

Checkpointing & Rollback Recovery

Chapter 13

Anh Huy Bui
Jason Wiggs
Hyun Seok Roh

Introduction

- Rollback recovery protocols
 - restore the system back to a consistent state after a failure
 - 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 processes that 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-free operation
- Rollback propagation
 - the dependencies may force some of the processes that did not fail to roll back
 - This phenomenon is called “*domino effect*”

Introduction

- If each process takes its checkpoints independently, then the system can not 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 a system-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-deterministic events
 - relies on piecewise deterministic (PWD) assumption

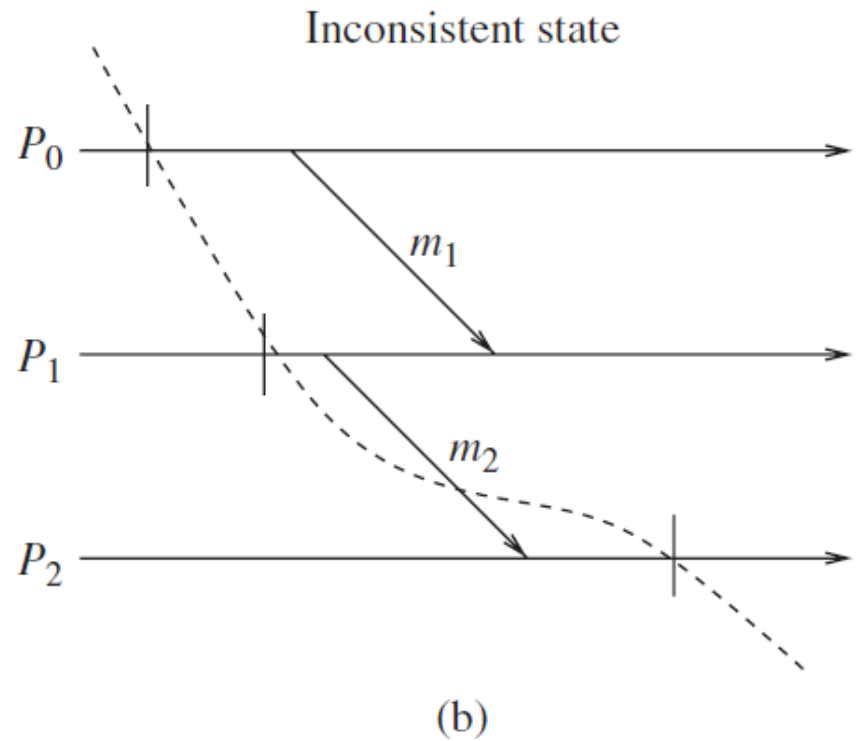
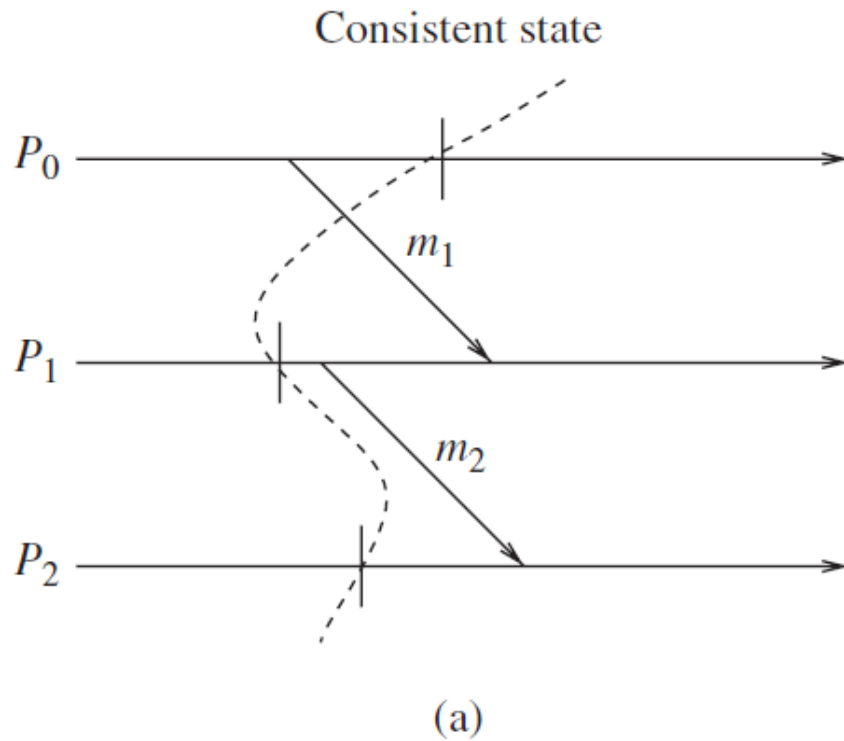
A local checkpoint

- 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
- $C_{i,k}$
 - The k th local checkpoint at process P_i
- $C_{i,0}$
 - A process P_i takes a checkpoint $C_{i,0}$ before it starts execution

Consistent states

- A global state of a distributed system
 - 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 distributed computation
 - if a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message
- A global checkpoint
 - a set of local checkpoints, one from each process
- A consistent global checkpoint
 - 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



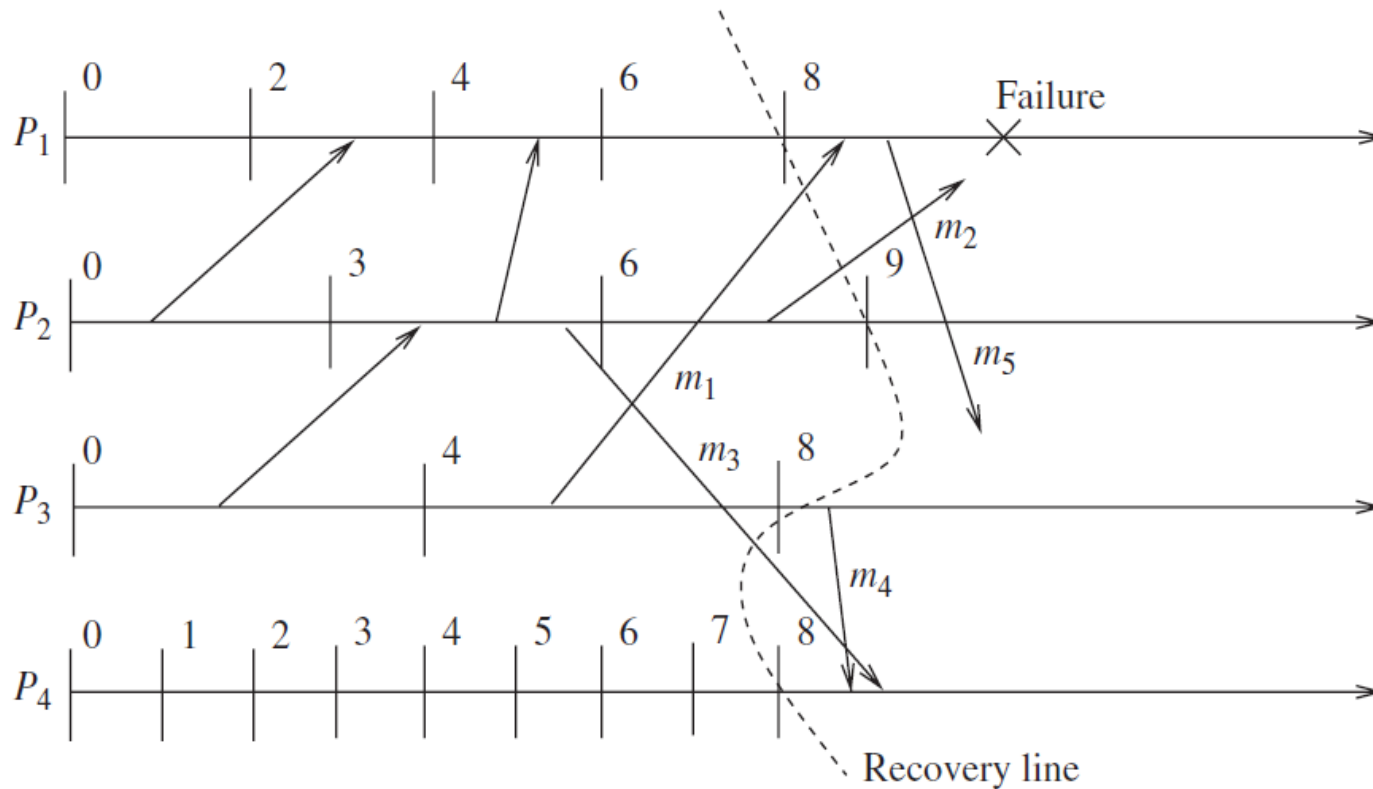
Interactions with outside world

- A distributed system often interacts with the outside world to receive input data or deliver the outcome of a computation
- Outside World Process (OWP)
 - a special process that interacts with the rest of the system through message passing
- A common approach
 - save each input message on the stable storage before allowing the application program to process it
- Symbol “||”
 - An interaction with the outside world to deliver the outcome of a computation

Messages

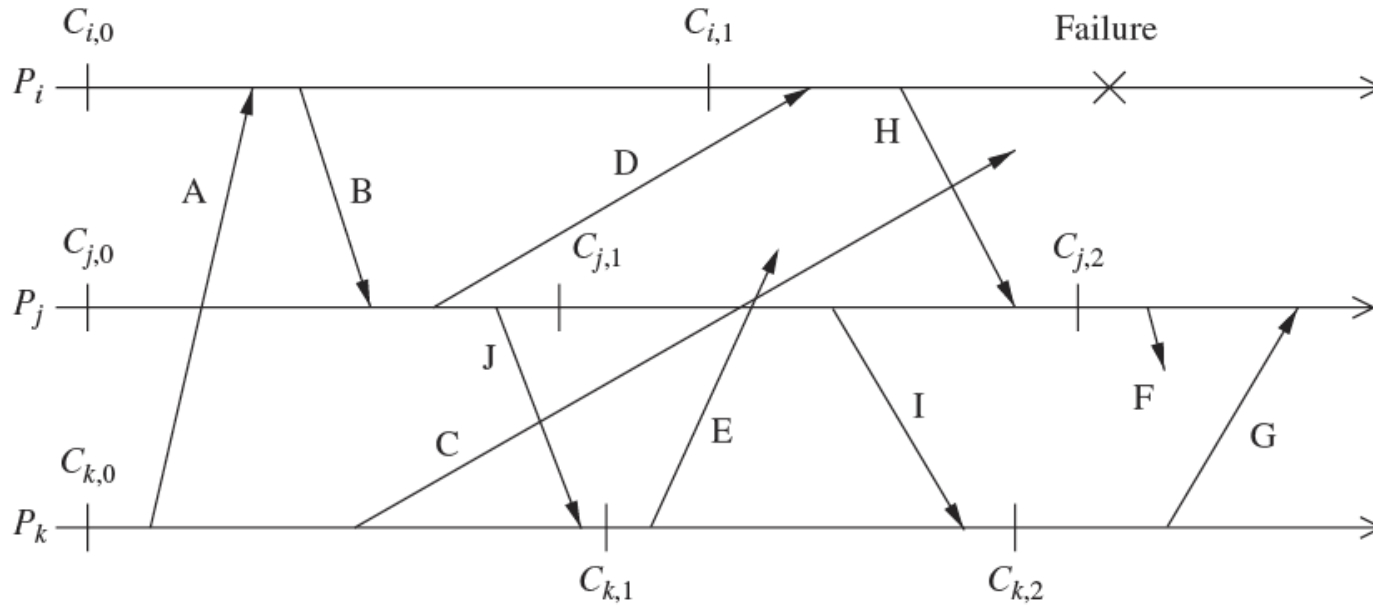
- In-transit message
 - messages that have been sent but not yet received
- Lost messages
 - messages whose ‘send’ is done but ‘receive’ is undone due to rollback
- Delayed messages
 - messages whose ‘receive’ is not recorded because the receiving process was either down or the message arrived after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

Messages – example



- In-transit
 - m_1, m_2
- Lost
 - m_1
- Delayed
 - m_1, m_5
- Orphan
 - none
- Duplicated
 - m_4, m_5

Issues in failure recovery



- 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 : $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

Issues in failure recovery

- The rollback of process P_i 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 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 resuming execution from their checkpoints, processes will generate both of these messages

Checkpoint-based Recovery

Y. V. Lokeswari

**Reference: Kshemkalyani, Ajay D., and Mukesh Singhal.
Distributed computing: principles,
algorithms, and systems. Cambridge University Press, 2011.**

Overview

- Checkpoint –based Recovery
 - Uncoordinated Checkpointing
 - Direct Dependency Tracking Technique.
 - Coordinated Checkpointing
 - Blocking Coordinated Checkpointing
 - Non-Blocking Coordinated Checkpointing
 - Communication-induced Checkpointing
 - Model-based Checkpointing
 - Index-based Checkpointing

Checkpoint-based Recovery

- In the checkpoint-based recovery approach, the **state of each process and the communication channel is checkpointed** frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints.
- Checkpoint-based protocols are therefore **less restrictive and simpler to implement** than log-based rollback recovery.
- Checkpoint-based rollback recovery does not guarantee that **prefailure execution can be deterministically regenerated after a rollback**.
- Checkpoint-based rollback recovery may **not be suitable for applications** that **require frequent interactions with the outside world**.

Uncoordinated checkpointing

- In uncoordinated checkpointing, each process has **autonomy in deciding when to take checkpoints**. This eliminates the synchronization overhead as there is **no need for coordination between processes** and it allows processes to take checkpoints when it is most convenient or efficient.
- **Advantages:**
 - **Lower runtime** overhead during normal execution, because **no coordination** among processes is necessary.
 - **Autonomy in taking checkpoints** also allows each process to select appropriate checkpoints positions.

Uncoordinated checkpointing

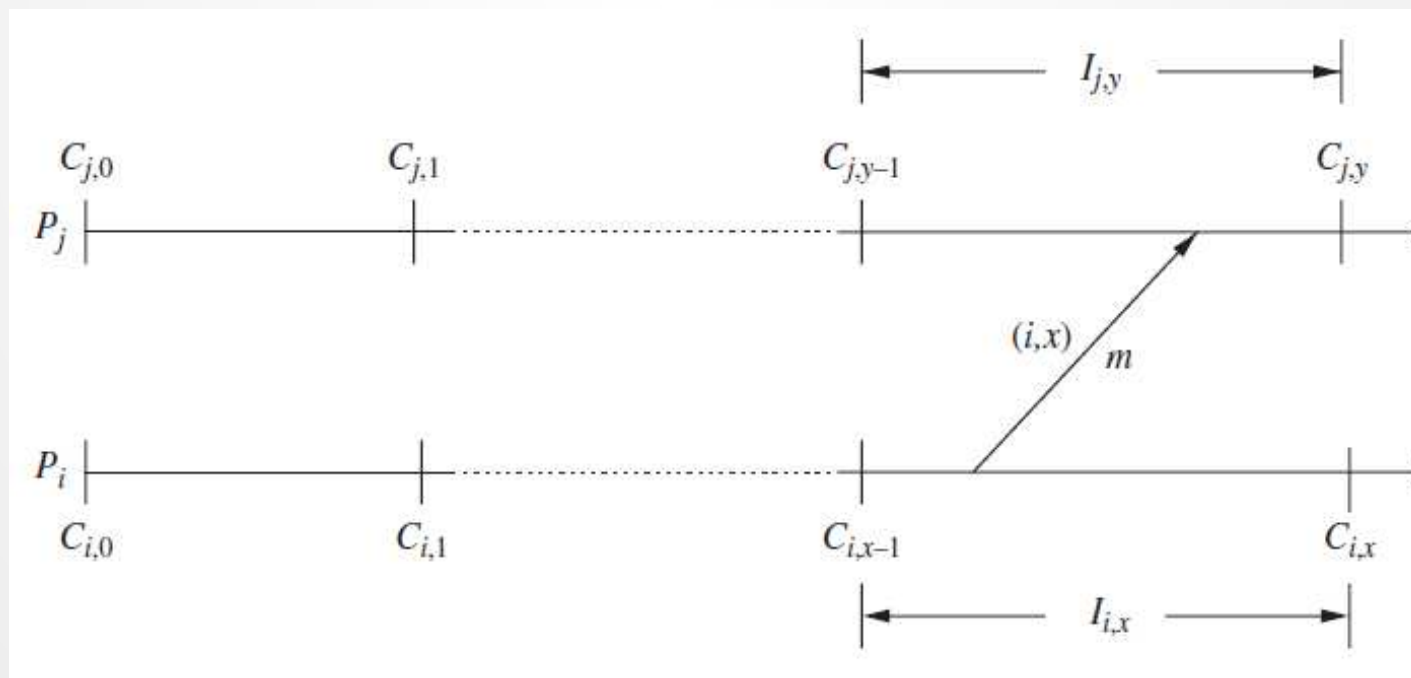
- **Disadvantages:**
 - The possibility of **the domino effect** during a recovery, which may cause the loss of a large amount of useful work.
 - **Recovery from a failure is slow** because processes need to iterate to find a consistent set of checkpoints.
 - Checkpoints taken by a process may be ***useless checkpoints***
 - Uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to **periodically invoke a garbage collection algorithm** to reclaim the checkpoints that are no longer required.
 - **Not suitable for applications with frequent output commits** because these require global coordination to compute the recovery line

Uncoordinated checkpointing

- To determine a consistent global checkpoint during recovery, the processes **record the dependencies among their checkpoints**.

Direct Dependency Tracking Technique

- Assume each process P_i starts its execution with an initial checkpoint $C_{i,0}$
- $I_{i,x}$: checkpoint interval, interval between $C_{i,x-1}$ and $C_{i,x}$
- When P_j receives a message m during $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes $C_{j,y}$



Uncoordinated checkpointing

Direct Dependency Tracking Technique

- 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.
- Receiving process stops its execution and *replies with the dependency information* saved on the stable storage as well as with the dependency information.
- 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.
- **Advantages**
 - Coordinated checkpointing **simplifies recovery** and is **not susceptible to the domino effect**, since every process always restarts from its most recent checkpoint
 - 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**.
- **Disadvantages:**
 - **Large latency** is involved in **committing output**, as a global checkpoint is needed before a message is sent to the OWP.

