by any process is the value that has been agreed upon as per the agreement condition.
- The termination condition is seen to be satisfied.

Complexity

- The complexity of this particular algorithm is it requires f + 1 rounds where f < n and the number of messages is $O(n^2)$ in each round and each message has one integers hence the total number of messages is $O((f+1)\cdot n^2)$ is the total number ofrounds and in each round n^2 messages are required.

Lower bound on the number of rounds

- At least f + 1 rounds are required, where f < n.
- In the worst-case scenario, one process may fail in each round; with f + 1 rounds, there is at least one round in which no process fails. In that guaranteed failure-free round, all messages broadcast can be delivered reliably, and all processes that have not failed can compute the common function of the received values to reach an

agreement value.

Consensus algorithms for Byzantine failures (synchronous system)

Upper bound on Byzantine processes

- In a system of n processes, the Byzantine agreement problem can be solved in a synchronous system only if the number of Byzantine processes f is such that

$$f \leq \left\lceil \frac{n-1}{3} \right\rceil$$

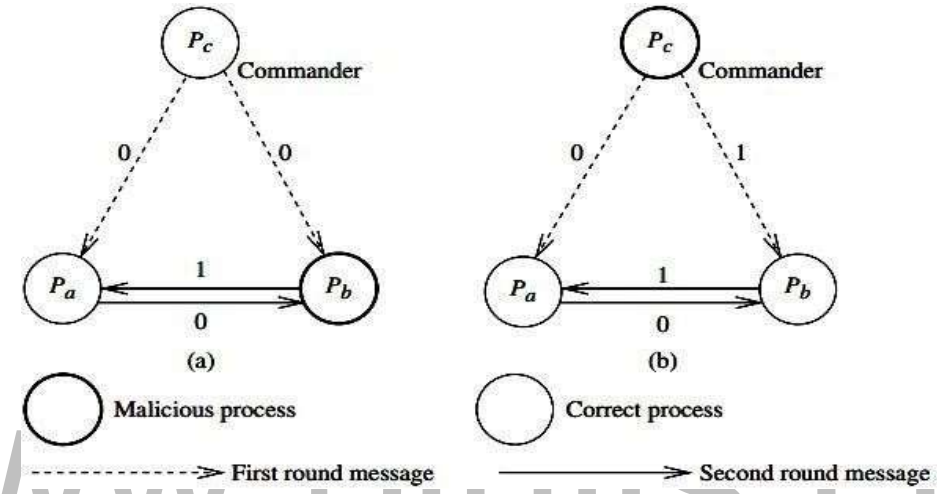


Fig: Impossibility of achieving Byzantine agreement with n = 3 processes and f = 1 malicious process

- The condition where f < (n – 1) / 2 is violated over here; that means, if f = 1 andn = 2 this particular assumption is violated
- (n – 1) / 2 is not 1 in that case, but we are assuming 1 so obviously, as per the previous condition agreement byzantine agreement is not possible.
- Here P 0 is faulty is non faulty and here P 0 is faulty so that means P 0 is the source, the source is faulty here in this case and source is non faulty in the other case.
- So, source is non faulty, but some other process is faulty let us say that P 2 is faulty. P 1 will send because it is non faulty same values to P 1 and P 2 and as far as the P 2s concerned it will send a different value because it is a faulty.

- Agreement is possible when f = 1 and the total number of processor is 4. So, agreement we can see how it is possible we can see about the commander P c.
- So, this is the source it will send the message 0 since it is faulty. It will send 0 to Pd 0 to P b, but 1 to pa in the first column. So, P a after receiving this one it will send one to both the neighbors, similarly P b after receiving 0 it will send 0 since itis not faulty.
- Similarity P d will send after receiving 0 at both the ends.
- If we take these values which will be received here it is 1 and basically it is 0 and this is also 0.
- So, the majority is basically 0 here in this case here also if you see the values 10 and 0. The majority is 0 and here also majority is 0.
- In this particular case even if the source is faulty, it will reach to an agreement, reach an agreement and that value will be agreed upon value or agreement variable will be equal to 0.

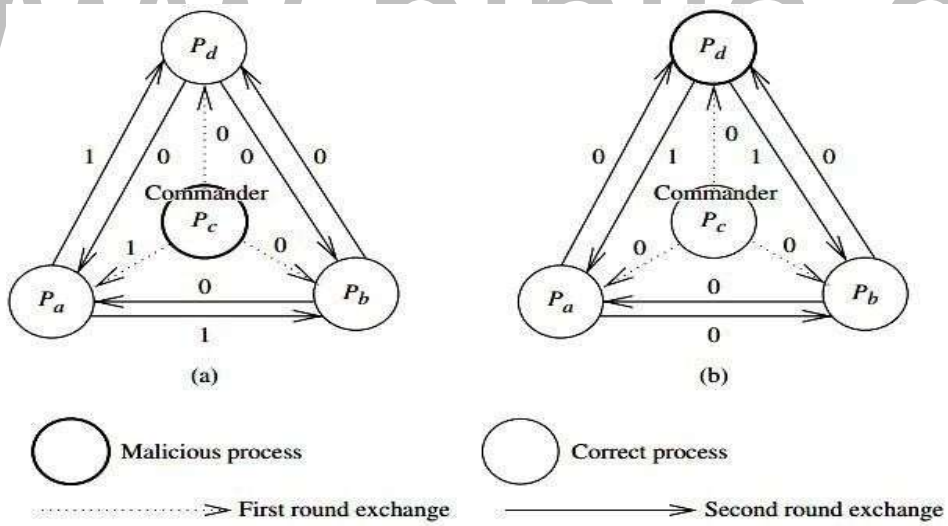


Fig : Achieving Byzantine agreement when n = 4 processes and f = 1

ALGORITHM FOR ASYNCHRONOUS CHECKPOINTING AND RECOVERY

(JUANG-VENKATESAN)

- This algorithm helps in recovery in asynchronous checkpointing.
- The following are the assumptions made:
  - communication channels are reliable
  - delivery messages in FIFO order
  - infinite buffers
  - message transmission delay is arbitrary but finite
- The underlying computation or application is event-driven: When process P is at states, receives message m, it processes the message; moves to state s' and send messages out. So the triplet (s, m, msgs_sent) represents the state of P.
- To facilitate recovery after a process failure and restore the system to a consistent state, two types of log storage are maintained:
  - **Volatile log:** It takes short time to access but lost if processor crash. Thecontents of volatile log are moved to stable log periodically.
  - **Stable log:** longer time to access but remained if crashed.

Asynchronous checkpointing

- After executing an event, a processor records a triplet (s, m, msg_sent) in its volatile storage.
  - s:state of the processor before the event
  - m: message
  - msgs_sent: 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.

Recovery Algorithm

The data structures followed in the algorithm are:

$RCVD_{i\rightarrow j}(CkPt_i)$ This represents the number of messages received by processor pi from processor pj,from the beginning of the computation until the checkpoint $CkPt_i$.

$$SENT_{i\rightarrow j}(CkPt_i)$$

This represents the number of messages sent by processor pi to processor pj, from the beginning of the computation until the checkpoint $CkPt_i$.

- The main idea of the algorithm is to find a set of consistent checkpoints, from theset 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 outif 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 thatit 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.

