UNIT IV CONSENSUS AND RECOVERY

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System(Synchronous and Asynchronous) – Agreement in Synchronous Systems with Failures; Checkpointing and Rollback Recovery: Introduction – Background and Definitions – Issuesin Failure Recovery – Checkpoint-based Recovery – Coordinated Checkpointing Algorithm - Algorithm for Asynchronous Checkpointing and Recovery

CONSENSUS PROBLEM IN ASYNCHRONOUS SYSTEMS.

Table: Overview of results on agreement.

f denotes number of failure-prone processes. n is the total number of processes.

Failure Mode	**Synchronous system (message-passing and shared memory)**	**Asynchronous system (message-passing and shared memory)**
No Failure	agreement attainable; common knowledge attainable	agreement attainable; concurrent common knowledge
Crash Failure	agreement attainable $f < n$ processes	agreement not attainable
Byzantie Failure	agreement attainable $f \leq [(n - 1)/3]$ Byzantine processes	agreement not attainable

In a failure-free system, consensus can be attained in a straightforward manner.

Consensus Problem (all processes have an initial value)

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 valueby all the non-faulty processes must be that same value.

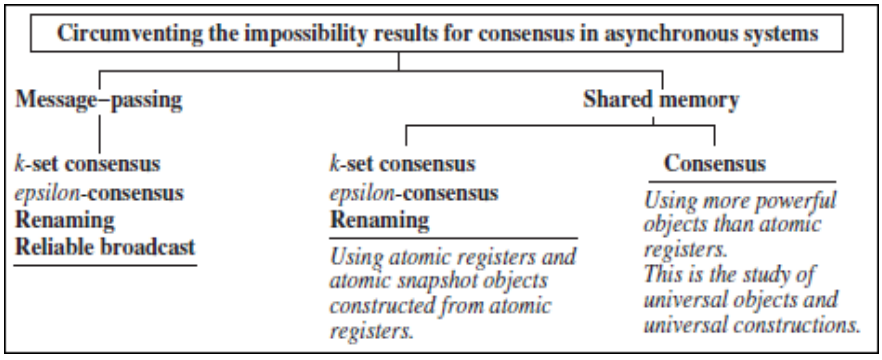
Termination: Each non-faulty process must eventually decide on a value.

Consensus Problem in Asynchronous Systems.

The overhead bounds are for the given algorithms, and not necessarily tight bounds for theproblem.

Solvable Variants	Failure model and overhead	Definition
Reliable broadcast	Crash Failure, $n > f$ (MP)	Validity, Agreement, Integrity conditions
k-set consensus	Crash Failure, $f < k < n$. (MP and SM)	size of the set of values agreed upon must be less than k
C-agreement	Crash Failure, $n \geq 5f + 1$ (MP)	values agreed upon are within ε of each other
Renaming	up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash Failure, $f \leq n - 1$ (SM)	select a unique name from a set of names

Circumventing the impossibility results for consensus in asynchronous systems:



STEPS FOR BYZANTINEGENERALS (ITERATIVE FORMULATION), SYNCHRONOUS, MESSAGE-PASSING:

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;
tree of boolean:

- level 0 root is v^{L}_{init}, where $L = \langle\rangle$;
- level $h (f \geq h > 0)$ nodes: for each v^{L}_{j} at level $h-1 = sizeof(L)$, its $n-2-sizeof(L)$ descendants at level h are $v^{concat(\langle j \rangle, L)}_{k}$, $\forall k$ such that $k \neq j$, i and k is not a member of list L.

(message type)
OM(v, *Dests*, *List*, *faulty*), where the parameters are as in the recursive formulation.

(1) Initiator (i.e., Commander) initiates Oral Byzantine agreement:
(1a) **send** $OM(v, N-\{i\}, \langle P_i \rangle, f)$ to $N-\{i\}$;
(1b) **return**(v).

(2) (Non-initiator, i.e., Lieutenant) receives Oral Message *OM*:
(2a) **for** $rnd = 0$ **to** f **do**
(2b) **for** each message OM that arrives in this round, **do**
(2c) **receive** $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty)$ from P_{k_1};
// $faulty + round = f$, $|Dests| + sizeof(L) = n$
(2d) $v^{tail(L)}_{head(L)} \longleftarrow v$; // $sizeof(L) + faulty = f+1$. fill in estimate.
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty - 1)$ to $Dests - \{i\}$ **if** $rnd < f$;
(2f) **for** $level = f - 1$ **down to** 0 **do**
(2g) **for** each of the $1 \cdot (n-2) \cdot \ldots \cdot (n-(level+1))$ nodes v^{L}_{x} in level *level*, **do**
(2h) $v^{L}_{x}(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v^{L}_{x}, v^{concat(\langle x \rangle, L)}_{y})$;

Byzantine Agreement (single source has an initial value)
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 source.

STEPS FOR BYZANTINE GENERALS (RECURSIVE FORMULATION),SYNCHRONOUS, MESSAGE-PASSING:

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
Oral_Msg(v, *Dests*, *List*, *faulty*), where
v is a boolean,
Dests is a set of destination process ids to which the message is sent,
List is a list of process ids traversed by this message, ordered from most recent to earliest,
faulty is an integer indicating the number of malicious processes to be tolerated.

Oral_Msg(f), where $f > 0$:

1. The algorithm is initiated by the Commander, who sends his source value v to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value v and terminates.
2. **[Recursion unfolding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process j, the process i uses the value v_j it receives from the source, and using that value, acts as a *new* source. (If no value is received, a default value is assumed.)

   To act as a new source, the process i initiates $Oral\_Msg(f'-1)$, wherein it sends
   $OM(v_j, Dests - \{i\}, concat(\langle i \rangle, L), (f'-1))$
   to destinations not in $concat(\langle i \rangle, L)$
   in the next round.
3. **[Recursion folding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in Step 2, each process i has computed the agreement value v_k, for each k not in *List* and $k \neq i$,corresponding to the value received from P_k after traversing the nodes in *List*, at one level lower in the recursion. If it receives no value in this round, it uses a default value. Process i then uses the value $majority_{k \notin List, k \neq i}(v_j, v_k)$ as the agreement value and returns it to the next higher level in the recursive invocation.

Oral_Msg(0):

1. **[Recursion unfolding:]** Process acts as a source and sends its value to each other process.
2. **[Recursion folding:]** Each process uses the value it receives from the other sources, and uses that value as the agreement value. If no value is received, a default value is assumed.