Coordinated checkpointing

- If perfectly **synchronized clocks were available** at processes, 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**.
- Perfectly synchronized clocks are not available, either the **sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking**.
- **Blocking coordinated checkpointing**
- **Non-Blocking coordinated checkpointing**

Coordinated checkpointing

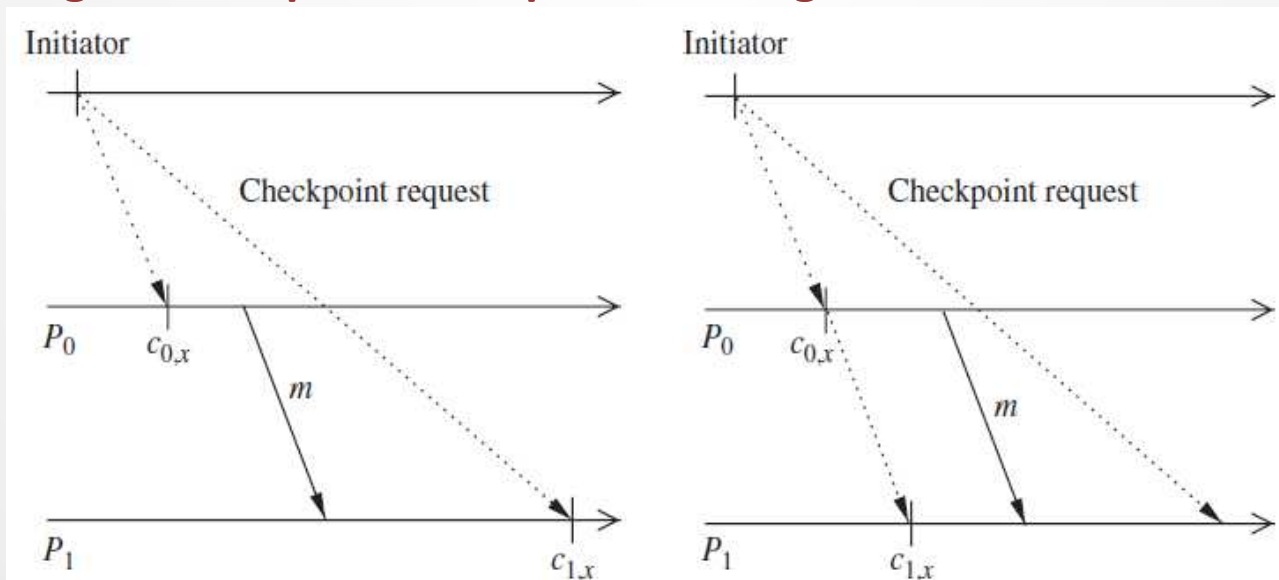
Blocking coordinated checkpointing

- Block communications 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.
- A problem with this approach is that the **computation is blocked during the checkpointing**

Coordinated checkpointing

Non-Blocking coordinated checkpointing

- The processes **need not stop their execution while taking checkpoints.**
- **Prevent a process** from **receiving application** messages that could make the checkpoint inconsistent.
- **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 . **Forcing each process to take a checkpoint before receiving the first post-checkpoint message.**



Coordinated checkpointing

Non-Blocking coordinated checkpointing

- Communication Channels are Reliable & FIFO
- Chandy and Lamport [1] in which *markers play the role of the checkpoint request* messages.
- The **initiator takes a checkpoint and sends a marker** (a checkpoint request) on **all outgoing** channels.
- Each process takes a checkpoint upon receiving the **first marker and sends the marker on all outgoing channels before sending any application message**.
- Communication Channels are Non- FIFO
- The marker can be **piggybacked on every post-checkpoint message**.
- When a process receives an **application message with a marker**, it treats it as if it has received a marker message, followed by the application message.

Coordinated checkpointing

- Coordinated checkpointing **requires all processes to participate in every checkpoint.**
- This **affects scalability.**
- **Reduce the number of processes** involved in a coordinated checkpointing session.
- Only those **processes that have communicated with the checkpoint initiator** either directly or indirectly since the last checkpoint, need to take new checkpoints.
- A two-phase protocol by **Koo and Toueg [2]** achieves minimal checkpoint coordination.

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** (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
- **Autonomous checkpoints**: The checkpoints that a process takes independently are called ***local checkpoints***.
- **Forced checkpoints** : A process is **forced** to take a checkpoint are called ***forced checkpoints***

Communication Induced Checkpointing

- 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**, possibly incurring some latency and overhead.
- No special coordination messages are exchanged unlike coordinated checkpointing.

Communication Induced Checkpointing

- **Two types: Model-based checkpointing and Index-based checkpointing.**
- **Model-Based:** The system maintains **checkpoints** and **communication structures** that prevent the domino effect or achieve some even stronger properties.
- **Index-Based:** The system uses an **indexing scheme** for the **local** and **forced checkpoints**, such that the checkpoints of the **same index at all processes form a consistent state.**

Communication Induced Checkpointing

- **Model-based checkpointing**
- **Model-based checkpointing prevents patterns of communications and checkpoints** that could result in inconsistent states among the existing checkpoints.
- **All information** necessary to execute the protocol is ***piggybacked on application messages.*** The decision to take a forced checkpoint is done locally using the information available.
- The ***MRS (mark, send, and receive) model of Russell [3]*** avoids the domino effect by ensuring that **within every checkpoint interval all message receiving events precede all message-sending events.**
- This model can be maintained by taking an **additional checkpoint before every message-receiving event**
- Another way to prevent the domino effect by avoiding rollback propagation completely is by taking a **checkpoint immediately after every message-sending event.**
- Recent work has **focused** on ensuring that every checkpoint can belong to a **consistent global checkpoint** and therefore is not useless.

Communication Induced Checkpointing

- **Index-based checkpointing**
- This 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.
- The protocol by **Briatico et al [4]**. *forces a process to take a checkpoint upon receiving a message with a piggybacked index greater than the local index.*

References

1. K. M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, *ACM Transactions on Computer Systems* 3(1), 1985, 63–75.
2. R. Koo and S. Toueg, Checkpointing and rollback-recovery for distributed systems, *IEEE Transactions on Software Engineering*, 13(1) 1987, 23–31.
3. D. L. Russell, State restoration in systems of communicating processes, *IEEE Transactions of Software Engineering*, 6(2), 1980, 183–194.
4. D. Briatico, A. Ciuffoletti, and L. Simoncini, A distributed domino-effect free recovery algorithm, *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, MD, October 1984, 207–215.

Summary

- Checkpoint –based Recovery
 - Uncoordinated Checkpointing
 - Direct Dependency Tracking Technique.
 - Coordinated Checkpointing
 - Blocking Coordinated Checkpointing
 - Non-Blocking Coordinated Checkpointing
 - Communication-induced Checkpointing
 - Model-based Checkpointing
 - Index-based Checkpointing

SYNCHRONOUS AND ASYNCHRONOUS CHECK POINT AND RECOVERY ALGORITHMS

BY
Y.V.LOKESHWARI
&
K.NIVETHAA SHREE

AGENDA

1. Introduction
2. Consistent Set of Checkpoints
3. Synchronous Checkpoint and Recovery
 - 3.1 Checkpoint Algorithm
 - 3.2 Recovery Algorithm
4. Drawbacks Synchronous Algorithm
5. Asynchronous Checkpoint and Recovery
 - 5.1 Checkpoint Algorithm
 - 5.2 Recovery Algorithm

INTRODUCTION

- Check-Pointing

The process of saving state

- Checkpoint

The recovery point at which check-pointing occurs

- Rolling Back

The process of restoring a process to a prior-state

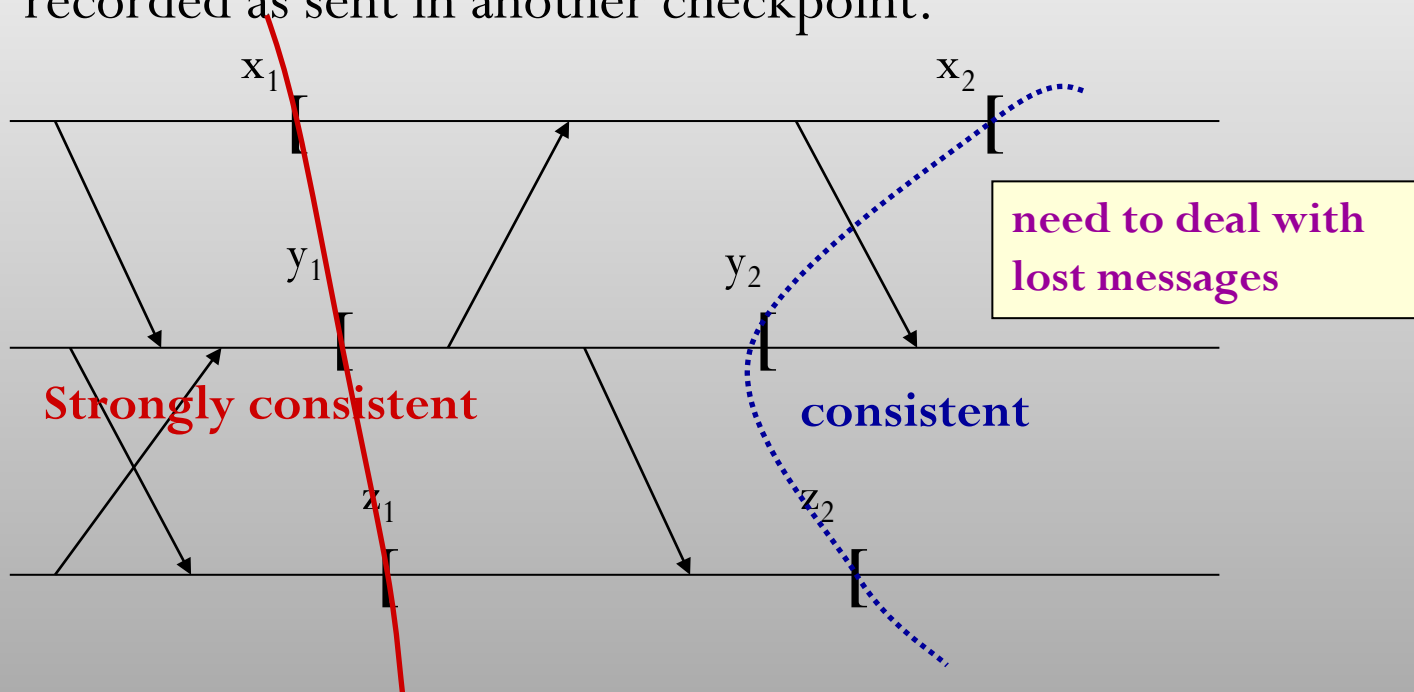
Consistency of Checkpoint

- Strongly consistent set of checkpoints

No information flow takes place between any pair of processes in the set , during the interval spanned by the checkpoints

- Consistent set of checkpoints

Each message recorded as received in a checkpoint should also be recorded as sent in another checkpoint.



Difference between Synchronous and Asynchronous Checkpoints

✓ **Synchronous Checkpoint**

Set of all recent checkpoints are guaranteed to be consistent.

✓ **Asynchronous Checkpoint**

Set of all recent checkpoints are not guaranteed to be consistent.

SYNCHRONOUS CHECK-POINTING AND RECOVERY

~Synchronous Checkpoint~

Goal

To make a consistent global checkpoint

Preliminary Assumptions

- Communication channels are FIFO
- No partition of the network
- End-to-end protocols cope with message loss due to rollback recovery and communication failure
- No failure during the execution of the algorithm
- The Checkpoint Algorithm assumes that single process invokes the Algorithm and not as several processes concurrently invoking the algorithm to take permanent checkpoint.

Preliminary (Two types of checkpoint)

~Synchronous Checkpoint~

Tentative checkpoint :

- a temporary checkpoint
- a candidate for permanent checkpoint

Permanent checkpoint :

- a local checkpoint at a process
- a part of a **consistent** global checkpoint

Checkpoint Algorithm

Algorithm

~Synchronous Checkpoint~

First Phase

1. An initiating process P_i (a single process that invokes this algorithm) takes a tentative checkpoint
2. It requests all the processes to take tentative checkpoints
3. It waits for receiving from all the processes whether taking a tentative checkpoint has been succeeded
4. If it learns all the processes has succeeded, it decides all tentative checkpoints should be made permanent; otherwise, should be discarded.

Second Phase

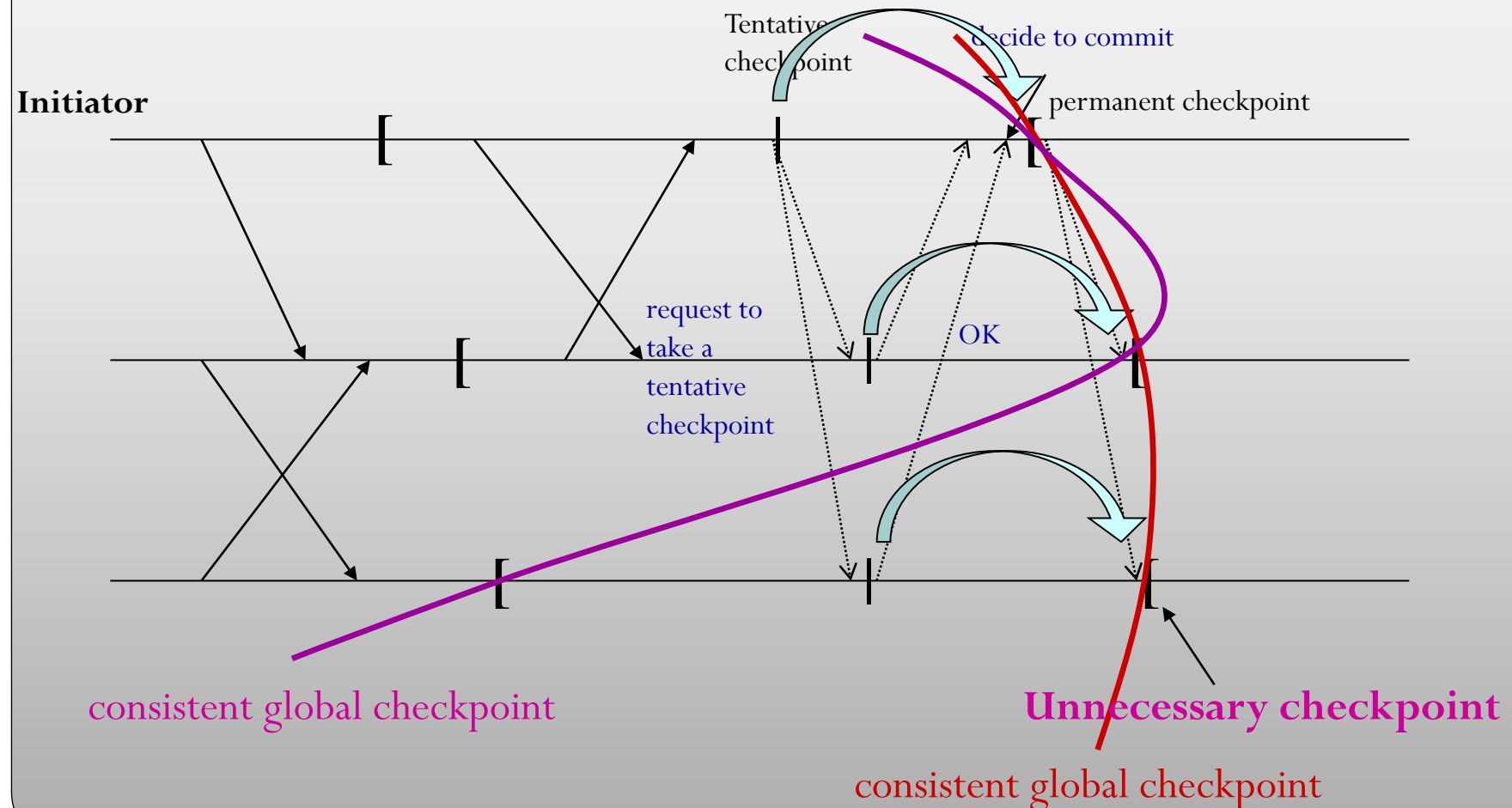
1. P_i It informs all the processes of the decision
2. The processes that receive the decision act accordingly

Supplement

Once a process has taken a tentative checkpoint, it shouldn't send messages until it is informed of initiator's decision.

Diagram of Checkpoint Algorithm

~Synchronous Checkpoint~



Optimized Algorithm

~Synchronous Checkpoint~

Each message is labeled by order of sending

Labeling Scheme

\perp : smallest label

T : largest label

$\text{last_label_rcvd}_X[Y] : y2$

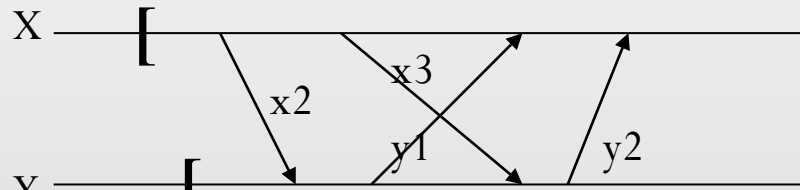
the last message that X received from Y after X has taken its last permanent or tentative checkpoint. if not exists, \perp is in it.

$\text{first_label_sent}_X[Y] : x2$

the first message that X sent to Y after X took its last permanent or tentative checkpoint . if not exists, \perp is in it.

$\text{ckpt_cohort}_X :$

the set of all processes that may have to take checkpoints when X decides to take a checkpoint.



Checkpoint request need to be sent to only the processes included in ckpt_cohort

Optimized Algorithm

~Synchronous Checkpoint~

$$\text{ckpt_cohort}_X : \{ Y \mid \text{last_label_rcvd}_X[Y] > \perp \}$$

Y takes a tentative checkpoint only if

$$\text{last_label_rcvd}_X[Y] \geq \text{first_label_sent}_Y[X] > \perp$$

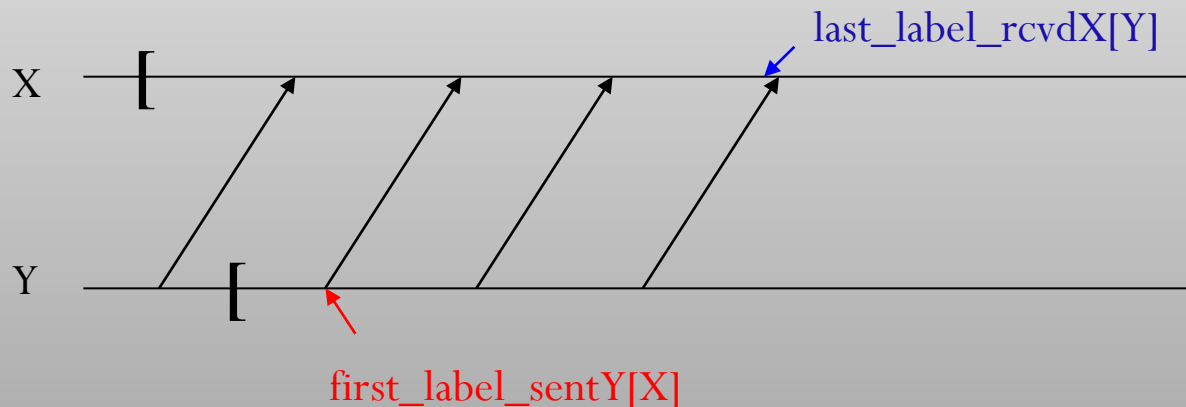
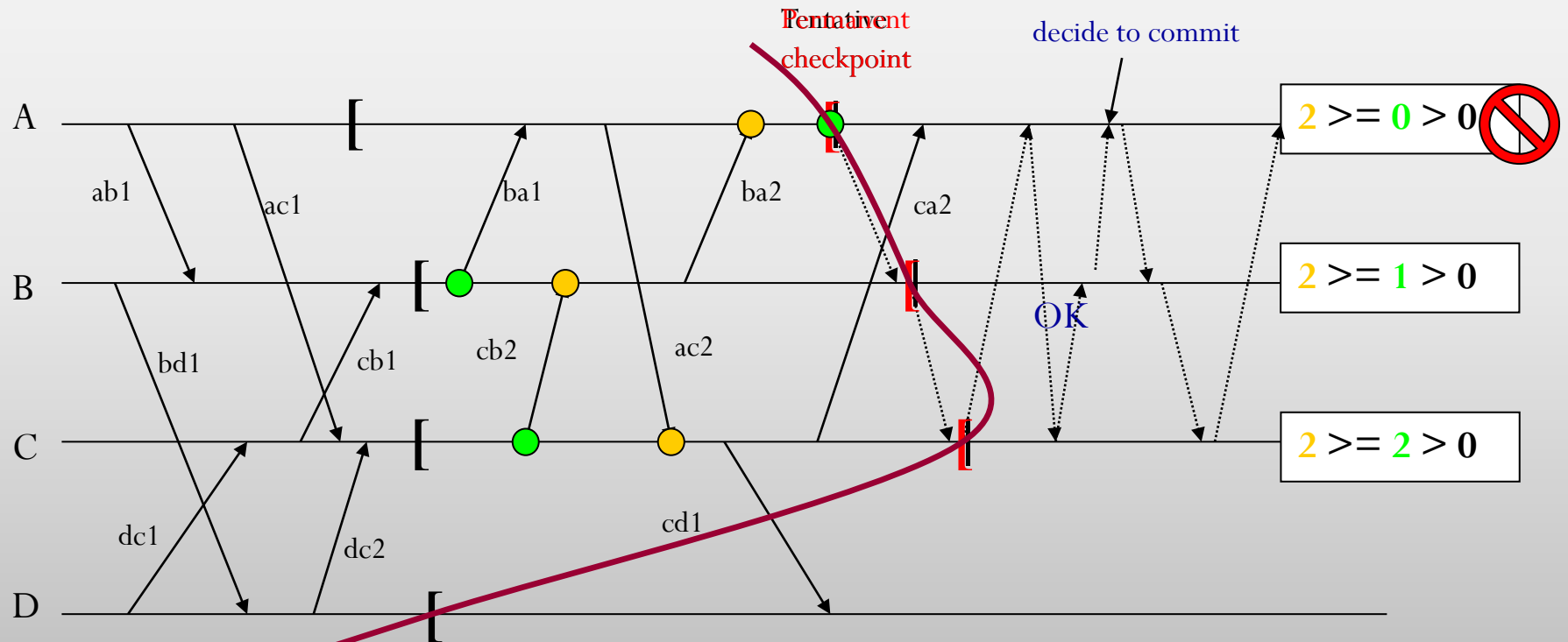


Diagram of Optimized Algorithm

~Synchronous Checkpoint~



$\text{ckpt_cohortX} : \{ Y \mid \text{last_label_rcvdX}[Y] > \perp \}$

$\text{last_label_rcvdX}[Y] \geq \text{first_label_sentY}[X] > \perp$

Correctness

~Synchronous Checkpoint~

- A set of permanent checkpoints taken by this algorithm is consistent
 - No process sends messages after taking a tentative checkpoint until the receipt of the decision
 - New checkpoints include no message from the processes that don't take a checkpoint
 - The set of tentative checkpoints is fully either made to permanent checkpoints or discarded.

