UNIT IV RECOVERY & CONSENSUS

Checkpointing and rollback recovery: Introduction – Background and definitions – Issues in failure recovery – Checkpoint-based recovery – Log-based rollback recovery – Coordinated checkpointing algorithm – Algorithm for asynchronous checkpointing and recovery. Consensus and agreement algorithms: Problem definition – Overview of results – Agreement in a failure – free system – Agreement in synchronous systems with failures.

4.1 Introduction

Rollback recovery treats a distributed system application as a collection of processes that communicate over a network. It achieves fault tolerance by periodically saving the state of a process during the failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work.

The saved state is called a ***checkpoint,*** and the procedure of restarting from a previously checkpointed state is called ***rollback recovery***. A checkpoint can be saved on either the stable storage or the volatile storage depending on the failure scenarios to be tolerated.

- In a distributed system, if each participating process takes its checkpoints independently, then the system is susceptible to the **domino effect**. This approach is called ***independent* or *uncoordinated checkpointing***.
- It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it. One such technique is ***coordinated check-pointing*** where processes coordinate their checkpoints to form a system-wide consistent state. In case of a process failure, the system state can be restored to such a consistent set of checkpoints, preventing the rollback propagation.
- Alternatively, ***communication-induced checkpointing*** forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes. Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

Log-based **rollback recovery**

- The approaches discussed so far implement *checkpoint-based* rollback recovery, which relies only on checkpoints to achieve fault-tolerance. ***Log-based*** **rollback recovery** combines checkpointing with logging of non-deterministic events. Log-based rollback recovery relies on the ***piecewise deterministic*** **(PWD)** assumption, which postulates that all non-deterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's *determinant*.

- By logging and replaying the non-deterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed. Log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints. It is therefore particularly attractive for applications that frequently interact with the *outside world*, which consists of input and output devices that cannot roll back.

Introduction

- **Rollback recovery protocols**

— restore the system back to a consistent state after afailure

— achieve fault tolerance by periodically saving the state of a process during the failure-free execution

— treats a distributed system application as a collection of processesthat communicate over a network

Checkpoints -> the saved states of a process

Why is rollback recovery of distributed systems complicated?

messages induce inter-process dependencies during failure-freeoperation

Rollback propagation

The dependencies may force some of the processes that did not fail to roll back. This phenomenon is called "*domino effect*"

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

➔ **Coordinated checkpointing rollback recovery**
processes coordinate their checkpoints to form asystem-wide consistent state

➔ **Communication-induced checkpointing rollback recovery**
forces each process to take checkpoints based on information piggybacked on the application

➔ **Log-based rollback recovery**
combines checkpointing with logging of non-deterministicevents relies on piecewise deterministic (PWD) assumption

4.2 Background and definitions

System model

Distributed system consists of a fixed number of processes, P_1, P_2 P_N , 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. Figure shows a system consisting of three processes and interactions with the outside world.

Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication. Some protocols assume that the com-munication subsystem delivers messages reliably, in first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can

A local checkpoint

- In distributed systems, all processes save their local states at certain instants of time. This saved state is known as a local checkpoint.

- A local checkpoint is a snapshot of the state of the process at a given instance and the event of recording the state of a process is called local checkpointing.

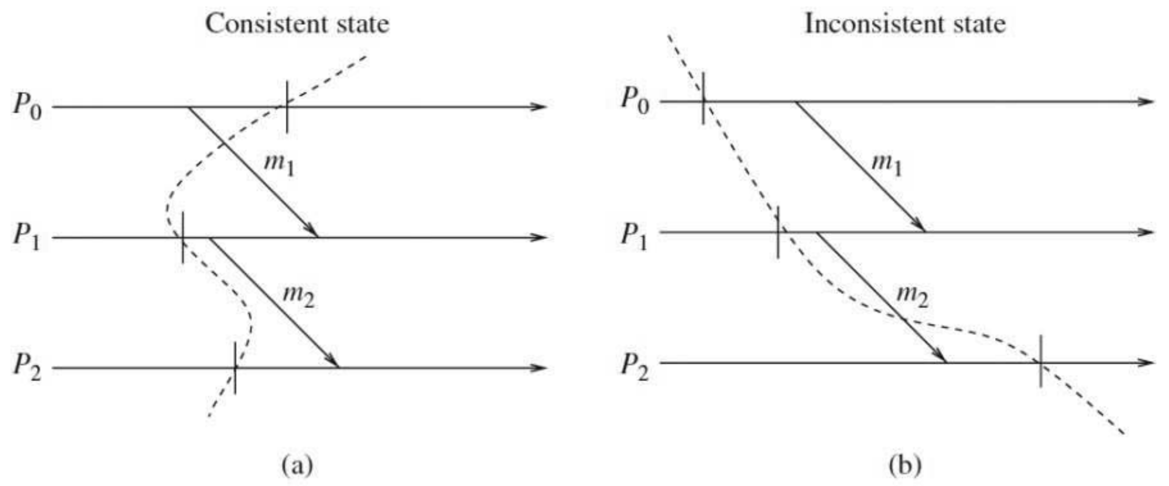
- The contents of a checkpoint depend upon the application context and the checkpointing method being used.

Assumption
A process stores all local checkpoints on the stablestorage
A process is able to roll back to any of its existing localcheckpoints ,*k*
The *k*th local checkpoint at process is $C_{i,0}$
Aprocess P_i takes a checkpoint $C_{i,0}$ before it startsexecution

Consistent system states

A global state of a distributedsystem
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 consistent global state is one that may occur during a failure-free execution of a distributed computation. More precisely, a *consistent system state* is one in which a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of that message
A global checkpoint
a set of local checkpoints, one from each process
A consistent globalcheckpoint
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 Figure (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. For instance, the situation shown in Figure (b) may occur if process P_1 fails after sending message m_2 to process P_2 and

then restarts at the state shown in Figure (b).

Thus, a local checkpoint is a snapshot of a local state of a process and 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 checkpoint that is received by another process before taking its local checkpoint. The consistency of global checkpoints strongly depends on the flow of messages exchanged by processes and an arbitrary set of local checkpoints at processes may not form a consistent global checkpoint.

The fundamental goal of any rollback-recovery protocol is to bring the system to a consistent state after a failure. The reconstructed consistent state is not necessarily one that occurred before the failure. It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free execution, provided that it is consistent with the interactions that the system had with the outside world.

4.3.4 Interactions with outside world

A distributed application 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, and an automatic teller machine cannot recover the money that it dispensed to a customer.

- ➔ **A distributed system often interacts with the outside worldto receive input data or deliver the outcome of a computation**
- ➔ **Outside World Process (OWP)**
  a special process that interacts with the rest of the systemthrough message passing

A common approach
save each input message on the stable storage before allowingthe application program to process it

Symbol "||"
An interaction with the outside world to deliver the outcome of a computation

Different types of messages

i. **In-transit message ->messages that have been sent but not yet received**
   In Figure, the global state $\{C_{1\,8}\, C_{2\,9}\, C_{3\,8}\, C_{4\,8}\}$ 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.

ii. **Lost messages**
   Messages whose send is not undone but receive is undone due to rollback are called *lost* messages. This type of messages occurs when the process rolls back to a checkpoint prior to reception of the message while the sender does not rollback beyond the send operation of the message. In Figure, message m_1 is a lost message.

iii. **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 m_2 and m_5 in Figure are delayed messages.

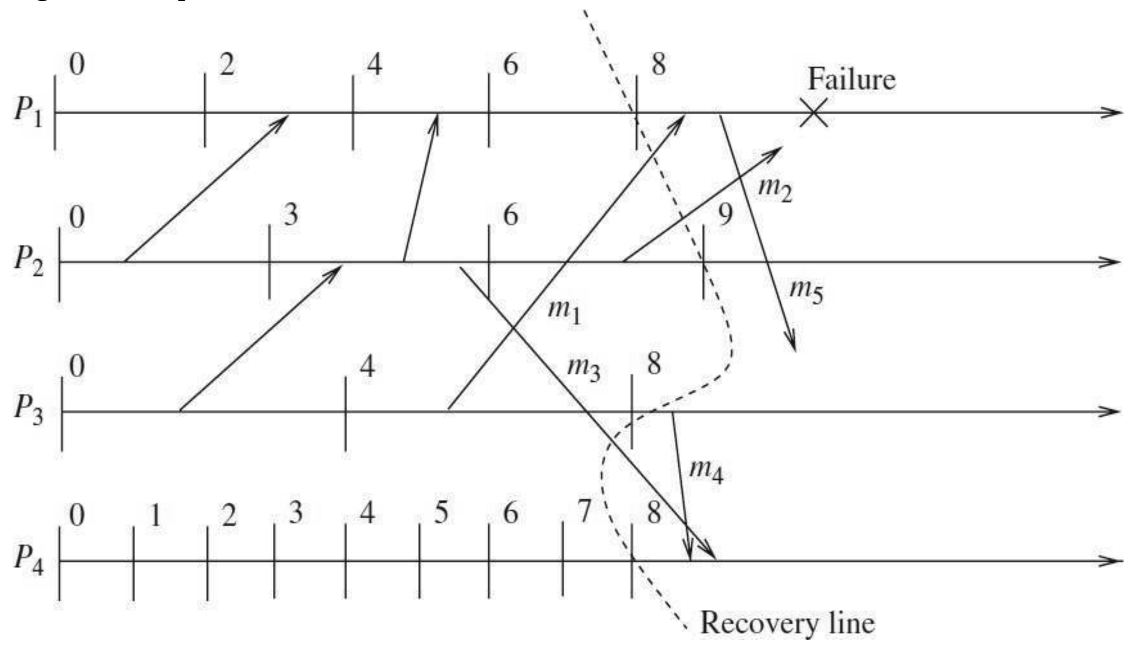
iv. ***orphan* messages**
   Messages with receive recorded but message send not recorded are called *orphan*

messages. For example, a rollback might have undone the send of such messages, leaving the receive event intact at the receiving process. **Orphan messages do not arise if processes roll back to a consistent global state.**

v. **Duplicate messages**

Duplicate messages arise due to message logging and replaying during process recovery.

