Consensus in Cloud Computing and Paxos

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the 'consensus problem' and its variants, its solvability under different failure models .
- We will also discuss the industry use of consensus using 'Paxos'.

Common issues in consensus

Consider a group of servers attempting together:

- Make sure that all of them receive the same updates in the same order as each other **[Reliable Multicast]**
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously **[Membership/Failure Detection]**
- Elect a leader among them, and let everyone in the group know about it **[Leader Election]**
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file **[Mutual Exclusion]**

So what is common?

- All of these were groups of processes attempting to ***coordinate*** with each other and reach ***agreement*** on the value of something
  - The ordering of messages
  - The up/down status of a suspected failed process
  - Who the leader is
  - Who has access to the critical resource
- All of these are related to the ***Consensus*** problem

What is a Consensus Problem?

Formal problem statement:

- N processes
- Each process p has

  input variable xp : initially either 0 or 1

  output variable yp : initially b (can be changed only once)
- **Consensus problem:** design a protocol so that at the end, either:
  1. All processes set their output variables to 0 (all-0's)
  2. Or All processes set their output variables to 1 (all-1's)

What is Consensus? (2)

- Every process contributes a value
- ***Goal is to have all processes decide same (some) value***
    - Decision once made can't be changed
- There might be other constraints:
    - **Validity:** if everyone proposes same value, then that's what's decided
    - **Integrity** : decided value must have been proposed by some process
    - **Non-triviality** : there is at least one initial system state that leads to each of the all-0's or all-1's outcomes

Why is it Important?

- Many problems in distributed systems are **equivalent to (or harder than)** consensus!
  - Perfect Failure Detection
  - Leader election (select exactly one leader, and every alive process knows about it)
  - Agreement (harder than consensus)
- So consensus is a very important problem, and solving it would be really useful!
- **So, is there a solution to Consensus?**

Two Different Models of Distributed Systems

- **Different Models of Distributed Systems:**
  (i) Synchronous System Model and
  (ii) **Asynchronous System Model**

(i) Synchronous Distributed System

- Each message is received within bounded time
- Drift of each process' local clock has a known bound
- Each step in a process takes lb < time < ub

E.g., A collection of processors connected by a communication bus, e.g., a Cray supercomputer or a multicore machine

Asynchronous System Model

(ii) Asynchronous Distributed System

- No bounds on process execution
- The drift rate of a clock is arbitrary
- No bounds on message transmission delays

E.g., The Internet is an asynchronous distributed system, so are ad-hoc and sensor networks

- This is a more **general (and thus challenging)** model than the synchronous system model.

- A protocol for an asynchronous system will also work for a synchronous system (but not vice-versa)

Possible or Not

- **In the synchronous system model:**
  - **Consensus is solvable**
- **In the asynchronous system model:**
  - **Consensus is impossible to solve**
  - Whatever protocol/algorithm you suggest, there is always a worst-case possible execution (with failures and message delays) that prevents the system from reaching consensus
  - **Powerful result (*FLP proof*)**
  - Subsequently, **safe or probabilistic solutions** have become quite popular to consensus or related problems.

FLP

- One of the most important results in distributed systems theory was published in April 1985 by **Fischer, Lynch and Patterson (FLP)**

- Their short paper 'Impossibility of Distributed Consensus with One Faulty Process', which won the Dijkstra award, placed an upper bound on what it is possible to achieve with distributed processes in an asynchronous environment.

Let's Try to Solve Consensus!

System model (Assumptions):

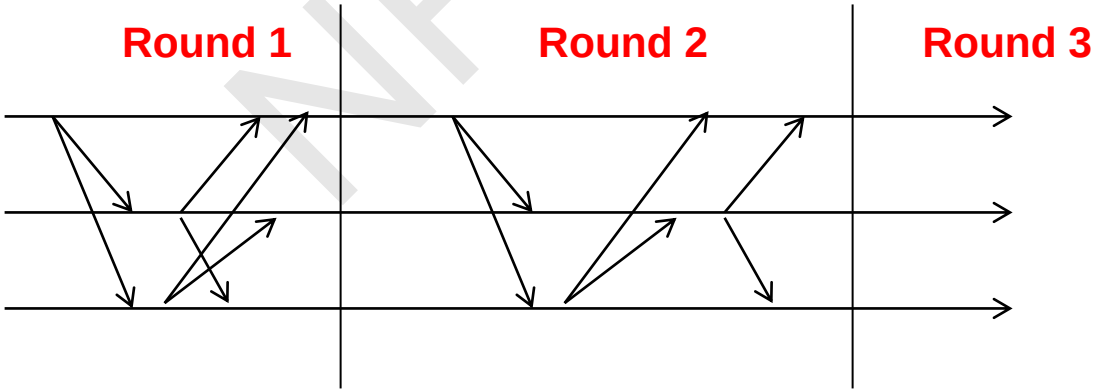
- **Synchronous system:** bounds on
  - Message delays
  - Upper bound on clock drift rates
  - Max time for each process step

  e.g., multiprocessor (common clock across processors)

- **Processes can fail by stopping (crash-stop or crash failures)**

Consensus in Synchronous Systems

- **For a system with at most *f* processes crashing**
  - All processes are synchronized and operate in **"rounds"** of time.
  - The algorithm proceeds in *f+1* rounds (with timeout), using reliable communication to all members
  - $Values^r_i$: the set of proposed values known to p_i at the beginning of round *r*.



Consensus in Synchronous System

Possible to achieve!

- For a system with at most *f* processes crashing
  - All processes are synchronized and operate in "rounds" of time
  - the algorithm proceeds in *f+1* rounds (with timeout), using reliable communication to all members.
  - $Values^r_i$: the set of proposed values known to p_i at the beginning of round *r*.

- Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i\}$

for round = 1 to *f+1* do

multicast ($Values^r_i - Values^{r-1}_i$) // iterate through processes, send each a message

$Values^{r+1}_i \leftarrow Values^r_i$

for each V_j received

$Values^{r+1}_i = Values^{r+1}_i \cup V_j$

end

end

d_i = **minimum**($Values^{f+1}_i$) // consistent minimum based on say, id (not minimum value)

Why does the Algorithm work?

- After *f+1* rounds, all non-faulty processes would have received the same set of Values. Proof by contradiction.
- Assume that two non-faulty processes, say p_i and p_j, differ in their final set of values (i.e., after *f+1* rounds)
- Assume that p_i possesses a value *v* that p_j does not possess.
  - p_i must have received *v* in the **very last** round
    - Else, p_i would have sent *v* to p_j in that last round
  - So, in the last round: a third process, p_k, must have sent *v* to p_i, but then crashed before sending *v* to p_j.
  - Similarly, a fourth process sending *v* in the **last-but-one round** must have crashed; otherwise, both p_k and p_j should have received *v*.
  - Proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds.
  - This means a total of *f+1* crashes, while we have assumed at most *f* crashes can occur => contradiction.

Consensus in an Asynchronous System

- **Impossible to achieve!**

- Proved in a now-famous result by **Fischer, Lynch and Patterson, 1983 (FLP)**
  - Stopped many distributed system designers dead in their tracks
  - A lot of claims of "reliability" vanished overnight

Recall

Asynchronous system: All message delays and processing delays can be arbitrarily long or short.

Consensus:

- Each process p has a state
  - program counter, registers, stack, local variables
  - input register xp : initially either 0 or 1
  - output register yp : initially b (undecided)
- **Consensus Problem:** design a protocol so that either
  - all processes set their output variables to 0 (all-0's)
  - Or all processes set their output variables to 1 (all-1's)
  - Non-triviality: at least one initial system state leads to each of the above two outcomes

Consensus Problem

- Consensus **impossible** to solve in asynchronous systems **(FLP Proof)**
  - **Key to the Proof:** It is impossible to distinguish a failed process from one that is just very very slow. Hence the rest of the alive processes may stay forever when it comes to deciding.

- But Consensus important since it maps to many important distributed computing problems.

Paxos

Paxos algorithm:

- Most popular "**consensus-solving**" algorithm
- Does not solve consensus problem (which would be impossible, because we already proved that)
- But provides **safety** and **eventual liveness.**
- A lot of systems use it
  - Zookeeper (Yahoo!), Google Chubby, and many other companies

Paxos

- Paxos is invented by **Leslie Lamport**

- Paxos provides **safety** and **eventual liveness**
  - **Safety:** Consensus is not violated
  - **Eventual Liveness:** If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.

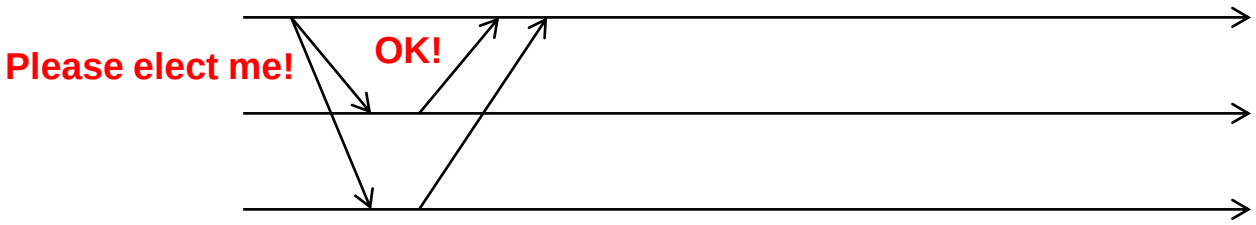
- FLP result still applies: Paxos is not ***guaranteed*** to reach Consensus (ever, or within any bounded time)

Paxos Algorithm

- Paxos has **rounds**; each round has a unique ballot id
- Rounds are asynchronous
  - Time synchronization not required
  - If you're in round *j* and hear a message from round *j+1*, abort everything and move over to round *j+1*
  - Use timeouts; may be pessimistic
- Each round itself broken into phases (which are also asynchronous)
  - Phase 1: A leader is elected (**Election**)
  - Phase 2: Leader proposes a value, processes ack
  - Phase 3: Leader multicasts final value