Procedure RollBack_Recovery: processor pi executes the following:STEP (a)

if processor pi is recovering after a failure **then**

$C_k Pt_i$:= latest event logged in the stable storage

else

$C_k Pt_i$:= latest event that look place in p_i {The latest event at pi can be either instable or in volatile storage}

end if

STEP(b)

for k=1 to N {N is the number of processors in the system} **do**

for each neighboring processor p_j **do**

compute $SENT_{i\rightarrow j}(C_k Pt_i)$

send a ROLLBACK(i, $SENT_{i\rightarrow j}(C_k Pt_i)$) message to p_j

end for

for every ROLLBACK(j,c) message received from a neighbor j **do**

if RCVD $_{i\rightarrow j}$ (C_k Pt_i) > c {Implies the presence of orphan message}

then

find the latest event e such that RCVD $_{i\rightarrow j}$ (e) = c {Such an event e may be in the volatile storage or stable storage}

C_k Pt_i := e

end if

end for

end for {for k}

Fig : Algorithm for Asynchronous Check pointing and Recovery (Juang- Venkatesan)

- The rollback starts at the failed processor and slowly diffuses into the entire systemthrough ROLLBACK messages.
- During the kth iteration (k != 1), a processor pi does the following:

(i) based on the state $CkPt_i$ it was rolled back in the (k − 1)th iteration, it computes $SENT_i \rightarrow j$ ($CkPt_i$) for each neighbor pj and sends this value in a ROLLBACK message to that neighbor

(ii) p_i waits for and processes ROLLBACK messages that it receives from its neighbors in kth iteration and determines a new recovery point $CkPt_i$ for p_i based on information in these messages.

Fig : Asynchronous Checkpointing And Recovery

At the end of each iteration, at least one processor will rollback to its final recovery point, unless the current recovery points are already consistent.

CONSENSUS AND AGREEMENT

A consensus algorithm is a process that achieves agreement on a single data value among distributed processes or systems.

- Consensus algorithms necessarily assume that some processes and systems will be unavailable and that some communications will be lost.
- Hence these algorithms must be fault-tolerant.

Examples of consensus algorithm:

- Deciding whether to commit a distributed transaction to a database.
- Designating node as a leader for some distributed task.
- Synchronizing state machine replicas and ensuring consistency among them.

Assumptions in Consensus algorithms

- **Failure models:**

– Some of the processes may be faulty in distributed systems.

– A faulty process can behave in any manner allowed by the failure model assumed.

– Some of the well known failure models includes fail-stop, send omission and receive omission, and Byzantine failures.

– **Fail stop model:** a process may crash in the middle of a step, which could be the execution of a local operation or processing of a message for a send or receive event. it may send a message to only a subset of the destination set before crashing.

– **Byzantine failure model:** a process may behave arbitrarily.

– The choice of the failure model determines the feasibility and complexity of solving consensus.

- **Synchronous/asynchronous communication:**

– If a failure-prone process chooses to send a message to process but fails, then intended process cannot detect the non-arrival of the message.

– This is because scenario is indistinguishable from the scenario in which the message

– takes a very long time in transit. This is a major hurdle in asynchronous system.

– In a synchronous system, a unsent message scenario can be identified by the intended recipient, at the end of the round.

– The intended recipient can deal with the non-arrival of the expected message by assuming the arrival of a message containing some default data, and then proceeding with the next round of the algorithm.

- **Network connectivity:**

– The system has full logical connectivity, i.e., each process can communicate with any other by direct message passing.

- **Sender identification:**

– A process that receives a message always knows the identity of the sender process.

– When multiple messages are expected from the same sender in a single round, a scheduling algorithm is employed that sends these messages in sub-rounds, so that each message sent within the round can be uniquely identified.

- **Channel reliability:**

– The channels are reliable, and only the processes may fail.

- Authenticated vs. non-authenticated messages:

– With unauthenticated messages, when a faulty process relays a message to other processes

(i) it can forge the message and claim that it was received from another process,

(ii) it can also tamper with the contents of a received message before relaying it.

– When a process receives a message, it has no way to verify its authenticity. This is known as **un authenticated message or oral message or an unsigned message.**

– Using authentication via techniques such as digital signatures, it is easier to solve the

agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering.

- Thus, faulty processes can inflict less damage.

- **Agreement variable:**

- The agreement variable may be boolean or multivalued, and need not be aninteger.

- This simplifying assumption does not affect the results for other data types, but helps in the abstraction while presenting the algorithms.

Byzantine General problem

- The Byzantine Generals' Problem (BGP) is a classic problem faced by any distributed computer system network.
- Imagine that the grand Eastern Roman empire aka Byzantine empire has decided to capture a city.
- There is fierce resistance from within the city.
- The Byzantine army has completely encircled the city.
- The army has many divisions and each division has a general.
- The generals communicate between each as well as between all lieutenants within their division only through messengers.
- All the generals or commanders have to agree upon one of the two plans of action.
- Exact time to attack all at once or if faced by fierce resistance then the time toretreat all at once. The army cannot hold on forever.
- If the attack or retreat is without full strength then it means only one thing—Unacceptable brutal defeat.
- If all generals and/or messengers were trustworthy then it is a very simple solution.
- However, some of the messengers and even a few generals/commanders are

traitors. They are spies or even enemy soldiers.

- There is a very high chance that they will not follow orders or pass on the incorrect message. The level of trust in the army is very less.
- Consider just a case of 1 commander and 2 Lieutenants and just 2 types of messages- 'Attack' and 'Retreat'.

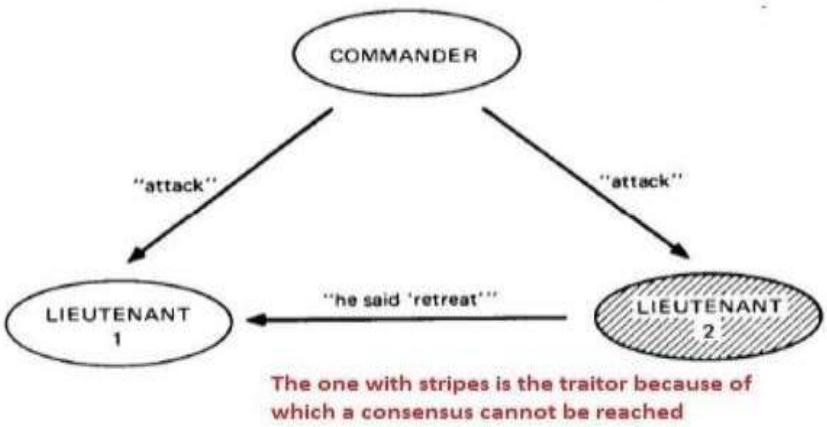


Fig : BGP algorithm

- In Fig, the Lieutenant 2 is a traitor who purposely changes the message that is to be passed to Lieutenant 1.
- Now Lieutenant 1 has received 2 messages and does not know which one to follow. Assuming Lieutenant 1 follows the Commander because of strict hierarchyin the army.
- Still, 1/3rd of the army is weaker by force as Lieutenant 2 is a traitor and this creates a lot of confusion.
- However what if the Commander is a traitor (as explained in Fig). Now 2/3rdof the total

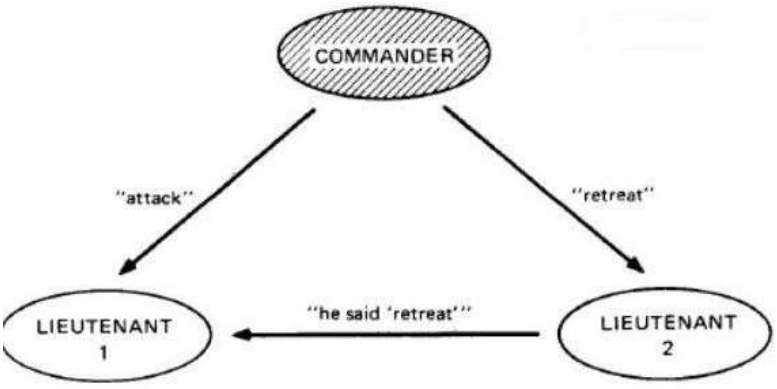


Fig: BGP algorithm

army has followed the incorrect order and failure is certain. After adding 1 more Lieutenant and 1 more type of message (Let's say the 3rd message is 'Not sure'), the complexity of finding a consensus between all the Lieutenants and the Commander is increased.