The Rollback Recovery Algorithm

Preliminary Assumptions

- The Rollback Recovery algorithm assumes that a single process invokes the algorithm and not several processes concurrently invoking the Algorithm.
- The Checkpoint and Rollback Recovery algorithms are not concurrently invoked.

Two phases of Rollback Recovery Algorithm

First Phase

- An initiating process P_i checks to see if all the processes are willing to restart from their previous checkpoints.
- A process may reply “No” to restart request if it is already participating in a check-pointing or a recovery process is initiated by some other process.
- If P_i learns that all the processes are willing to restart from their previous checkpoints
then

P_i decides that all the process should restart.

otherwise

All the processes should continue their normal activities.

Phases contd..

Second phase

- P_i Propagates its decision to all processes.
- On receiving P_i 's decision, a process will act accordingly.
- The recovery algorithm requires that every process should not send messages related to underlying computation while it is waiting for P_i 's decision.

Correctness

- All Co-operating processes will restart from an appropriate state.
- All processes either restart from their previous checkpoint or continue with their normal operation
- If processes decide to re-start, then they all will resume execution in a consistent checkpoint.

Recovery Algorithm

~Synchronous Recovery~

Labeling Scheme

\perp : smallest label

T : largest label

$last_label_sent_X[Y]$:

The last message that X sent to Y before X takes its latest permanent checkpoint. If not exist, T is in it.

$last_label_rcvd_Y[X]$:

The last message that Y received from X after X took its last permanent or tentative checkpoint. If not exists, \perp is in it.

When X request Y to restart from permanent checkpoint, it sends $last_label_sent_X[Y]$ along with its request.

Recovery Algorithm

~Synchronous Recovery~

Y will restart from the permanent checkpoint only if

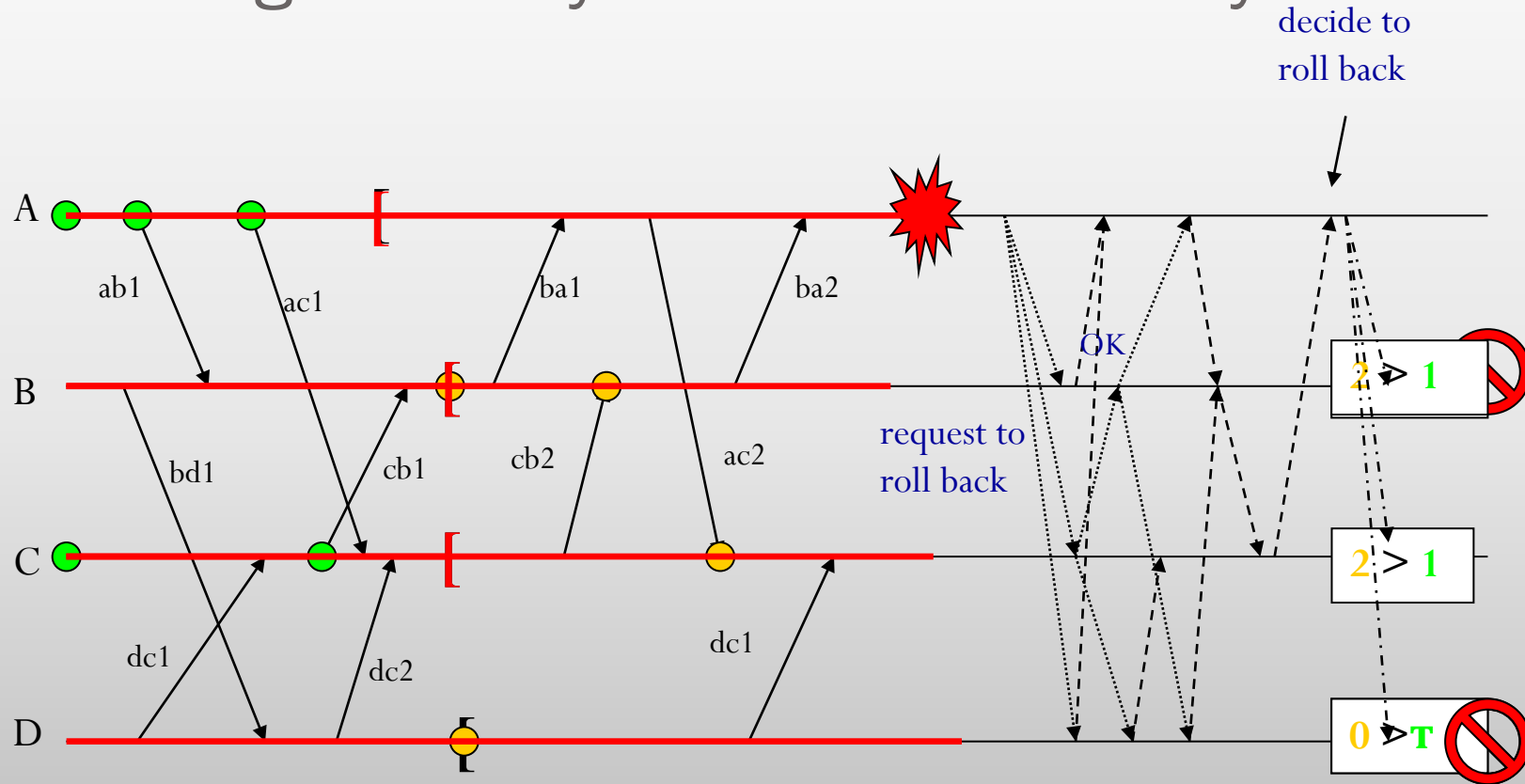
$$\text{last_label_rcvd}_Y[X] > \text{last_label_sent}_X[Y]$$

roll_cohort_X :

The set of all processes that may have to roll back to the latest checkpoint when process X rolls back.

$$\text{roll_cohort}_X = \{ Y \mid X \text{ can send messages to } Y \}$$

Diagram of Synchronous Recovery



$\text{roll_cohortX} = \{Y \mid X \text{ can send messages to } Y\}$

$\text{last_label_rcvdY}[X] > \text{last_label_sentX}[Y]$

Asynchronous Checkpoint/Recovery Algorithm

Synchronous Approach

- It simplifies recovery
 - Since consistent set of checkpoints readily available.
- Demerits
 - Additional messages are exchanged by checkpoint algorithm when it takes checkpoint.
 - Synchronization delays occurs.
 - No computational message can be sent while checkpoint algorithm is in progress.

Asynchronous Approach

Characteristic:

- Each process takes checkpoints independently without any synchronization among process.
- No guarantee that a set of local checkpoints is consistent.
- A recovery algorithm has to search consistent set of checkpoints before recovery initiated.
- No additional message
- No synchronization delay

Asynchronous Checkpoint (Message logging)

- To minimize amount of computation undone during recovery , all incoming message logged at each processor.
- Message received can be logged in two ways:
- Pessimistic message logging:
 - Incoming message is logged before it is processed.

Asynchronous Checkpoint(contd.)

- Optimistic message logging:
 - Processors continue to perform computation and message received are stored in volatile storage , logged at certain intervals.
 - In system failure , incoming message lost as it may not have been logged.

Asynchronous Checkpoint (contd.)

- Comparison:
 - During rollback , amount of computation redone during recovery more in system that use optimistic logging when compared to system tat use pessimistic logging.

Two types of log

~Asynchronous Checkpoint / Recovery~

- Two types of log storage, volatile and stable log.
- Volatile log:
 - Access time less.
 - Contents are lost if processor fails.
 - Periodically flushed to stable storage and cleared.
- Stable log:
 - Slow access.
 - Not lost even if processors fail.

Record events

- Each processor, after event, records triplet $\{s, m, \text{msg_sent}\}$ in volatile storage.
- S is state of the processor before the event.
- m is message whose arrival caused the events.
- msg_sent is the set of messages that were sent by processor during event.
- Local checkpoint at each processor consist of the record of an event occurring at processor
- Taken without any synchronization with other processors.

Preliminary (Assumptions)

~Asynchronous Checkpoint / Recovery~

- Assumptions
 - Communication channels are FIFO
 - Communication channels are reliable
 - Communication channel have infinite buffers.
 - Message transmission delay is arbitrary, but finite.

Preliminary (Notations)

~Asynchronous Checkpoint / Recovery~

Definition

$CkPt_i$: the checkpoint (stable log) that i rolled back when failure occurs

$RCVD_{i \leftarrow j}(CkPt_i)$:

the number of messages received by processor i from processor j , per the information stored in the checkpoint $CkPt_i$

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

the number of messages sent by processor i to processor j , per the information stored in the checkpoint $CkPt_i$

Recovery Algorithm

~Asynchronous Checkpoint / Recovery~

- Each processor keeps track of number of messages it has sent to other processor as well as number of messages it has received from other processor.
- When rollback occurs, other processor find out whether any message previously sent are orphan message.
- Discovered by comparing number of message sent and received.
- If number of message received is greater than number of message sent, it indicates orphan message.
- Processor have to rollback to state where number of messages received agrees with number of messages sent.

Recovery Algorithm

If i is a processor that is recovering after failure then

$CkPt_i$ = latest event logged in the stable storage

else

$CkPt_i$ = latest event that took place in it

for $k=1$ to N do

begin

for each neighboring processor j do

Send $ROLLBACK(I, SENT_{i \rightarrow j}(CkPt_i))$ message

wait for $ROLLBACK$ message from every neighbor.

Recovery Algorithm(contd.)

for each ROLLBACK(j, c) message received from a neighbor
 j

i does following

 if $RCVD_{i \leftarrow j}(CkPt_i) > c$ then

 (inplies presence of orphan message)

 begin

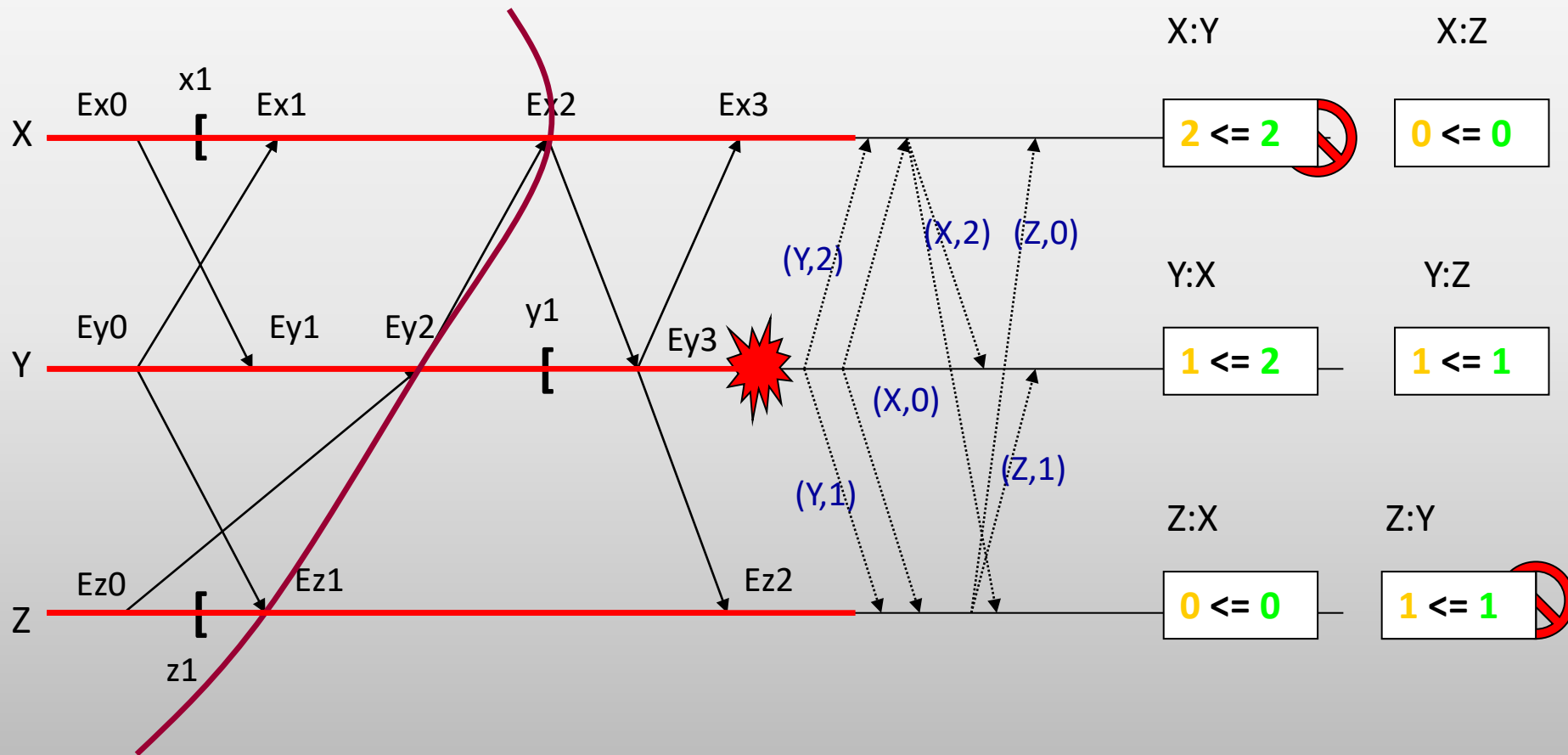
 find the latest event e such that $RCVD_{i \leftarrow j}(e) = c$

$CkPt_i = e$;

 end;

end(* for k^*)

Asynchronous Recovery



$$RCVDi \leftarrow j (CkPti) \leq SENTj \rightarrow i (CkPtj)$$

QUERIES???

THANK YOU

Livelock and Domino Effects

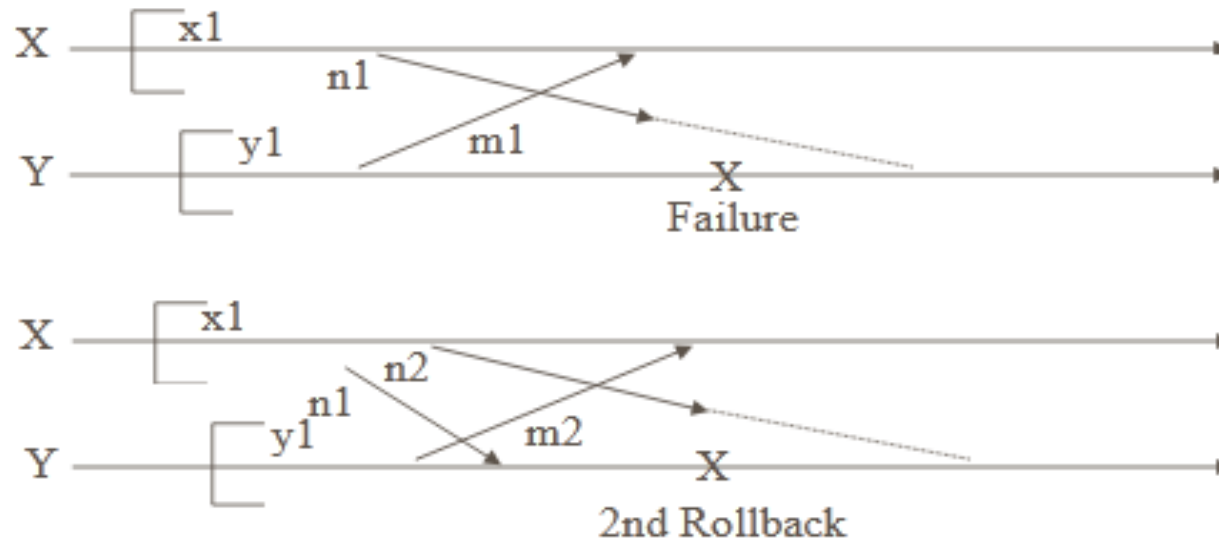
Dr. Lokeswari YV

ASP/ CSE

SSN College of Engineering

Livelocks due to delayed messages

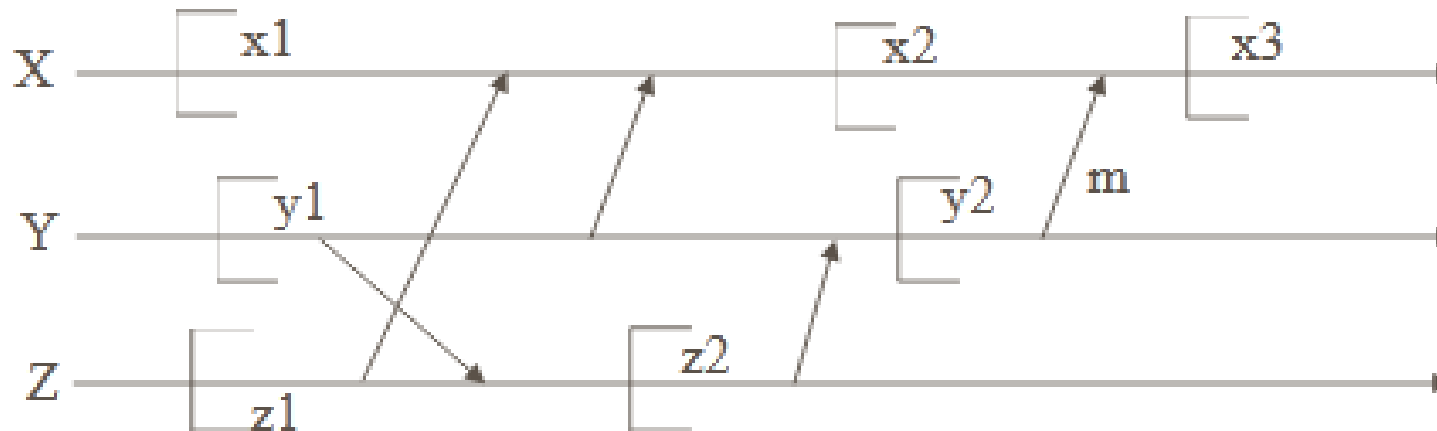
Livelocks



- Y crashes before receiving `n1`. Y rolls back to `y1` -> X to `x1`.
- Y recovers, receives `n1` and sends `m2`.
- X recovers, sends `n2` but has no record of sending `n1`
- Hence, Y is forced to rollback second time. X also rolls back as it has received `m2` but Y has no record of `m2`.
- Above sequence can repeat indefinitely, causing a livelock.