For example, in Figure, message m_4 was sent and received before the rollback. However, due to the rollback of process P_4 to $C_{4\ 8}$ and process P_3 to $C_{3\ 8}$, both send and receipt of message m_4 are undone. When process P_3 restarts from $C_{3\ 8}$, it will resend message m_4. Therefore, P_4 should not replay message m_4 from its log. If P_4 replays message m_4, then message m_4 is called a *duplicate* message.
Message m_5 is an excellent example of a duplicate message. No matter what, the receiver of m_5 will receive a duplicate m_5 message.
Messages – example

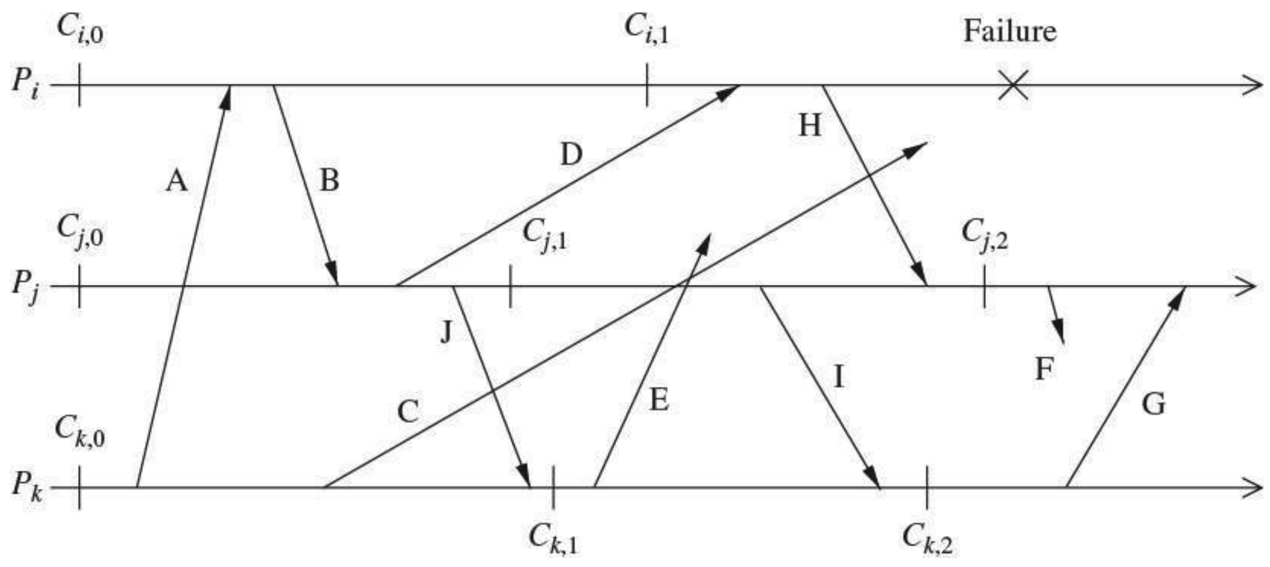


- In-transit – $m1$, $m2$
- Lost – $m1$
- Delayed – $m1$, $m5$
- Orphan – none
- Duplicated – $m4$, $m5$

4.3 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 P_i, P_j, and P_k, connected through a communication network. The processes communicate solely by exchanging messages over fault-free, FIFO communication channels. Processes P_i, P_j , and P_k have taken check-points $\{C_{i\ 0}, C_{i\ 1}\}$, $\{C_{j\ 0}, C_{j\ 1},$

$C_{j\,2}$}, and {$C_{k\,0}$, $C_{k\,1}$}, respectively, and these processes have exchanged messages A to J as shown in Figure.



- 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 to checkpoint $C_{i,1}$ created an orphan message H

- Orphan message I is created due to the roll back of process P_j to checkpoint $C_{j\,1}$
- 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 process P_j , but the receive event has been undone at process P_i.

- Lost messages can be handled by having processes keep a message log of all the sent messages

Messages E, F: delayed orphan messages. After resumingexecution from their checkpoints, processes will generate both of these messages

4.4 Checkpoint-based recovery

In the checkpoint-based recovery approach, the state of each process and the communication channel is check pointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints. It does not rely on the PWD assumption, and so does not need to detect, log, or replay non-deterministic events. Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback recovery. However, checkpoint-based rollback recovery does not guarantee that pre-failure execution can be deterministically regenerated after a rollback. There-fore, checkpoint-based rollback recovery may not be suitable for applications that require frequent interactions with the outside world.

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

- *uncoordi-nated checkpointing*,
- *coordinated checkpointing*, and
- *communication-induced checkpointing*

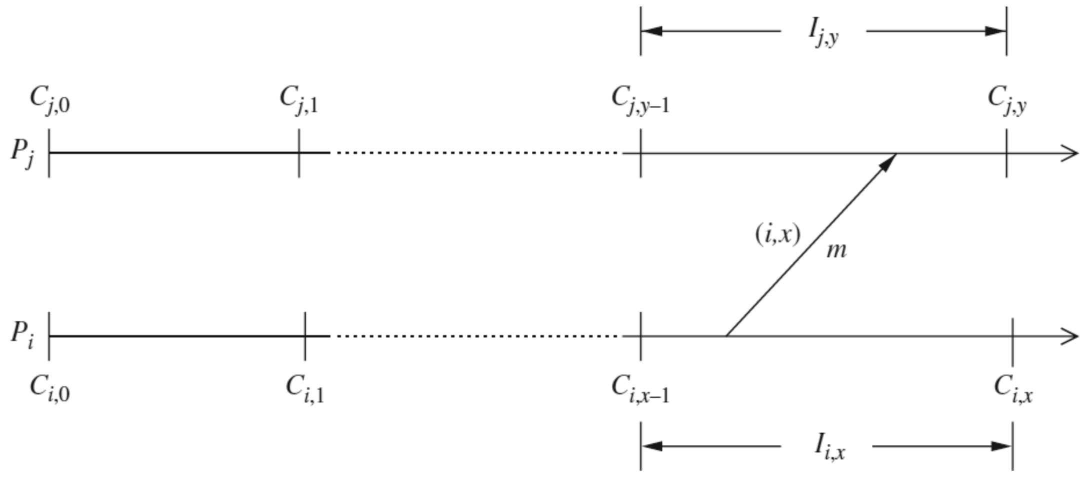
Uncoordinated Checkpointing

Each process has autonomy in deciding when to take checkpoints

- Advantages
  — The lower runtime overhead during normal execution
- **Disadvantages**
  — Domino effect during a recovery
  — Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints
  — Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm
  — Not suitable for application with frequent outputcommits
- **The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation**

Direct dependency tracking technique

Let $C_{i\,x}$ be the *x*th checkpoint of process P_i, where i is the process i.d. and x is the checkpoint index (we assume each process P_i starts its execution with an initial checkpoint $C_{i\,0}$). Let $I_{i\,x}$ denote the *checkpoint interval* or simply *interval* between checkpoints $C_{i\,x-1}$ and $C_{i\,x}$.



- When a failure occurs, the recovering process initiates rollback by broad-casting 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 activ-ities so that all local checkpoints form a consistent global state. Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint.
- Also, coordinated checkpointing requires each process to maintain only one checkpoint on the stable storage, reducing the storage overhead and eliminating the need for garbage collection. The main disadvantage of this method is that large latency is involved in committing output, as a global checkpoint is needed before a message is sent to the OWP. Also, delays and overhead are involved everytime a new global checkpoint is taken.
- If perfectly synchronized clocks were available at processes, the following simple method can be used for checkpointing: all processes agree at what instants of time they will take checkpoints, and the clocks at processes trigger the local checkpointing actions at all processes. Since perfectly synchronized clocks are not available, the following approaches are used to guarantee checkpoint consistency: either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking.

Blocking Checkpointing

- A straightforward approach to coordinated checkpointing is to block commu-nications while the checkpointing protocol executes. After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete.
- The coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint. When a process receives this message, it stops its execution, flushes all the communication channels, takes a *tentative* checkpoint, and sends an acknowledgment message back to the coordinator. After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol.
- After receiving the commit message, a process removes the old permanent checkpoint and atomically makes the *tentative* checkpoint permanent and then resumes its execution and exchange of messages with other processes. A problem with this approach is that the computation is blocked during the checkpointing and therefore, non-blocking checkpointing schemes are preferable.