CODE FOR THE PHASE KING ALGORITHM:

Each phase has a unique "phase king" derived, say, from PID. Each phase has two rounds:

- 1 in 1st round, each process sends its estimate to all other processes.
- 2 in 2nd round, the "Phase king" process arrives at an estimate based on the values itreceived in 1st round, and broadcasts its new estimate to all others.

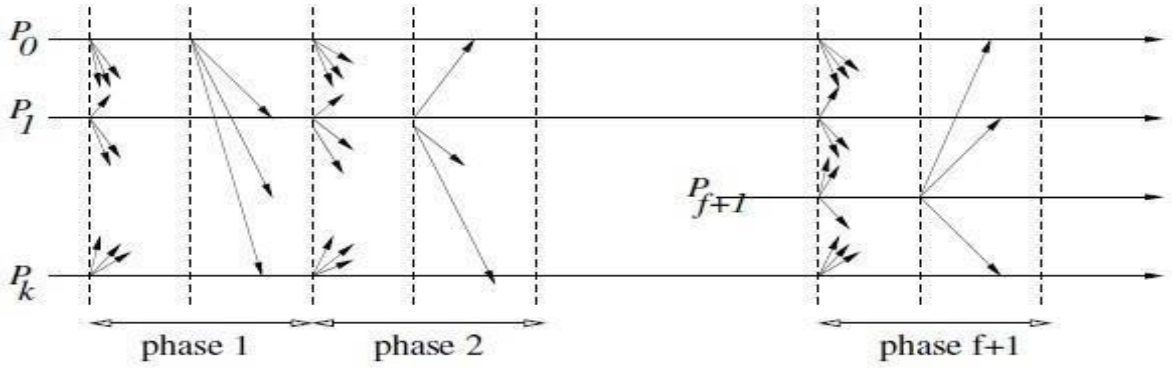


Fig. Message pattern for the phase-king algorithm.

```
(variables)
boolean: v ⟵ initial value;
integer: f ⟵ maximum number of malicious processes, f < ⌈n/4⌉;

(1) Each process executes the following f + 1 phases, where f < n/4:
(1a) for phase = 1 to f + 1 do
(1b)   Execute the following Round 1 actions:          // actions in round one of each phase
(1c)        broadcast v to all processes;
(1d)        await value v_j from each process P_j;
(1e)        majority ⟵ the value among the v_j that occurs > n/2 times (default if no maj.);
(1f)        mult ⟵ number of times that majority occurs;
(1g)   Execute the following Round 2 actions:          // actions in round two of each phase
(1h)        if i = phase then // only the phase leader executes this send step
(1i)             broadcast majority to all processes;
(1j)        receive tiebreaker from P_phase (default value if nothing is received);
(1k)        if mult > n/2 + f then
(1l)             v ⟵ majority;
(1m)        else v ⟵ tiebreaker;
(1n)        if phase = f + 1 then
(1o)             output decision value v.
```

PHASE KING ALGORITHM CODE:

(f + 1) phases, (f + 1)[(n - 1)(n + 1)] messages, and can tolerate up to f < dn=4e malicious processesCorrectness Argument

- 1 Among f + 1 phases, at least one phase k where phase-king is non-malicious.
- 2 In phase k, all non-malicious processes Pi and Pj will have same estimate of consensusvalue as Pk does.
- Pi and Pj use their own majority values. Pi 's mult > n=2 + f)
- Pi uses its majority value; Pj uses phase-king's tie-breaker value. (Pi’s mult > n=2 + f ,Pj 's mult > n=2 for same value)
- Pi and Pj use the phase-king's tie-breaker value. (In the phase in which Pk is non-malicious, it sends same value to Pi and Pj)

In all 3 cases, argue that Pi and Pj end up with same value as estimate

- If all non-malicious processes have the value x at the start of a phase, they will continueto have x as the consensus value at the end of the phase.

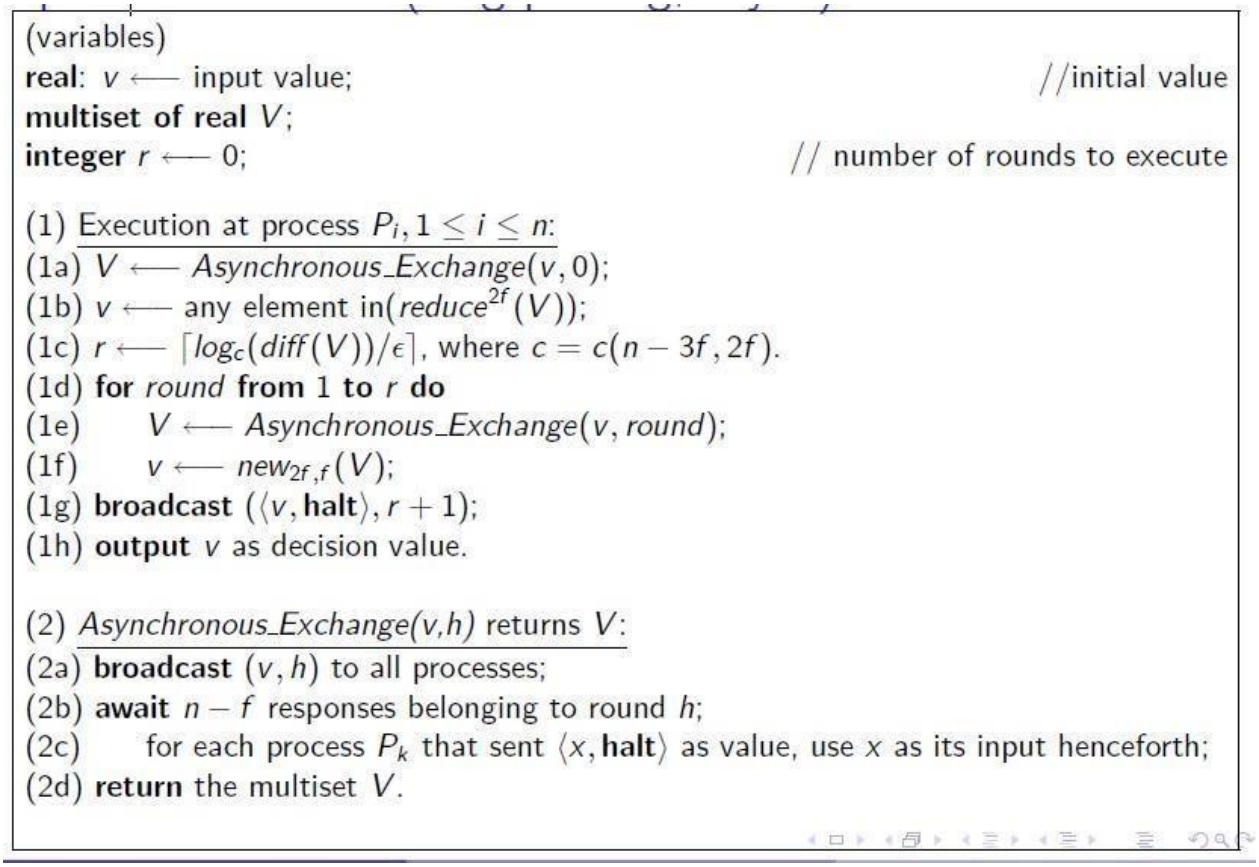
CODE FOR THE EPSILON CONSENSUS (MESSAGE-PASSING, ASYNCHRONOUS):

Agreement: All non-faulty processes must make a decision and the values decided upon by anytwo non-faulty processes must be within range of each other.