Domino Effect due to Orphan Messages

- Orphan messages & the Domino effect: Assume Y fails after sending m.
 - X has record of m at x3 but Y has no record. m -> orphan message.
 - Y rolls back to y2 -> X should go to x2.
 - If Z rolls back, X and Y has to go to x1 and y1 -> Domino effect, roll back of one process causes one or more processes to roll back.



Log- based Rollback Recovery

Y. V. Lokeswari

Reference: Kshemkalyani, Ajay D., and Mukesh Singhal. Distributed computing: principles, algorithms, and systems. Cambridge University Press, 2011.

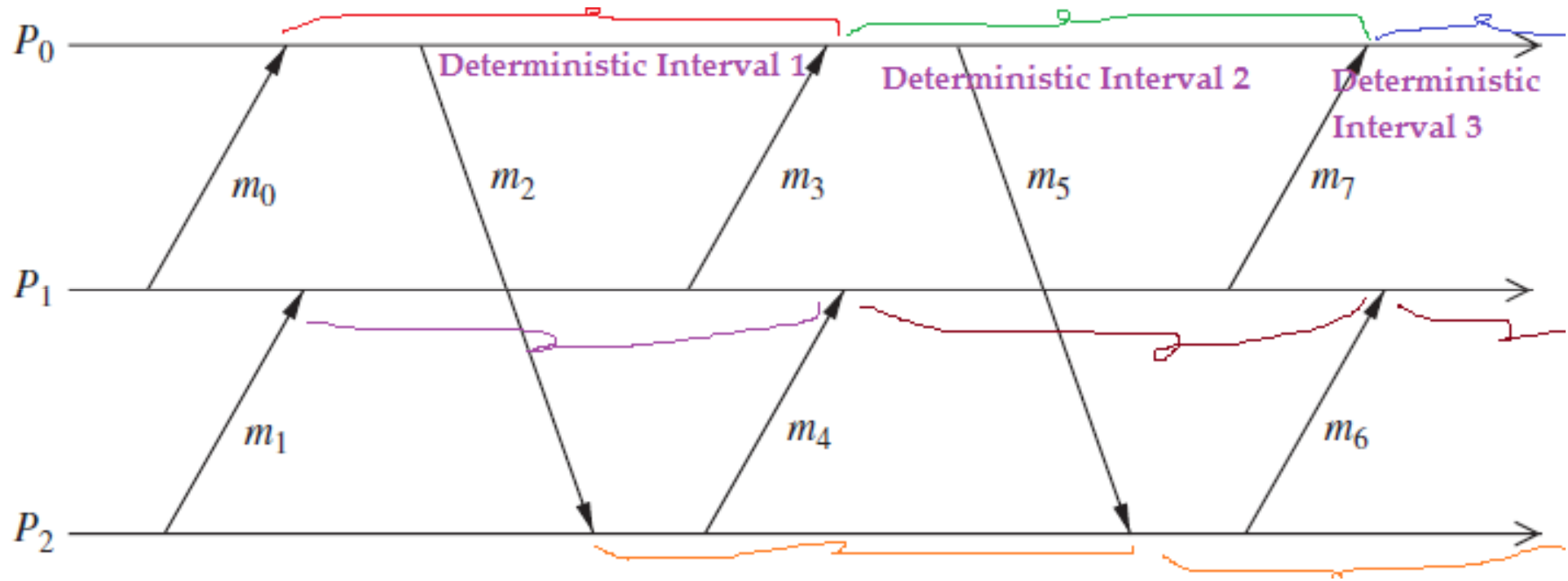
Overview

- **Log-based Recovery.**
- **Deterministic and Non-Deterministic Events.**
- **No-Orphans consistency condition.**
- **Types of Logging Protocols.**
 - **Pessimistic logging.**
 - **Optimistic logging.**
 - **Causal logging.**

Log-based Rollback Recovery

- Log-based rollback recovery exploits the fact that a process execution can be modelled as a **sequence of deterministic state intervals, each starting with the execution of a non-deterministic event.**
- **Send event is Deterministic**
- **Receive and internal events are non-deterministic.**
- **Non-determinism** means that the path of execution isn't fully determined by the specification of the computation, so the same input can produce different outcomes, while **deterministic** execution is guaranteed to be the same, given the same input.

Deterministic State Intervals



Log-based Rollback Recovery

- 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.
- Additionally, each process **also takes checkpoints** to reduce the extent of rollback during recovery.
- After a failure occurs, the failed processes **recover** by using the **checkpoints and logged determinants to replay the corresponding non-deterministic events** precisely as they occurred during the pre-failure execution.
- The pre-failure execution of a failed process can be reconstructed during recovery **up to the first non-deterministic event whose determinant is not logged**.

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

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

No-Orphans consistency condition

Always-no-orphans condition

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

- This property is called the *always-no-orphans condition*.

It states that if any surviving process depends on an event e , then either event e is logged on the stable storage, or the process has a copy of the determinant of event e .

- *If neither condition is true, then the process is an orphan because it depends on an event e that cannot be generated during recovery since its determinant is lost.*

Log-based rollback-recovery

- Log-based rollback-recovery protocols guarantee that **upon recovery** of all failed processes, the **system does not contain any orphan process**.

Types of Log-based rollback-recovery

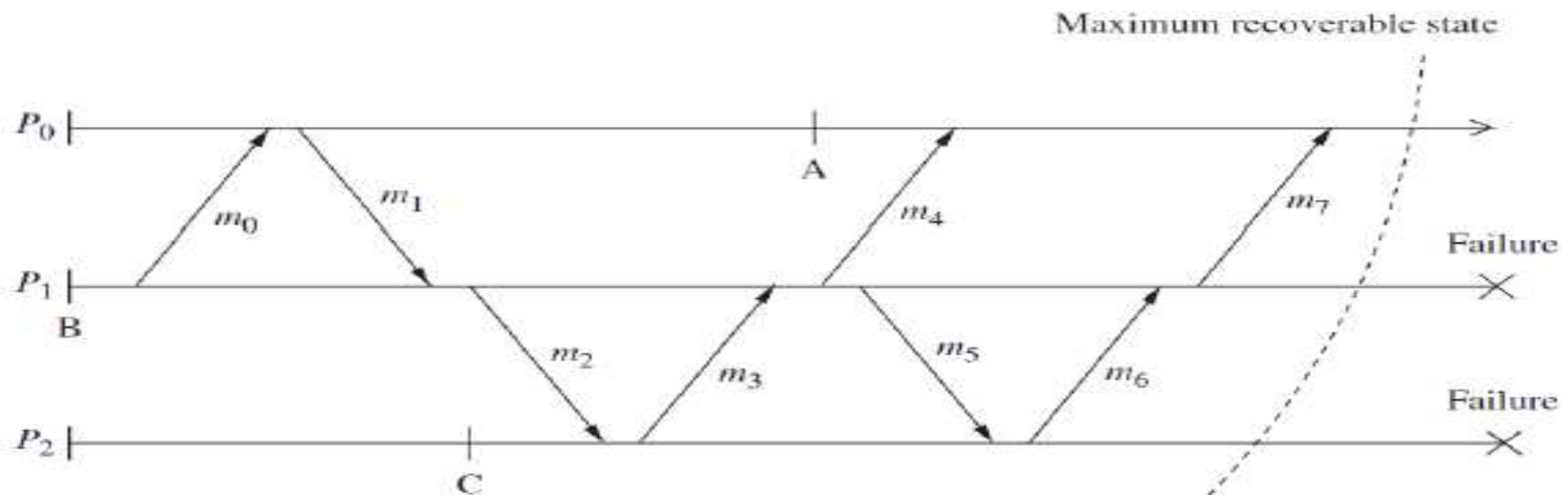
- **Pessimistic logging.**
- **Optimistic logging.**
- **Causal logging.**
- They differ in their failure-free performance overhead, latency of output commit, simplicity of recovery and garbage collection, and the potential for rolling back surviving processes.

Pessimistic Logging

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation.
- Pessimistic protocols log to the stable storage the determinant of each non-deterministic event before the event affects the computation.
- **This protocol implements *Synchronous Logging***
- **$\forall e: \neg \text{Stable}(e) \Rightarrow |\text{Depend}(e)| = 0$**
 - if an event has not been logged on the stable storage, then no process can depend on it.
 - stronger than the always-no-orphans condition.
- In addition to logging determinants, processes also take periodic checkpoints to minimize the amount of work that has to be repeated during recovery.
- When a process fails, the process is restarted from the most recent checkpoint and the logged determinants are used to recreate the pre-failure execution.

Pessimistic Logging

- During failure-free operation the logs of processes ***P0***, ***P1***, and ***P2*** **contain the determinants** needed to **replay messages *m0*, *m4*, *m7*, *m1*, *m3*, *m6*, and *m2*, *m5***, respectively.
- 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.
- This guarantees that *P1* and *P2* will repeat exactly their pre-failure execution and re-send the same messages.



Pessimistic Logging

- Synchronous logging can potentially result in a **high performance overhead**.
- Special techniques to reduce the effects of synchronous logging on the performance. This overhead can be lowered using special hardware.
- **To Overcome Performance Overhead:**
- Fast non-volatile semiconductor memory can be used to implement the stable storage.
- Limit the number of failures that can be tolerated.
- Delivering a message or executing an event and deferring its logging until the process communicates with another process or with the outside world.
- ***Sender-Based Message Logging (SBML)*** protocol keeps the determinants corresponding to the delivery of each message *m* in the volatile memory of its sender.

Optimistic Logging

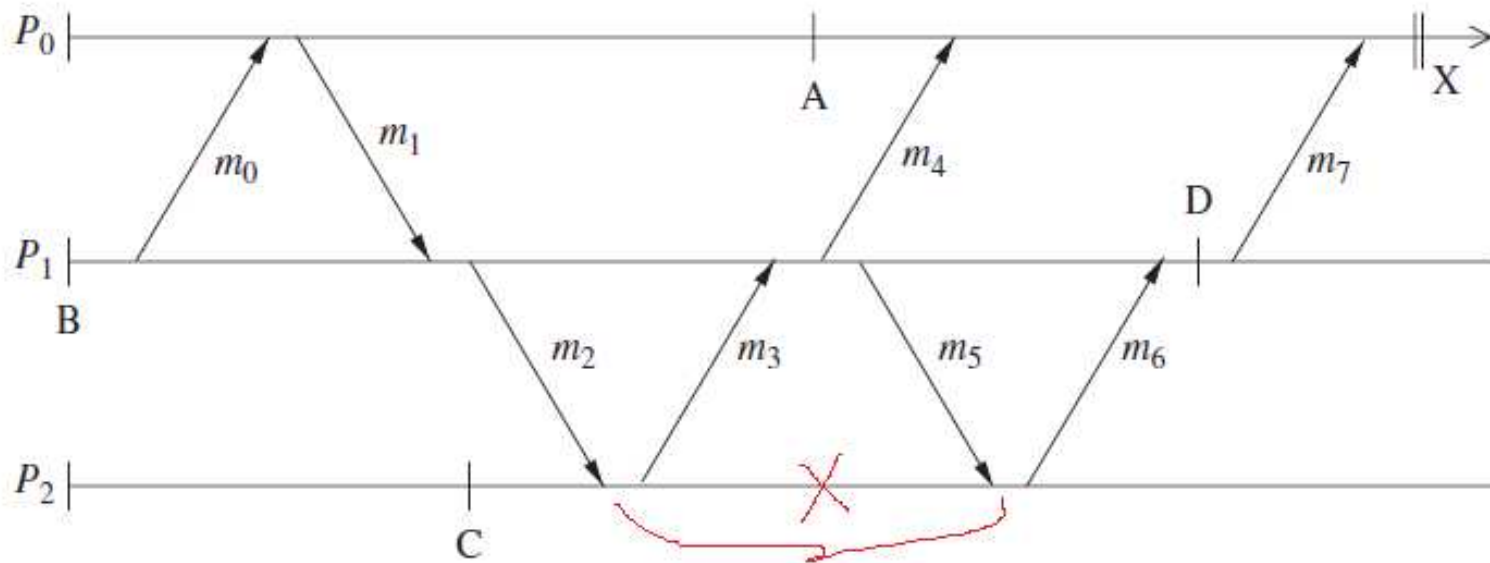
- Processes log determinants *asynchronously* to the stable storage.
- These protocols optimistically assume that logging will be complete before a failure occurs.
- Determinants are kept in a volatile log, and are periodically flushed to the stable storage.
- **Incurs much less overhead during failure-free execution.**
- **But more complicated recovery, garbage collection, and slower output commit.**

Optimistic Logging

- 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**.
- **Optimistic logging protocols do not implement the *always-no-orphans condition*.**
- The ***always-no-orphans condition*** *holds after the recovery is complete.*
- This is achieved by rolling back orphan processes until their states do not depend on any message whose determinant has been lost.

Optimistic Logging

- 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 .
- 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 .



Optimistic Logging

- To perform rollbacks correctly, **optimistic logging protocols** track **causal dependencies** during failure free execution.
- Upon a failure, the **dependency information** is used to **calculate and recover the latest global state** of the pre-failure execution in which **no process is in an orphan**.
- **Pessimistic protocols** need **only keep the most recent checkpoint** of each process.
- **Optimistic protocols** may need to **keep multiple checkpoints** for each process.

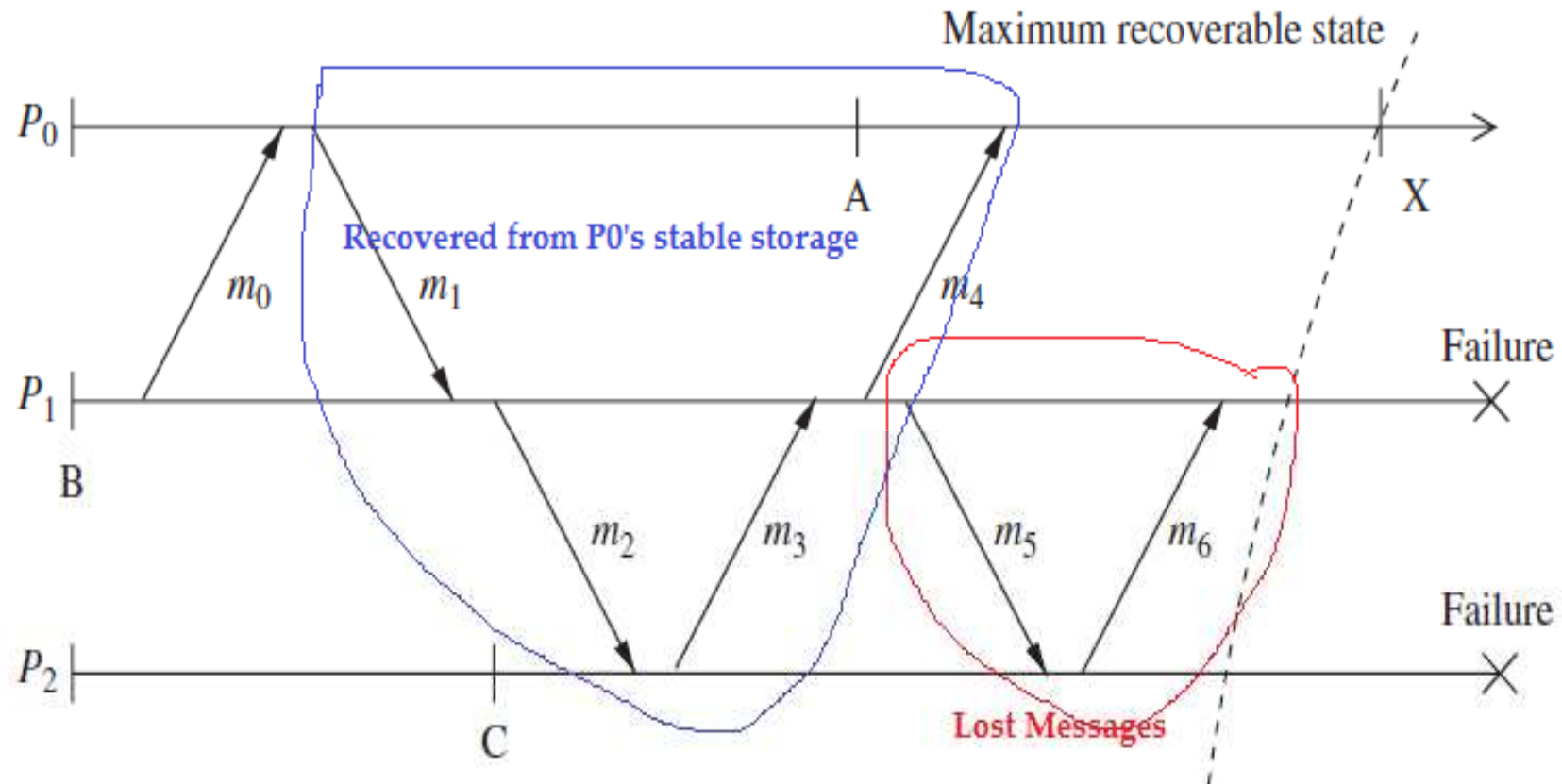
Causal Logging

- 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**.
- 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.
- **Causal logging protocols make sure that the *always-no-orphans property* holds.**

Causal Logging

- Each process **maintains information** about **all** the **events** that have **causally affected** its state.
- This information **protects** it from the **failures** of other processes and also **allows** the **process** to make its **state recoverable** by simply logging the information available locally.
- It can **commit output independently**.

Causal Logging



Causal Logging

- Messages m5 and m6 are likely to be lost on the failures of P1 and P2 at the indicated instants.
- Process P0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's happened-before relation.
- These events consist of the delivery of messages m0, m1, m2, m3, and m4. 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 P0.
- 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.
- 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.

Summary

- **Log-based Recovery.**
- **Deterministic and Non-Deterministic Events.**
- **No-Orphans consistency condition.**
- **Types of Logging Protocols.**
 - **Pessimistic logging.**
 - **Optimistic logging.**
 - **Causal logging.**

Consensus and Agreement

Y. V. Lokeswari

Reference: Ajay Kshemkalyani and
Mukesh Singhal, Distributed Computing:
Principles, Algorithms, and Systems



Agreement

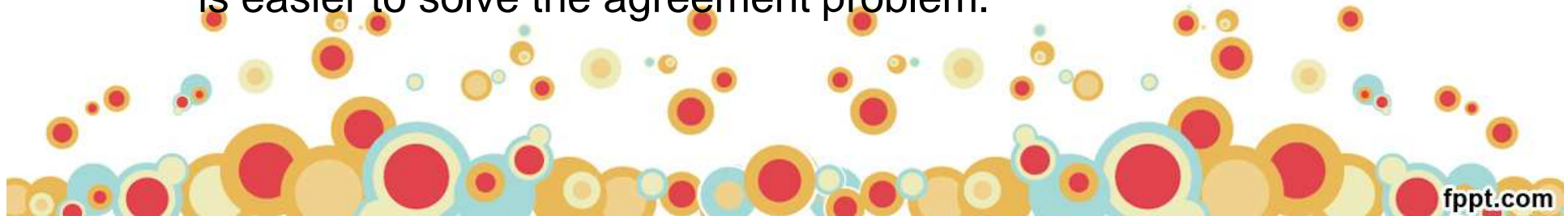
- Co-ordination require the **processes** to **exchange information** to negotiate with one another and eventually reach a common understanding or **agreement**, before taking application-specific actions.
- A classical example is that of the **commit decision** in database systems, processes collectively decide whether to **commit** or **abort** a transaction.
- **Assumptions**
 - Failure models
 - Synchronous / Asynchronous Communication
 - Network Connectivity
 - Sender Identification
 - Channel Reliability
 - Authentication vs Non-Authenticated messages
 - Agreement Variable

Agreement

- **Failure models**: Among the n processes in the system, at most f processes can be faulty.
- The various failure models – **fail-stop, send omission and receive omission and Byzantine failures**.
- It may send a message to only a **subset** of the **destination set** before crashing.
- In **Byzantine failure** model, a process may **behave arbitrarily**.
- **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**.