— After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete

—

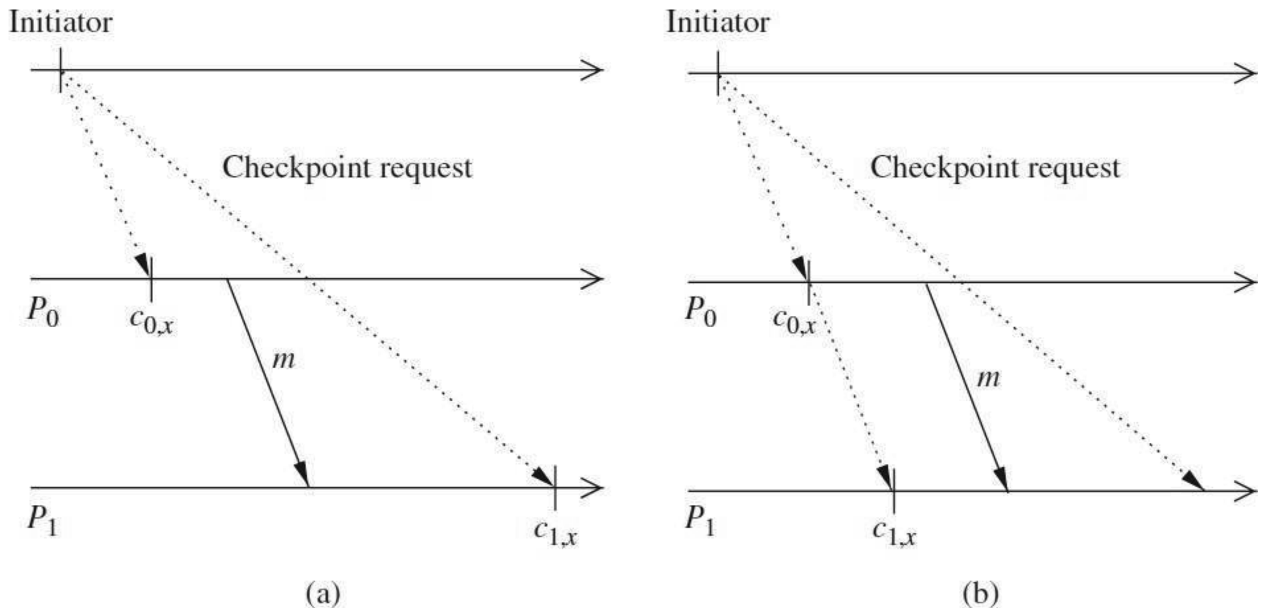
Disadvantages

- the computation is blocked during the checkpointing

Non-blocking Checkpointing

In this approach the processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to pre-vent a process from receiving application messages that could make the checkpoint inconsistent.
Consider the example in Figure (a): message *m* is sent by P_0 *after* receiving a checkpoint request from the checkpoint coordinator. Assume *m* reaches P_1 *before* the checkpoint request. This sit-uation results in an inconsistent checkpoint since checkpoint $c_{1\ x}$ shows the receipt of message *m* from P_0, while checkpoint $c_{0\ x}$ does not show *m* being sent from P_0.

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, as illustrated in Figure 13.6(b).



Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) a solution with FIFO channels

- The processes need not stop their execution while takingcheckpoints
- A fundamental problem in coordinated check pointing is to preventa process from receiving application messages that could make the checkpoint inconsistent.

- **Example (a) : checkpoint inconsistency**

message *m* is sent by P_0 *after* receiving a checkpoint request from the checkpoint coordinator. Assume *m* reaches P_1 *before* the checkpoint request. This situation results in an inconsistent checkpoint since checkpoint $c_{1\ x}$ shows the receipt of message *m* from P_0, while checkpoint $c_{0\ x}$ does not show *m* being sent from P_0.

- **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 thefirst post-checkpoint message

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. Clearly, such checkpointing algorithms will be very attractive. Cao and Singhal showed that it is impossible to design a min-process, non-blocking checkpointing algorithm.

The following type of min-process checkpointing algorithms are possible. 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.

Based on a concept called "**Z-dependency**," Cao and Singhal proved that there does not exist a non-blocking algorithm that will allow a minimum number of processes to take their checkpoints. Here we give only a sketch of the proof and readers are referred to the original source for a detailed proof.

Z-dependency is defined as follows: if a process P_p sends a message to process P_q during its ith checkpoint interval and process P_q receives the message during its jth checkpoint interval, then P_q Z-depends on P_p during P_p's ith checkpoint interval and P_q 's jth checkpoint interval, denoted by $P_p \rightarrow^i_j P_q$. If $P_p \rightarrow^i_j P_q$ and $P_q \rightarrow^j_k P_r$, then P_r transitively Z-depends depends on P_p during P_r 's kth checkpoint interval and P_p's ith checkpoint interval, and this is denoted as $P_p \, {}^* \!\rightarrow^i_k P_r$.

A min process algorithm is one that satisfies the following condition: when a process P_p initiates a new checkpoint and takes checkpoint $C_{p\,i}$, a process P_q takes a checkpoint $C_{q\,j}$ associated with $C_{p\,i}$ if and only if $P_q \, {}^* \!\rightarrow^{j-1}_{i-1} P_p$. In a min-process non-blocking algorithm, process P_p initiates a new checkpoint and takes a checkpoint $C_{p\,i}$ and if a process P_r sends a message m to P_q after it takes a new checkpoint associated with $C_{p\,i}$, then P_q takes a checkpoint $C_{q\,i}$ before processing m if and only if $P_q \, {}^* \!\rightarrow^{j-1}_{i-1} P_p$. According to the min-process definition, P_q takes checkpoint $C_{q\,j}$ if and only if $P_q \, {}^* \!\rightarrow^{j-1}_{i-1} P_p$, but P_q should take $C_{q\,i}$ before processing m. If it takes $C_{q\,j}$ after processing m, m becomes an orphan. Therefore, when a process receives a message m, it must know if the initiator of a new checkpoint transitively Z-depends on it during the previous checkpoint interval. But it has been proved that there is not enough information at the receiver of a message to decide whether the initiator of a new checkpoint transitively Z-depends on the receiver. Therefore, no min-process, non-blocking algorithm exists.

Communication-induced Checkpointing

Communication-induced checkpointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints inde-pendently. Processes may be forced to take additional checkpoints (over and above their autonomous checkpoints), and thus process independence is constrained to guarantee the eventual progress of the recovery line. Communication-induced checkpointing reduces or completely eliminates the useless checkpoints.

• **Two types of checkpoints**

— autonomous and 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

— **(i) model-based checkpointing and**

— **(ii) index-based checkpointing.**

Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and check-points that could result in inconsistent states among the existing checkpoints. \
- A process detects the potential for inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns.
- A forced checkpoint is generally used to prevent the undesirable patterns from occurring. No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggy-backed on application messages. The decision to take a forced checkpoint is done locally using the information available.

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.
- Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint.

4.5 Log-based rollback recovery

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

Deterministic and Non-deterministic events

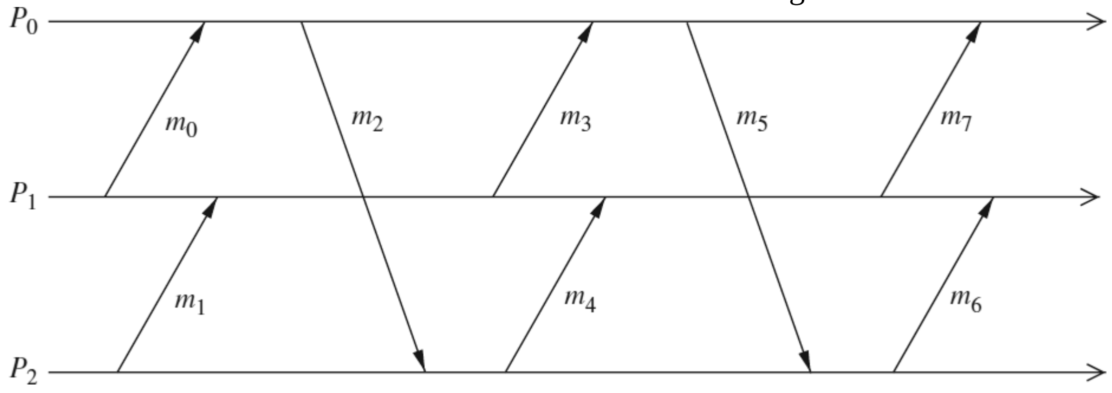
— Non-deterministic events can be the receipt of a message from another process or an event internal to the process

— a message send event is *not* a non-deterministic event.

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

— During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage



No-orphans consistency condition

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

***Depend*(*e*) ->** the set of processes that are affected by a non-deterministic event *e*.This set consists of *p*, and any process whose state depends on the event *e* according to Lamport's *happened before* relation

***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**

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

Pessimistic Logging

Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation

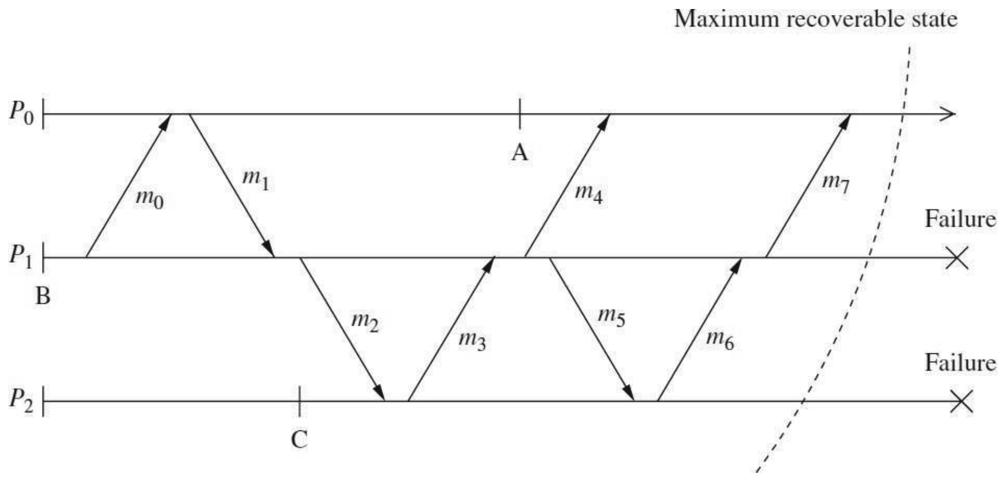
- However, in reality failures are rare

synchronous logging

$$\forall e: \neg \mathbf{Stable(e)} \Rightarrow |\mathbf{Depend(e)}| = 0$$

— if an event has not been logged on the stable storage, then no processca n depend on it.

— stronger than the always-no-orphans condition



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. This guarantees that P_1 and P_2 will repeat exactly their pre-failure execution and re-send the same messages. Hence, once the recovery is complete, both processes will be consistent with the state of P_0 that includes the receipt of message m_7 from P_1. In a pessimistic logging system, the observable state of each process is always recoverable.