Validity: If a non-faulty process Pi decides on some value vi , then that value must be within therange of values initially proposed by the processes.

Termination: Each non-faulty process must eventually decide on a value. The algorithm for themessage-passing model assumes $n \geq 5f + 1$, although the problem is solvable for $n > 3f + 1$.

- Main loop simulates sync rounds.
- Main lines (1d)-(1f): processes perform all-all msg exchange
- Process broadcasts its estimate of consensus value, and awaits n - f similar
- msgs from other processes
- the processes' estimate of the consensus value converges at a particular rate,
- until it is _ from any other processes estimate.
- # rounds determined by lines (1a)-(1c).



TWO-PROCESS WAIT-FREE CONSENSUS USING FIFO QUEUE, COMPARE & SWAP:

Wait-free Shared Memory Consensus using Shared Objects:

Not possible to go from bivalent to univalent state if even a single failure is allowed. Difficulty isnot being able to read & write a variable atomically.

- It is not possible to reach consensus in an asynchronous shared memory system usingRead/Write atomic registers, even if a single process can fail by crashing.

  There is no wait-free consensus algorithm for reaching consensus in an asynchronous shared memory system using Read/Write atomic registers.

To overcome these negative results:

- Weakening the consensus problem, e.g., k-set consensus, approximate consensus, and renaming using atomic registers.

- Using memory that is stronger than atomic Read/Write memory to design wait- free consensus algorithms. Such a memory would need corresponding access primitives.

Are there objects (with supporting operations), using which there is a wait-free (i.e., (n -1)-crashresilient) algorithm for reaching consensus in a n-process system? Yes, e.g., Test&Set, Swap, Compare&Swap. The crash failure model requires the solutions to be wait-free.

TWO-PROCESS WAIT-FREE CONSENSUS USING FIFO QUEUE:

```
(shared variables)
queue: Q ⟵ ⟨0⟩;                                          // queue Q initialized
integer: Choice[0, 1] ⟵ [⊥, ⊥]                  // preferred value of each process
(local variables)
integer: temp ⟵ 0;
integer: x ⟵ initial choice;

(1) Process P_i, 0 ≤ i ≤ 1, executes this for 2-process consensus using a FIFO queue:
(1a) Choice[i] ⟵ x;
(1b) temp ⟵ dequeue(Q);
(1c) if temp = 0 then
(1d)      output(x)
(1e) else output(Choice[1 − i]).
```

WAIT-FREE CONSENSUS USING COMPARE & SWAP:

```
(shared variables)
integer: Reg ⟵ ⊥;                          // shared register Reg initialized
(local variables)
integer: temp ⟵ 0;                         // temp variable to read value of Reg
integer: x ⟵ initial choice;               // initial preference of process

(1) Process P_i, (∀i ≥ 1), executes this for consensus using Compare&Swap:
(1a) temp ⟵ Compare&Swap(Reg, ⊥, x);
(1b) if temp = ⊥ then
(1c)     output(x)
(1d) else output(temp).
```

NONBLOCKING UNIVERSAL ALGORITHM:

Universality of Consensus Objects

An object is defined to be universal if that object along with read/write registers can simulate any other object in a wait-free manner. In any system containing up to k processes, an object Xsuch that CN(X) = k is universal.

For any system with up to k processes, the universality of objects X with consensus number k isshown by giving a universal algorithm to wait-free simulate any object using objects of type X and read/write registers.

This is shown in two steps.

- 1 A universal algorithm to wait-free simulate any object whatsoever using read/writeregisters and arbitrary k-processor consensus objects is given. This is the main step.
- 2 Then, the arbitrary k-process consensus objects are simulated with objects of type X,having consensus number k. This trivially follows after the first step.

Any object X with consensus number k is universal in a system with $n \leq k$ processes.

A nonblocking operation, in the context of shared memory operations, is an operation that maynot complete itself but is guaranteed to complete at least one of the pending operations in a finite number of steps.

Nonblocking Universal Algorithm:

The linked list stores the linearized sequence of operations and states following each operation.

Operations to the arbitrary object Z are simulated in a nonblocking way using an arbitraryconsensus object (the field op.next in each record) which is accessed via the Decide call.

Each process attempts to thread its own operation next into the linked list.

- There are as many universal objects as there are operations to thread.
- A single pointer/counter cannot be used instead of the array Head. Because reading andupdating the pointer cannot be done atomically in a wait-free manner.
- Linearization of the operations given by the sequence number. As algorithm is non block

4.1 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 communicateover 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 storageor 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 dominoeffect – this scheme is called independent or uncoordinated check pointing

Techniques that avoid domino effect

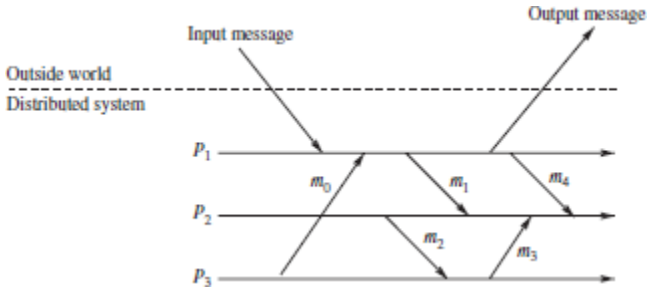
1. Coordinated check pointing rollback recovery - Processes coordinate their checkpoints toform a system-wide consistent state
2. Communication-induced check pointing rollback recovery - Forces each process to take checkpoints based on information piggybacked on the application.
3. Log-based rollback recovery - Combines check pointing 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 worldby receiving and sending input and output messages, respectively.