Agreement

- **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.
- **Channel reliability**: The channels are **reliable**, only the **processes** may **fail**.
 - **With unauthenticated messages**, when a faulty process relays a message to other processes, it can **forge** and claim that it was received from another processor or **tamper** the **contents** of the message.
 - Using **authentication** via techniques such as **digital signatures**, it is easier to solve the agreement problem.

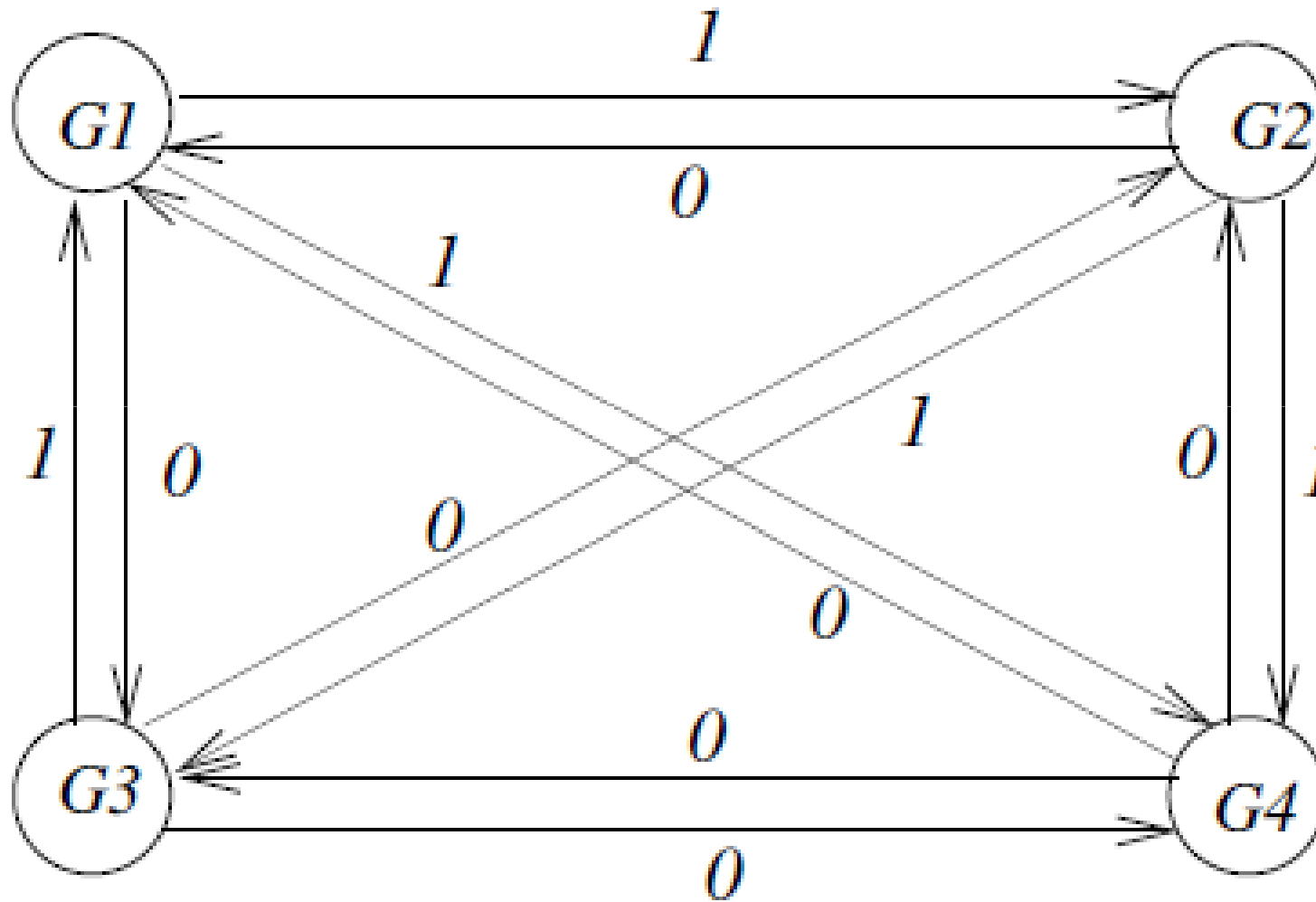


Agreement

- **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.
- **Channel reliability**: The channels are **reliable**, only the **processes** may **fail**.
 - **With unauthenticated messages**, when a faulty process relays a message to other processes, it can **forge** and claim that it was received from another processor or **tamper** the **contents** of the message.
 - Using **authentication** via techniques such as **digital signatures**, it is easier to solve the agreement problem.
- **Agreement variable** The agreement variable may be **Boolean** or **multi-valued** and **need not** be an **integer**.



Byzantine General sending Confusing Messages



Problem Specification

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 $A[v_1 \dots v_n]$.

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 .



Overall 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.

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



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

- In a **failure-free** system, **consensus** can be **reached** by **collecting information** from the **different processes**, arriving at a “**decision**,” and **distributing** this **decision** in the **system**.
- A distributed mechanism would have each process **broadcast** its **values** to others, and each **process** **computes** the **same function** on the **values received**.
- Examples being the **majority, max, and min functions**.
- Distribute the decision may be based on the **token** **circulation** on a **logical ring**, or the **three-phase tree-based broadcast, convergecast–broadcast**, or **direct communication** with all nodes.

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

- In a **synchronous system**, this can be done simply in a **constant number of rounds**.
- **common knowledge** of the decision value can be obtained using an **additional round**.
- In an **asynchronous system**, consensus can similarly be reached in a **constant number of message hops**.
- **concurrent common knowledge** of the **consensus** value can also be attained, **using** the **algorithms**.
- **Reaching agreement** is **straightforward** in a **failure-free system**.
- **Focus** on **failure-prone** systems.

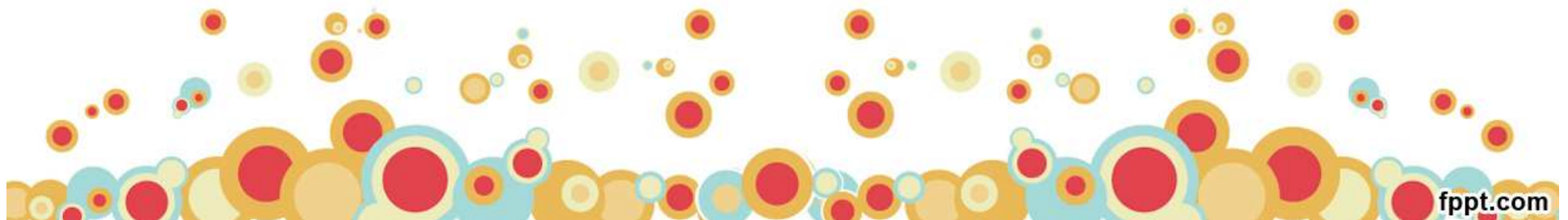


Summary

- Agreement.
- Problem Definition:
 - Byzantine Problem.
 - Consensus.
 - Interactive Consistency.
- Agreement in a failure-free system.



Thank You



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: $x \leftarrow$ 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);

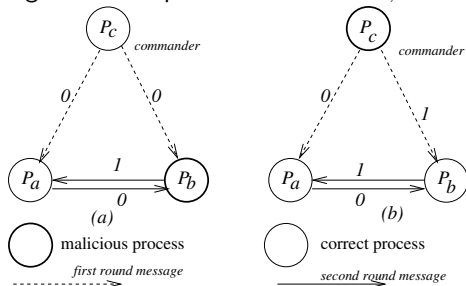
(1d) $y_j \leftarrow$ value (if any) received from process j in this round;

(1e) $x \leftarrow \min(x, y_j)$;

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

Upper Bound on Byzantine Processes (sync)

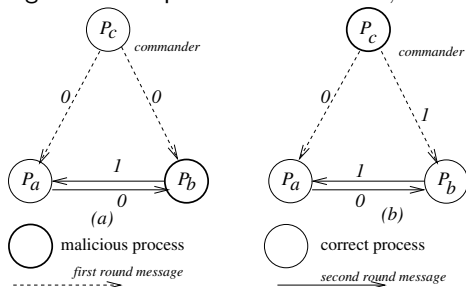
Agreement impossible when $f = 1, n = 3$.



- 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 $f \leq \lfloor \frac{n-1}{3} \rfloor$. See text.

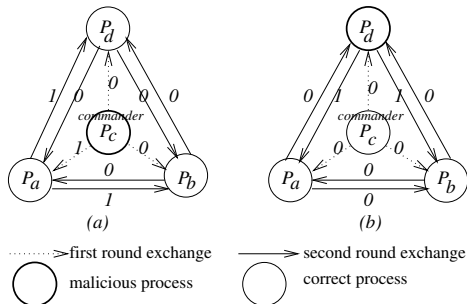
Upper Bound on Byzantine Processes (sync)

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 $f \leq \lfloor \frac{n-1}{3} \rfloor$. See text.

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: $v \leftarrow$ initial value;

integer: $f \leftarrow$ 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	$(f + 1) - x$	$n - x$	$(n - 1)(n - 2) \dots (n - x)$
$x + 1$	$x + 1$	$(f + 1) - x - 1$	$n - x - 1$	$(n - 1)(n - 2) \dots (n - x - 1)$
$f + 1$	$f + 1$	0	$n - f - 1$	$(n - 1)(n - 2) \dots (n - f - 1)$

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

Complexity: $f + 1$ rounds, exponential amount of space, and

$$(n - 1) + (n - 1)(n - 2) + \dots + (n - 1)(n - 2) \dots (n - f - 1) \text{ messages}$$

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

(variables)

boolean: $v \leftarrow$ initial value;

integer: $f \leftarrow$ 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 $h - 1 = \text{sizeof}(L)$, its $n - 2 - \text{sizeof}(L)$ descendants at level h are $v_k^{\text{concat}(\langle j \rangle, L)}$, $\forall k$ 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** $OM(v, Dests, L = \langle P_{k_1} \dots P_{k_{f+1}-faulty} \rangle, faulty)$ from P_{k_1} ;
 $// faulty + round = f, |Dests| + \text{sizeof}(L) = n$

(2d) $v_{head(L)}^{tail(L)} \leftarrow v$; $// \text{sizeof}(L) + faulty = f + 1$. fill in estimate.

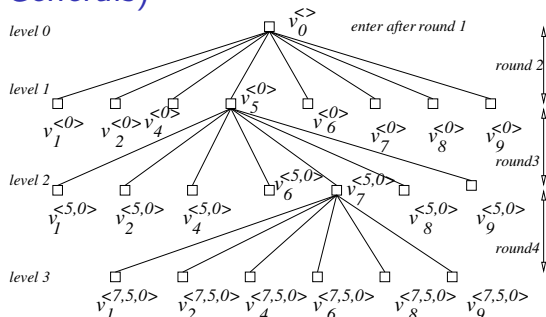
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \dots P_{k_{f+1}-faulty} \rangle, faulty - 1)$ to $Dests - \{i\}$ if $rnd < f$;

(2f) **for** $level = f - 1$ **down to** 0 **do**

(2g) **for** each of the $1 \cdot (n - 2) \cdot \dots \cdot (n - (level + 1))$ nodes v_x^L in level $level$, **do**

(2h) $v_x^L (x \neq i, x \notin L) = \text{majority}_y \notin \text{concat}(\langle x \rangle, L); y \neq i (v_x^L, v_y^{\text{concat}(\langle x \rangle, L)})$;

Tree Data Structure for Agreement Problem (Byzantine Generals)



Some branches of the tree at P_3 . In

this example, $n = 10, f = 3$, commander is P_0 .

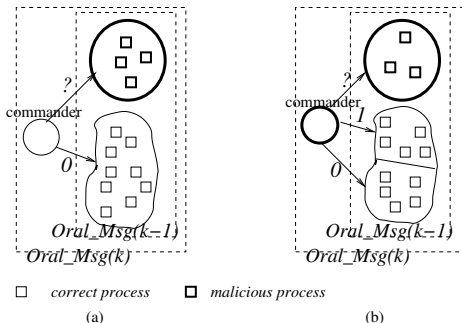
- (round 1) P_0 sends its value to all other processes using $Oral_Msg(3)$, including to P_3 .
- (round 2) P_3 sends 8 messages to others (excl. P_0 and P_3) using $Oral_Msg(2)$. P_3 also receives 8 messages.
- (round 3) P_3 sends $8 \times 7 = 56$ messages to all others using $Oral_Msg(1)$; P_3 also 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.

Exponential Algorithm: An example

An example of the majority computation is as follows.

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

Impact of a Loyal and of a Disloyal Commander



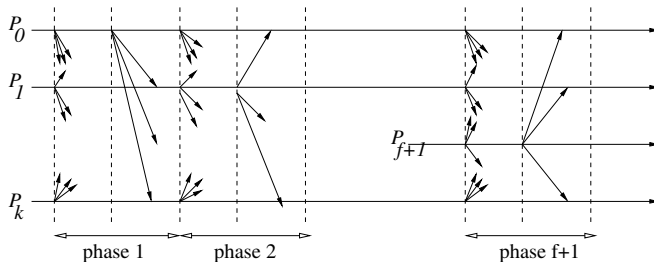
effects of a loyal or a disloyal commander in a system with $n = 14$ and $f = 4$. The subsystems that need to tolerate k and $k - 1$ traitors are shown for two cases. (a) Loyal commander. (b) No assumptions about commander.

- (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:
 - 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 it received in 1st round, and broadcasts its new estimate to all others.



The Phase King Algorithm: Code

(variables)

boolean: $v \leftarrow$ initial value;

integer: $f \leftarrow$ 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) $majority \leftarrow$ the value among the v_j that occurs $> n/2$ times (default if no maj.);

(1f) $mult \leftarrow$ 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) $v \leftarrow majority$;

(1m) **else** $v \leftarrow tiebreaker$;

(1n) **if** $phase = f + 1$ **then**

(1o) output decision value v .

The Phase King Algorithm

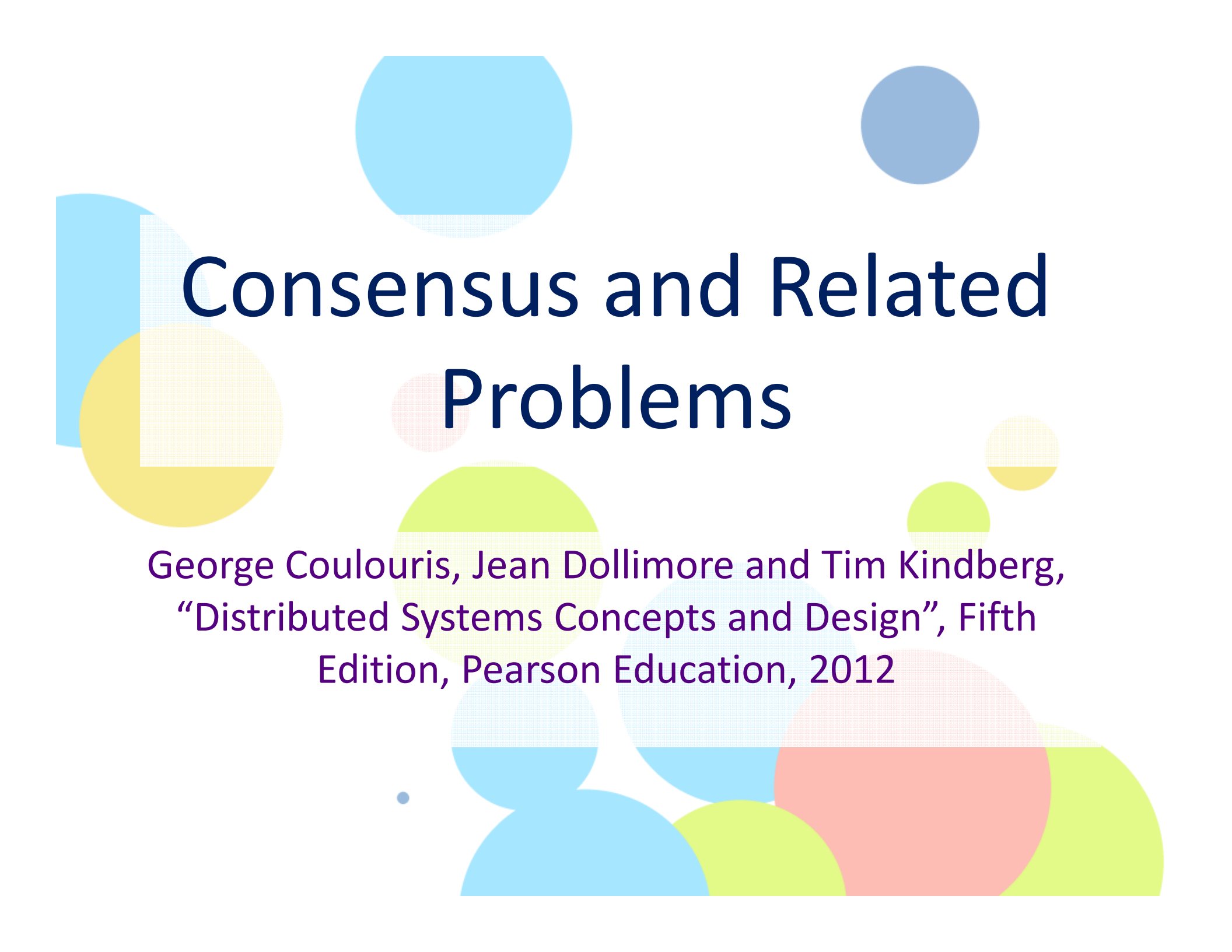
- $(f + 1)$ phases, $(f + 1)[(n - 1)(n + 1)]$ messages, and can tolerate up to $f < \lceil n/4 \rceil$ malicious processes

Correctness 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 P_i and P_j will have same estimate of consensus value as P_k does.
 - 1 P_i and P_j use their own majority values (Hint: $\implies P_i$'s *mult* $> n/2 + f$)
 - 2 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)
 - 3 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

- 3 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.