- Now imagine the exponential increase when there are hundreds of Lieutenants.

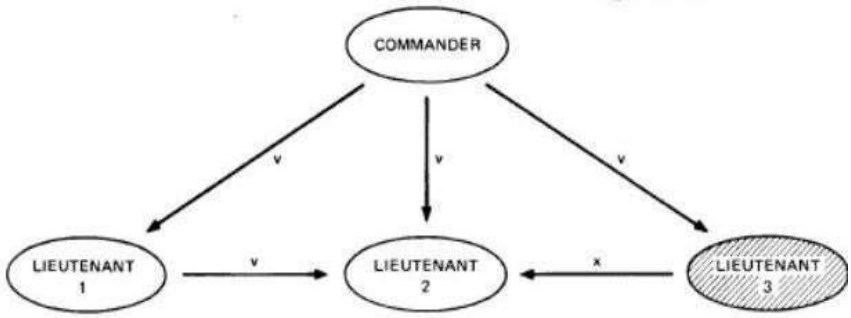


Fig : Adding one more lieutenant

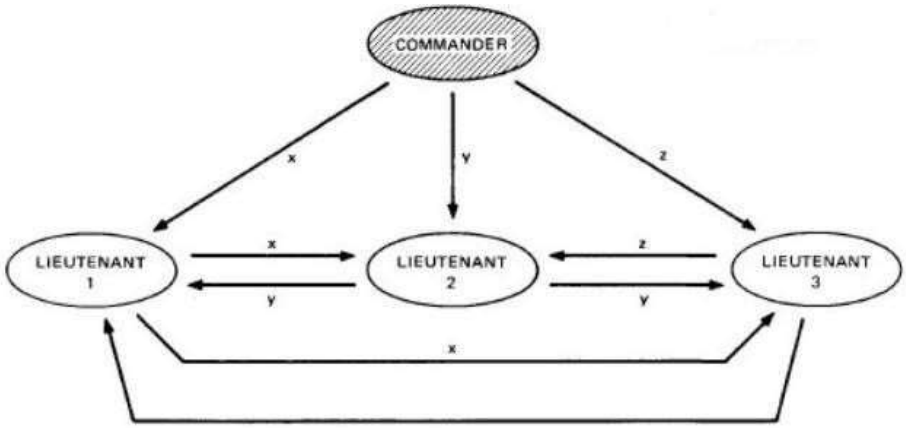


Fig : 1 commandant, 3 lieutenant and 3 types of messages

- This is BGP. It is applicable to every distributed network. All participants or nodes ('Lieutenant') are exactly of equal hierarchy. If agreement is reachable, then protocols to reach it need to be devised.

- All participating nodes have to agree upon every message that is transmitted between the nodes.
- If a group of nodes is corrupt or the message that they transmit is corrupt then still the network as a whole should not be affected by it and should resist this 'Attack'.

- The network in its entirety has to agree upon every message transmitted in the network. This agreement is called as **consensus.**

The Byzantine agreement problem requires a designated source process, with an initial value, to reach agreement with the other processes about its initial value, subject to:

- **Agreement:** All non-faulty processes must agree on the same value.
- **Validity:** If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the

There are two other versions of the Byzantine agreement problem:

- Consensus problem
- Interactive consistency problem.
- A correct process is a process that does not exhibit a Byzantine behaviour.
- A process is Byzantine if, during its execution, one of the following faults occurs:

– **Crash:** The process stops executing statements of its program and halts.

– **Corruption**: The process changes arbitrarily the value of a local variable with respect to its program specification. This fault could be propagated toother processes by including incorrect values in the content of a message sent by the process.

– **Omission:** The process omits to execute a statement of its program. If a process omits to execute an assignment, this could lead to a corruption fault.

– **Duplication:** The process executes more than one time a statement of its program. If a process executes an assignment more than one time, this could lead to a corruption fault.

– **Misevaluation:** The process misevaluates an expression included in its program. This fault is different from a corruption fault: misevaluating an expression does not imply the update of the variables involved in the expression and, in some cases the result of an evaluation is not as-signed toa variable.

Consensus Problem

> ***All the process has an initial value and all the correct processes must agree on single value. This is consensus problem.***

Consensus is a fundamental paradigm for fault-tolerant asynchronous distributed systems. Each process proposes a value to the others. All correct processes have to agree (Termination) on the same value (Agreement) which must be one of the initially proposed values (Validity).

The requirements of the consensus problem are:

- **Agreement:** All non-faulty processes must agree on the same (single) value.
- **Validity:** If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
- **Termination:** Each non-faulty process must eventually decide on a value.

Interactive Consistency Problem

> ***All the process has an initial value, and all the correct processes must agree upon a set of values, with one value for each process. This is interactive consistency problem.***

The formal specifications are:

- **Agreement:** All non-faulty processes must agree on the same array of values A [v1, …,vn].
- **Validity:** If process i is non-faulty and its initial value is vi, then all non faulty processes agree on vi as the ith element of the array A. If process jis faulty, then the non-faulty processes can agree on any value for A[j].
- **Termination:** Each non-faulty process must eventually decide on the array A.

The difference between the agreement problem and the consensus problem is that, in the agreement problem, a single process has the initial value, whereas in the consensus problem, all processes have an initial value.

RESULTS OF CONSENSUS PROBLEM

Some important facts to remember are:

- Consensus is not solvable in asynchronous systems even if one process can fail by crashing. Consensus is attainable for no failure case.
- In a synchronous system, common knowledge of the consensus value is also attainable.In asynchronous case, concurrent common knowledge of the consensus value is attainable.

The results are tabulated below. f indicates the number of processes that can failand n indicates the total number of processes.

S.No	Failure Mode	Synchronous System	Asynchronous System
1.	No failure	Agreement is attainable. Common knowledge is also attainable.	Agreement is attainable. Concurrent common knowledge is also attainable.
2.	Crash failure	Agreement is attainable. f < n process Ω(f+1) rounds	Agreement is not attainable.
3.	Byzantine (malicious) failure	Agreement is attainable. f<= floor((n-1)/3) Byzantine process Ω(f+1) rounds	Agreement is not attainable.