  Rollback-recovery protocols generally make assumptions about the reliability of theinter- process communication.
- Some protocols assume that the communication uses first-in-first-out (FIFO) order, whileother 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

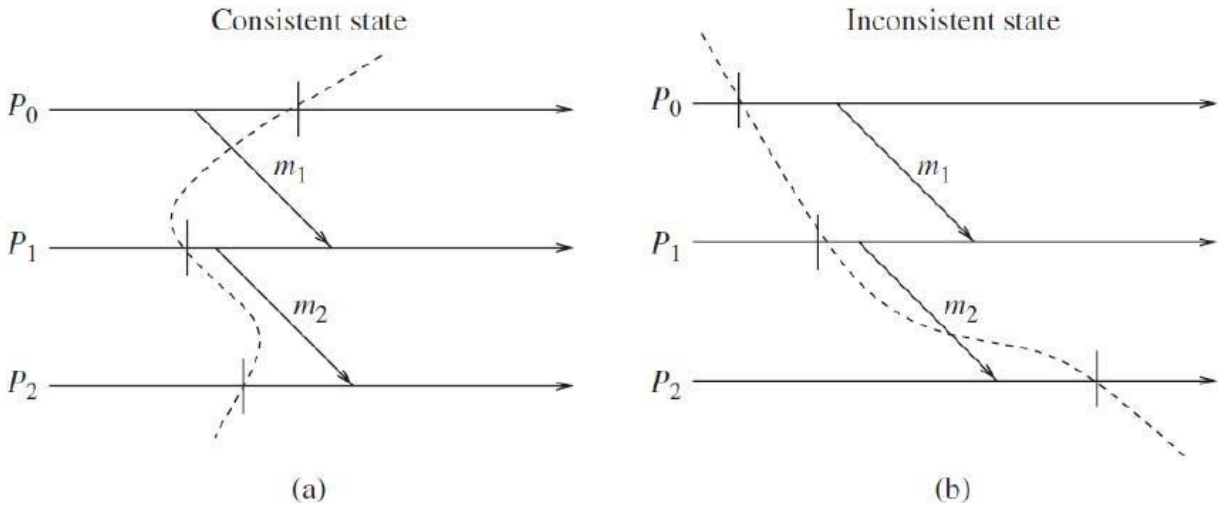
Consistent states

- 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 correspondingsender 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 notyet received, but that is alright.
- The state in Figure (a) is consistent because it represents a situation in which everymessage 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*2but the state of process *P*1 does not reflect having sent it.
- Such a state is impossible in any failure-free, correct computation. Inconsistent statesoccur 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 systemdespite failures.
- Thus, before sending output to the OWP, the system must ensure that the state from whichthe 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.

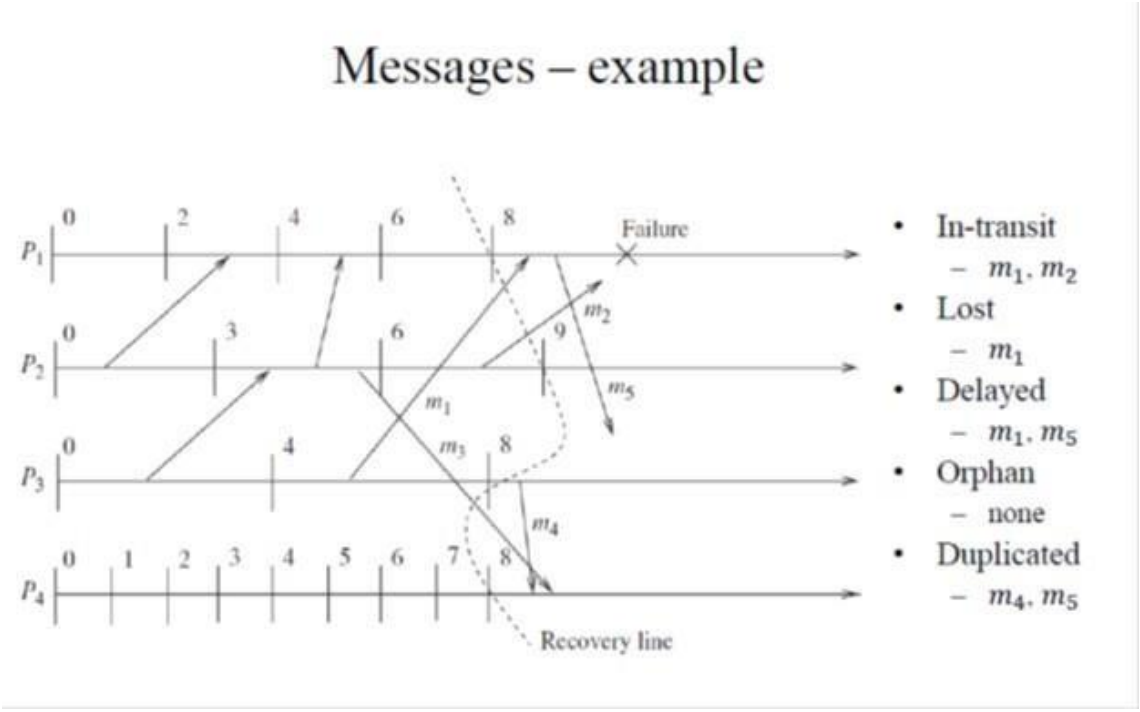
An interaction with the outside world to deliver the outcome of a computation is shown on theprocess-line by the symbol "||".

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 waseither

down or the message arrived after rollback

4. Orphan messages
    - messages with "receive" recorded but message "send" not recorded
    - 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 rollback are called lostmessages.This type of messages occurs when the process rolls back to a checkpoint prior to reception of themessage while the sender does not rollback beyond the send operation of the message. In Figure ,message m1 is

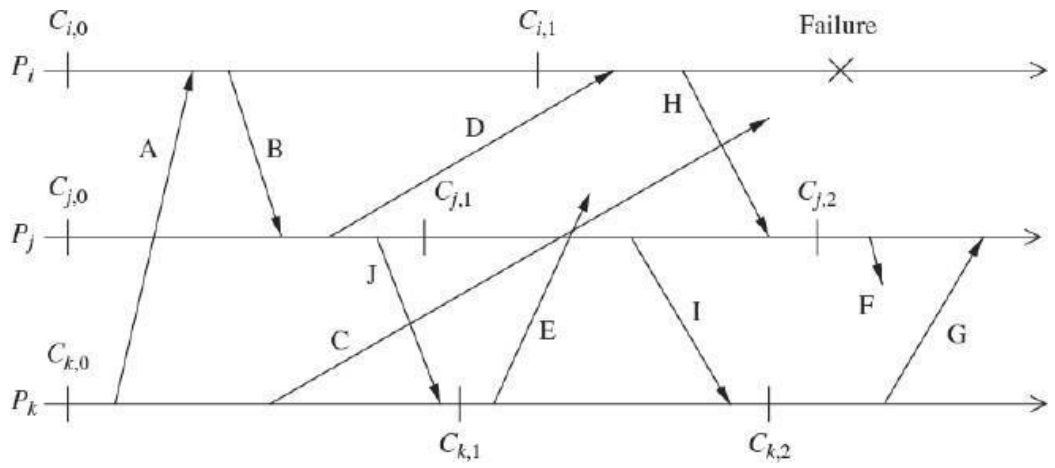
a lost message.