Optimistic Logging

- In optimistic logging protocols, processes log determinants *asynchronously* to the stable storage . These protocols optimistically assume that logging will be complete before a failure occurs. Determinants are kept in a volatile log, and are periodically flushed to the stable storage. Thus, optimistic logging does not require the application to block waiting for the determinants to be written to the stable storage, and therefore incurs much less overhead during failure-free execution.
- However, the price paid is more complicated recovery, garbage collection, and slower output commit. If a process fails, the determinants in its volatile log are lost, and the state intervals that were started by the non-deterministic events corresponding to these determinants cannot be recovered.
- Furthermore, if the failed process sent a message during any of the state intervals that cannot be recovered, the receiver of the message becomes an orphan process and must roll back to undo the effects of receiving the message.

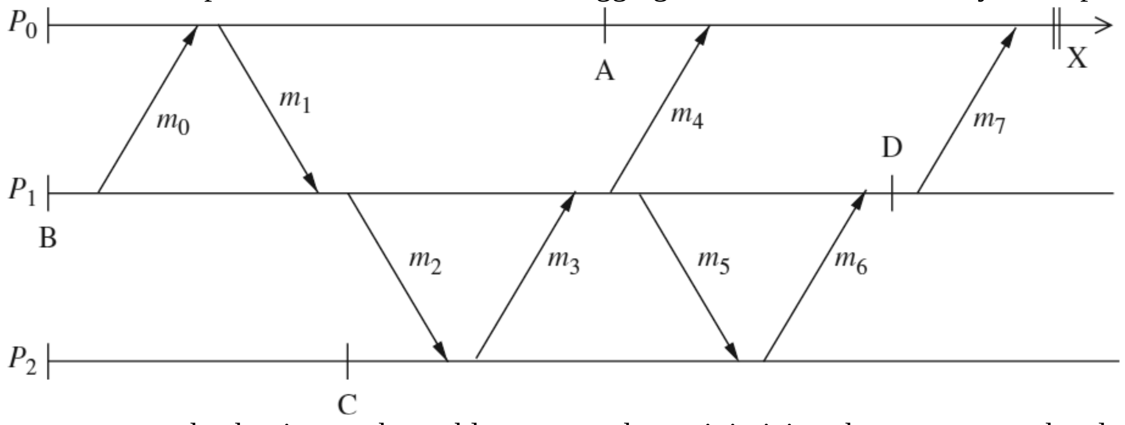
To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution

- Optimistic logging protocols require a non-trivial garbage collect ion 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

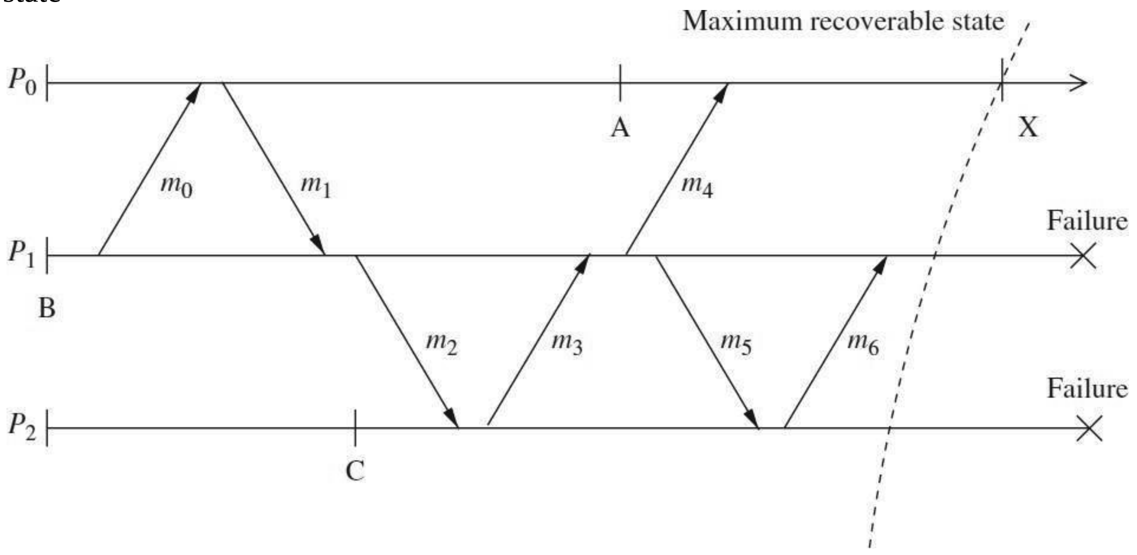
Causal Logging

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. Moreover, causal logging limits the rollback of any failed process to



the most recent checkpoint on the stable storage, thus minimizing the storage overhead and the amount of lost work.

- Make sure that the always-no-orphans property holds
- Each process maintains information about all the events that have causally affected its state



4.6 Koo-Toueg coordinated checkpointing algorithm

- A coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and avoids domino effect and livelock problems during the recovery

- Includes 2 parts: the checkpointing algorithm and the recovery algorithm

<u>4.6.1 Checkpointing algorithm</u>

– Assumptions: FIFO channel, end-to-end protocols, communication failures do not partition the network, single process initiation, no process fails during the execution of the algorithm

<u>Two kinds of checkpoints: permanent and tentative</u>

- Permanent checkpoint: local checkpoint, part of a consistent global checkpoint
- Tentative checkpoint: temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully

<u>Checkpointing algorithm</u>

2 phases

- The initiating process takes a tentative checkpoint and requests all other processes to take tentative checkpoints. Every process can not send messages after taking tentative checkpoint. All processes will finally have the single same decision: do or discard
- All processes will receive the final decision from initiating process and act accordingly

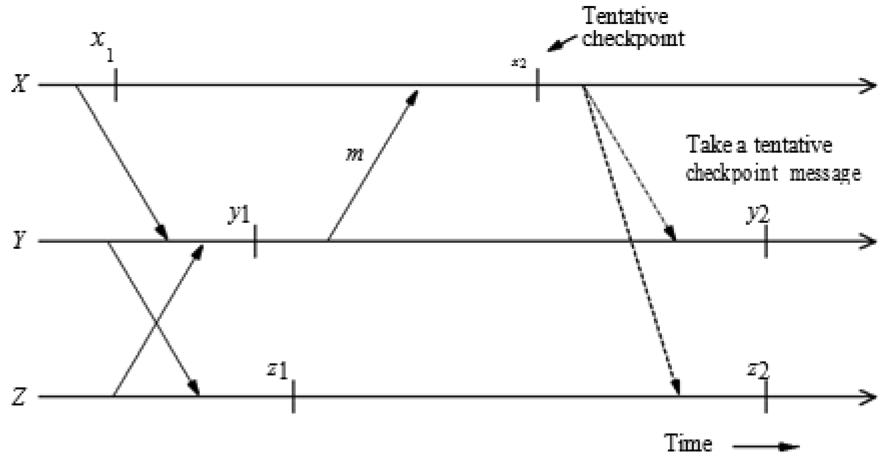
Correctness: for 2 reasons

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

Optimization: maybe not all of the processes need to take checkpoints (if not change since the last checkpoint)

<u>The rollback recovery algorithm</u>

- Restore the system state to a consistent state after a failure with assumptions: single initiator, checkpoint and rollback recovery algorithms are not invoked concurrently



Example of checkpoints taken unnecessarily

<u>2 phases</u>

First phase

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

Second phase

P_i propagates its decision to all the processes. On receiving P_i's decision, a process acts accordingly.

During the execution of the recovery algorithm, a process cannot send messages related to the underlying computation while it is waiting for P_i's decision.

Correctness

All processes restart from an appropriate state because, if they decide to restart, they resume execution from a consistent state (the checkpointing algorithm takes a consistent set of checkpoints).

An optimization

The above recovery protocol causes all processes to roll back irrespective of whether a process needs to roll back or not. Consider the example shown in Figure. In the event of failure of process X, the above protocol will require processes X, Y, and Z to restart from checkpoints x_2, y_2, and z_2, respectively. However, note that 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.

4.7 Juang-Venkatesan algorithm for asynchronous checkpointing and recovery

- Assumptions: communication channels are reliable, delivery messages in FIFO order, infinite buffers, 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

Asynchronous checkpointing:

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

Recovery algorithm

Notation and data structure

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

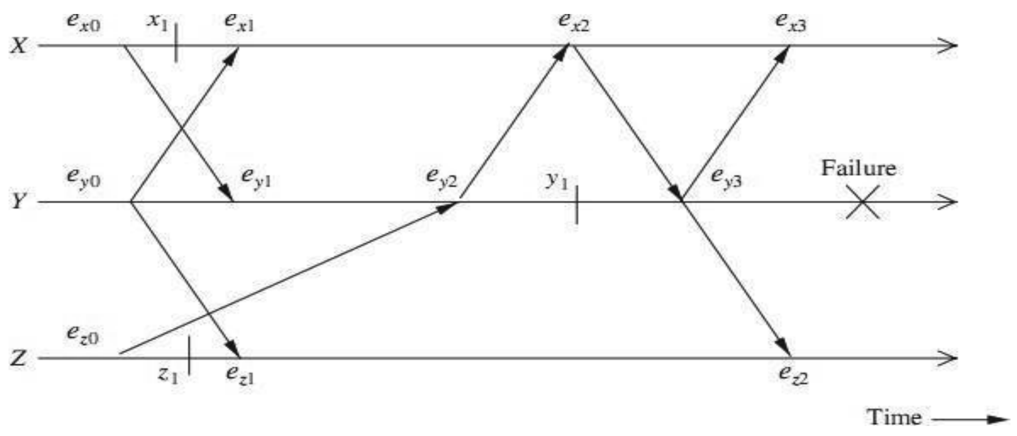
- **$RCVD_{i\leftarrow j}$ $CkPt_i$** represents the number of messages received by processor p_i from processor p_j , from the beginning of the computation until the checkpoint $CkPt_i$.