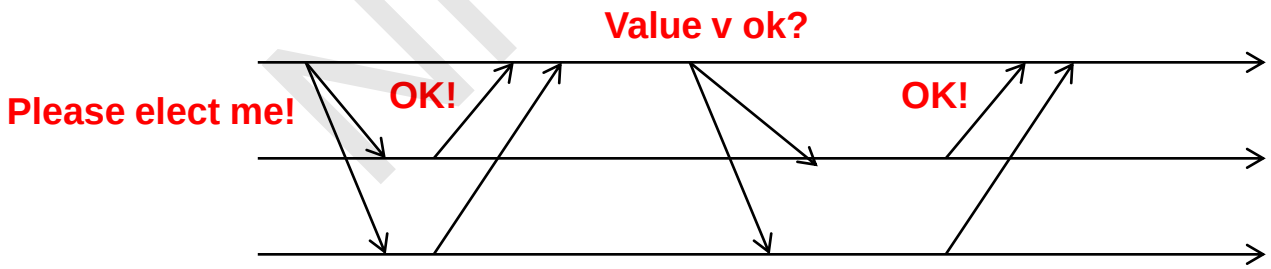
Phase 1: Election

- Potential leader chooses a unique ballot id, higher than seen anything so far
- Sends to all processes
- Processes wait, respond once to highest ballot id
  - If potential leader sees a higher ballot id, it can't be a leader
  - Paxos tolerant to multiple leaders, but we'll only discuss 1 leader case
  - Processes also **log** received ballot ID on disk
- If a process has in a previous round decided on a value v', it includes value v' in its response
- If **majority (i.e., quorum)** respond OK then you are the leader
  - If no one has majority, start new round
- (If things go right) A round cannot have two leaders (why?)



Phase 2: Proposal

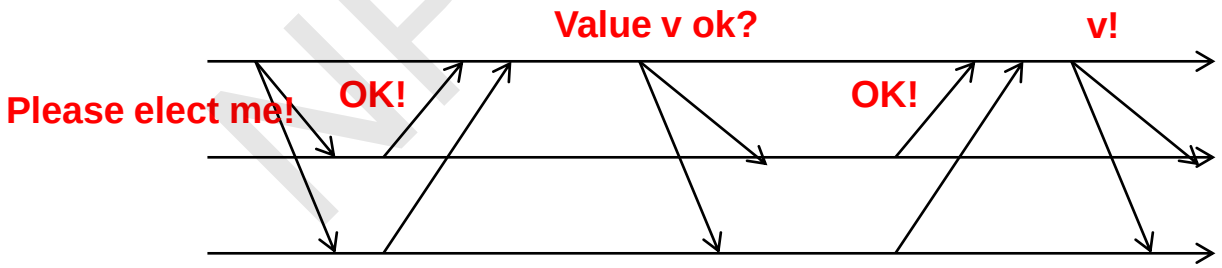
- Leader sends proposed value v to all
  - use v=v’ if some process already decided in a previous round and sent you its decided value v’
  - If multiple such v’ received, use latest one
- Recipient logs on disk; responds OK



Phase 3: Decision

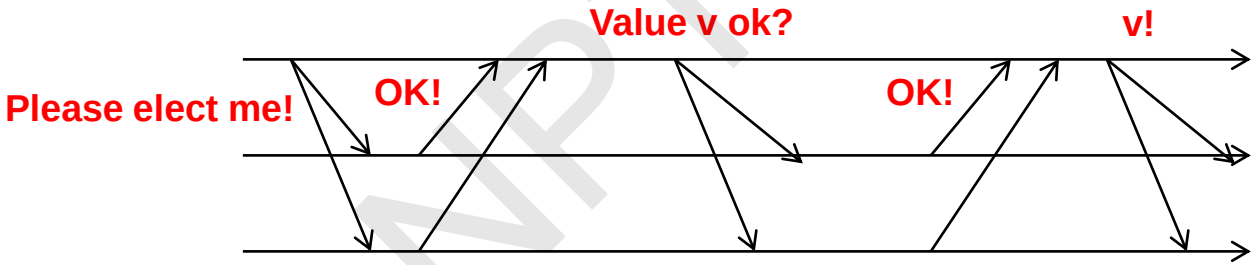
- If leader hears a **majority** of OKs, it lets everyone know of the decision

- Recipients receive decision, log it on disk



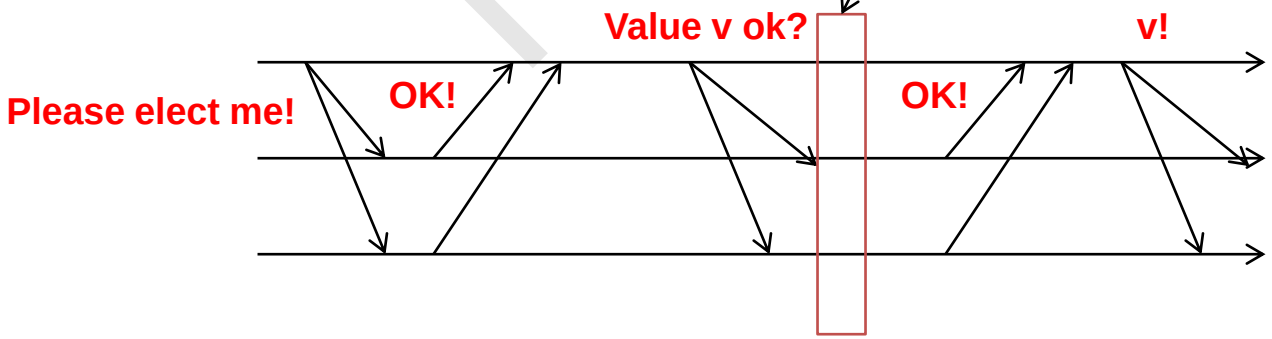
Which is the point of No-Return?

- That is, when is consensus reached in the system



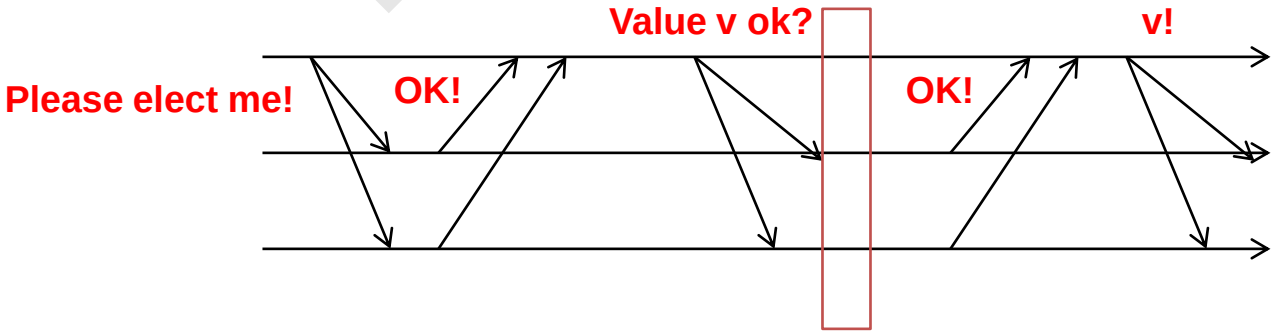
Which is the point of No-Return?

- If/when a majority of processes hear proposed value and accept it (i.e., are about to/have respond(ed) with an OK!)
- Processes ***may not know it yet,*** but a decision has been made for the group
  - Even leader does not know it yet
- What if leader fails after that?
  - Keep having rounds until some round completes



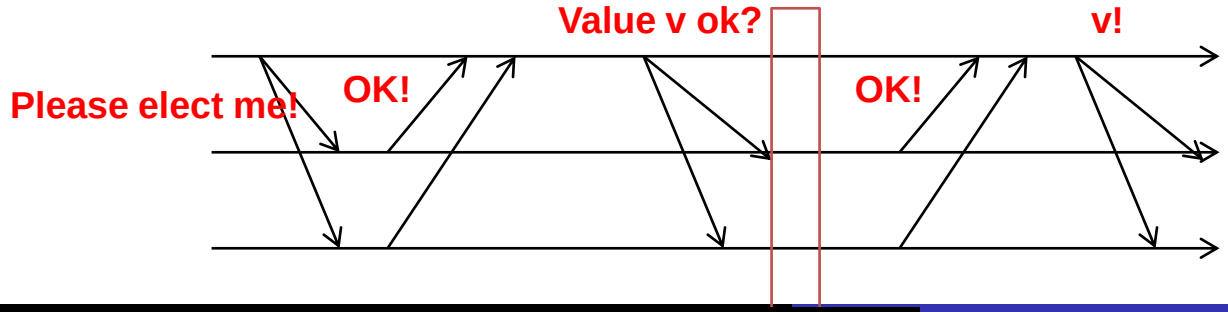
Safety

- If some round has a majority (i.e., quorum) hearing proposed value v' and accepting it, then subsequently at each round either: 1) the round chooses v' as decision or 2) the round fails
- **Proof:**
  - Potential leader waits for majority of OKs in Phase 1
  - At least one will contain v' (because two majorities or quorums always intersect)
  - It will choose to send out v' in Phase 2
- Success requires a majority, and any two majority sets intersect



What could go Wrong?