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 message m4 is called a duplicate message

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. U*ncoordinated checkpointing*
2. *Coordinated checkpointing*
3. *Communication-induced checkpointing*

1. Uncoordinated Checkpointing

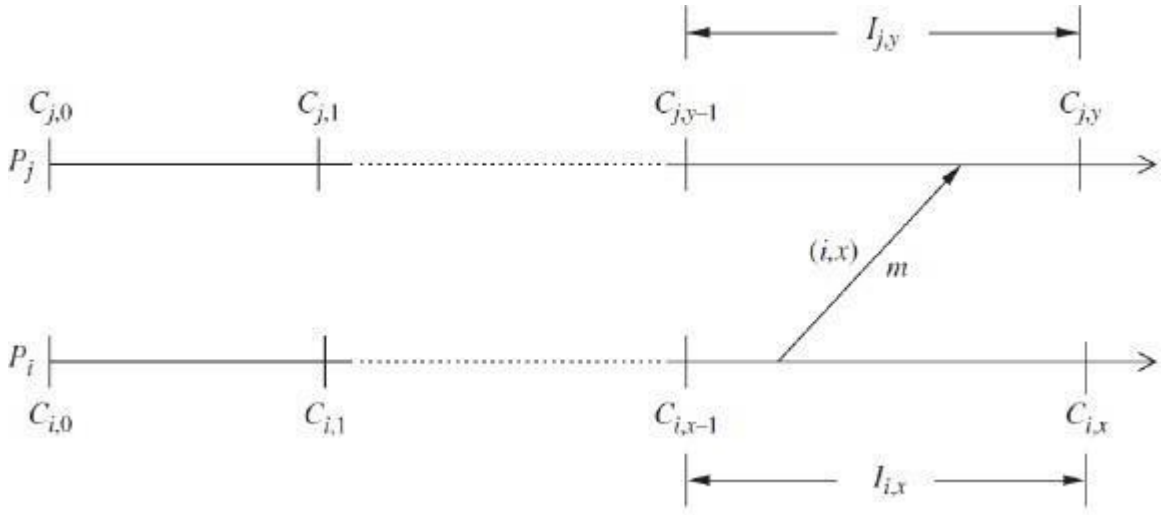
- Each process has autonomy in deciding when to take checkpoints
- Advantages

  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 checkpointing.

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.

Coordinated Checkpointing

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

Types

5. 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

6. 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.

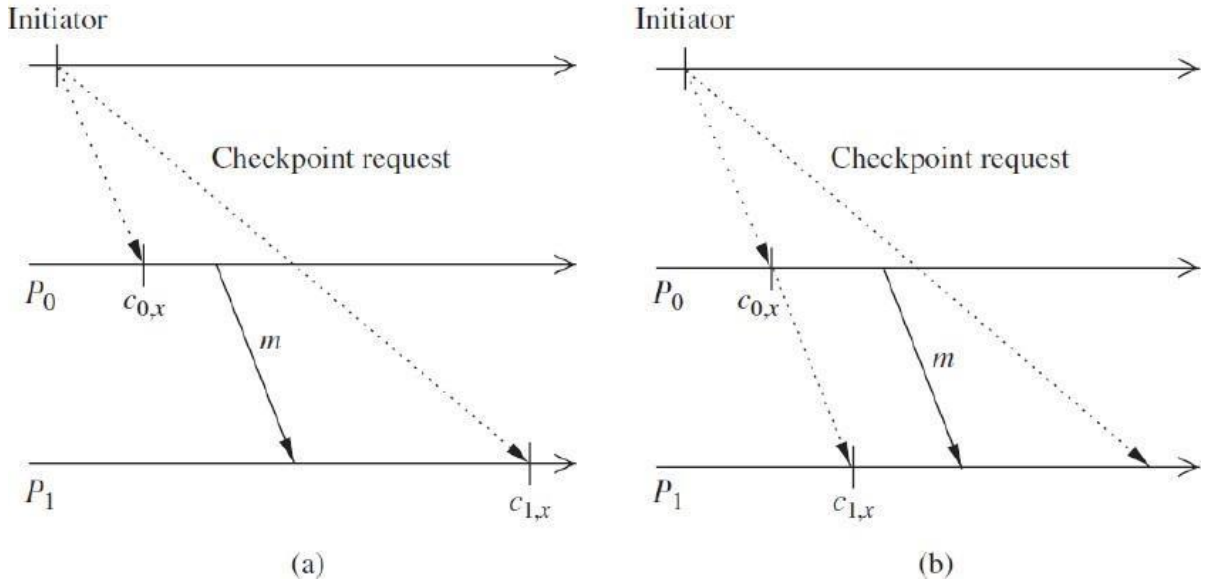
Example (a) : Checkpoint inconsistency

- 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.

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 check pointing 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 check pointing, 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.