Solvable variants of agreement problem

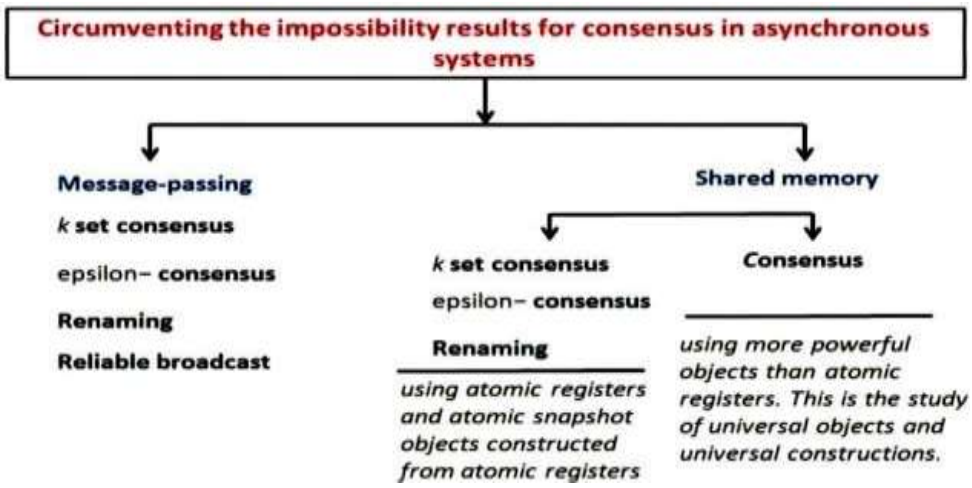


Fig : Circumventing the impossibility results

- A synchronous message passing system and a shared memory system can be used solve the consensus problem. The following are the weaker consensus problem in asynchronous system:
- **Terminating reliable broadcast:** A correct process will always get a message even if the sender crashes while sending. If the sender crashes while sending the message, the message may be even null, but still it has to be delivered to the correct process.
- **K-set consensus:** It is solvable as long as the number of crashes is less than the parameter k, which indicates the non-faulty processes that agree on different values, as long as the size of the set of values agreed upon is bounded by k.
- **Approximate agreement:** The consensus value is from multi valued domain. The agreed upon values by the non-faulty processes be within of each other.
- **Renaming problem:** It requires the processes to agree on necessarily distinct values.
- **Reliable broadcast**: A weaker version of reliable terminating broadcast (RTB), isthe one in which the terminating condition is dropped and is solvable under crash failures.

Solvable variants	Failure model and overhead	Definition
Reliable broadcast	Crash failures, $n > f$ (MP)	Validity, agreement, integrity conditions
k-set consensus	Crash failures, $f < k < n$ (MP and SM)	Size of the set of values agreed upon must be at most k
ϵ-agreement	Crash failures, $n \geq 5f + 1$ (MP)	Values agreed upon are within ϵ of each other
Renaming	Up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash failures, $f \leq n - 1$ (SM)	Select a unique name from a set of names

Fig : Solvable variants of agreement problem in asynchronous system

COORDINATED CHECKPOINTING ALGORITHM (KOO-TOUEG)

- Koo and Toueg coordinated check pointing and recovery technique takes a consistent set of checkpoints and avoids the domino effect and live lock problems during the recovery.
- Includes 2 parts: the check pointing algorithm and the recovery algorithm

A. The Check pointing Algorithm

The checkpoint algorithm makes the following assumptions about the distributed system:

- Processes communicate by exchanging messages through communication channels.
- Communication channels are FIFO.
- Assume that end-to-end protocols (the sliding window protocol) exist to handle with message loss due to rollback recovery and communication failure.
- Communication failures do not divide the network.
- The checkpoint algorithm takes two kinds of checkpoints on the stable storage: 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 that is made a permanent checkpoint on the successful termination of the checkpoint algorithm.

The algorithm consists of two phases.

First Phase

1. An initiating process Pi takes a tentative checkpoint and requests all other processes to take tentative checkpoints. Each process informs Pi whether it succeeded in taking a tentative checkpoint.
2. A process says “no” to a request if it fails to take a tentative checkpoint
3. 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 thrown-away.

Second Phase

1. Pi informs all the processes of the decision it reached at the end of the first phase.
2. A process, on receiving the message from Pi will act accordingly.

3. Either all or none of the processes advance the checkpoint by taking permanent checkpoints.
4. The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the basic computation until it is informed of Pi's decision.

Correctness: for two reasons

i. Either all or none of the processes take permanent checkpoint
ii. No process sends message after taking permanent checkpoint

An Optimization

The above protocol may cause a process to take a checkpoint even when it is not necessary for consistency. Since taking a checkpoint is an expensive operation, we avoid taking checkpoints.

B. The Rollback Recovery Algorithm

The rollback recovery algorithm restores the system state to a consistent state after a failure. The rollback recovery algorithm assumes that a single process invokes the algorithm. It assumes that the checkpoint and the rollback recovery algorithms are not invoked concurrently. The rollback recovery algorithm has two phases.

First Phase

1. An initiating process Pi sends a message to all other processes to check if they all are willing to restart from their previous checkpoints.
2. A process may reply "no" to a restart request due to any reason (e.g., it is already participating in a check pointing or a recovery process initiated by some other process).
3. If Pi learns that all processes are willing to restart from their previous checkpoints, Pi decides that all processes should roll back to their previous checkpoints. Otherwise,
4. Pi aborts the roll back attempt and it may attempt a recovery at a later time.

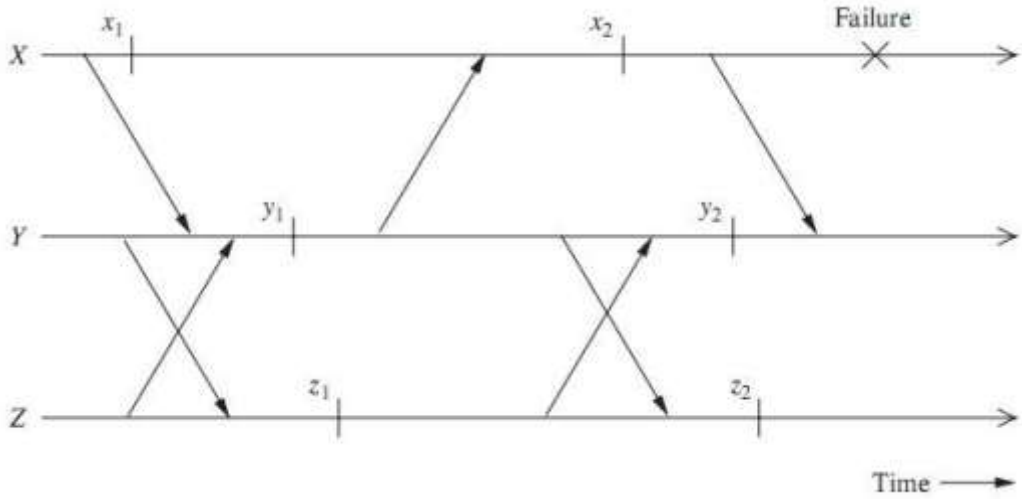
Second Phase

1. Pi propagates its decision to all the processes.
2. On receiving Pi's decision, a process acts accordingly.
3. During the execution of the recovery algorithm, a process cannot send messages related to the underlying computation while it is waiting for Pi's decision.