- **Process fails**
  - Majority does not include it
  - When process restarts, it uses log to retrieve a past decision (if any) and past-seen ballot ids. Tries to know of past decisions.
- **Leader fails**
  - Start another round
- **Messages dropped**
  - If too flaky, just start another round
- **Note that anyone can start a round any time**
- **Protocol may never end**
  - Impossibility result not violated
  - If things go well sometime in the future, consensus reached



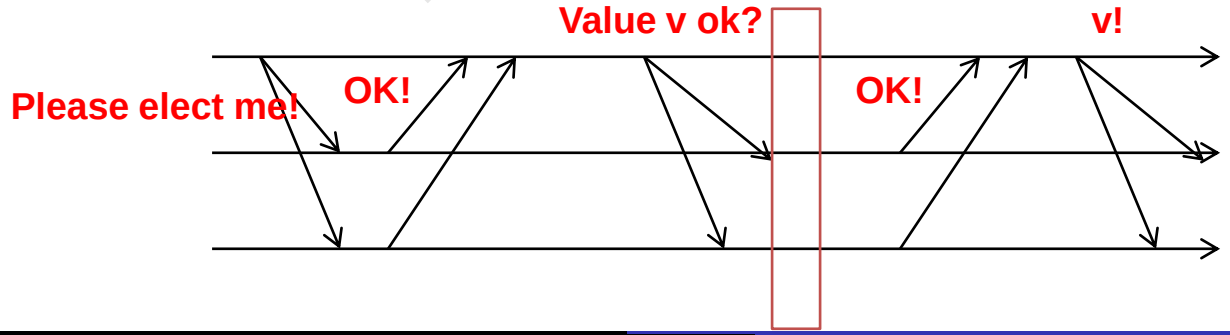
What could go Wrong?

- A lot more!
- This is a highly simplified view of Paxos.
- Lamport's original paper:

The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.



Conclusion

- Consensus is a very important problem
  - **Equivalent to many important distributed computing problems that have to do with reliability**
- **Consensus is possible to solve in a synchronous system where message delays and processing delays are bounded**
- **Consensus is impossible to solve in an asynchronous system where these delays are unbounded**
- **Paxos protocol:** widely used implementation of a safe, eventually-live consensus protocol for asynchronous systems
  - Paxos (or variants) used in Apache Zookeeper, Google's Chubby system, Active Disk Paxos, and many other cloud computing systems

Byzantine Agreement

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

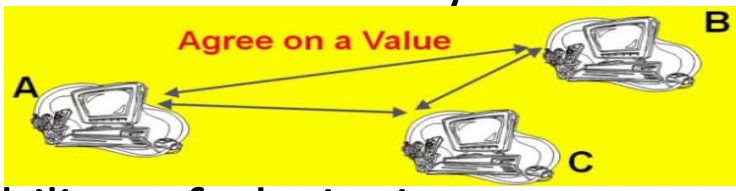
Preface

Content of this Lecture:

- In this lecture, we will discuss about **'Agreement Algorithms for Byzantine processes'**.
- This lecture first covers different forms of the **'consensus problem'** then gives an overview of what forms of consensus are solvable under different failure models and different assumptions on the synchrony/asynchrony.
- Also covers agreement in the category of:
  (i) Synchronous message-passing systems with failures and
  (ii) Asynchronous message-passing systems with failures.

Introduction

- **Agreement among the processes** in a distributed system is a fundamental requirement for a wide range of applications.
- Many forms of coordination 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**, wherein the processes collectively decide whether to commit or abort a transaction that they participate in.



- In this lecture, we will study the feasibility of designing algorithms to reach agreement under various system models and failure models, and, where possible, examine some representative algorithms to reach agreement.

Classification of Faults: Overview

- **Based on components that failed**

– Program / process
– Processor / machine
– Link
– Storage

- **Based on behavior of faulty component**

– Crash – just halts
–Fail stop – crash with additional conditions
– Omission– fails to perform some steps
– Byzantine– behaves arbitrarily
– Timing – violates timing constraints

Classification of Tolerance: Overview

- **Types of tolerance:**

– **Masking –** System always behaves as per specifications even in presence of faults.
–**Non-masking–** System may violate specifications in presence of faults. Should at least behave in a well-defined manner.

- **Fault tolerant system should specify:**

– Class of faults tolerated
– What tolerance is given from each class

Measuring Reliability and Performance

- **Distributed systems:**
  - Improve performance
  - Improve reliability
- Or do they? Need to measure to know.
- Need a vocabulary

SLIs, SLOs, SLAs, TLAs

- SLI = Service Level Indicator
  ⇒ What you are measuring

- SLO = Service Level Objective
  ⇒ How good should it be?

- SLA = Service Level Agreement
  ⇒ SLO + consequences

- TLA = Three Letter Acronym

Why study SLIs, SLOs, and SLAs?

- If you measure it, you can improve it
- Learn what matters
  - Don't waste time on things that don't matter!
- Reliability promises are part of business

Reading an SLA

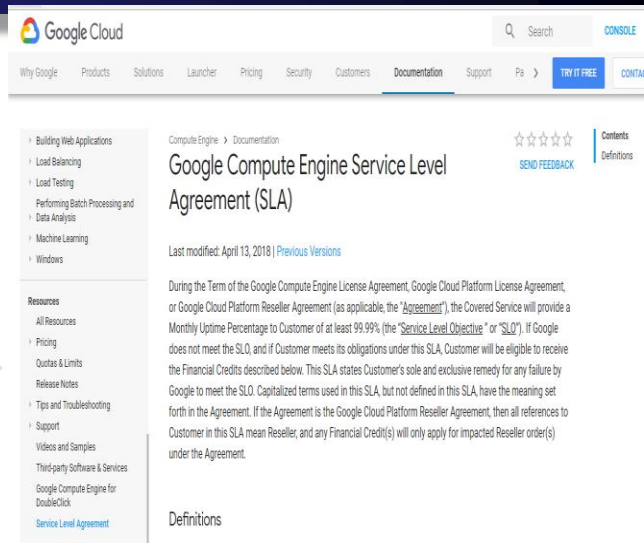
- **"I promise 99% uptime"**
- How often do you check if your system is up?
  - Sampling frequency
- What does it mean to be "up"?
  - Domain of responsibility
- Over what time interval do you promise 99% uptime?
  - Measurement interval

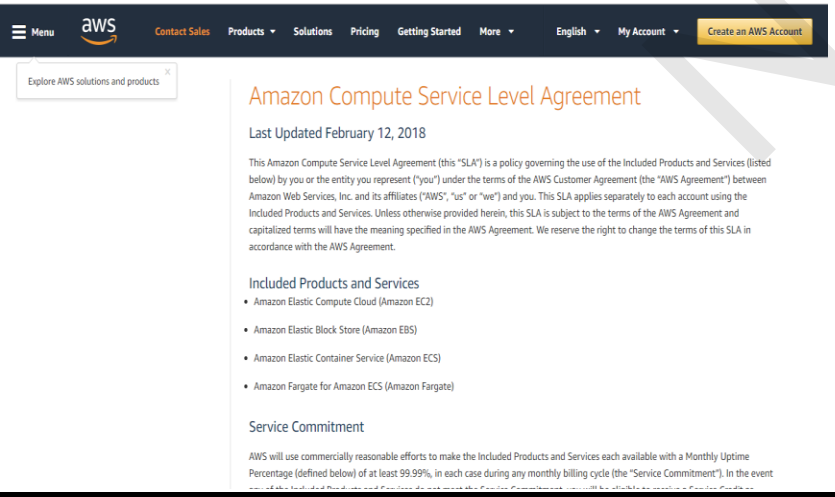
How many nines?

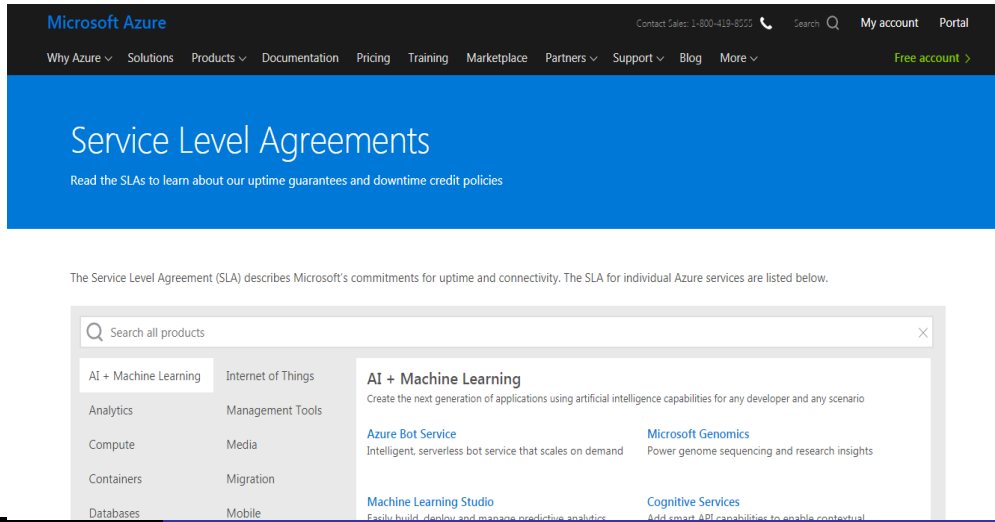
Nines	Uptime	Downtime/month
1	90%	3 days
2	99%	7 hours
3	99.9%	43 minutes
4	99.99%	4 minutes
5	99.999%	25 seconds (5m/year)

Cloud VM providers

- **Consider Microsoft Azure, Amazon EC2, Google GCE (Google Compute Engine)**
- Promise 99.95% uptime (22 minutes downtime/month)
  - Better than my net connection… right?
- 1-minute sampling frequency
  - GCE doesn't count <5 minute outages







What does the SLA imply for provider?