4.4 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

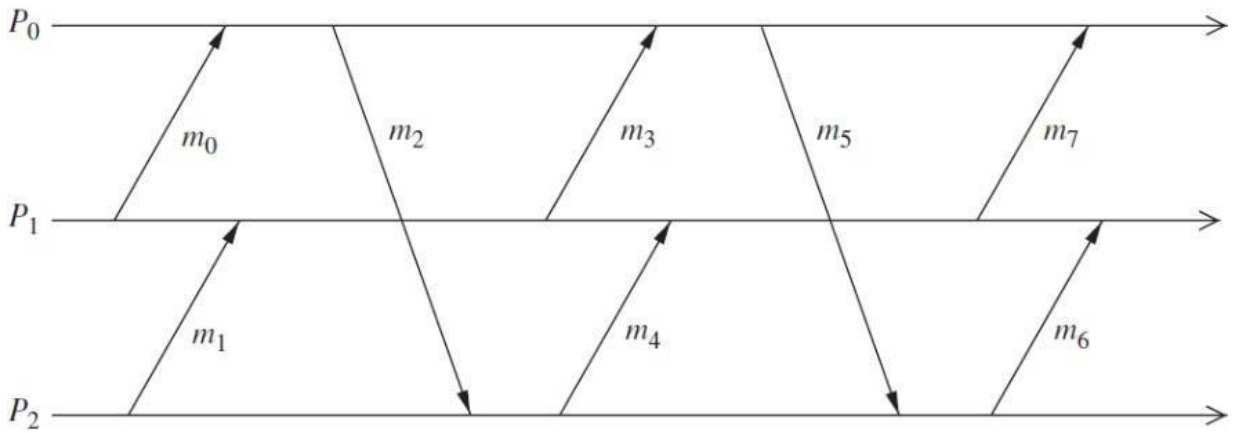
receipt of message *m*0, and is therefore not a non-deterministic event.

- 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.

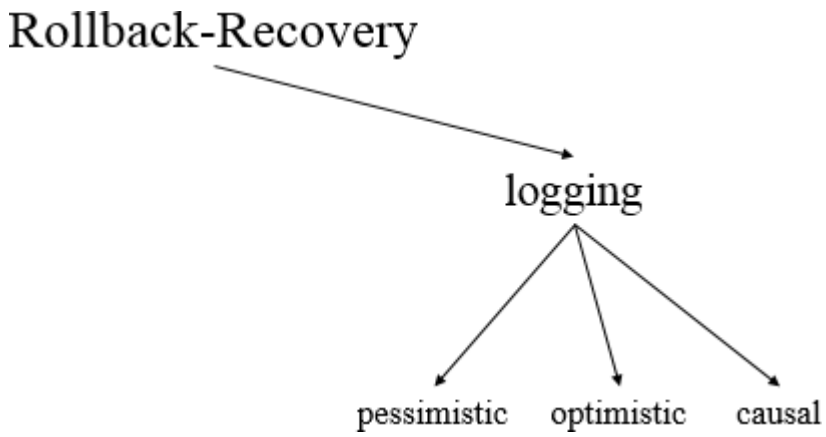
• *Stable*(*e*): a predicate that is true if *e*'s determinant is logged on the stable storage.

Suppose a set of processes Ψ crashes. A process *p* in Ψ becomes an orphan when *p* itself does not fail and *p*'s state depends on the execution of a nondeterministic event *e* whose determinant cannot be recovered from the stable storage or from the volatile memory of a surviving process. storage or from the volatile memory of a surviving process. Formally, it can be stated as follows

always-no-orphans condition

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

Types



1. Pessimistic Logging

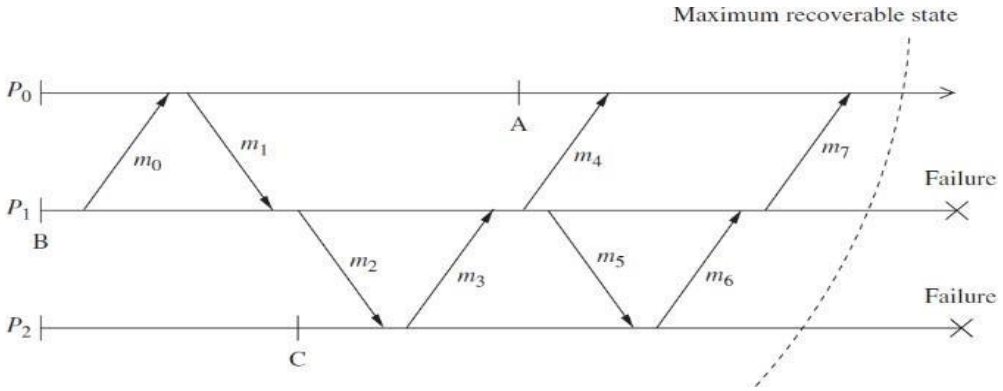
- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation. However, in reality failures are rare
- Pessimistic protocols implement the following property, often referred to as *synchronous logging*,which is a stronger than the always-no-orphans 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 $P1$ and $P2$ 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 $P1$



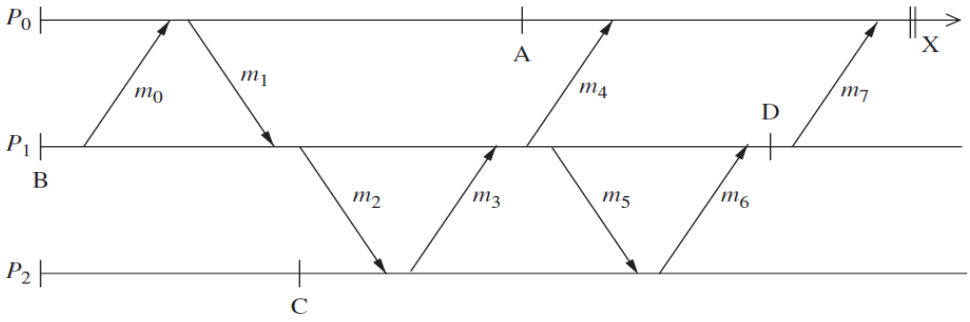
- Disadvantage: performance penalty for synchronous logging
- Advantages:
    - immediate output commit
    - restart from most recent checkpoint
    - recovery limited to failed process(es)
    - simple garbage collection
- Some pessimistic logging systems reduce the overhead of synchronous logging 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.