- **$SENT_{i\rightarrow j}$ $CkPt_i$** represents the number of messages sent by processor p_i to processor p_j , from the beginning of the computation until the checkpoint $CkPt_i$.

Idea:

- From the set of checkpoints, find a set of consistent checkpoints
- Doing that based on the number of messages sent and received

Example



<u>Procedure RollBack_Recovery:</u>

processor p_i executes the following:

```
STEP (a)

if processor $p_i$ is recovering after a failure then $CkPt_i$ = latest event
  logged in the stable storage
else

  $CkPt_i$ = latest event that took place in $p_i$ {The latest event at $p_i$ can be either in stable 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 $SENT_{i\rightarrow j}$ $CkPt_i$

    send a ROLLBACK i $SENT_{i\rightarrow j}$ $CkPt_i$ message to $p_j$ end for

  for every ROLLBACK j c  message received from a neighbor j do
```

```
        if $RCVD_{i \leftarrow j}$ $CkPt_i$ > c {Implies the presence of orphan messages}
          then
          find the latest event e such that $RCVD_{i \leftarrow j}$ e = c {Such an event e may be in the
          volatile storage or stable storage.}

          $CkPt_i$ = e
        end if

     end for

   end for{for k}
```

Consensus and Agreement

Assumptions

Assumptions underlying our study of agreement algorithms:

Failure models Among the n processes in the system, at most f processes can be faulty. A faulty process can behave in any manner allowed by the failure model assumed.

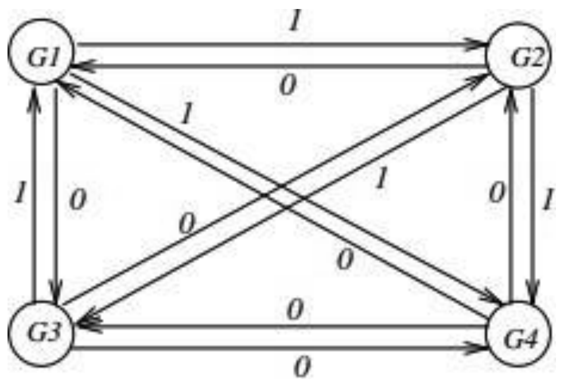
Synchronous/asynchronous communication If a failure-prone process chooses to send a message to process P_i but fails, then P_i cannot detect the non-arrival of the message in an asynchronous system because this scenario is indistinguishable from the scenario in which the message takes a very long time in transit.

- **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. This assumption is important – because even with Byzantine behavior, even though the payload of the message can contain fictitious data sent by a malicious sender, the underlying network layer protocols can reveal the true identity of the sender process.
- **Channel reliability** The channels are reliable, and only the processes may fail (under one of various failure models). This is a simplifying assumption in our study. As we will see even with this simplifying assumption, the agreement problem is either unsolvable, or solvable in a complex manner.

- **Authenticated vs. non-authenticated messages** In our study, we will be dealing only with *unauthenticated* messages. With unauthenticated mes-sages, 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, and (ii) it can also tamper with the contents of a received message before relaying it. An unauthenticated message is also called an *oral* message or an *unsigned* message.

- **Agreement variable** The agreement variable may be boolean or multi-valued, and need not be an integer. When studying some of the more complex algorithms, we will use a boolean variable. This simplifying assumption does not affect the results for other data types, but helps in the abstraction while presenting the algorithms.



Problem Specifications

The Byzantine agreement and other problems

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.

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

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 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 (all processes have an initial value)

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

Validity: If process i is non-faulty and its initial value is v_i, then all non-faulty processes agree on v_i as the i th element of the array A. If process j is 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. These problems are equivalent to one another! Show using reductions.

Overview of Results

Failure mode	Synchronous system (message-passing and shared memory)	Asynchronous system (message-passing and shared memory)
No failure	agreement attainable; common knowledge also attainable	agreement attainable; concurrent common knowledge attainable
Crash failure	agreement attainable $f < n$ processes $\Omega(f + 1)$ rounds	agreement not attainable
Byzantine failure	agreement attainable $f \leq \lfloor (n - 1)/3 \rfloor$ Byzantine processes $\Omega(f + 1)$ rounds	agreement not attainable

Table:Overview of results on agreement. f denotes number of failure-prone processes. n is the total number of processes.

Agreement in a failure-free system (synchronous or asynchronous)

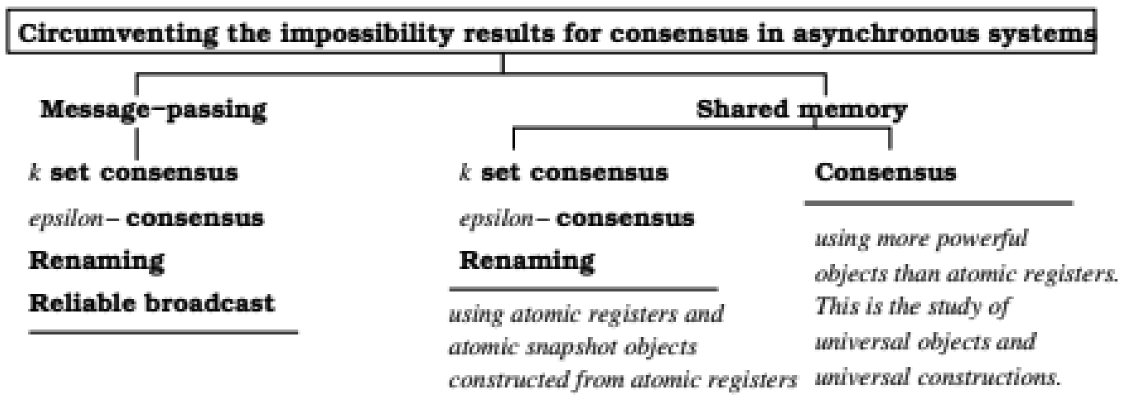
In a failure-free system, consensus can be attained in a straightforward manner

Some Solvable Variants of the Consensus Problem in Async Systems

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 less than k
s-agreement	crash failures $n \geq 5f + 1$ (MP)	values agreed upon are within s of eachother
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

Table:Some solvable variants of the agreement problem in asynchronous system. The overhead bounds are for the given algorithms, and not necessarily tight bounds for the problem

Solvable Variants of the Consensus Problem in Async Systems

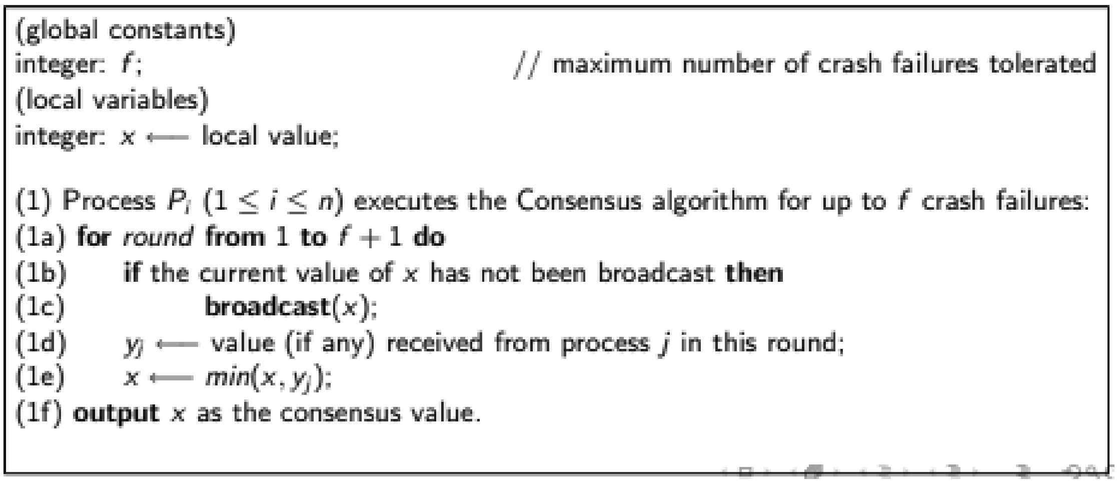

Circumventing the impossibility results for consensus in asynchronous systems
Message−passing
k set consensus
epsilon− consensus
Renaming
Reliable broadcast
Shared memory
k set consensus
epsilon− consensus
Renaming
using atomic registers and atomic snapshot objects constructed from atomic registers
Consensus
using more powerful objects than atomic registers. This is the study of universal objects and universal constructions.

Agreement in a failure-free system

Agreement in (message-passing) synchronous systems with failures

Consensus Algorithm for Crash Failures (MP, synchronous)

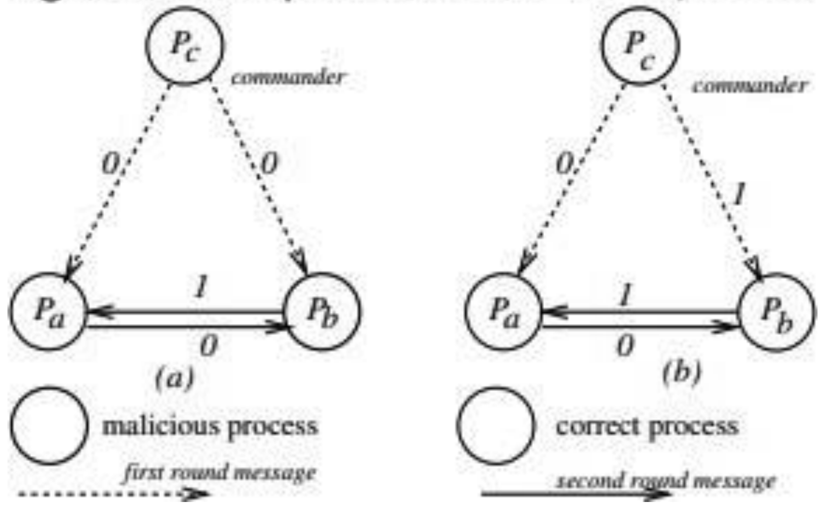
- Up to f ($< n$) crash failures possible.
- In $f + 1$ rounds, at least one round has no failures.
- Now justify: agreement, validity, termination conditions are satisfied.
- Complexity: $O(f + 1)n^2$ messages
- $f + 1$ is lower bound on number of rounds



```
(global constants)
integer: $f$;                          // maximum number of crash failures tolerated
(local variables)
integer: $x \longleftarrow$ local value;

(1) Process $P_i$ ($1 \leq i \leq n$) executes 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)     $y_j \longleftarrow$ value (if any) received from process $j$ in this round;
(1e)     $x \longleftarrow min(x, y_j)$;
(1f) output $x$ as the consensus value.
```

Upper Bound on Byzantine Processes (sync)

Agreement impossible when $f = 1, n = 3$.