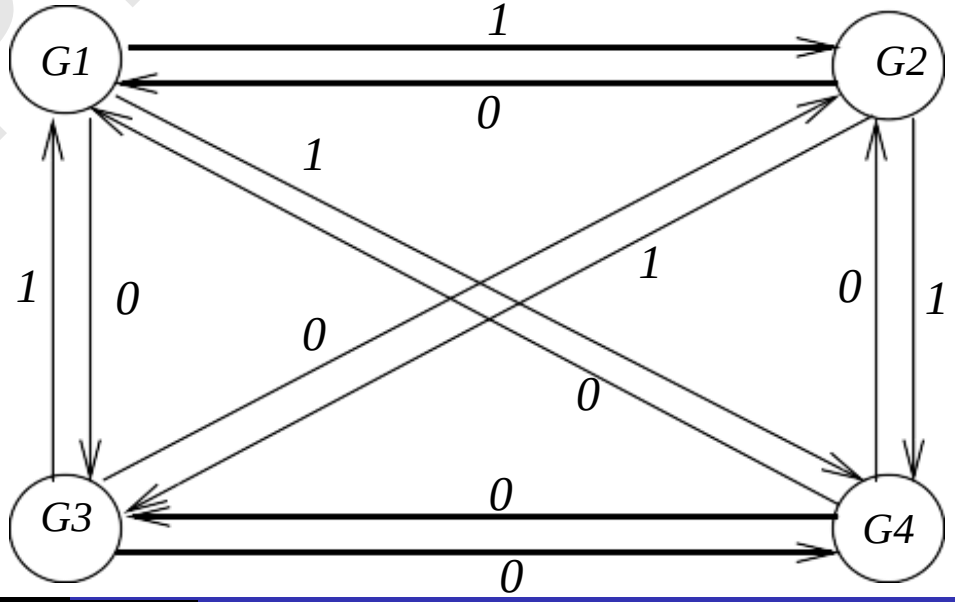
- **99.95% (or 22 minutes/month downtime) means either:**
  - They rarely expect their hardware or software to fail
  - When it fails they think they can fix it quickly

What does the SLA imply for you?

- **SLA requires you to have:**
  - Multiple VMs
  - Over multiple failure domains
  - Automatic failover
  - Monitoring
  - Tolerance of planned outages
  - Automatic machine provisioning (GCE)

Assumptions

1. Failure models
2. Synchronous/ Asynchronous communication
3. Network connectivity
4. Sender identification
5. Channel reliability
6. Authenticated vs. Non-authenticated messages
7. Agreement variable



1) Failure models

- A failure model specifies the manner in which the component(s) of the system may fail.

- There exists a rich class of **well-studied failure models**. The various process failure models are: (i) Fail-stop, (ii) Crash, (iii) Receive omission, (iv) Send omission, (v) General omission, and (vi) Byzantine or malicious failures

- 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.

Type of Process Failure Models

i. **Fail-stop:** In this model, a properly functioning process may fail by stopping execution from some instant thenceforth. Additionally, other processes can learn that the process has failed.

ii. **Crash:** In this model, a properly functioning process may fail by stopping to function from any instance thenceforth. Unlike the fail-stop model, other processes do not learn of this crash.

iii. **Receive omission:** A properly functioning process may fail by intermittently receiving only some of the messages sent to it, or by crashing.

iv. **Send omission:** A properly functioning process may fail by intermittently sending only some of the messages it is supposed to send, or by crashing.

Contd…

v. **General omission:** A properly functioning process may fail by exhibiting either or both of send omission and receive omission failures.

vi. **Byzantine or malicious failure:** In this model, a process may exhibit any arbitrary behavior and no authentication techniques are applicable to verify any claims made.

2) Synchronous/Asynchronous Computation

-Synchronous Computation:

i.Processes run in lock step manner **[Process receives a message sent to it earlier, performs computation and sends a message to other process.]**
ii.Step of Synchronous computation is called ***'round'***

-Asynchronous Computation:

i.Computation does not proceed in lock step.
ii.Process can send receive messages and perform computation at any time.

3) Network connectivity

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

4) Sender identification

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

5) 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.

6) Authenticated vs. Non-authenticated messages

- In this study, we will be **dealing only with unauthenticated messages**.

- With **unauthenticated messages**, when a faulty process relays a message to other processes, (i) it can forge the message and claim that it was received from another process, and (ii) it can also tamper with the contents of a received message before relaying it.

- Using **authentication** via techniques such as digital signatures, it is easier to solve the agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering.

7) Agreement variable

- The agreement variable **may be boolean or multivalued**, 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.

Performance Aspects of Agreement Protocols

Few Performance Metrics are as follows:

(i) Time: No of rounds needed to reach an agreement

(ii) Message Traffic: Number of messages exchanged to reach an agreement.

(iii) Storage Overhead: Amount of information that needs to stored at processors during execution of the protocol.

Problem Specifications

1. Byzantine Agreement Problem (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.

2. 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.

Contd…

3. Interactive Consistency Problem (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.

Equivalence of the Problems

- The three problems defined above are equivalent in the sense that a solution to any one of them can be used as a solution to the other two problems. This equivalence can be shown using a reduction of each problem to the other two problems.
- **If problem A is reduced to problem B, then a solution to problem B can be used as a solution to problem A in conjunction with the reduction.**

- Formally, the **difference between the agreement problem and the consensus problem** is that, in the agreement problem, a single process has the initial value, whereas in the consensus problem, all processes have an initial value.
- However, the two terms are used interchangeably in much of the literature and hence we shall also use the terms interchangeably.

Overview of Results

- **Table 10.1** gives an overview of the results and lower bounds on solving the consensus problem under different assumptions.

- It is worth understanding the relation between the consensus problem and the problem of attaining common knowledge of the agreement value. For the **"no failure"** case, consensus is attainable.

- Further, in a synchronous system, common knowledge of the consensus value is also attainable, whereas in the asynchronous case, concurrent common knowledge of the consensus value is attainable.

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 10.1: 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

Contd…

- **Consensus is not solvable in asynchronous systems** even if one process can fail by crashing.

- **Figure 10.1** shows further how asynchronous message-passing systems and shared memory systems deal with trying to solve consensus.

Solvable Variants of the Consensus Problem in Asynchronous 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.

Weaker Consensus Problems in Asynchronous System

Consensus Problem	Description
Terminating reliable broadcast	It states that a correct process always gets a message even if the sender crashes while sending. If the sender crashes while sending the message, the message may be a null message but it must be delivered to each correct process.
k-set consensus	It is solvable as long as the number of crash failures f is less than the parameter k. The parameter k indicates that the non-faulty processes agree on different values, as long as the size of the set of values agreed upon is bounded by k.
Approximate agreement	Like k-set consensus, approximate agreement also assumes the consensus value is from a multi-valued domain. However, rather than restricting the set of consensus values to a set of size k, ϵ-approximate agreement requires that the agreed upon values by the non-faulty processes be within ϵ of each other.
Renaming problem	It requires the processes to agree on necessarily distinct values.
Reliable broadcast	A weaker version of reliable terminating broadcast(RTB), namely reliable broadcast, in which the termination condition is dropped, is solvable under crash failures.

Contd…

- To circumvent the impossibility result, weaker variants of the consensus problem are defined in **Table 10.2**.

- The overheads given in this table are for the algorithms described.

Some Solvable Variants of the Consensus Problem in Asynchronous 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
ϵ-agreement	crash failures $n \geq 5f + 1$ (MP)	values agreed upon are within s of each other
Renaming	up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash failures $f \leq n - 1$ (SM)	select a unique name from a set of names

Table 10.2: 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.

Here MP- Message Passing, SM- Shared Memory

Agreement in Synchronous Message-Passing Systems with Failures

Byzantine Failure

- "Not fail stop"

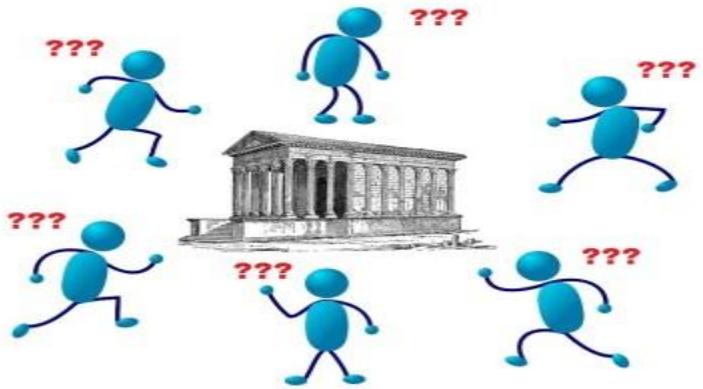
- ***Traitor*** nodes send conflicting messages
  - Which leads to an incorrect result

- **Cause:**
  - Flaky node(s)
  - Malicious node(s)

Why study Byzantine failure?

- **Extreme fault tolerance:**
  - Bitcoin
  - Boeing 777 & 787 flight controls

- **Solving this problem is fun!**
  - This reason has really driven a lot of research, since at least the **1980's**



What assumptions are you making?