Consensus and Related Problems

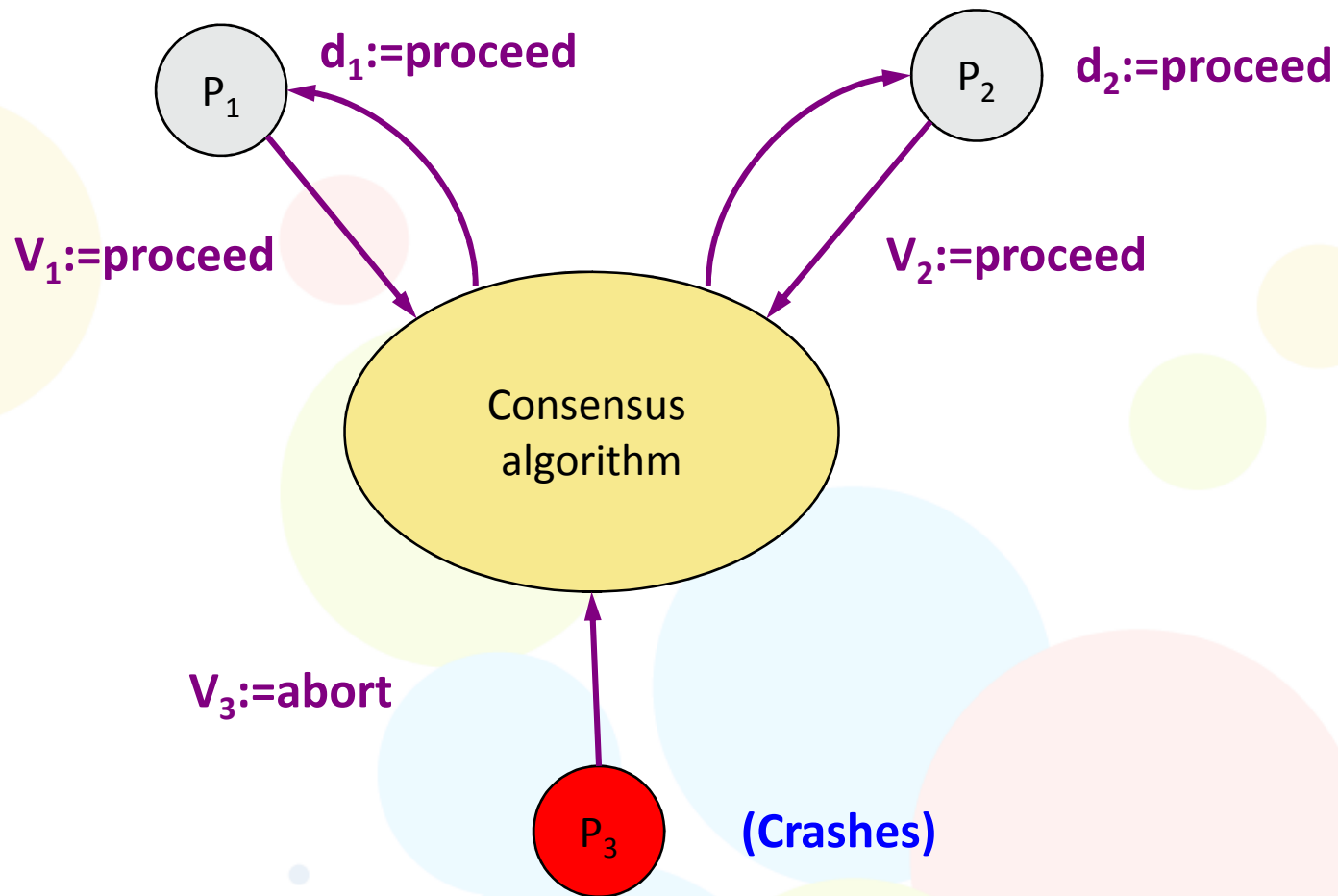
George Coulouris, Jean Dollimore and Tim Kindberg,
“Distributed Systems Concepts and Design”, Fifth
Edition, Pearson Education, 2012

Consensus ₍₁₎

- **Objective:** processes must agree on a value after one or more of the processes has proposed what that value should be
- **Hypotheses:** reliable communication, but processes may fail
- **Consensus problem:**
 - Every process P_i begins in the *undecided* state
 - Proposes a value $V_i \in D$ ($i=1, \dots, N$)
 - Processes communicate with one another, exchanging values
 - Each process then sets the value of a decision variable d_i

Enters the state *decided*, in which it may no longer change d_i ($i=1, \dots, N$)

Consensus (2)



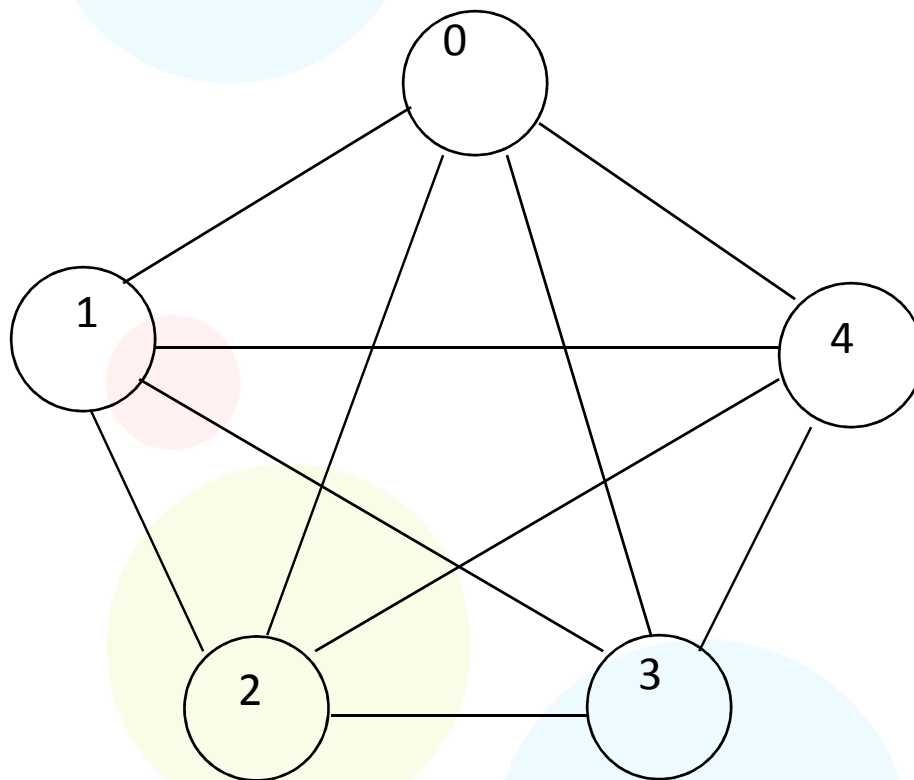
A simple algorithm for fault-free consensus

Each processor:

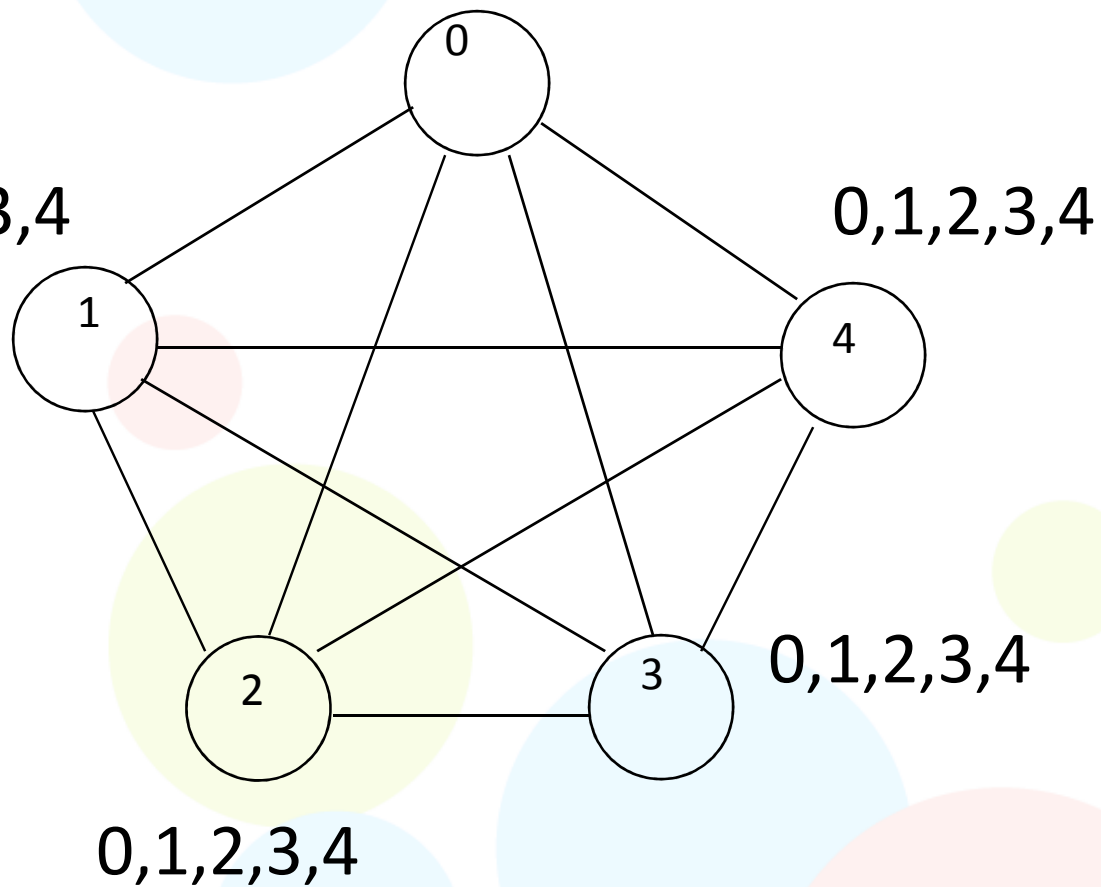
1. Broadcast its input to all processors
2. Decide on the minimum

(only one round is needed, since the graph is complete)

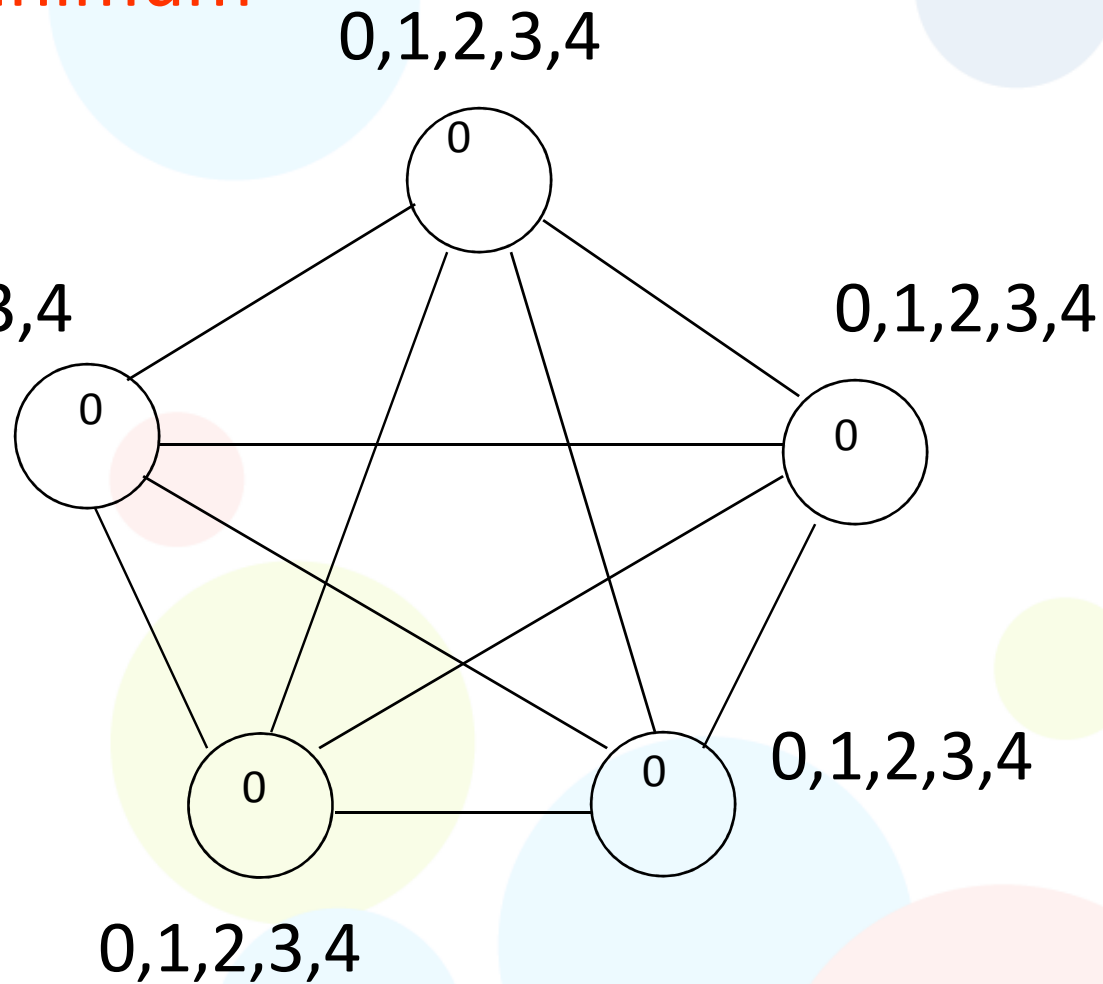
Start



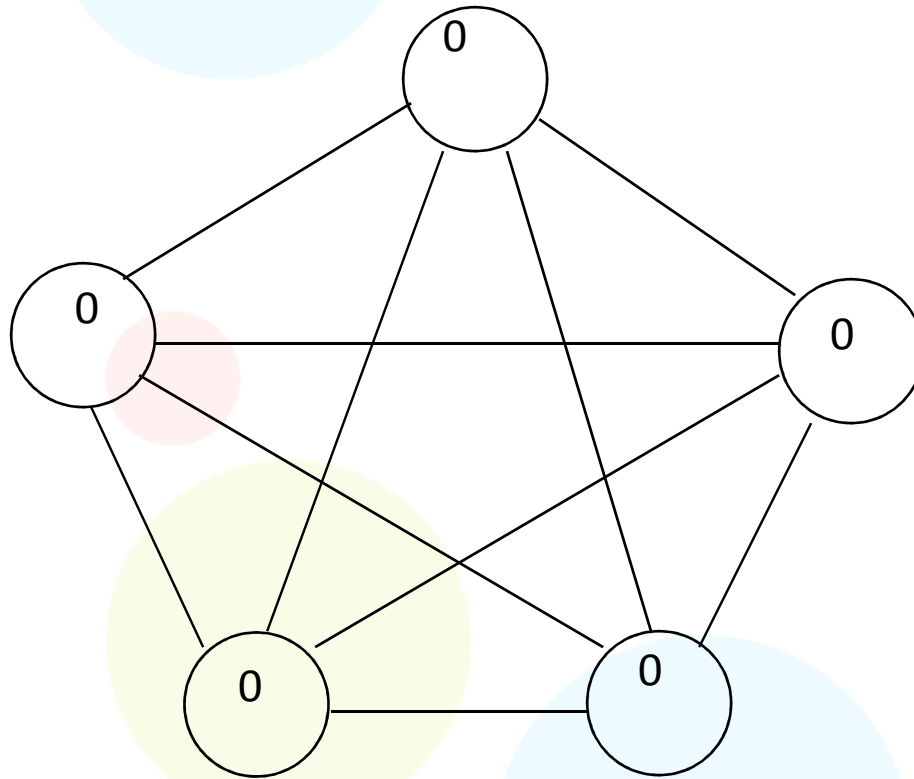
Broadcast values



Decide on minimum

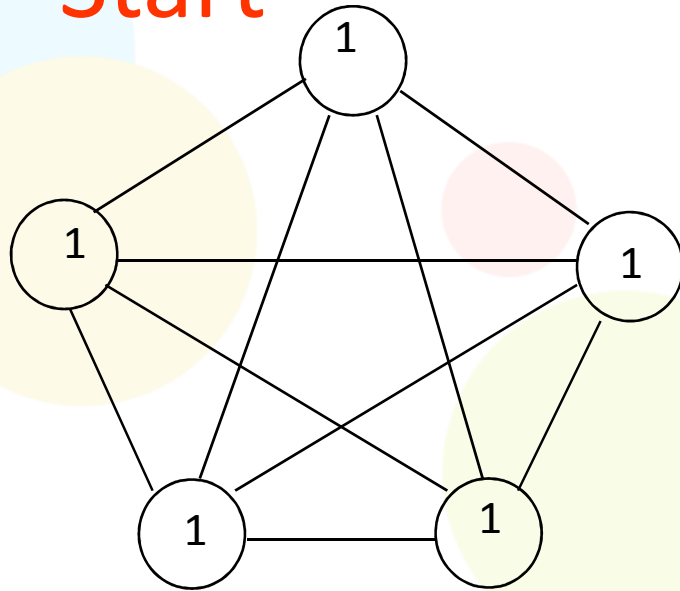


Finish

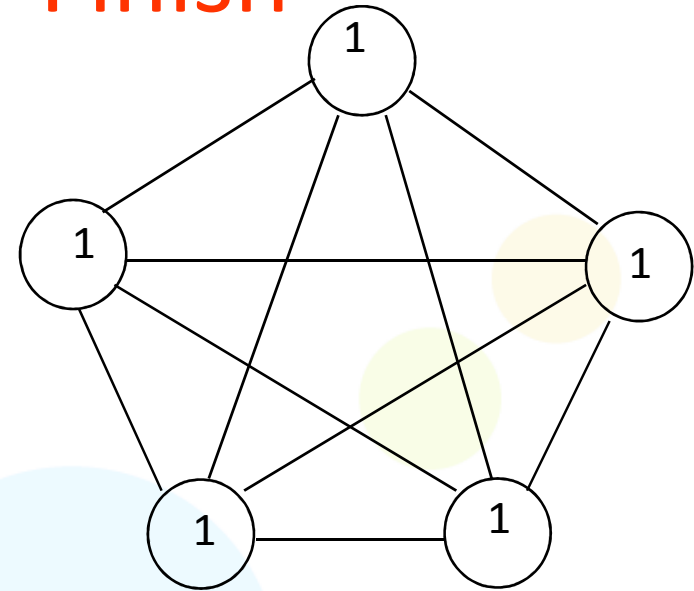


This algorithm satisfies the **validity** condition

Start



Finish



If everybody starts with the same initial value, everybody decides on that value (minimum)

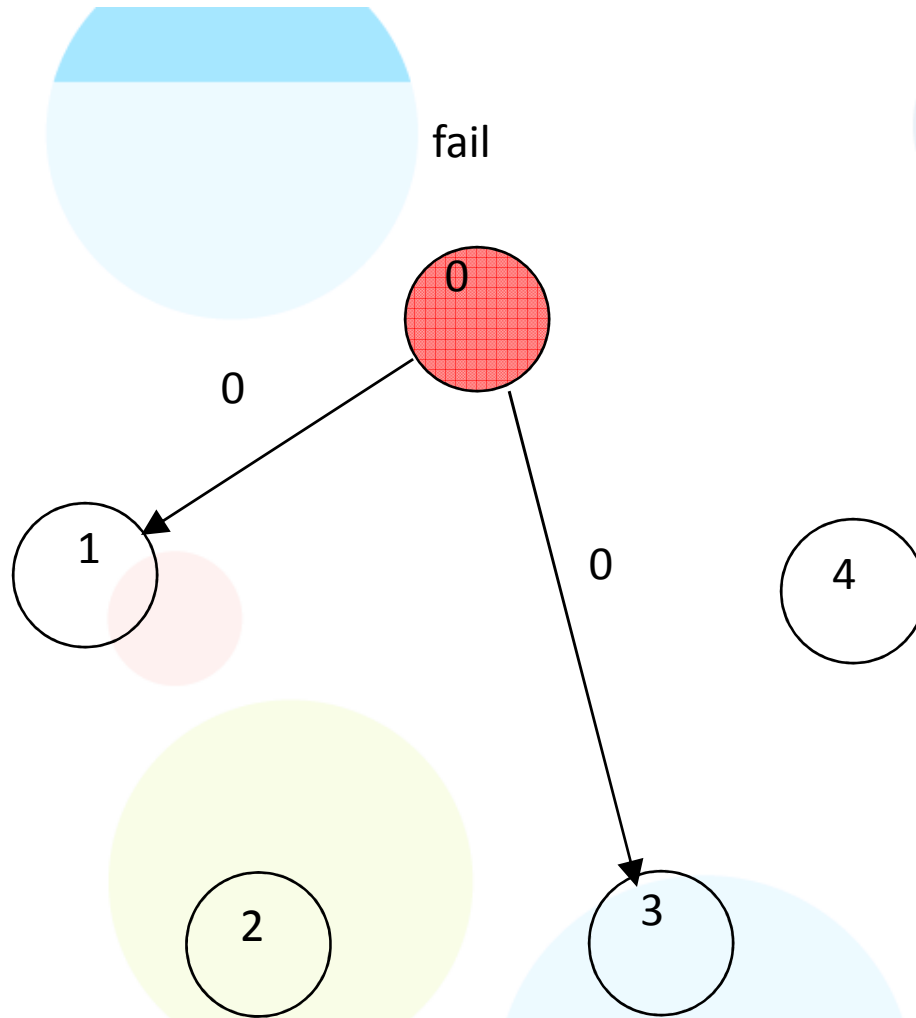
Consensus with **Crash** Failures

The simple algorithm doesn't work

Each processor:

1. Broadcast value to all processors
2. Decide on the minimum

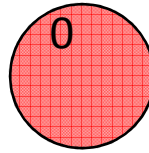
Start



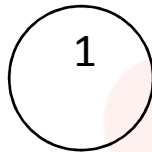
The failed processor doesn't broadcast its value to all processors

Broadcasted values

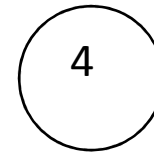
fail



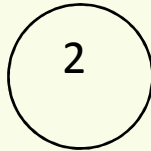
0,1,2,3,4



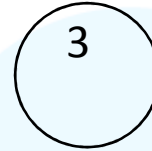
1,2,3,4



1,2,3,4

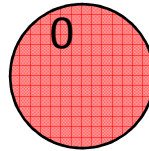


0,1,2,3,4

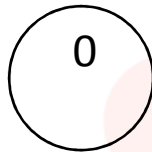


Decide on minimum

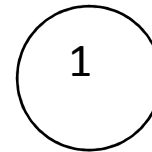
fail



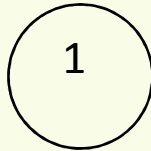
0,1,2,3,4



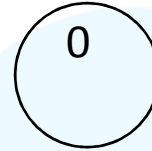
1,2,3,4



1,2,3,4

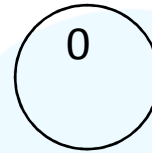
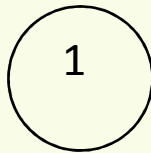
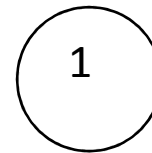
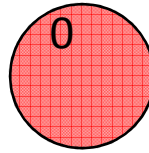


0,1,2,3,4



Finish

fail



No Consensus!!!