Taking simple majority decision does not help because loyal commander P_a cannot distinguish between the possible scenarios (a) and (b); hence does not know which action to take.

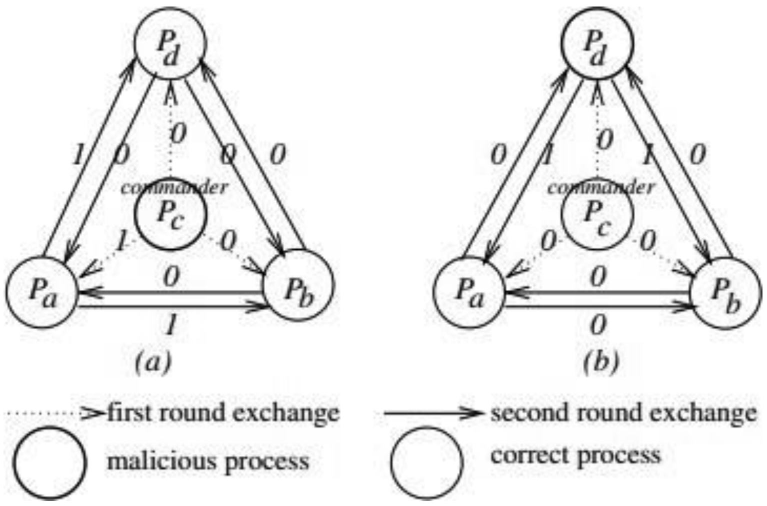
Proof using induction that problem solvable if $f \leq \lfloor \frac{n-1}{3} \rfloor$.

Byzantine agreement tree algorithm: exponential (synchronous system)

Recursive formulation

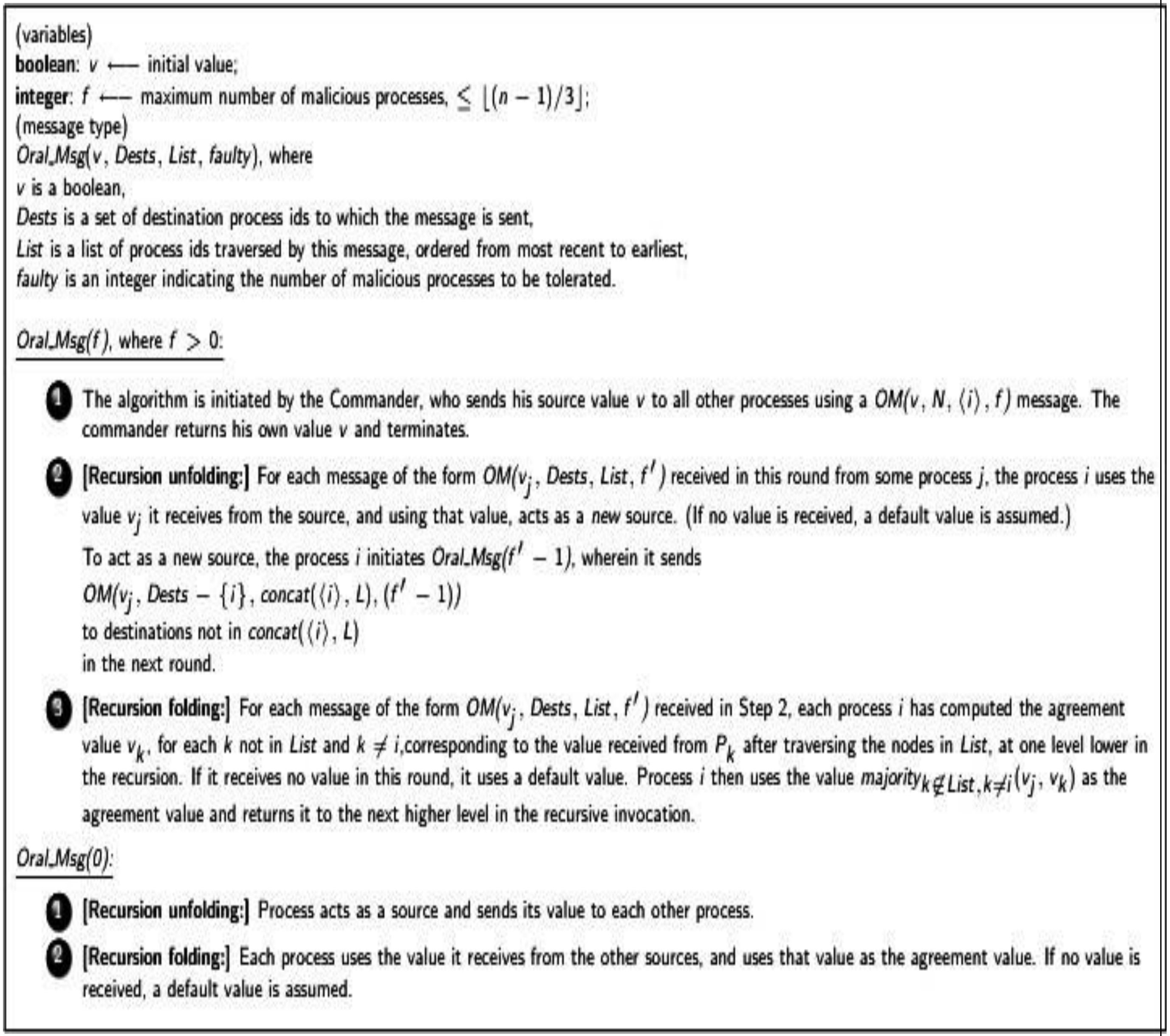
- In the first round, the commander P_c sends its value to the other three lieutenants, as shown by dotted arrows.
- In the second round, each lieutenant relays to the other two lieutenants, the value it received from the commander in the first round. At the end of the second round, a lieutenant takes the majority of the values it received (i) directly from the commander in the first round, and (ii) from the other two lieutenants in the second round.
- The majority gives a correct estimate of the commander’s value.

Consensus Solvable when f = 1, n = 4



- There is no ambiguity at any loyal commander, when taking majority decision
- Majority decision is over 2nd round messages, and 1st round message received directly from commander-in-chief process.

Byzantine Generals (recursive formulation), (sync, msg-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.

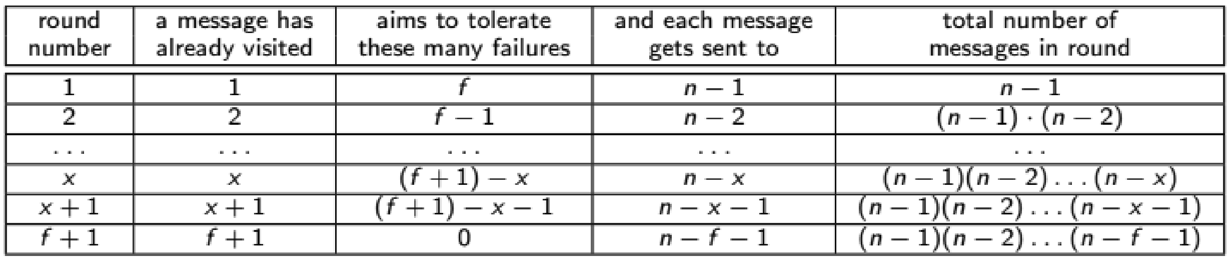
Oral_Msg(f), where $f > 0$:

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.