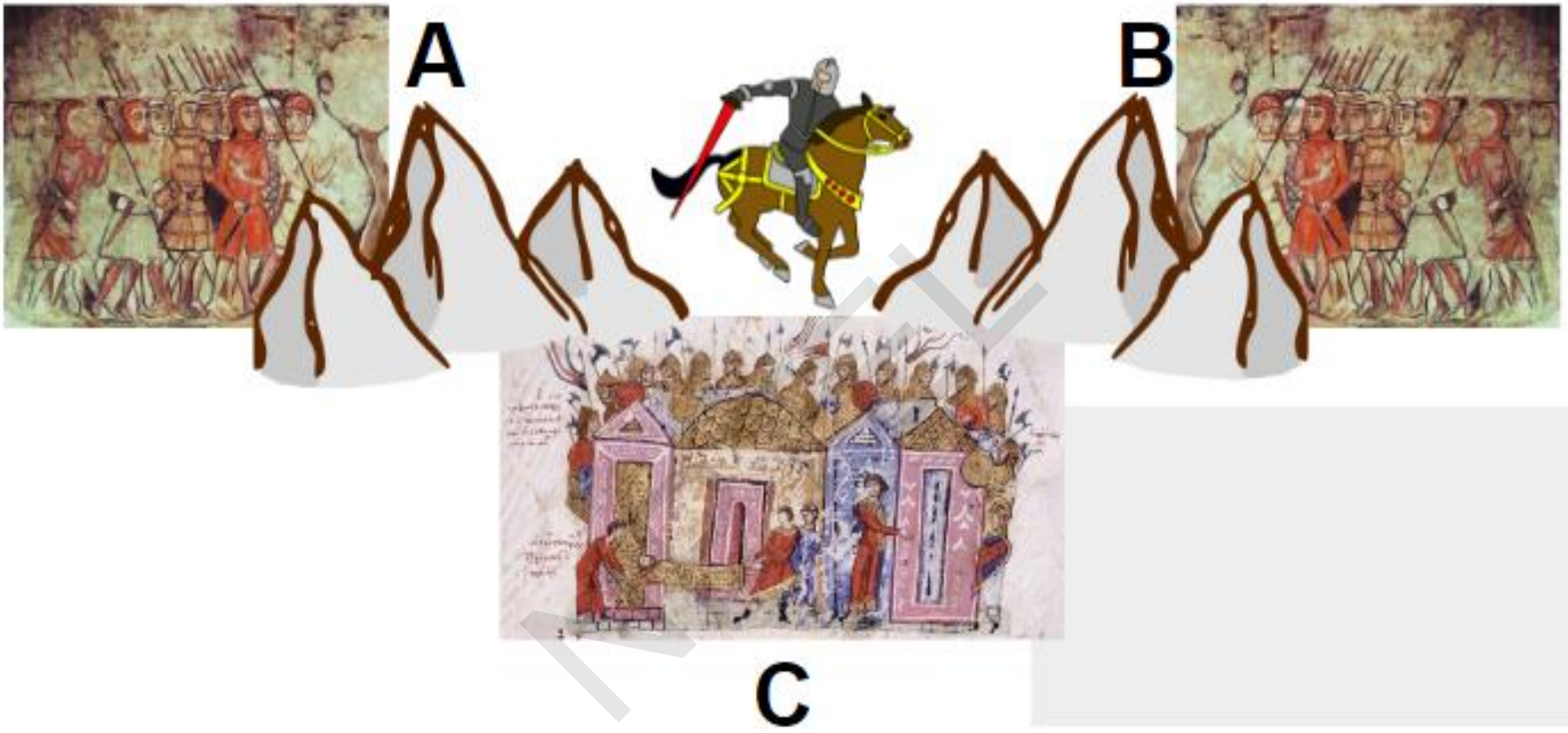
- Can all nodes see all message? Some? None?
- Do nodes fail? How about the network?
- Finite computation?
- Static or dynamic adversary?
- Bounded communication time?
- Fully connected network?
- Randomized algorithms?
- Quantum or binary computers?

The Two Generals Problem

Consensus: The Two Generals Problem

Two armies, A and B in separate valleys.
Want to attack third army, C, in valley between them.
Must decide: attack tomorrow or not?
If they both attack: victory!
If neither attack: survival!
If just one attacks: defeat!
All messages sent by horse -- *through enemy territory*.
Each messenger may or may not make it through.

Consensus: The Two Generals Problem



See if you can figure out a series of messages to solve this problem.

Two Generals Problem: solved ?

A

B

A wants to attack

If you respond, I'll attack!

A will attack;
B wants to attack

If you respond, I'll attack!

A will attack;
B will attack

We'll attack!

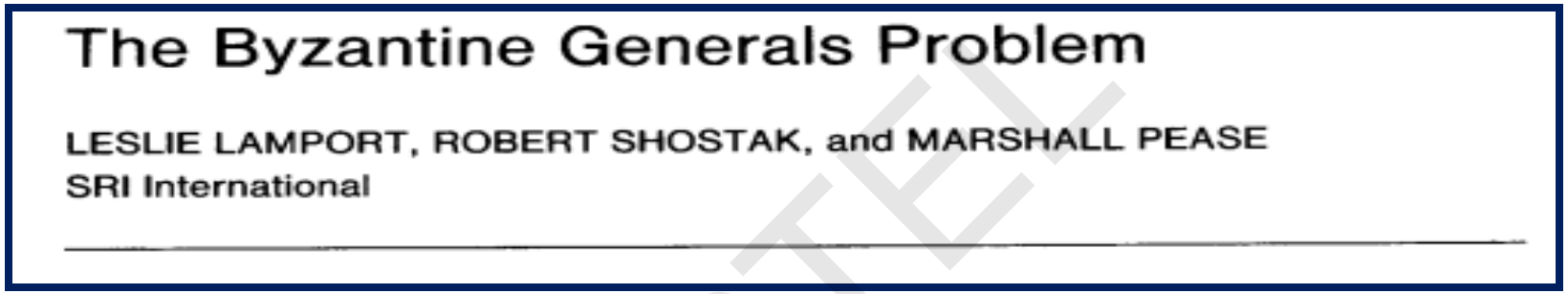
A will attack;
B will attack

There is no perfect solution!

The Byzantine Generals Problem

Byzantine Generals

- **The Byzantine Generals Problem**, Leslie Lamport, Robert Shostack and Mashall Peace. ACM TOPLAS 4.3, 1982.



The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

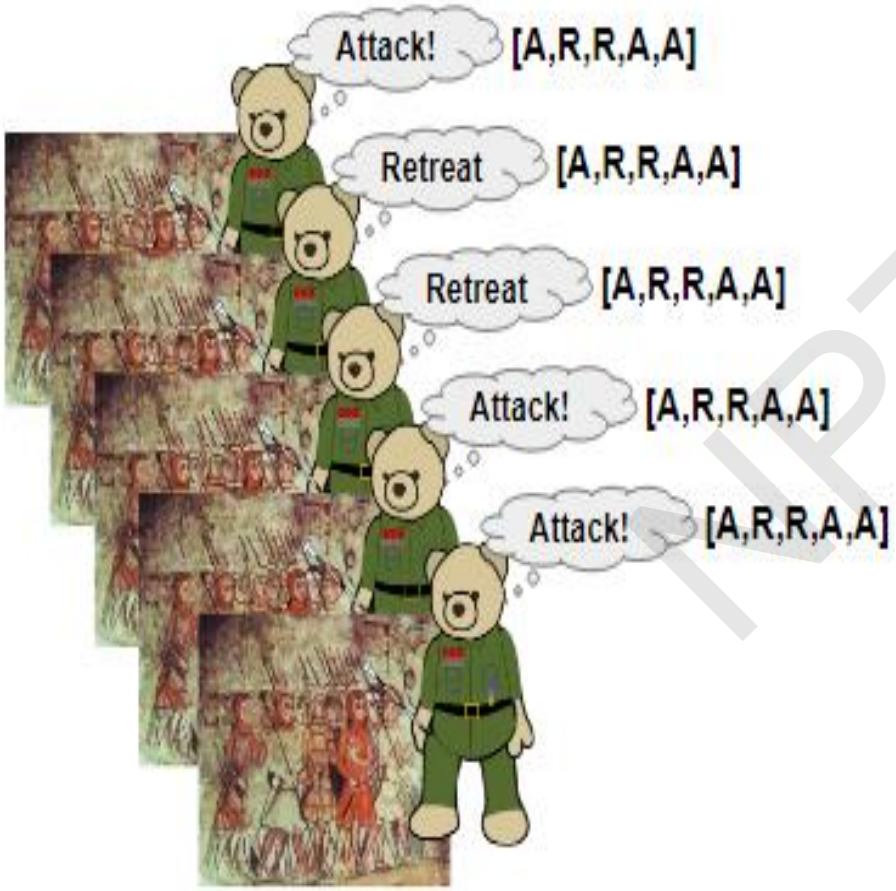
Answers:

- How many byzantine node failures can a system survive?
- How might you build such a system?

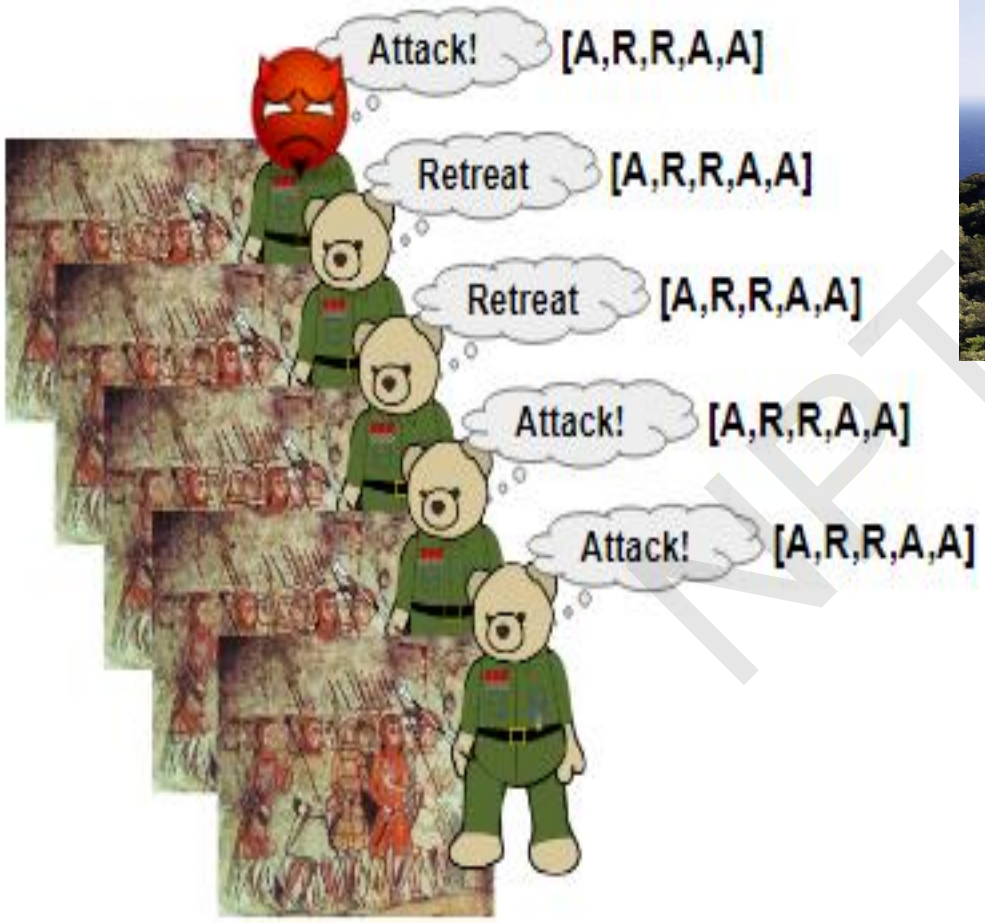
Doesn't answer:

- Is it worth doing at all?