Correctness: Resume from a consistent state

Optimization: May not to recover all, since some of the processes did not change anything

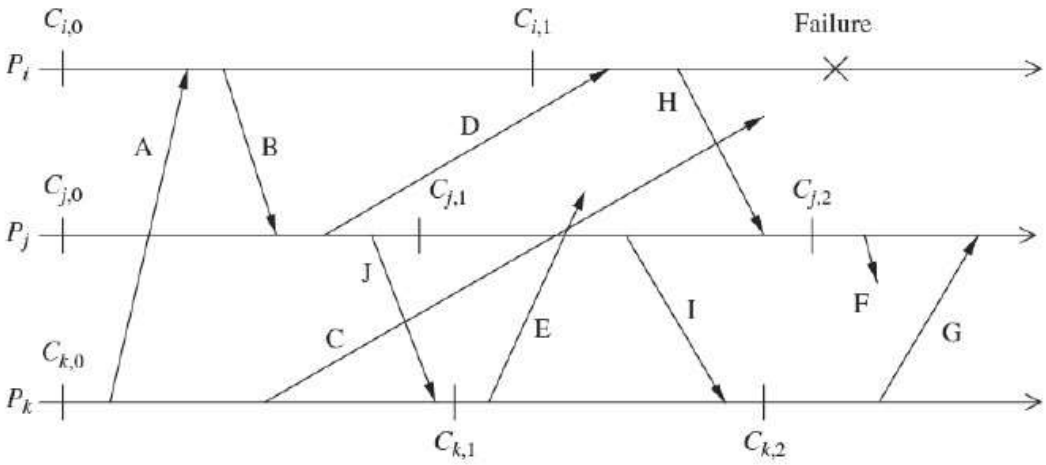
Optimization: May not to recover all, since some of the processes did not change anything



The above protocol, in the event of failure of process X, the above protocol will require processes X, Y, and Z to restart from checkpoints x2, y2, and z2, respectively. Process Z need not roll back because there has been no interaction between process Z and the other two processes since the last checkpoint at Z.

ISSUES IN FAILURE RECOVERY

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery



The computation comprises of three processes Pi, Pj , and Pk, connected through a communication network. The processes communicate solely by exchanging messages over fault-free, FIFO communication channels.

Processes Pi, Pj , and Pk have taken checkpoints

- Checkpoints : $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}, C_{k,2}\}$
- Messages : A - J
- The restored global consistent state : $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

- The rollback of process Pi to checkpoint Ci,1 created an orphan message H
- Orphan message I is created due to the roll back of process Pj to checkpoint Cj,1
- Messages C, D, E, and F are potentially problematic
  - Message C: a delayed message
  - Message D: a lost message since the send event for D is recorded in the restored state for Pj, but the receive event has been undone at process Pi.
  - Lost messages can be handled by having processes keep a message log of all the sent messages
  - Messages E, F: delayed orphan messages. After resuming execution from their checkpoints, processes will generate both of these messages

CHECKPOINT-BASED RECOVERY

Checkpoint-based rollback-recovery techniques can be classified into three categories:

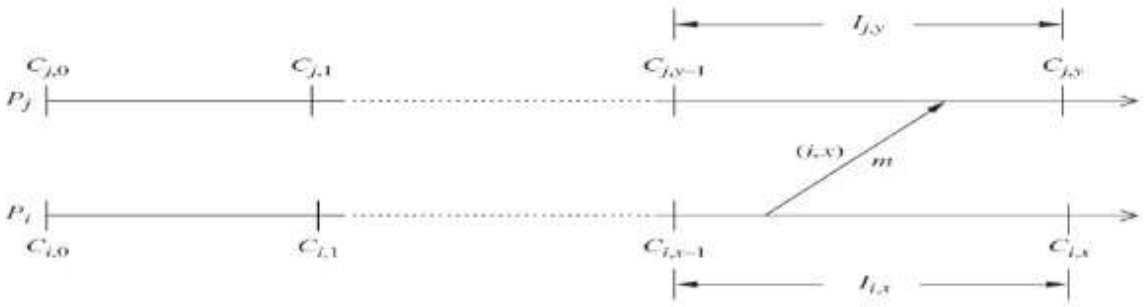
1. Uncoordinated check pointing
2. Coordinated check pointing
3. Communication-induced check pointing

1. Uncoordinated Check pointing

- Each process has autonomy in deciding when to take checkpoint
- Advantage: The lower runtime overhead during normal execution
- Disadvantages
    1. Domino effect during a recovery
    2. Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints
    3. Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm
    4. Not suitable for application with frequent output commits
- The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation
- The following direct dependency tracking technique is commonly used in uncoordinated check pointing.

Direct dependency tracking technique

- Assume each process Pi starts its execution with an initial checkpoint $Ci,0$
- Ii,x : checkpoint interval, interval between $Ci,x-1$ and Ci,x
- When Pj receives a message m during Ij,y , it records the dependency from Ii,x to Ij,y, which is later saved onto stable storage when Pj takes Cj,y



- When a failure occurs, the recovering process initiates rollback by broadcasting a dependency *request* message to collect all the dependency information maintained by each process.
- When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state.
- The initiator then calculates the recovery line based on the global dependency information and broadcasts a rollback request message containing the recovery line.
- Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

2. Coordinated Checkpointing

In coordinated check pointing, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state

Types

1. Blocking Checkpointing: After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete
   Disadvantages: The computation is blocked during the checkpointing
2. Non-blocking Checkpointing: The processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

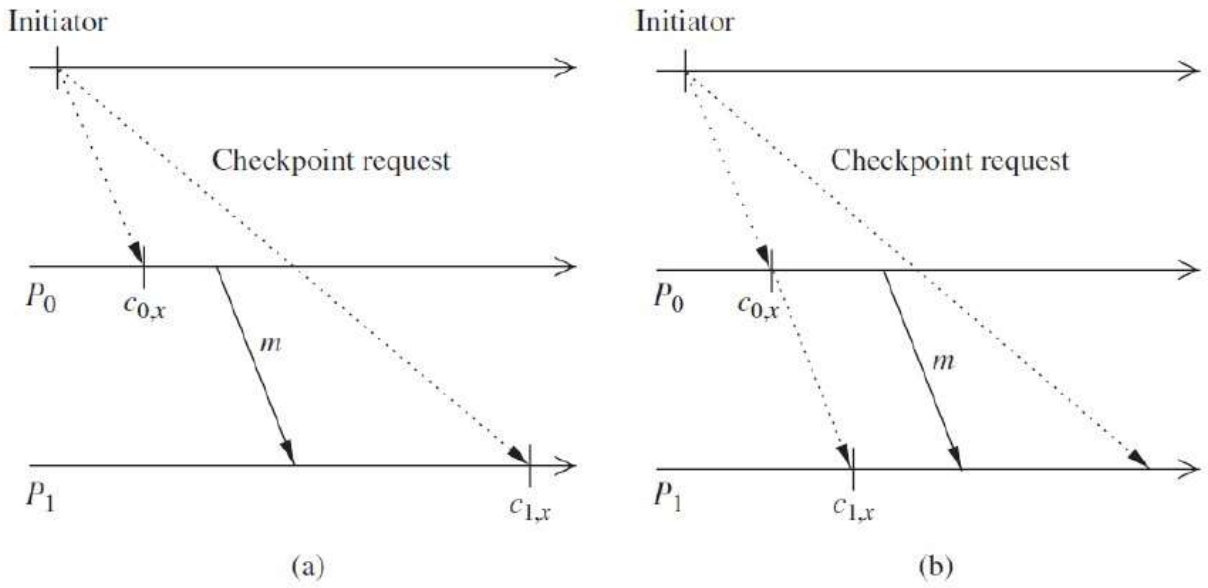
Example (a) : Checkpoint inconsistency

- Message m is sent by $P0$ after receiving a checkpoint request from the checkpoint coordinator
- Assume m reaches $P1$ before the checkpoint request
- This situation results in an inconsistent checkpoint since checkpoint $C1,x$ shows the receipt of message m from $P0$, while checkpoint $C0,x$ does not show m being sent from $P0$

Example (b) : A solution with FIFO channels

- If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message

Coordinated Checkpointing



Impossibility of min-process non-blocking checkpointing

- A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation.

Algorithm

- The algorithm consists of two phases. During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request.
- Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified.
- During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes.