Relationship between # Messages and Rounds

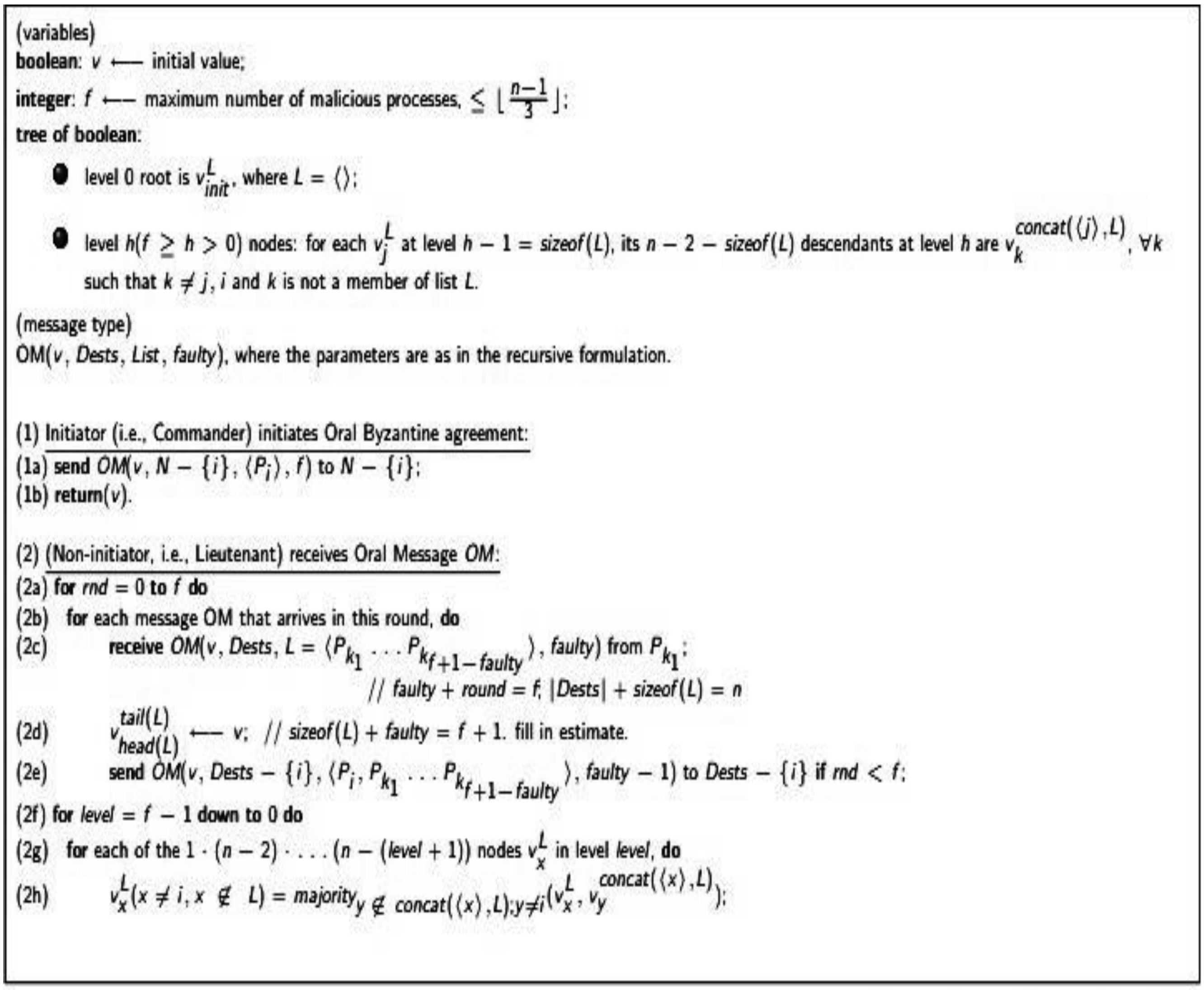


round number	a message has already visited	aims to tolerate these many failures	and each message gets sent to	total number of messages in round
1	1	f	$n-1$	$n-1$
2	2	$f-1$	$n-2$	$(n-1)\cdot(n-2)$
...	...	...	...	...
x	x	$(f+1)-x$	$n-x$	$(n-1)(n-2)\ldots(n-x)$
$x+1$	$x+1$	$(f+1)-x-1$	$n-x-1$	$(n-1)(n-2)\ldots(n-x-1)$
$f+1$	$f+1$	0	$n-f-1$	$(n-1)(n-2)\ldots(n-f-1)$

Relationships between messages and rounds in the Oral Messages algorithm for Byzantine agreement.

Complexity: f + 1 rounds, exponential amount of space, and (n − 1) + (n − 1)(n − 2) + . . . + (n − 1)(n − 2)..(n − f − 1)messages

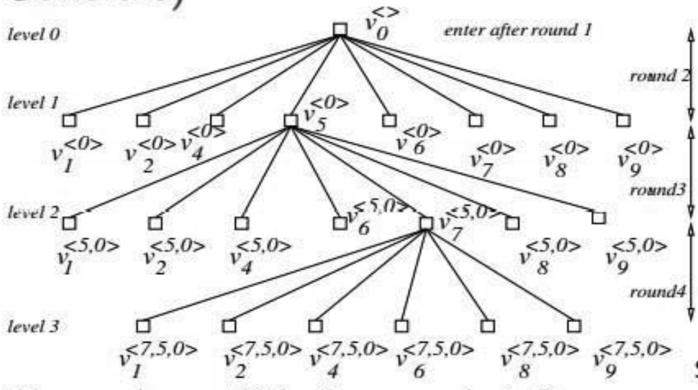
Bzantine Generals (iterative formulation), Sync, Msg-passing



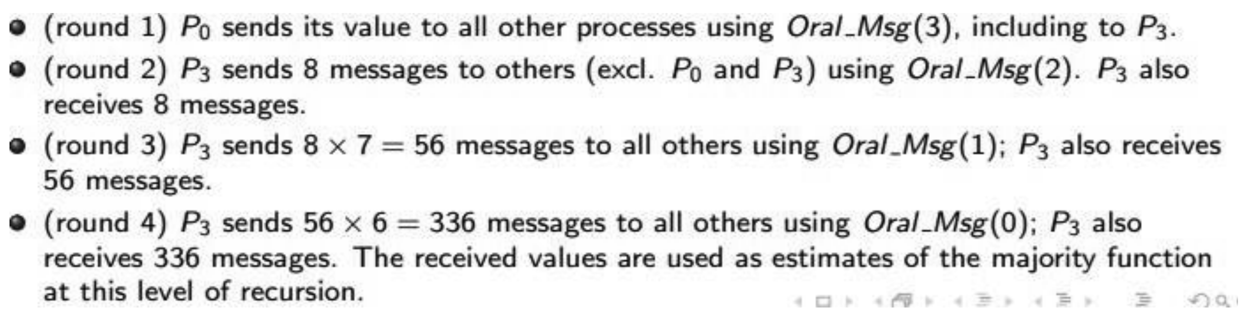
```
(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;
tree of boolean:
  ● level 0 root is $v_{init}^{L}$, where $L = \langle\rangle$;
  ● level $h(f \geq h > 0)$ nodes: for each $v_j^L$ at level $h - 1 = sizeof(L)$, its $n - 2 - sizeof(L)$ descendants at level $h$ are $v_k^{concat(\langle j \rangle, L)}$, $\forall k$
    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)     $v_{head(L)}^{tail(L)} \longleftarrow v$;  // $sizeof(L) + faulty = f + 1$. 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 $v_x^L$ in level $level$, do
(2h)     $v_x^L(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v_x^L, v_y^{concat(\langle x \rangle, L)})$;
```

Tree Data Structure for Agreement Problem (Byzantine Generals)

Tree Data Structure for Agreement Problem (Byzantine Generals)

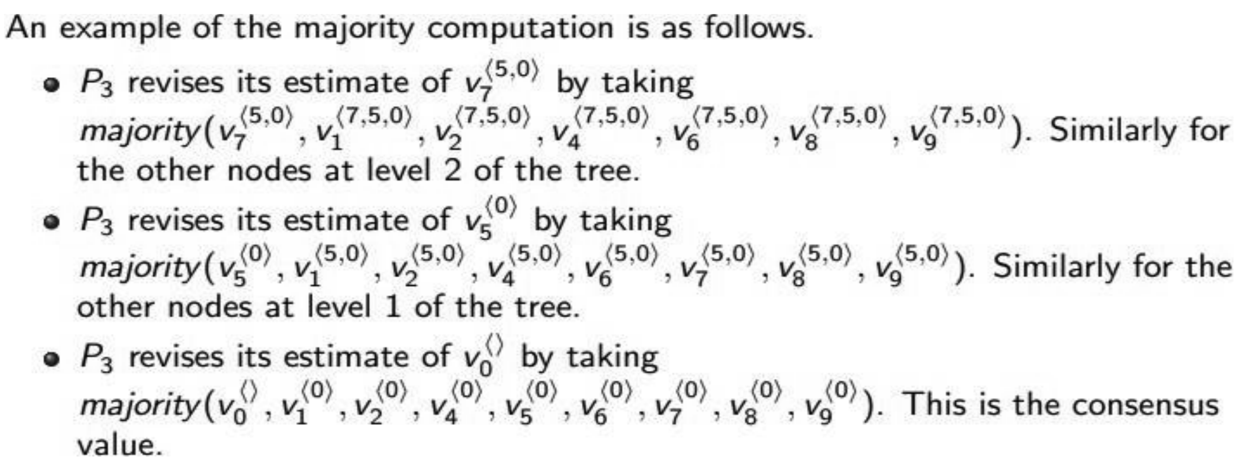

level 0
$v_0^{<>}$
enter after round 1
round 2
level 1
$v_1^{<0>}$ $v_2^{<0>}$ $v_4^{<0>}$ $v_5^{<0>}$ $v_6^{<0>}$ $v_7^{<0>}$ $v_8^{<0>}$ $v_9^{<0>}$
round3
level 2
$v_1^{<5,0>}$ $v_2^{<5,0>}$ $v_4^{<5,0>}$ $v_6^{<5,0>}$ $v_7^{<5,0>}$ $v_8^{<5,0>}$ $v_9^{<5,0>}$
round4
level 3
$v_1^{<7,5,0>}$ $v_2^{<7,5,0>}$ $v_4^{<7,5,0>}$ $v_6^{<7,5,0>}$ $v_8^{<7,5,0>}$ $v_9^{<7,5,0>}$

Some branches of the tree at P_3. In this example, $n = 10$, $f = 3$, commander is P_0.



- (round 1) P_0 sends its value to all other processes using $Oral\_Msg(3)$, including to P_3.
- (round 2) P_3 sends 8 messages to others (excl. P_0 and P_3) using $Oral\_Msg(2)$. P_3 also receives 8 messages.
- (round 3) P_3 sends $8 \times 7 = 56$ messages to all others using $Oral\_Msg(1)$; P_3 also receives 56 messages.
- (round 4) P_3 sends $56 \times 6 = 336$ messages to all others using $Oral\_Msg(0)$; P_3 also receives 336 messages. The received values are used as estimates of the majority function at this level of recursion.