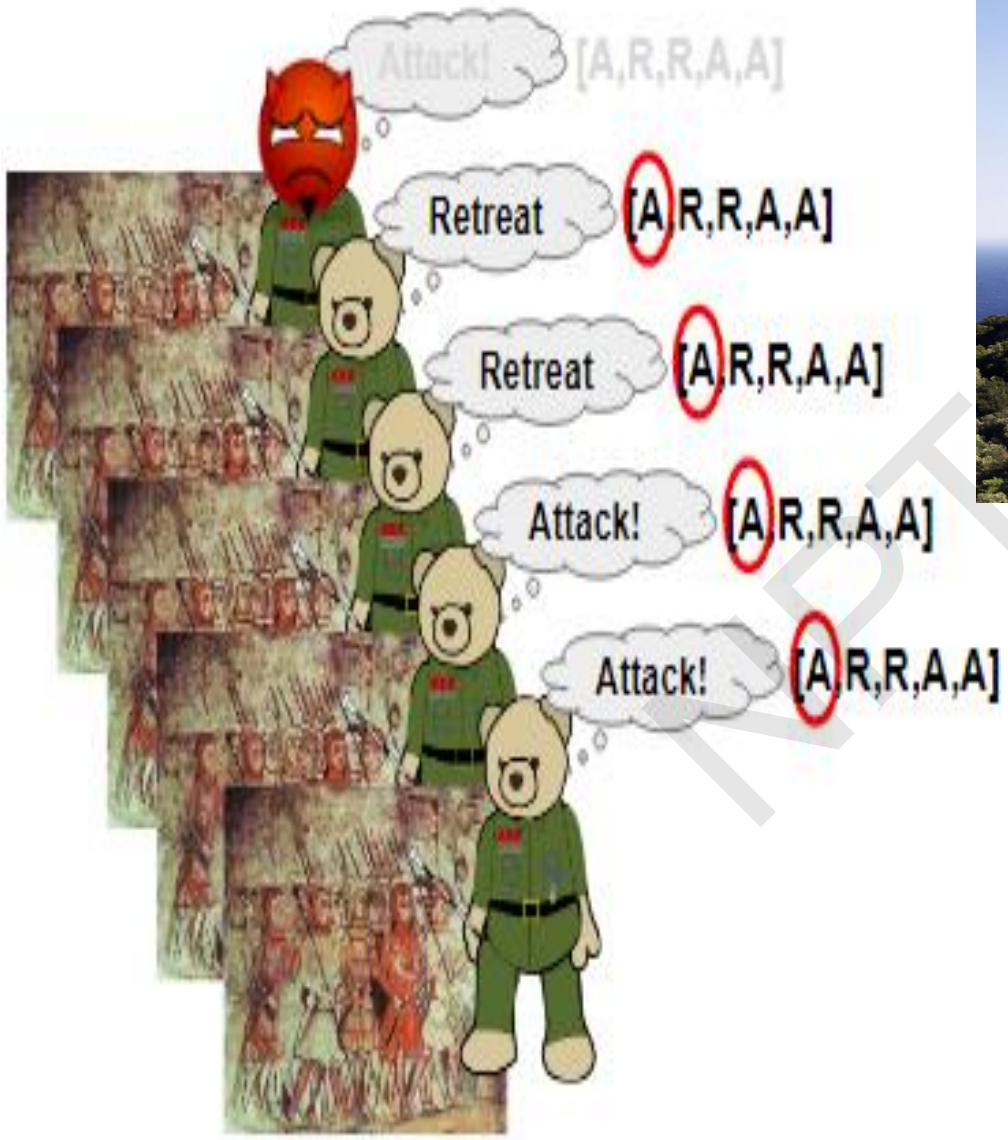
The Problem


Attack!
[A,R,R,A,A]
Retreat
[A,R,R,A,A]
Retreat
[A,R,R,A,A]
Attack!
[A,R,R,A,A]
Attack!
[A,R,R,A,A]