If an algorithm solves consensus for
f failed (crashing) processors we say it is:

an f-resilient consensus algorithm

An f -resilient algorithm

Round 1:

Broadcast my value

Round 2 to round $f+1$:

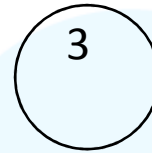
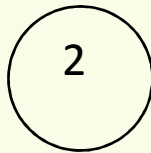
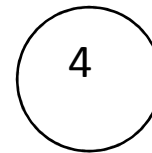
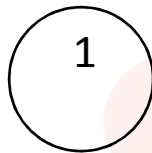
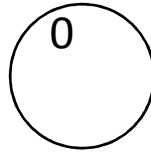
Broadcast any new received values

End of round $f+1$:

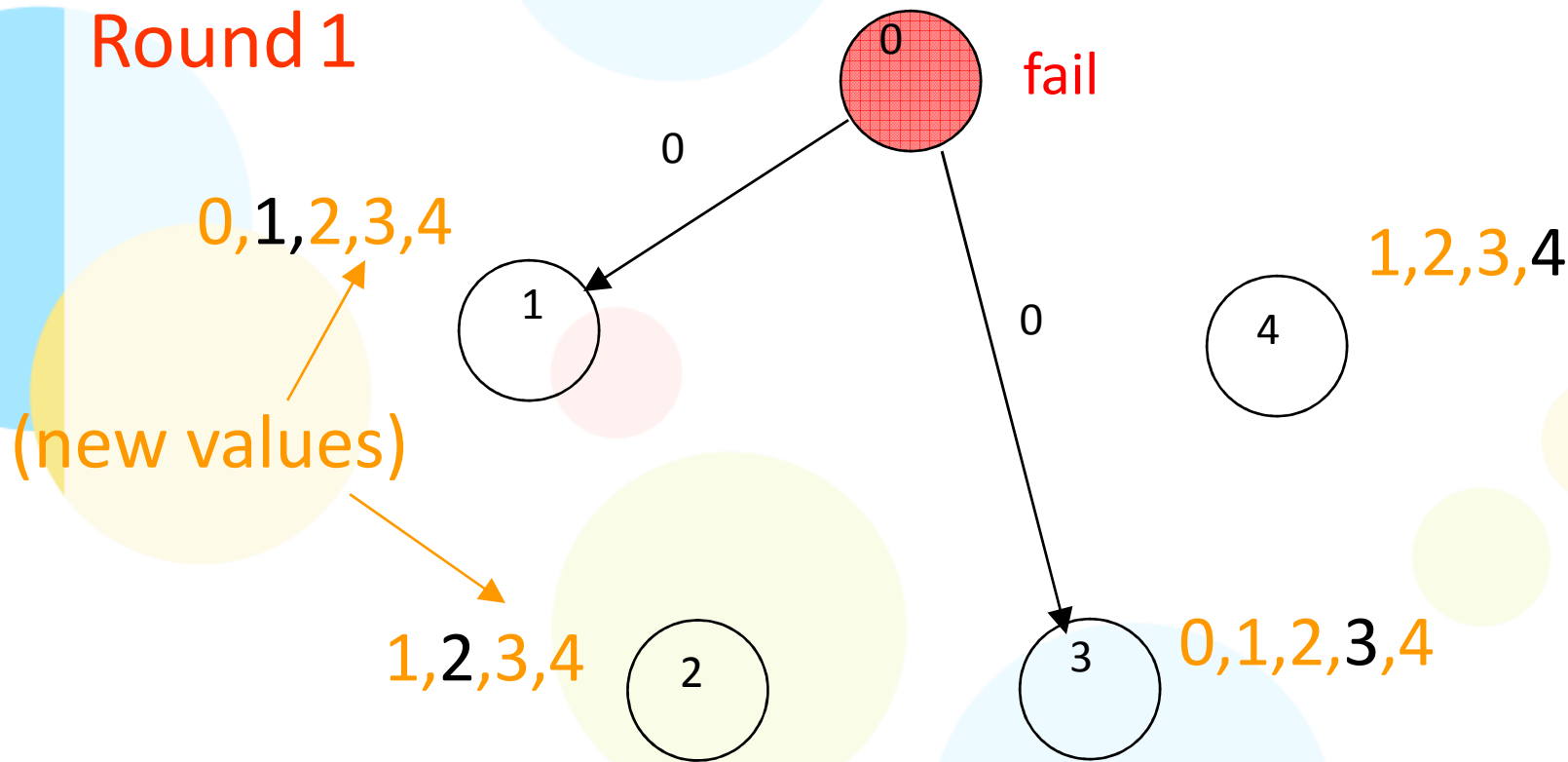
Decide on the minimum value received

Example: $f=1$ failures, $f+1 = 2$ rounds needed

Start



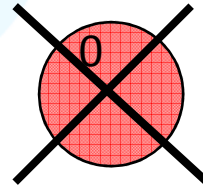
Example: $f=1$ failures, $f+1 = 2$ rounds needed



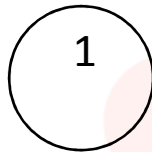
Broadcast all values to everybody

Example: $f=1$ failures, $f+1 = 2$ rounds needed

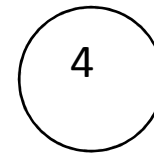
Round 2



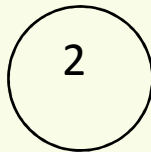
0,1,2,3,4



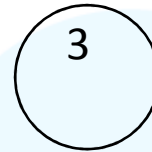
0,1,2,3,4



0,1,2,3,4



0,1,2,3,4

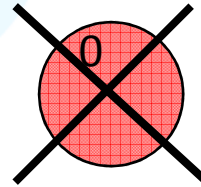
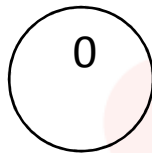


Broadcast all new values to everybody

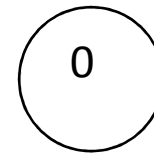
Example: $f=1$ failures, $f+1 = 2$ rounds needed

Finish

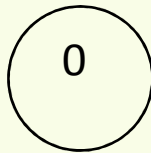
0,1,2,3,4



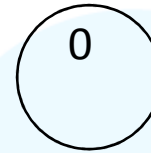
0,1,2,3,4



0,1,2,3,4



0,1,2,3,4



Decide on minimum value

Example: $f=2$ failures, $f+1 = 3$ rounds needed

Start

0

1

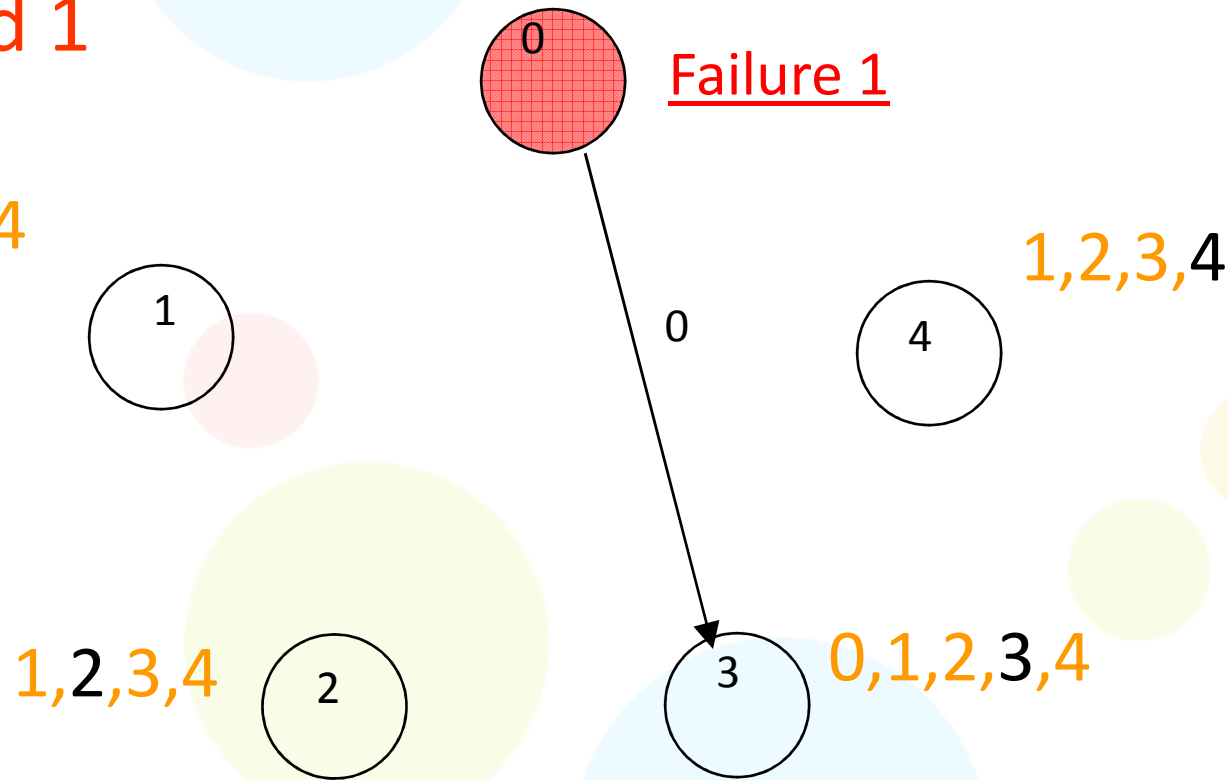
4

2

3

Example: $f=2$ failures, $f+1 = 3$ rounds needed

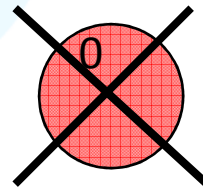
Round 1



Broadcast all values to everybody

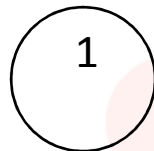
Example: $f=2$ failures, $f+1 = 3$ rounds needed

Round 2

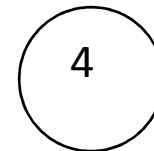


Failure 1

0,1,2,3,4

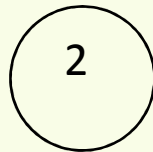


1,2,3,4

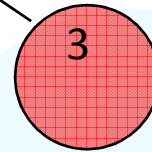


0

1,2,3,4



0,1,2,3,4

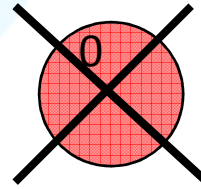


Failure 2

Broadcast new values to everybody

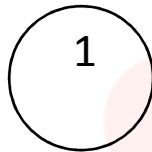
Example: $f=2$ failures, $f+1 = 3$ rounds needed

Round 3

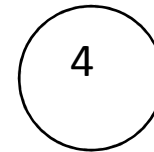


Failure 1

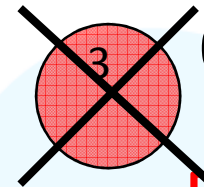
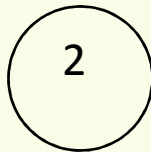
0,1,2,3,4



0,1,2,3,4



0,1,2,3,4



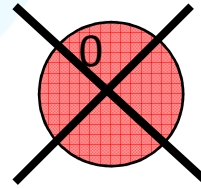
0,1,2,3,4

Failure 2

Broadcast new values to everybody

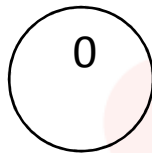
Example: $f=2$ failures, $f+1 = 3$ rounds needed

Finish

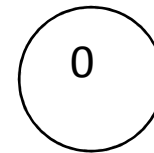


Failure 1

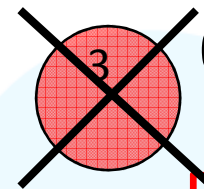
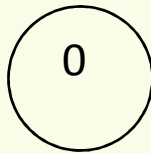
0,1,2,3,4



0,1,2,3,4



0,1,2,3,4



0,1,2,3,4

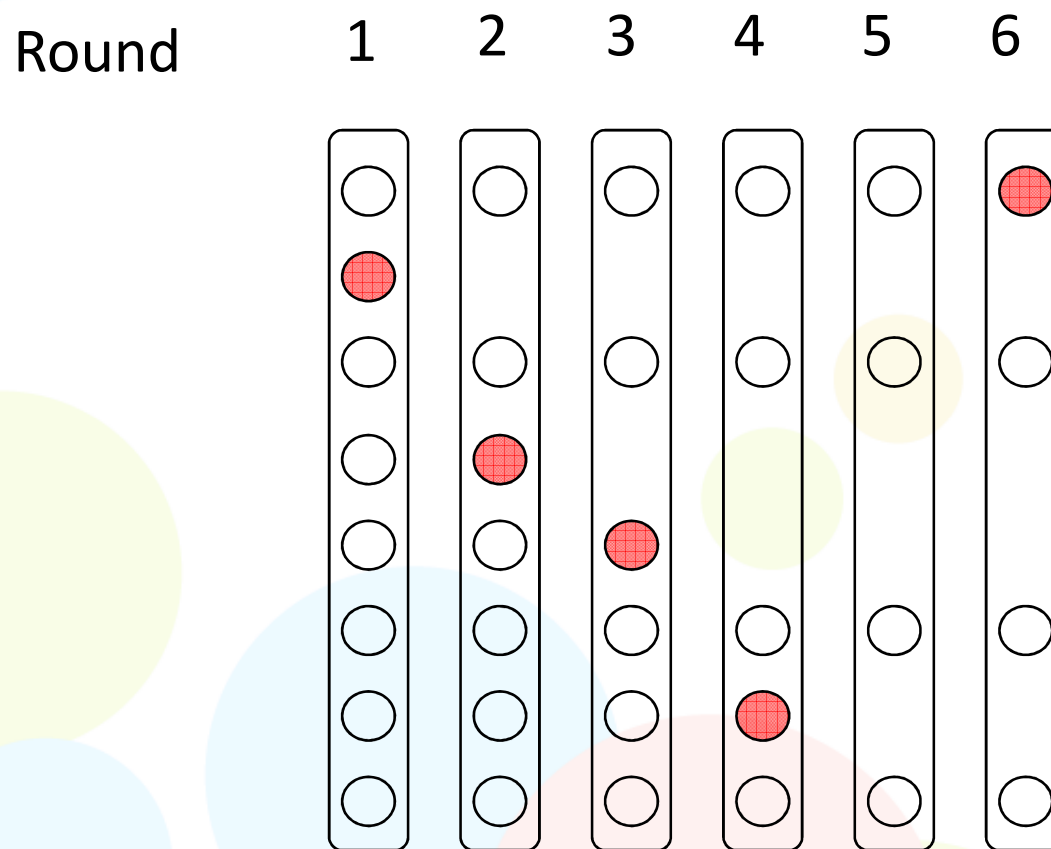
Failure 2

Decide on the minimum value

If there are f failures and $f+1$ rounds then there is at least a round with no failed processors:

Example:
5 failures,
6 rounds

No failure





In the algorithm, at the end of the round with no failure:

- Every (non faulty) process knows about all the values of all other participating processes
- This knowledge doesn't change until the end of the algorithm



Therefore, at the end of the round with no failure:

everybody would decide the same value

However, we don't know the exact position of this round, so we have to let the algorithm execute for $f+1$ rounds



Validity of algorithm

when all processes start with the same input value then the consensus is that value

This holds, since the value decided from each process is some input value

A Lower Bound

Theorem: Any f -resilient consensus algorithm requires at least $f+1$ rounds

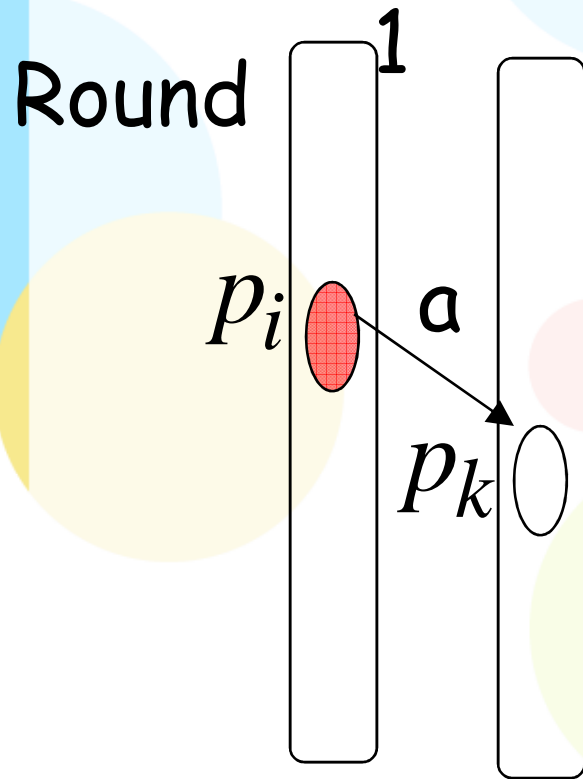
Proof sketch:

Assume for contradiction that f or less rounds are enough

Worst case scenario:

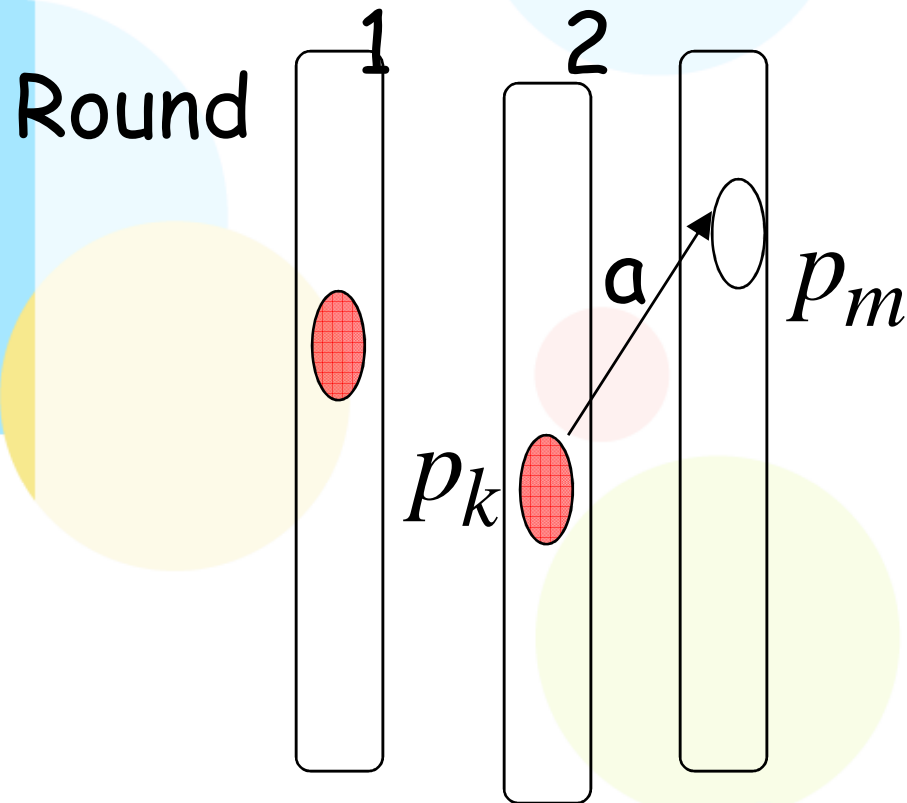
There is a process that fails in each round

Worst case scenario



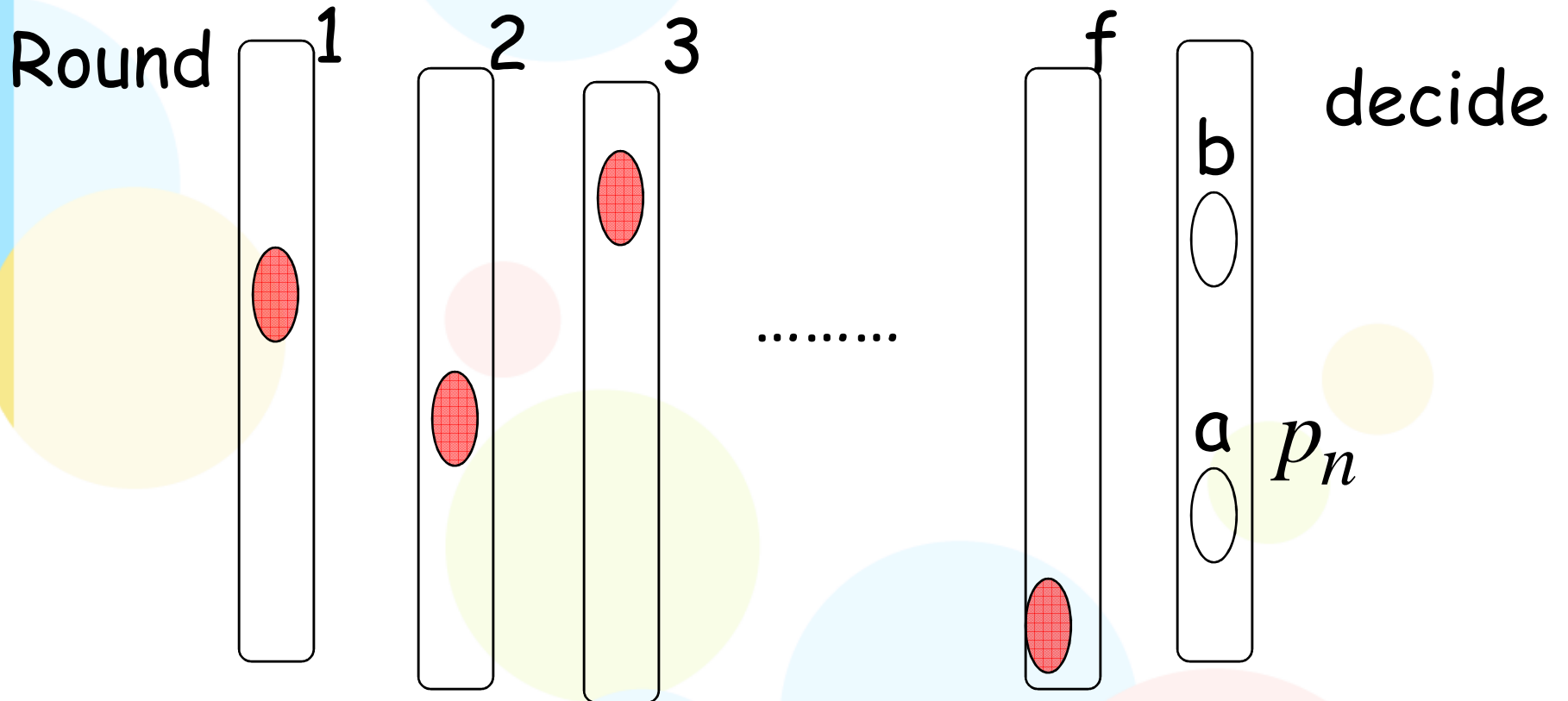
before process p_i fails, it sends its value a to only one process p_k

Worst case scenario



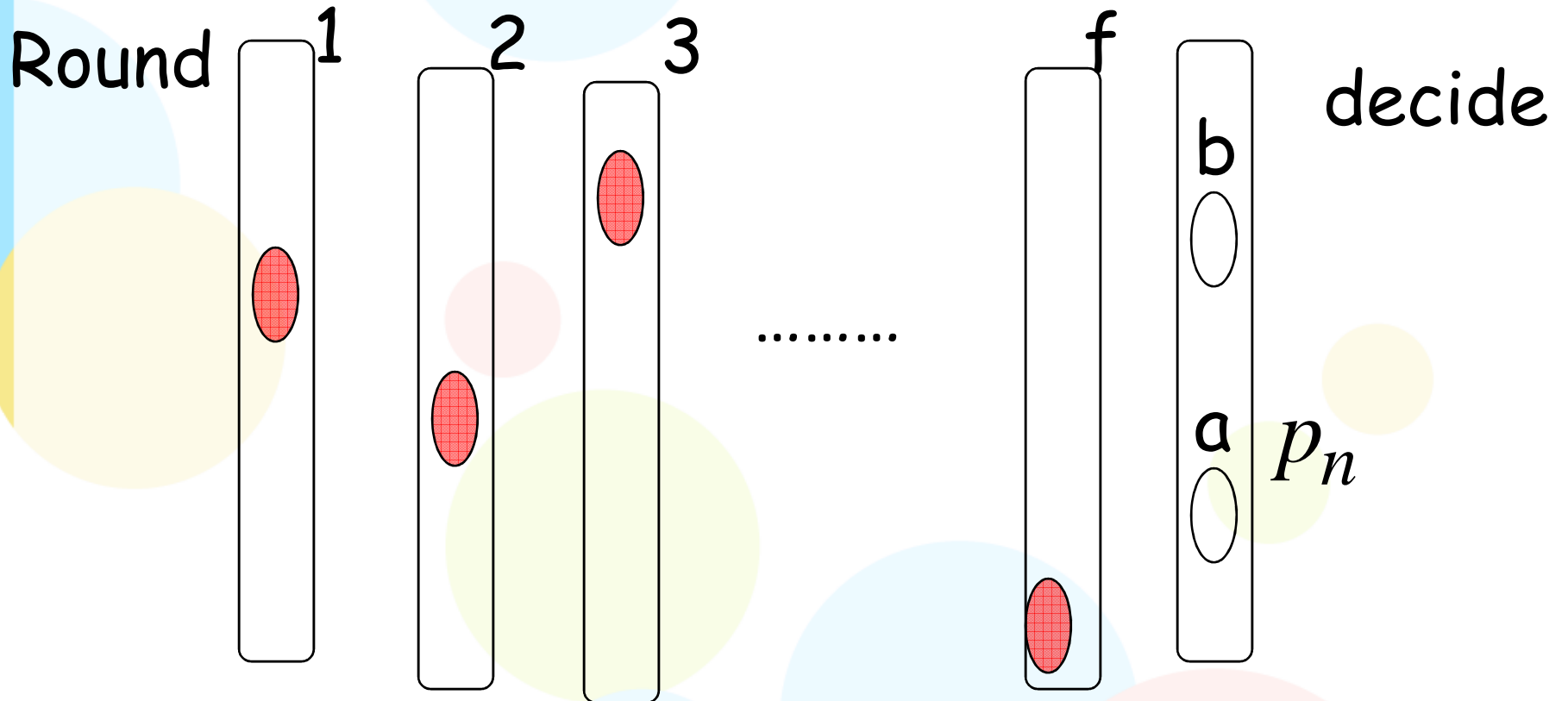
before process p_k fails, it sends value a
to only one process p_m

Worst case scenario



Process p_n may decide a, and all other processes may decide another value (b)

Worst case scenario



Therefore f rounds are not enough
At least $f+1$ rounds are needed

Consensus in synchronous systems

Up to f faulty processes

Duration of round:
max. delay of B-multicast

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}; Values_i^0 = \{\};$

In round r ($1 \leq r \leq f + 1$)

$B\text{-multicast}(g, Values_i^r - Values_i^{r-1});$ // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r;$

while (in round r)

{

On B-deliver(V_j) from some p_j
 $Values_i^{r+1} := Values_i^{r+1} \cup V_j;$

}

After $(f + 1)$ rounds

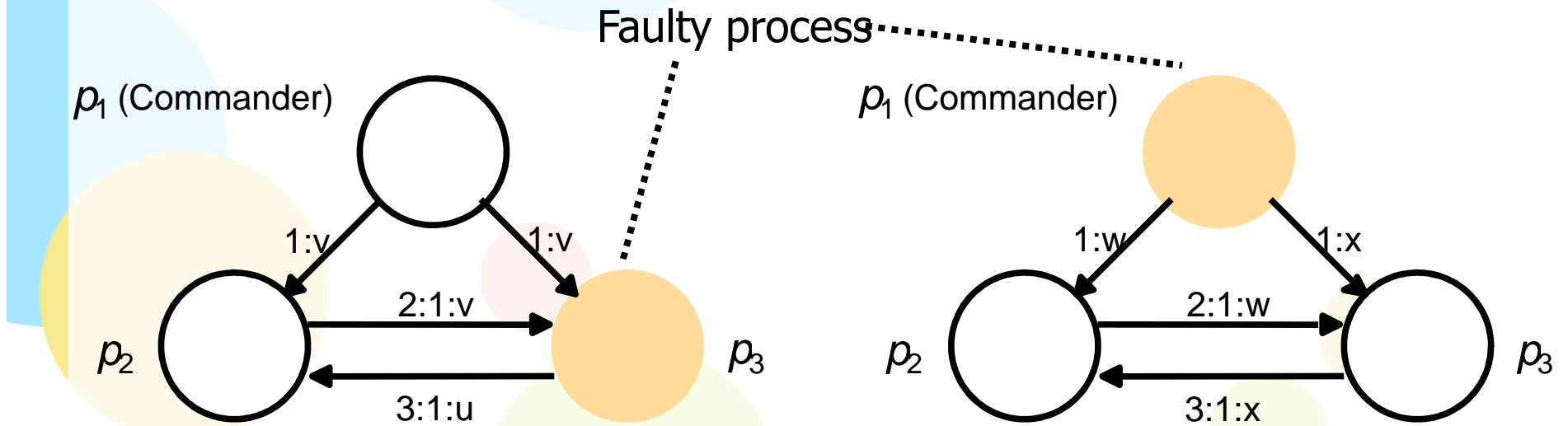
Assign $d_i = \text{minimum}(Values_i^{f+1});$

Only crashes,
no Byzantine
faults

Dolev & Strong, 1983:

Any algorithm to reach consensus despite
up to f failures requires $(f + 1)$ rounds.

Byzantine agreement: synchronous



3 says 1 says 'u'

Nothing can be done to improve a correct process' knowledge beyond the first stage:
- It cannot tell which process is faulty.

Lamport et al, 1982:

No solution for $N = 3, f = 1$

Pease et al, 1982:

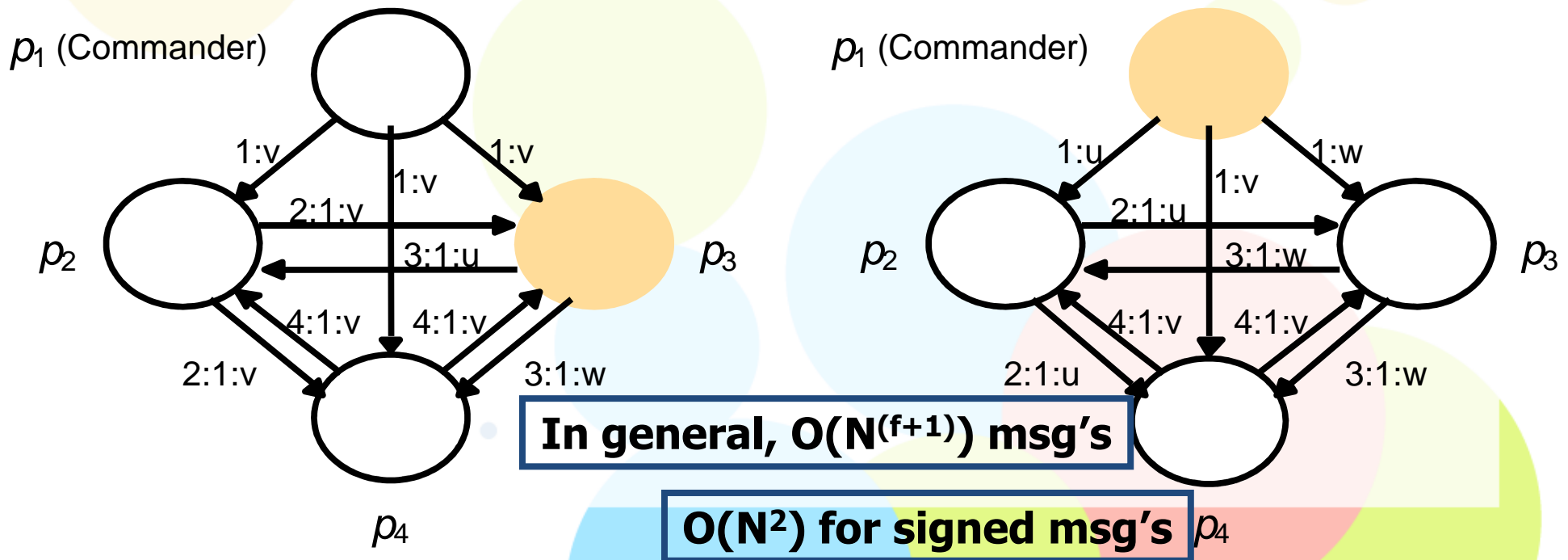
No solution for $N \leq 3*f$

(assuming private comm. channels)

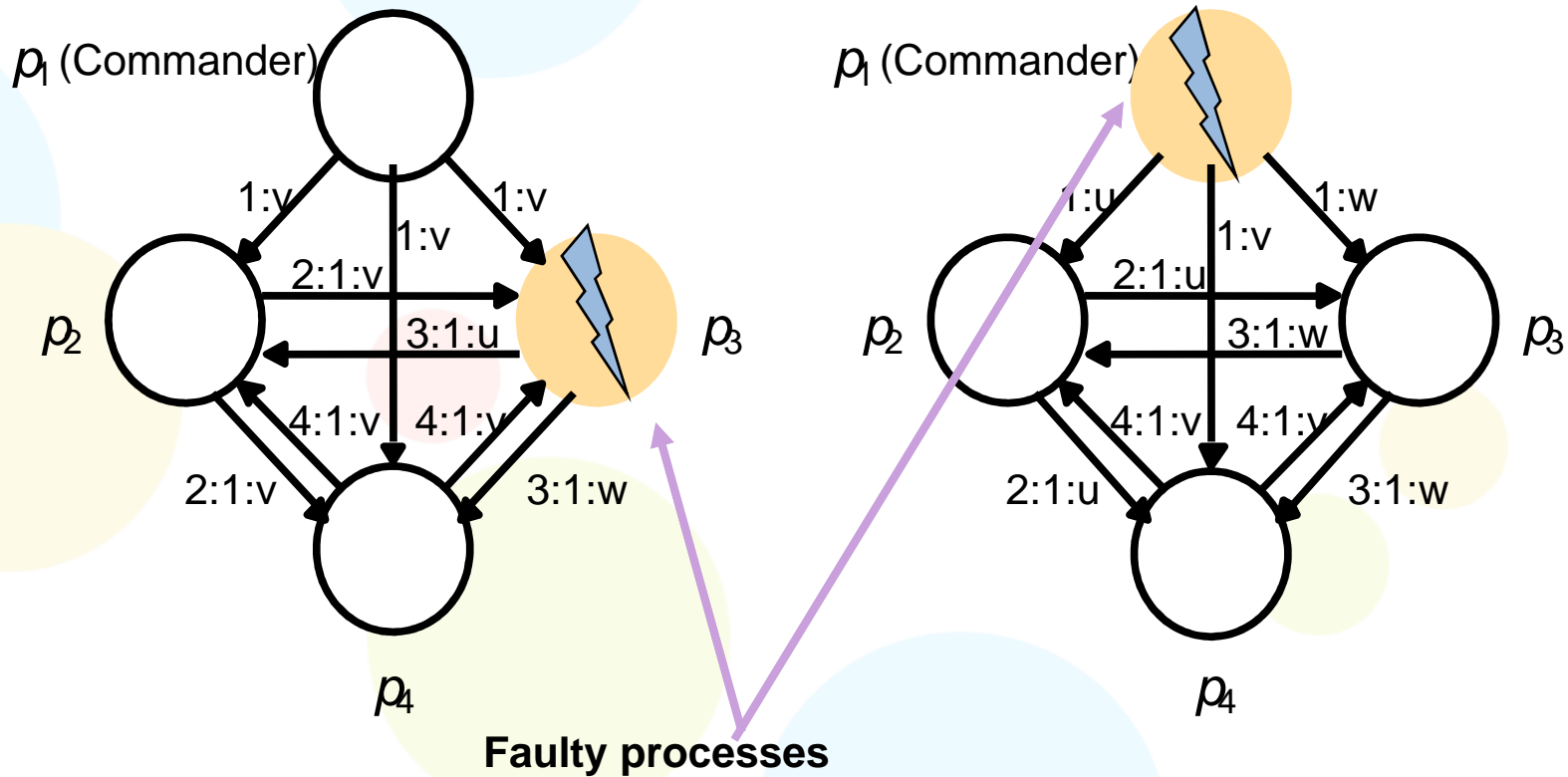
Byzantine agreement for $N > 3*f$

Example with $N=4, f=1$:

- 1st round: Commander sends a value to each lieutenant
- 2nd round: Each of the lieutenants sends the value it has received to each of its peers.
- A lieutenant receives a total of $(N - 2) + 1$ values, of which $(N - 2)$ are correct.
- By majority(), the correct lieutenants compute the same value.



Four Byzantine Generals: $N = 4, f = 1$ in a Synchronous DS



p_2 decides on $\text{majority}(v, u, v) = v$
 p_4 decides on $\text{majority}(v, v, w) = v$

p_2, p_3, p_4 decide on $\text{majority}(u, v, w) = \perp$

The background is a light gray gradient. It features several abstract geometric shapes: a large yellow circle on the left, a large light blue circle at the top center, a smaller dark blue circle at the top right, a small pink circle to the left of the text, a large light green circle below the text, a large light blue circle below that, a large pink circle at the bottom right, and a large light green circle at the bottom right. There are also several smaller circles and shapes in blue, yellow, and green, and a small gray dot near the bottom left.

Thank You