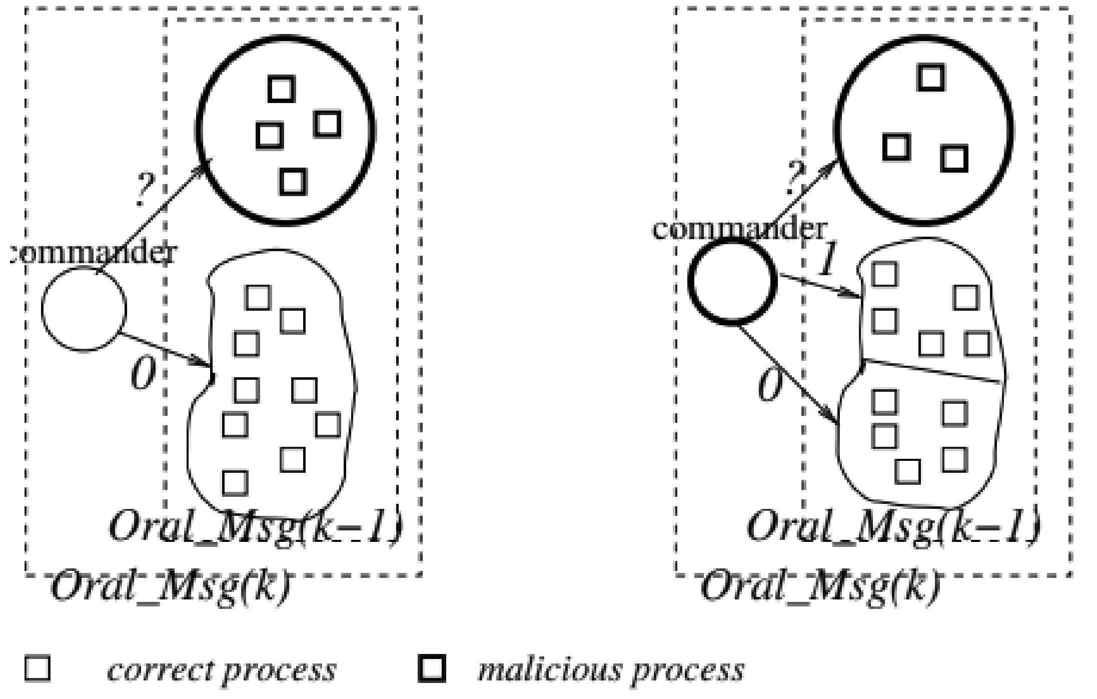
<u>Exponential Algorithm: An example</u>



An example of the majority computation is as follows.

- P_3 revises its estimate of $v_7^{\langle 5,0\rangle}$ by taking $majority(v_7^{\langle 5,0\rangle}, v_1^{\langle 7,5,0\rangle}, v_2^{\langle 7,5,0\rangle}, v_4^{\langle 7,5,0\rangle}, v_6^{\langle 7,5,0\rangle}, v_8^{\langle 7,5,0\rangle}, v_9^{\langle 7,5,0\rangle})$. Similarly for the other nodes at level 2 of the tree.
- P_3 revises its estimate of $v_5^{\langle 0\rangle}$ by taking $majority(v_5^{\langle 0\rangle}, v_1^{\langle 5,0\rangle}, v_2^{\langle 5,0\rangle}, v_4^{\langle 5,0\rangle}, v_6^{\langle 5,0\rangle}, v_7^{\langle 5,0\rangle}, v_8^{\langle 5,0\rangle}, v_9^{\langle 5,0\rangle})$. Similarly for the other nodes at level 1 of the tree.
- P_3 revises its estimate of $v_0^{\langle\rangle}$ by taking $majority(v_0^{\langle\rangle}, v_1^{\langle 0\rangle}, v_2^{\langle 0\rangle}, v_4^{\langle 0\rangle}, v_5^{\langle 0\rangle}, v_6^{\langle 0\rangle}, v_7^{\langle 0\rangle}, v_8^{\langle 0\rangle}, v_9^{\langle 0\rangle})$. This is the consensus value.

Impact of a Loyal and of a Disloyal Commander



(a) (b)

The effects of a loyal or a disloyal commander in a system with n = 14 and f = 4. The subsystems that need to tolerate k and k − 1 traitors are shown for two cases. (a) Loyal commander.
(b) No assumptions about commander.

(a) the commander who invokes Oral Msg(x) is loyal, so all the loyal processes have the same estimate. Although the subsystem of 3x processes has x malicious processes, all the loyal processes have the same view to begin with. Even if this case repeats for each nested invocation of Oral Msg, even after x rounds, among the processes, the loyal processes are in a simple majority, so the majority function works in having them maintain the same common view of the loyal commander's value.
(b) the commander who invokes Oral Msg(x) may be malicious and can send conflicting values to the loyal processes. The subsystem of 3x processes has x − 1 malicious processes, but all the loyal processes do not have the same view to begin with.

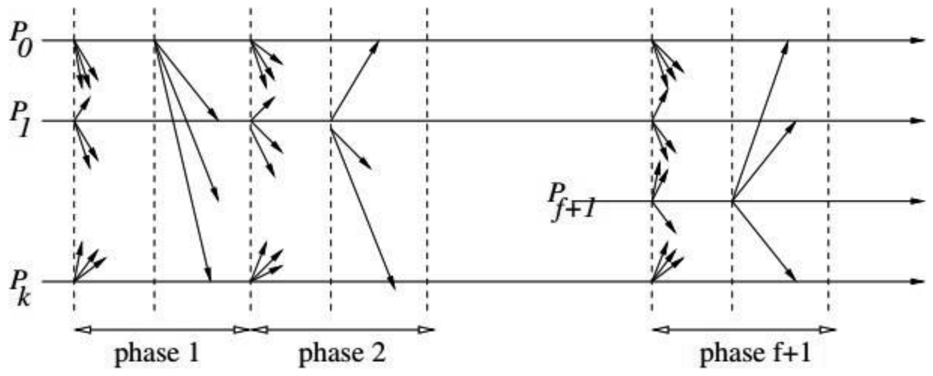
The Phase King Algorithm

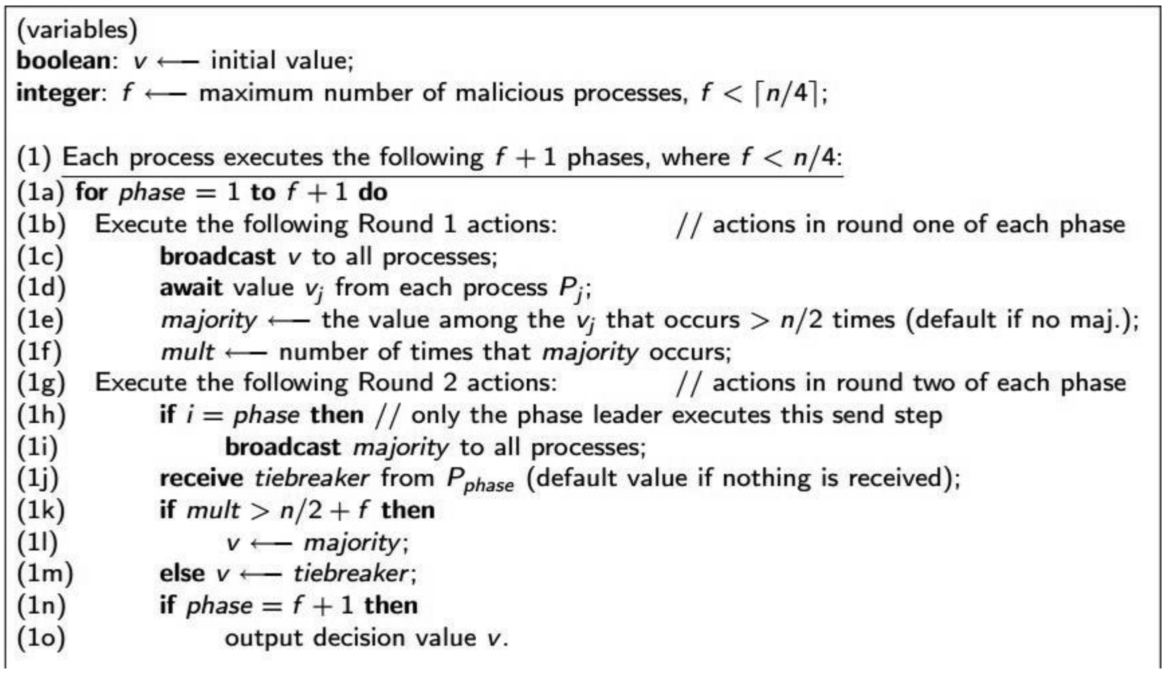
Operation

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

in 1st round, each process sends its estimate to all other processes.

in 2nd round, the "Phase king" process arrives at an estimate based on the values it received in 1st round, and broadcasts its new estimate to all others.





```
(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $f < \lceil n/4 \rceil$;

(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 $\longleftarrow$ the value among the $v_j$ that occurs $> n/2$ times (default if no maj.);
(1f)       mult $\longleftarrow$ 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)           $v \longleftarrow$ majority;
(1m)       else $v \longleftarrow$ tiebreaker;
(1n)       if phase = $f + 1$ then
(1o)           output decision value $v$.
```

(f + 1) phases, (f + 1)[(n − 1)(n + 1)] messages, and can tolerate up to $f < |n/4|$ malicious processes

<u>Correctness Argument</u>

Among f + 1 phases, at least one phase k where phase-king is non-malicious.

In phase k, all non-malicious processes P_i and P_j will have same estimate of consensus value as P_k does.

- **P_i and P_j use their own majority values (Hint: $\Longrightarrow$ P_i's *mult* $> n/2 + f$)**
- **P_i uses its majority value; P_j uses phase-king's tie-breaker value. (Hint: P_i"s *mult* $> n/2 + f$, P_j's *mult* $> n/2$ for same value)**
- **P_i and P_j use the phase-king's tie-breaker value. (Hint: In the phase in which P_k is non-malicious, it sends same value to P_i and P_j)**

In all 3 cases, argue that P_i and P_j end up with same value as estimate

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