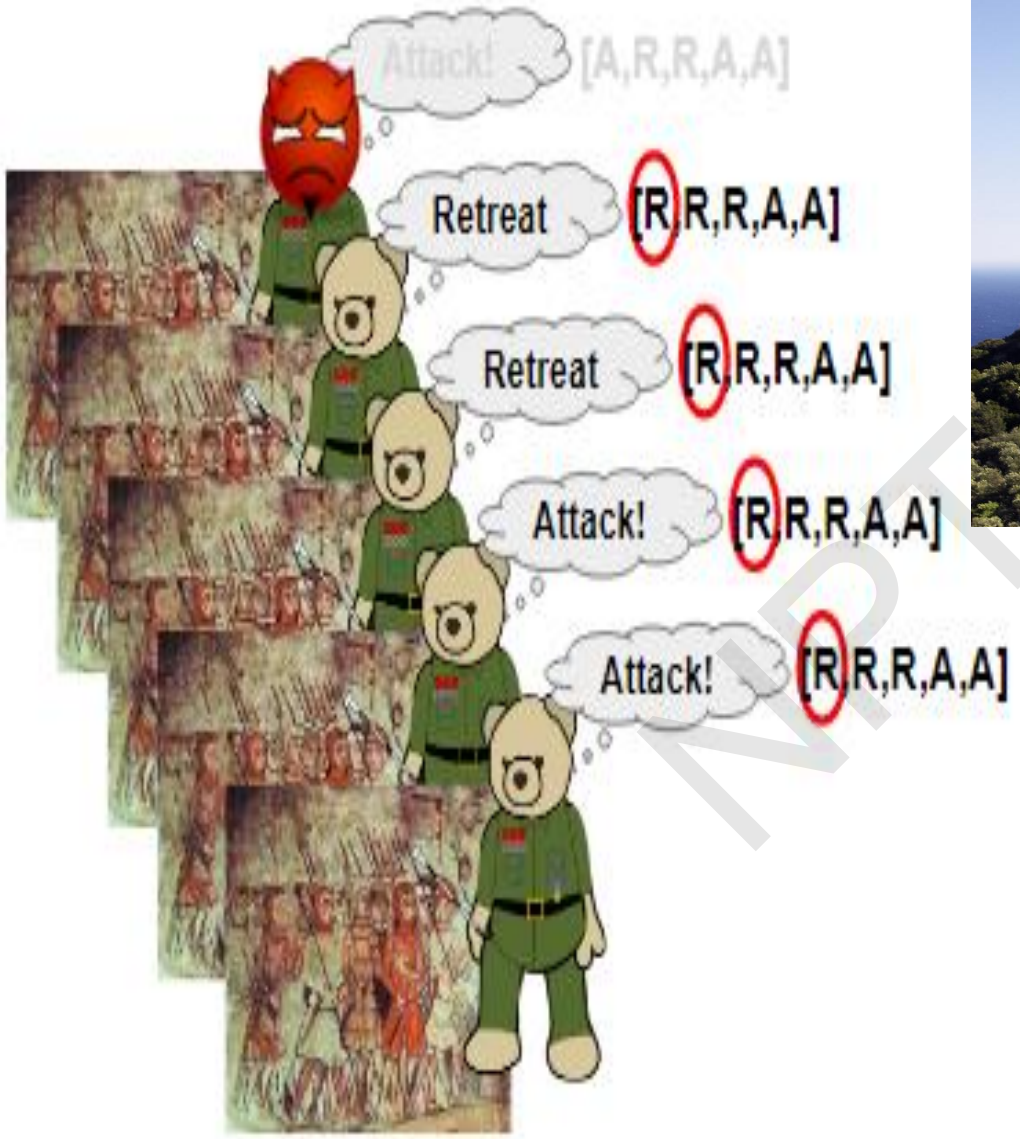
The Problem



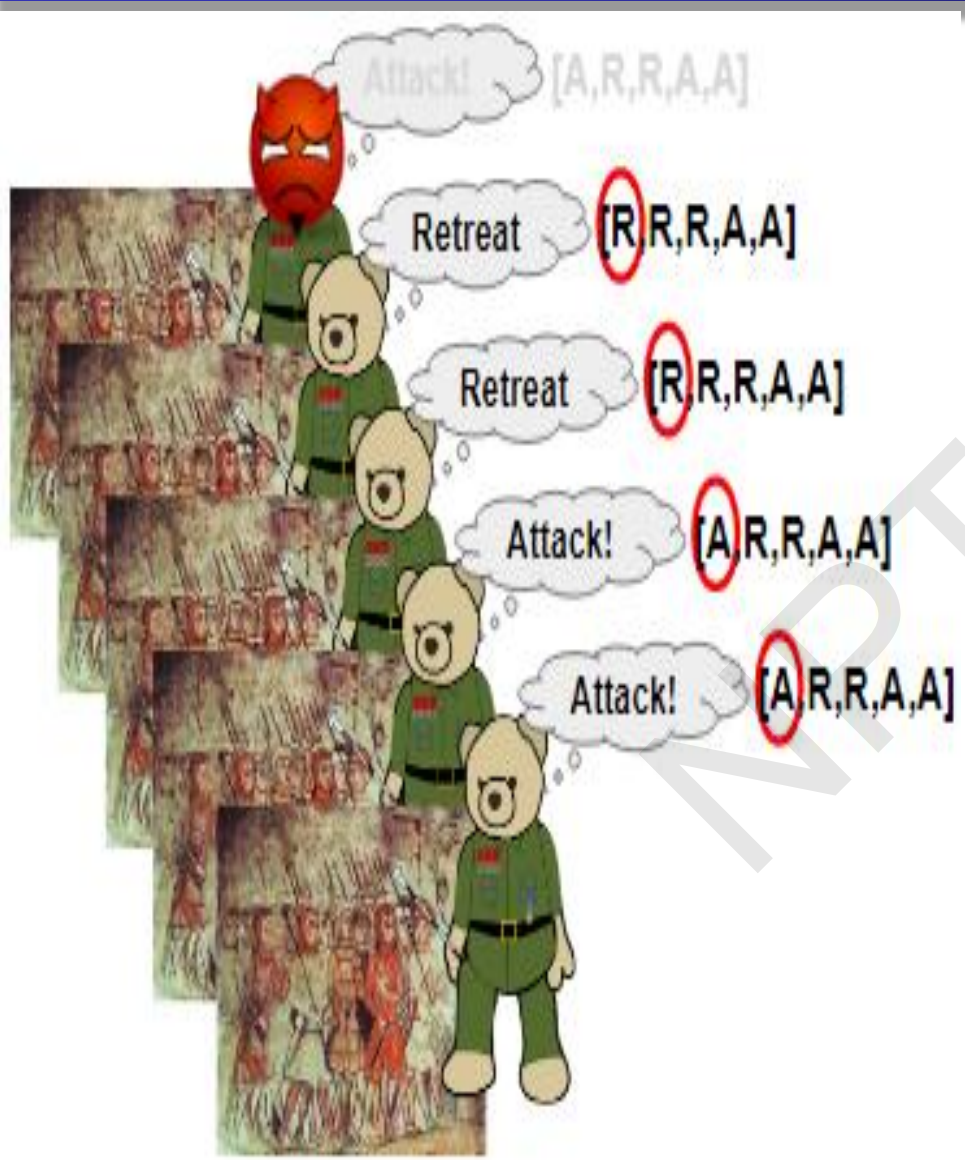
The Problem


Attack! [A,R,R,A,A]
Retreat [A,R,R,A,A]
Retreat [A,R,R,A,A]
Attack! [A,R,R,A,A]
Attack! [A,R,R,A,A]

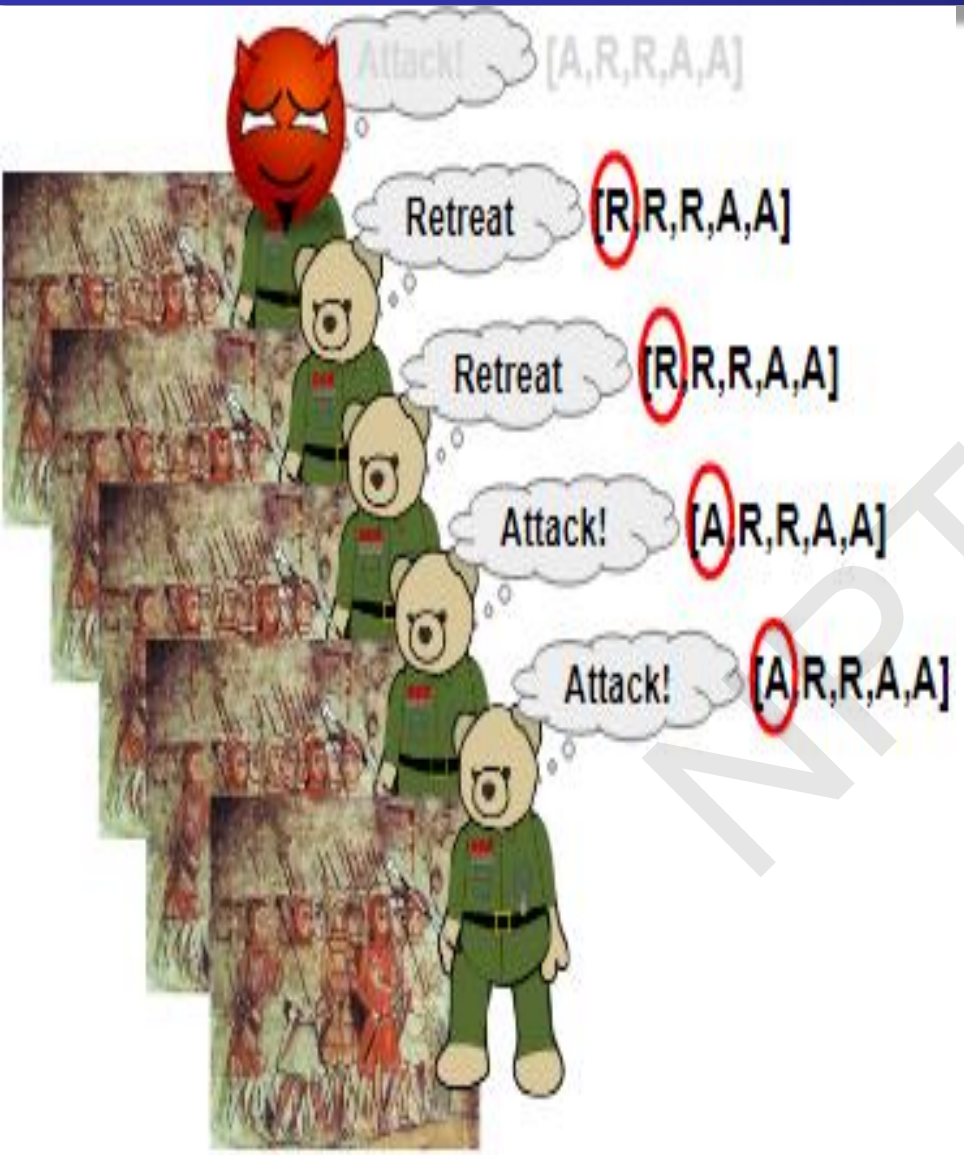
The Problem



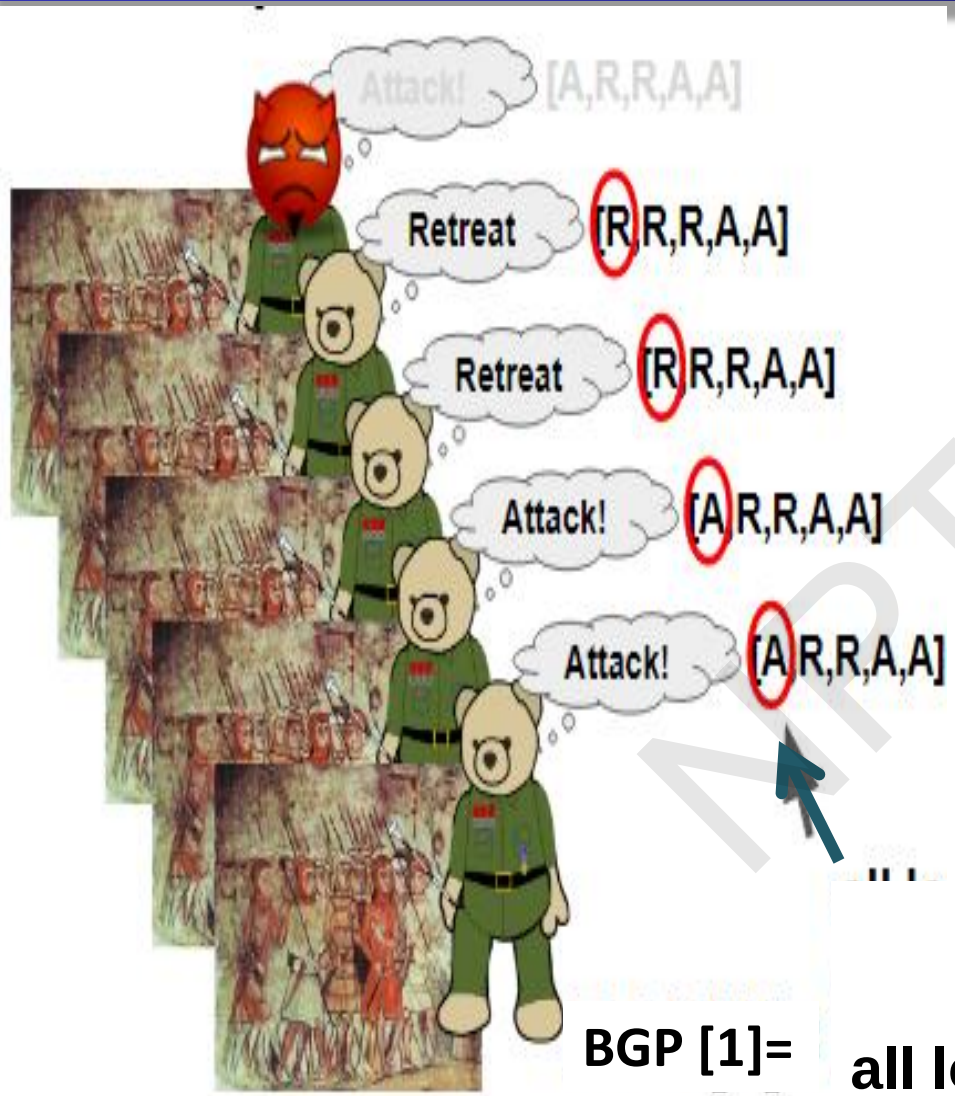
The Problem



The Problem



The Problem



BGP [1]= all loyal generals agree on what 1st general wants

How many traitors can there be?