- In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

3. Communication-induced Checkpointing

*Communication-induced checkpointing*is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently. Processes may be forced to take additional checkpoints

Two types of checkpoints

1. Autonomous checkpoints
2. Forced checkpoints

The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints.

- Communication-induced checkpointing piggybacks protocol- related information on each application message
- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line
- The forced checkpoint must be taken before the application may process the contents of the message
- In contrast with coordinated checkpointing, no special coordination messages are exchanged

Two types of communication-induced checkpointing

1. Model-based checkpointing
2. Index-based checkpointing.

Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints.
- No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggybacked on application messages
- There are several domino-effect-free checkpoint and communication model.

- The MRS (mark, send, and receive) model of Russell avoids the domino effect by ensuring that within every checkpoint interval all message receiving events precede all message-sending events.

Index-based checkpointing.

- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.

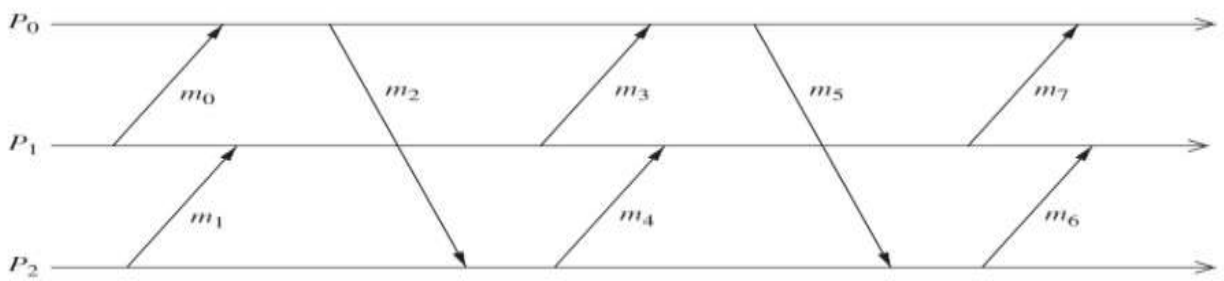
LOG-BASED ROLLBACK RECOVERY

A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.

Deterministic and non-deterministic events

- Log-based rollback recovery exploits the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a non-deterministic event.
- A non-deterministic event can be the receipt of a message from another process or an event internal to the process.
- Note that a message send event is *not* a non-deterministic event.
- For example, in Figure, the execution of process *P*0 is a sequence of four deterministic intervals. The first one starts with the creation of the process, while the remaining three start with the receipt of messages *m*0, *m*3, and*m*7, respectively.
- Send event of message *m*2 is uniquely determined by the initial state of *P*0 and by the receipt of message *m*0, and is therefore not a non-deterministic event.
- Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage.
- Determinant: the information need to "replay" the occurrence of a non-deterministic event (e.g., message reception).
- During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage. Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery.

Log-based Rollback Recovery



The no-orphans consistency condition

Let *e* be a non-deterministic event that occurs at process *p*.

We define the following:

• ***Depend*(*e*):** the set of processes that are affected by a non-deterministic event *e.*

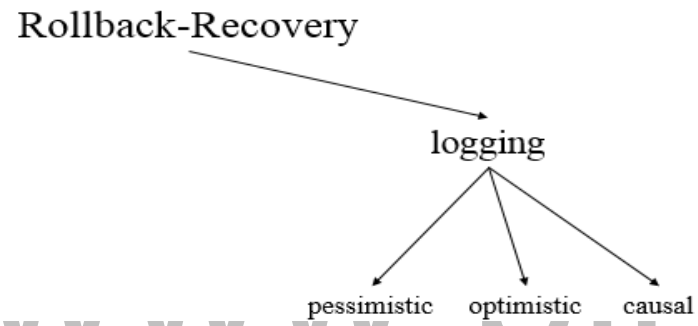
• ***Log*(*e*):** the set of processes that have logged a copy of *e*'s determinant in their volatile memory.

• ***Stable*(*e*):** a predicate that is true if *e*'s determinant is logged on the stable storage.

always-no-orphans condition

– $\forall(e) : \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$

Types



1. Pessimistic Logging

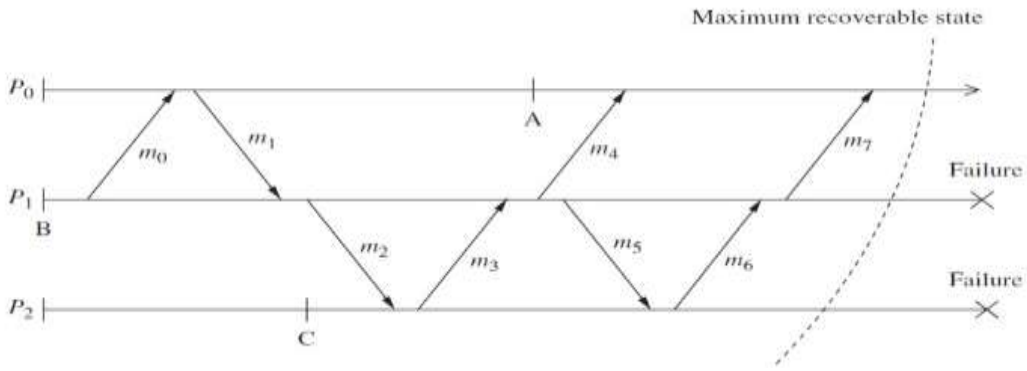
- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation. However, in reality failures are rare
- Pessimistic protocols implement the following property, often referred to as *synchronous logging*, which is a stronger than the always-no-orphans condition
- *Synchronous logging*

  – $\forall e: \neg Stable(e) \Rightarrow |Depend(e)| = 0$
- Thai is, if an event has not been logged on the stable storage, then no process can depend on it.

Example:

Suppose processes *P*1 and *P*2 fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to deliver again the same sequence of messages as in the pre-failure execution

- Once the recovery is complete, both processes will be consistent with the state of $P0$ that includes the receipt of message $m7$ from $P1$



- **Disadvantage**: performance penalty for synchronous logging
- **Advantages:**
  - immediate output commit
  - restart from most recent checkpoint
  - recovery limited to failed process(es)
  - simple garbage collection
- Some pessimistic logging systems reduce the overhead of synchronous logging without relying on hardware. For example, the *sender-based message logging* (SBML) protocol keeps the determinants corresponding to the delivery of each message *m* in the volatile memory of its sender.
- The ***sender-based message logging*** **(SBML) protocol**

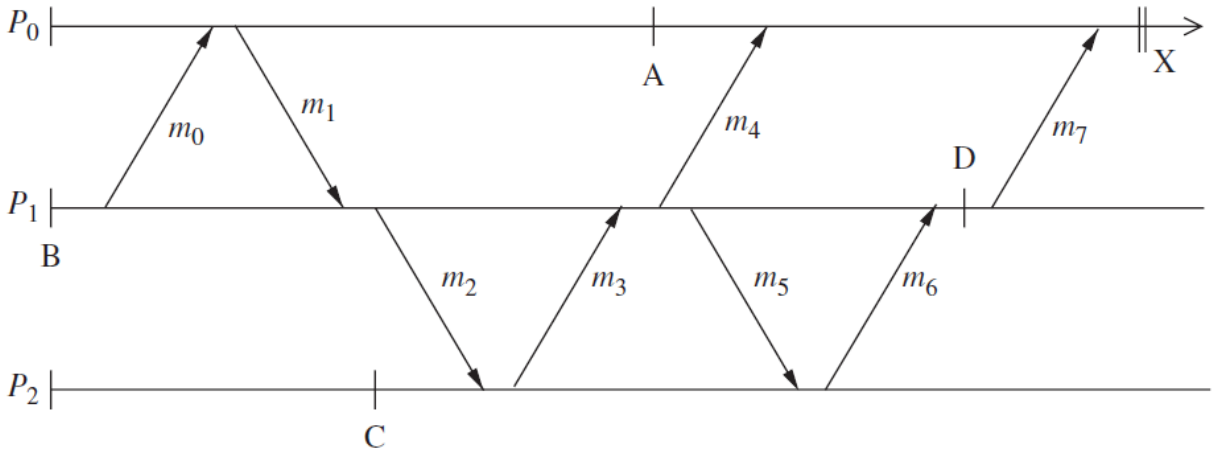
  Two steps.