Optimistic Logging

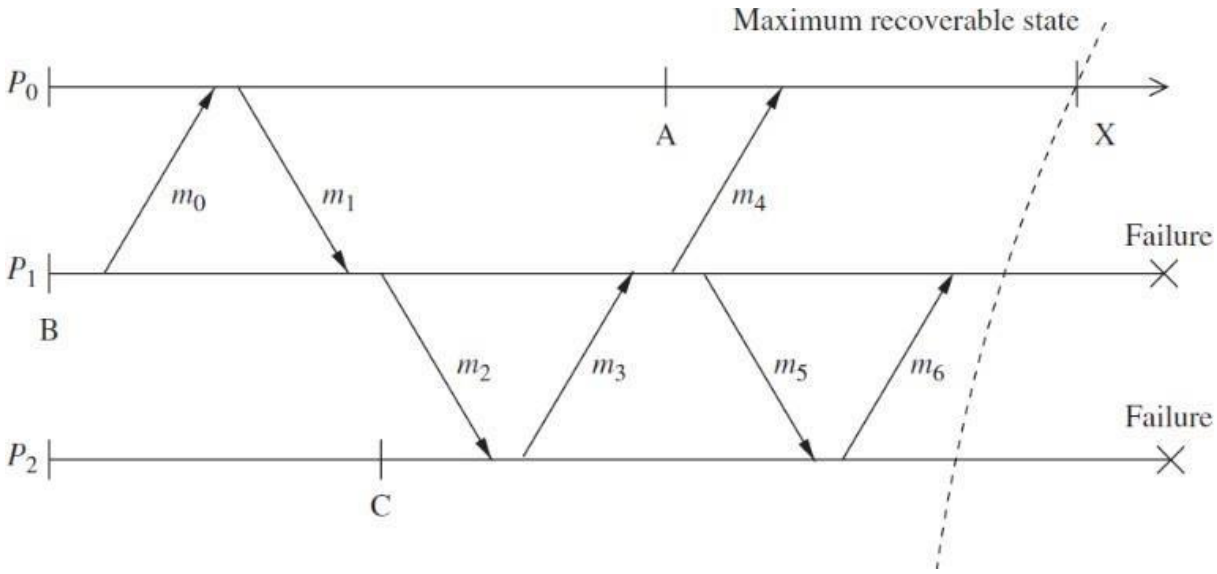
- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a failure occurs
- Do not implement the *always-no-orphans* condition
- To perform rollbacks correctly, optimistic logging protocols track causal dependenciesduring failure free execution
- Optimistic logging protocols require a non-trivial garbage collection scheme
- Pessimistic protocols need only keep the most recent checkpoint of each process, whereasoptimistic protocols may need to keep multiple checkpoints for each process



- Consider the example shown in Figure Suppose process *P*2 fails before the determinant for*m*5 is logged to the stable storage. Process *P*1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message *m*6. The rollback of *P*1 further forces *P*0 to roll back to undo the effects of receiving message *m*7.
  - **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 loggingprotocols requires a guarantee that no failure scenario can revoke the output. For example,if process *P*0 needs to commit output at state X, it must log messages *m*4 and*m*7 to the stable storage and ask *P*2 to log *m*2 and *m*5. In this case, if any process fails, the computation can be reconstructed up to state X.

Causal Logging

- Combines the advantages of both pessimistic and optimistic logging at the expense of a morecomplex recovery protocol
- Like optimistic logging, it does not require synchronous access to the stable storage exceptduring output commit
- Like pessimistic logging, it allows each process to commit output independently and nevercreates 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 failuresof P1 and P2 at the indicated instants. Process
- *P*0 at state X will have logged the determinants of the nondeterministic events thatcausally 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 $P0$ will be able to "guide" the recovery of $P1$ and $P2$ since it knows the order in which $P1$ should replay messages $m1$ and $m3$ to reach the state from which $P1$ sent message$m4$.
- 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.

KOO AND TOUEG COORDINATED CHECKPOINTING AND RECOVERY TECHNIQUE:

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

A. The Checkpointing 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 withmessage 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 andTentative.

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

A *tentative checkpoint* is a temporary checkpoint that is made a permanent checkpoint onthe 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 decidesthat all tentative checkpoints should be made permanent; otherwise, Pi decides that all thetentative 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 permanentcheckpoints.
4. The algorithm requires that after a process has taken a tentative checkpoint, it cannotsend 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 arewilling 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).

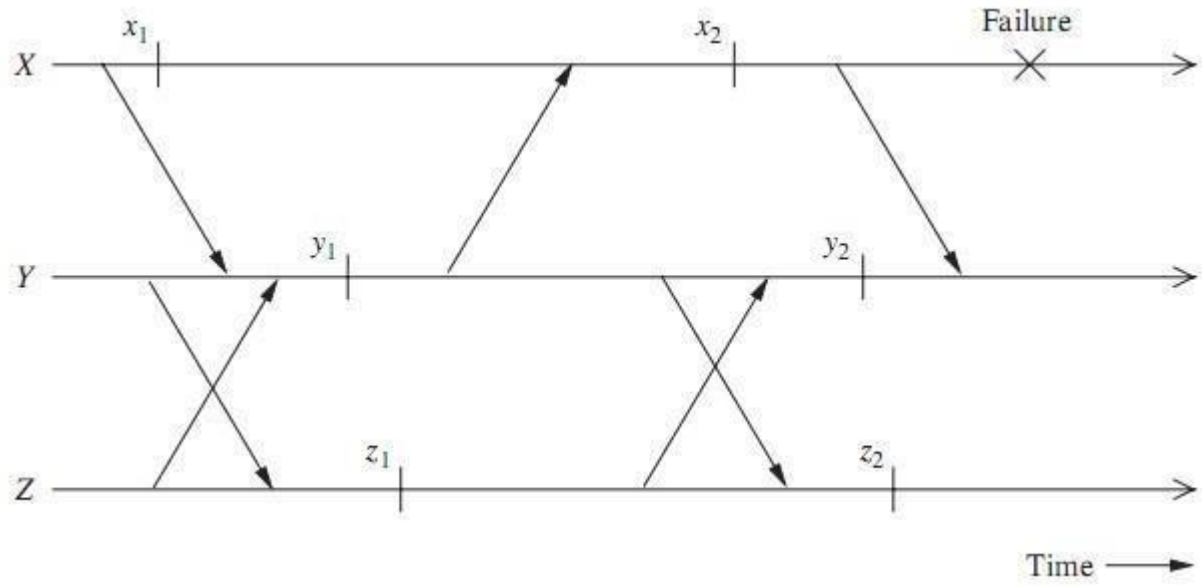
3. If Pi learns that all processes are willing to restart from their previous checkpoints, Pidecides that all processes should roll back to their previous checkpoints. Otherwise,
4. Pi aborts the roll back attempt and it may attempt a recovery at a later time.

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 relatedto 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



The above protocol, in the event of failure of process X, the above protocol will requireprocesses 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 theother two processes since the last checkpoint at Z.

ALGORITHM FOR ASYNCHRONOUS CHECKPOINTING AND RECOVERY:

The algorithm of Juang and Venkatesan for recovery in a system that uses asynchronous checkpointing.

A. 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 haveinfinite buffers.
- The message transmission delay is arbitrary, but finite.
- Underlying computation/application is event-driven: process P is at state s, receives message m, processes the message, moves to state s᾽ and send messages out. So the triplet (*s, m, msgs_sent*) represents the state of P

  Two type of log storage are maintained:
  - Volatile log: short time to access but lost if processor crash. Move to stable log periodically.
  - Stable log: longer time to access but remained if crashed

A. Asynchronous Check pointing

- After executing an event, the triplet is recorded without any synchronization withother processes.
- Local checkpoint consist of set of records, first are stored in volatile log, thenmoved to stable log.