  1. First, before sending m, the sender logs its content in volatile memory.
  2. Then, when the receiver of m responds with an acknowledgment that includes the order in which the message was delivered, the sender adds to the determinant the ordering information.

2. Optimistic Logging

- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a failure occurs
- Do not implement the *always-no-orphans* condition

- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution
- Optimistic logging protocols require a non-trivial garbage collection scheme
- Pessimistic protocols need only keep the most recent checkpoint of each process, whereas optimistic protocols may need to keep multiple checkpoints for each process

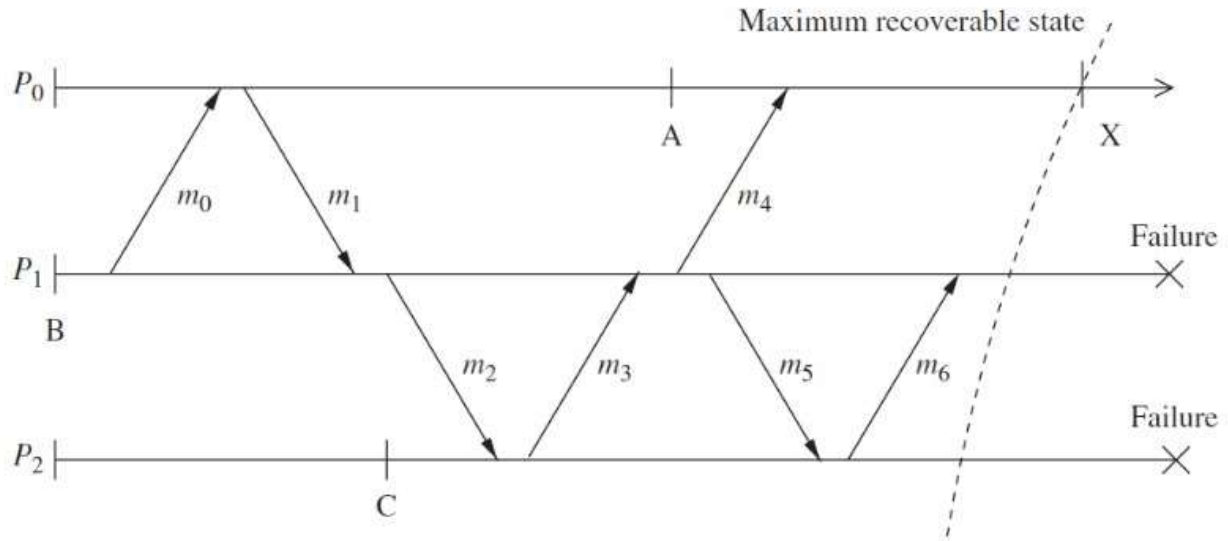


- Consider the example shown in Figure. Suppose process $P2$ fails before the determinant for $m5$ is logged to the stable storage. Process $P1$ then becomes an orphan process and must roll back to undo the effects of receiving the orphan message $m6$. The rollback of $P1$ further forces $P0$ to roll back to undo the effects of receiving message $m7$.
- **Advantage:** better performance in failure-free execution
- **Disadvantages:**
  - coordination required on output commit
  - more complex garbage collection
- Since determinants are logged asynchronously, output commit in optimistic logging 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 $m4$ and $m7$ to the stable storage and ask $P2$ to log $m2$ and $m5$. In this case, if any process fails, the computation can be reconstructed up to state X.

3. Causal Logging

- Combines the advantages of both pessimistic and optimistic logging at the expense of a more complex recovery protocol

- Like optimistic logging, 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 and never creates orphans, thus isolating processes from the effects of failures at other processes
- Make sure that the always-no-orphans property holds
- Each process maintains information about all the events that have causally affected its state



- Consider the example in Figure Messages *m*5 and *m*6 are likely to be lost on the failures of P1 and P2 at the indicated instants. Process
- *P*0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's *happened-before* relation.
- These events consist of the delivery of messages *m*0, *m*1, *m*2, *m*3, and *m*4.
- The determinant of each of these non-deterministic events is either logged on the stable storage or is available in the volatile log of process *P*0.
- The determinant of each of these events contains the order in which its original receiver delivered the corresponding message.
- The message sender, as in sender-based message logging, logs the message content. Thus, process *P*0 will be able to "guide" the recovery of *P*1 and *P*2 since it knows the order in which *P*1 should replay messages *m*1 and *m*3 to reach the state from which *P*1 sent message *m*4.

- Similarly, $P0$ has the order in which $P2$should replay message $m2$ to be consistent with both $P0$ and $P1$.
- The content of these messages is obtained from the sender log of $P0$ or regenerated deterministically during the recovery of $P1$ and $P2$.
- Note that information about messages $m5$ and $m6$ is lost due to failures. These messages may be resent after recovery possibly in a different order.
- However, since they did not causally affect the surviving process or the outside world, the resulting state is consistent.
- Each process maintains information about all the events that have causally affected its state.

RECOVERY & CONSENSUS

CHECK POINTING AND ROLLBACK RECOVERY: INTRODUCTION

- Rollback recovery protocols restore the system back to a consistent state after a failure,
- It achieves fault tolerance by periodically saving the state of a process during the failure-free execution
- It treats a distributed system application as a collection of processes that communicate over a network

Checkpoints

The saved state is called a checkpoint, and the procedure of restarting from a previously check pointed state is called rollback recovery. A checkpoint can be saved on either the stable storage or the volatile storage

Why is rollback recovery of distributed systems complicated?

Messages induce inter-process dependencies during failure-free operation

Rollback propagation

The dependencies among messages may force some of the processes that did not fail to roll back.This phenomenon of cascaded rollback is called the domino effect.

Uncoordinated check pointing

If each process takes its checkpoints independently, then the system cannot avoid the domino effect – this scheme is called independent or uncoordinated checkpointing

Techniques that avoid domino effect

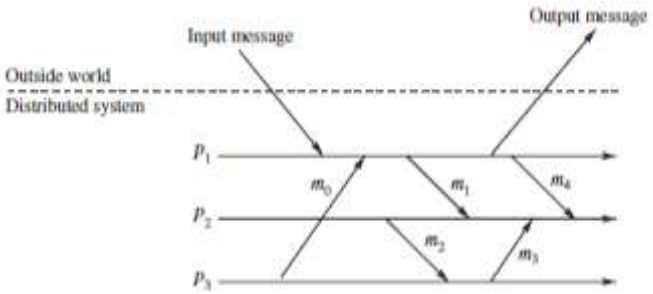
1. Coordinated checkpointing rollback recovery - Processes coordinate their checkpoints to form a system-wide consistent state
2. Communication-induced checkpointing rollback recovery - Forces each process to take checkpoints based on information piggybacked on the application.
3. Log-based rollback recovery - Combines checkpointing with logging of non-deterministic events • relies on piecewise deterministic (PWD) assumption.