The Recovery Algorithm Notations and data structure

The following notations and data structure are used by the algorithm:

• RCVDi←j(CkPti) represents the number of messages received by processor pi from processorpj , from the beginning of the computation till the checkpoint CkPti.

• SENTi→j(CkPti) represents the number of messages sent by processor pi to processor pj , fromthe beginning of the computation till the checkpoint CkPti.

Basic idea

- Since the algorithm is based on asynchronous check pointing, the main issue in therecovery is to find a consistent set of checkpoints to which the system can be restored.

  The recovery algorithm achieves this by making each processor keep track of both the number of messages it has sent to other processors as well as the number of messages it has received from other processors.

- Whenever a processor rolls back, it is necessary for all other processors to find out if any message has become an orphan message. Orphan messages are discovered by comparing the number of messages sent to and received from neighboring processors. For example, if RCVDi←j(CkPti) > SENTj→i(CkPtj) (that is, the number of messages received by processor pi from processor pj is greater than the number of messages sent by processor pj to processor pi, according to the current states the processors), then one or more messages at processor pj are orphan messages.

The Algorithm

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

Procedure RollBack_Recovery

processor pi executes the following:

STEP (a)

if processor pi is recovering after a

failure **then** CkPti := latest event

logged in the stable storage**else**

CkPti := latest event that took place in pi {The latest event at pi can be either in stable or involatile storage.}

end if

STEP (b)

for k = 1 1 to N {N is the number of processors in the

system} **dofor** each neighboring processor pj **do**

compute SENTi→j(CkPti)

send a ROLLBACK(i, SENTi→j(CkPti)) message to pj

end for

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

if RCVDi←j(CkPti) > c {Implies the presence of orphan messages} **then**

find the latest event e such that RCVDi←j(e) = c {Such an event e may be in the volatile storageor stable storage.}

CkPti := e

end if

end for

end for{for k}

D. An Example

Consider an example shown in Figure 2 consisting of three processors. Suppose processor Y fails and restarts. If event ey2 is the latest checkpointed event at Y, then Y will restart from thestate corresponding to ey2.

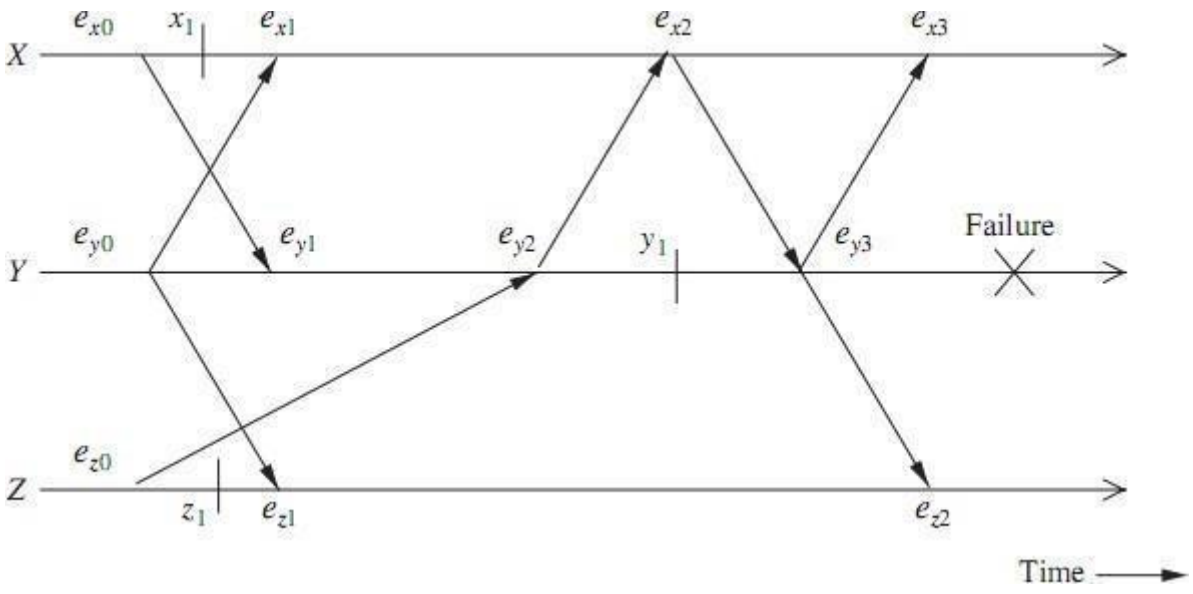


Figure 2: An example of Juan-Venkatesan algorithm.

- Because of the broadcast nature of ROLLBACK messages, the recovery algorithm isinitiated at processors X and Z.
- Initially, X, Y, and Z set CkPtX ← ex3, CkPtY ← ey2 and CkPtZ ← ez2, respectively,and X, Y, and Z send the following messages during the first iteration:
- Y sends ROLLBACK(Y,2) to X and ROLLBACK(Y,1) to Z;
- X sends ROLLBACK(X,2) to Y and ROLLBACK(X,0) to Z;
- Z sends ROLLBACK(Z,0) to X and ROLLBACK(Z,1) to Y.

Since RCVDX←Y (CkPtX) = 3 > 2 (2 is the value received in the ROLLBACK(Y,2) messagefrom Y), X will set CkPtX to ex2 satisfying RCVDX←Y (ex2) = 1≤ 2.

Since RCVDZ←Y (CkPtZ) = 2 > 1, Z will set CkPtZ to ez1 satisfying RCVDZ←Y (ez1) = 1 ≤1.

At Y, RCVDY←X(CkPtY) = 1 < 2 and RCVDY←Z(CkPtY) = 1 = SENTZ←Y (CkPtZ).

Y need not roll back further.

In the second iteration, Y sends ROLLBACK(Y,2) to X and ROLLBACK(Y,1) to Z;

Z sends ROLLBACK(Z,1) to Y and ROLLBACK(Z,0) to X;

X sends ROLLBACK(X,0) to Z and ROLLBACK(X, 1) to Y.

If Y rolls back beyond ey3 and loses the message from X that caused ey3, X can resend this message to Y because ex2 is logged at X and this message available in the log. The second and third iteration will progress in the same manner. The set of recovery points chosen at the end of the first iteration, {ex2, ey2, ez1}, is consistent, and no further rollback occurs.