BACKGROUND AND DEFINITIONS

System model

- A distributed system consists of a fixed number of processes, P1, P2,…_ PN , which communicate only through messages.

- Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively.
- Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication.
- Some protocols assume that the communication uses first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can lose, duplicate, or reorder messages.
- Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.



An example of a distributed system with three processes.

A local checkpoint

- 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
- Assumption
  - A process stores all local checkpoints on the stable storage
  - A process is able to roll back to any of its existing local checkpoints

- Ci,k – The kth local checkpoint at process Pi
- $Ci,0$ – A process Pi takes a checkpoint $Ci,0$ before it starts execution

Consistent states

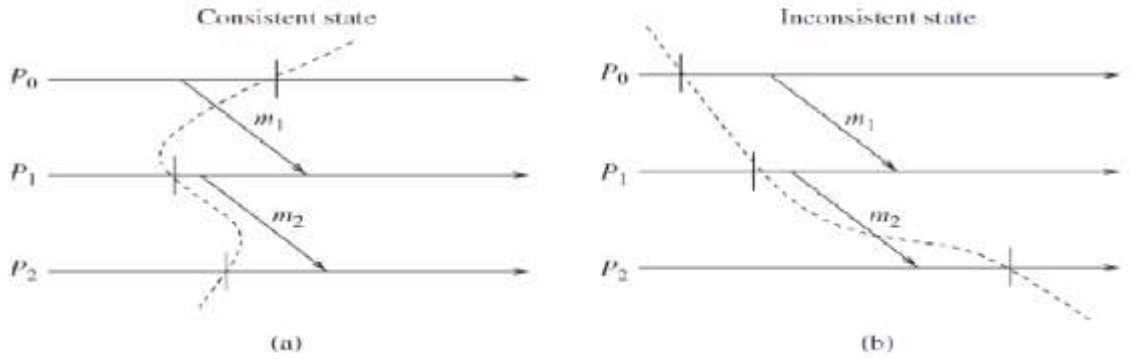
- A global state of a distributed system is a collection of the individual states of all participating processes and the states of the communication channels
- Consistent global state

– a global state that may occur during a failure-free execution of distribution of distributed computation

– if a process‟s state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message

- A global checkpoint is a set of local checkpoints, one from each process
- A consistent global checkpoint is a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before taking its checkpoint.

Consistent states - examples



- For instance, Figure shows two examples of global states.
- The state in fig (a) is consistent and the state in Figure (b) is inconsistent.
- Note that the consistent state in Figure (a) shows message *m*1 to have been sent but not yet received, but that is alright.
- The state in Figure (a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event.
- The state in Figure (b) is inconsistent because process *P*2 is shown to have received *m*2 but the state of process *P*1 does not reflect having sent it.
- Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures.

Interactions with outside world

A distributed system often interacts with the outside world to receive input data or deliver the outcome of a computation. If a failure occurs, the outside world cannot be expected to roll back. For example, a printer cannot roll back the effects of printing a character

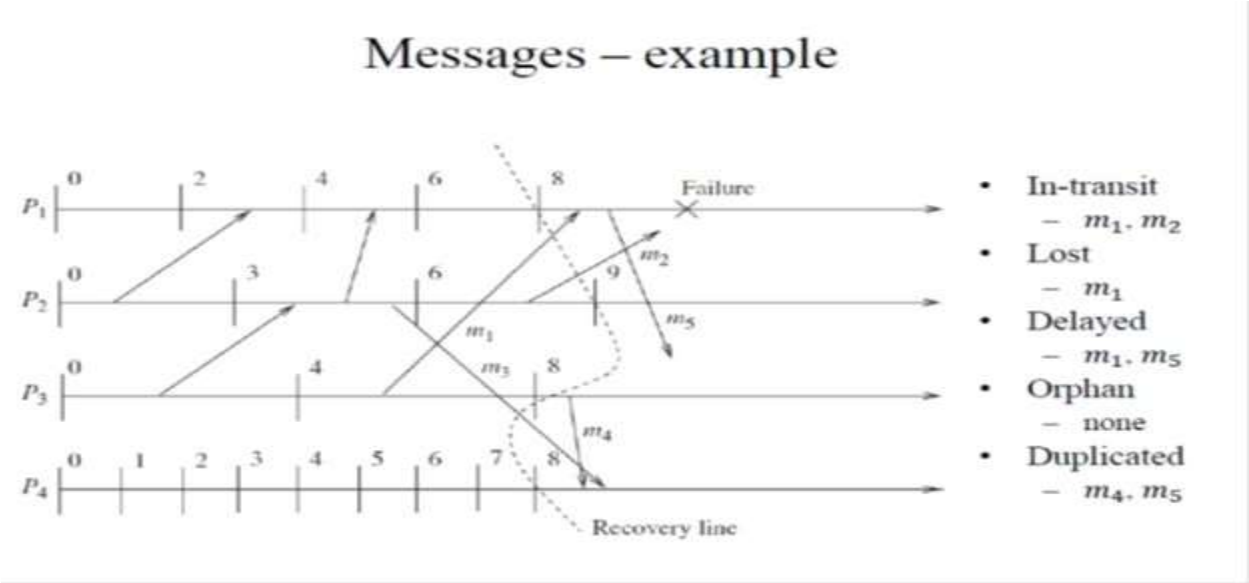
Outside World Process (OWP)

- It is a special process that interacts with the rest of the system through message passing.
- It is therefore necessary that the outside world see a consistent behavior of the system despite failures.
- Thus, before sending output to the OWP, the system must ensure that the state from which the output is sent will be recovered despite any future failure.

A common approach is to save each input message on the stable storage before allowing the application program to process it. An interaction with the outside world to deliver the outcome of a computation is shown on the process-line by the symbol "||".

Different types of Messages

1. In-transit message
    - messages that have been sent but not yet received
2. Lost messages
    - messages whose "send" is done but "receive" is undone due to rollback
3. Delayed messages
    - messages whose "receive" is not recorded because the receiving process was either down or the message arrived after rollback
4. Orphan messages
    - messages with "receive" recorded but message "send" notrecorded
    - do not arise if processes roll back to a consistent global state
5. Duplicate messages
    - arise due to message logging and replaying during process recovery



In-transit messages

In Figure , the global state {C1,8 , C2, 9 , C3,8, C4,8} shows that message *m*1 has been sent but not yet received. We call such a message an *in-transit* message. Message *m*2 is also an in-transit message.

Delayed messages

Messages whose receive is not recorded because the receiving process was either down or the message arrived after the rollback of the receiving process, are called *delayed* messages. For example, messages m2 and m5 in Figure are delayed messages.

Lost messages

Messages whose send is not undone but receive is undone due to rollbackare called lostmessages. This type of messages occurs when the processrolls back to a checkpoint prior to reception of the message while the senderdoes not rollback beyond the send operation of the message. In Figure ,message m1 is a lost message.

Duplicate messages

- Duplicate messages arise due to message logging and replaying during process recovery. For example, in Figure, message m4 was sent and received before the rollback. However, due to the rollback of process P4 to C4,8 and process P3 to C3,8, both send and receipt of message m4 are undone.
- When process P3 restarts from C3,8, it will resend message m4.
- Therefore, P4 should not replay message m4 from its log.
- If P4 replays message m4, then messagem4 is called a duplicate message.