The question

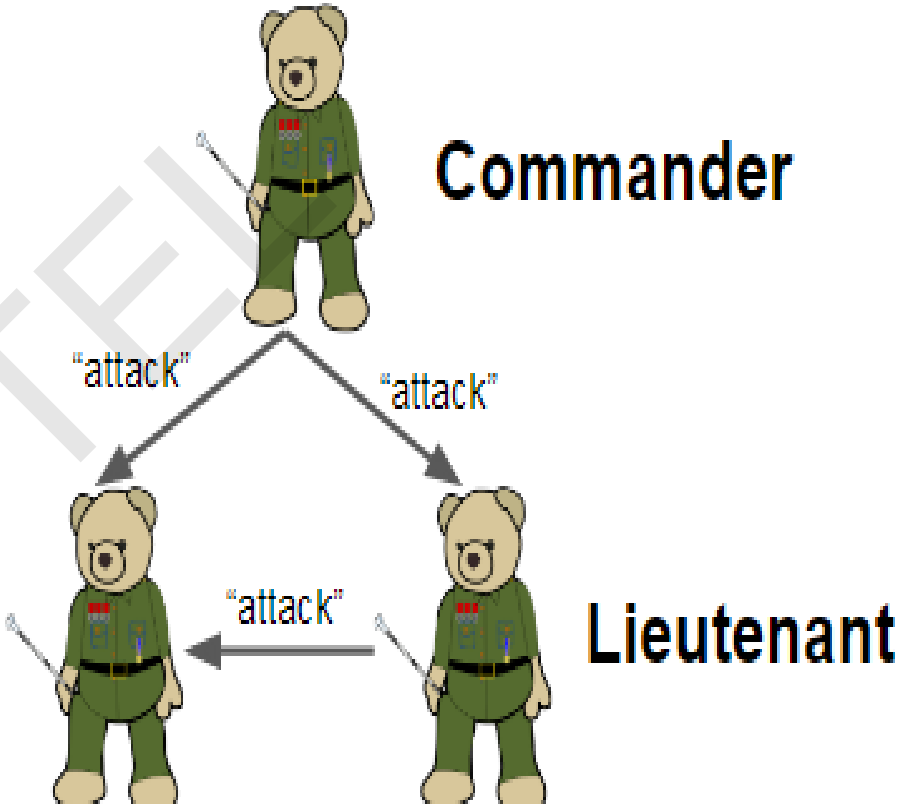
How many traitors can you have and still solve BGP ?

Assuming:
point-to-point (“oral”) messages
No crypto

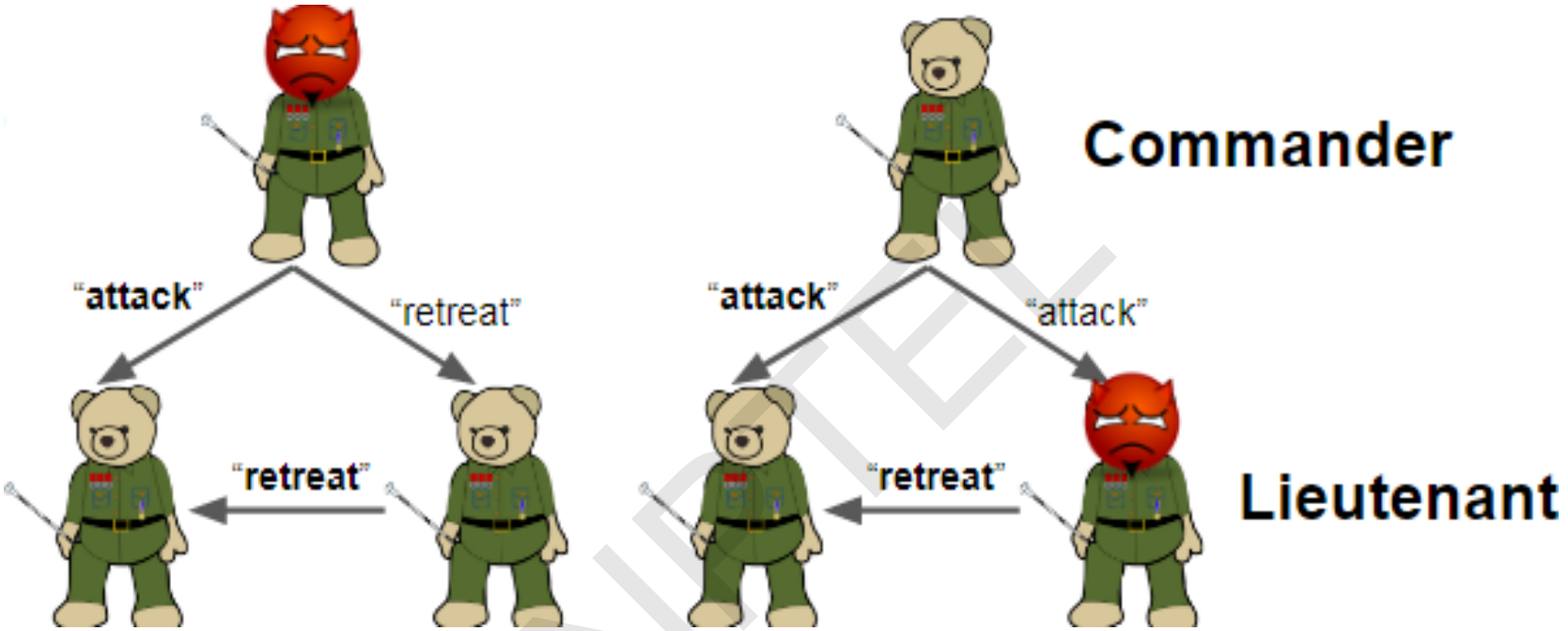
n = 1, or n = 2?

Trivial, no consensus required.

n = 3



n = 3



There is no solution for 3 Generals, 1 Traitor.

How many traitors can be tolerated?

- Lemma: **No** solution for **3m+1** generals with **>m** traitors.

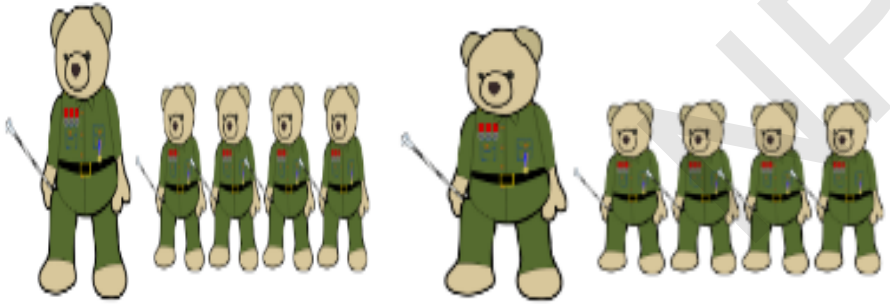
Proof:

1. Assume solution exists.
2. Use solution to solve **1** traitor **3** generals case.

We know 2 is impossible!

- ⇒ Hence solution must not exist. ⇐

Simulation proof



- **Assume solved:** 4 traitors; 12 generals
- Each general simulates (pretends to be) 4 simulated generals
- Run solution on simulated generals
- Each general chooses value chosen by its simulated generals

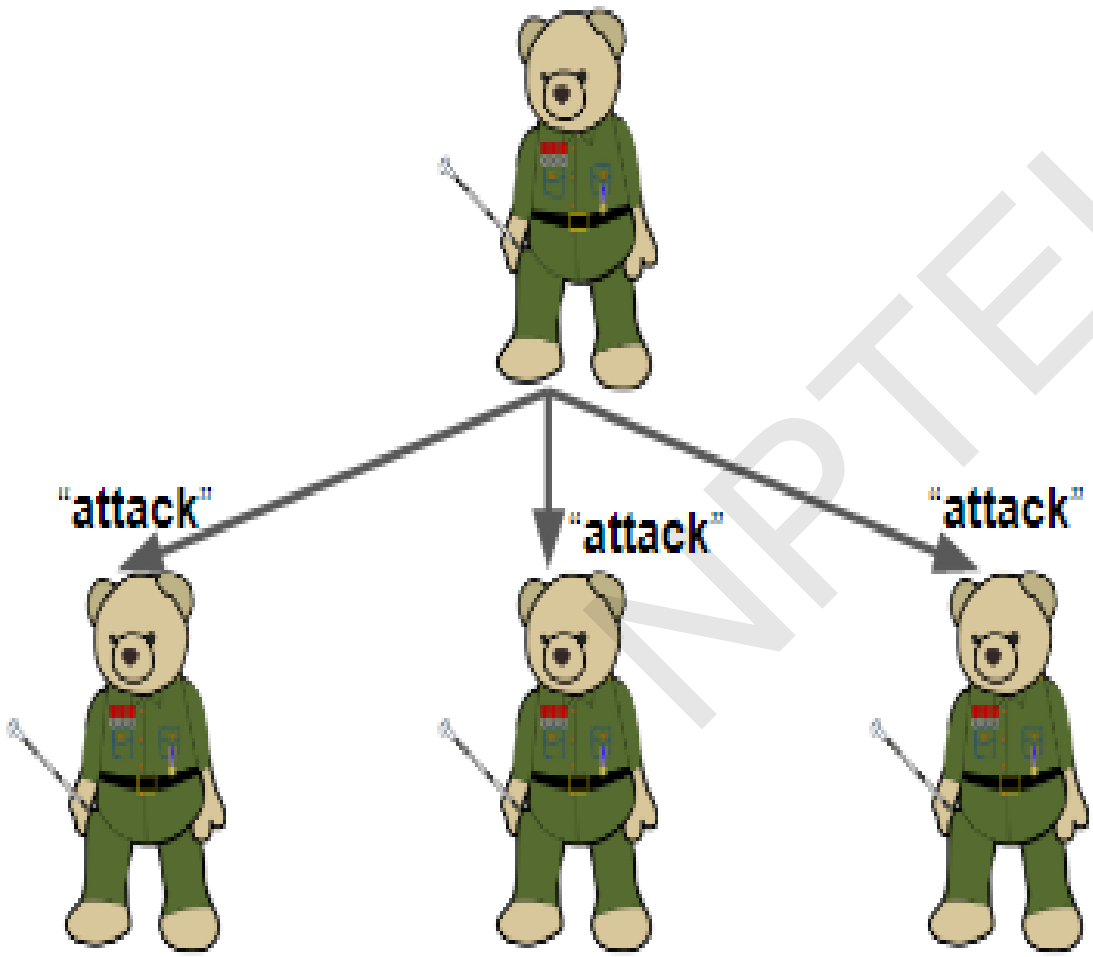
There is no solution for 3m+1 Generals, >m Traitors.

Solving Byzantine Generals

What can you do?

- **Assuming:**
  - Less than ⅓ of generals are traitors
  - Oral messages
  - No crypto
- OM(m): solution to BGP for ≤m traitors

Inductive Solution for Oral Messages

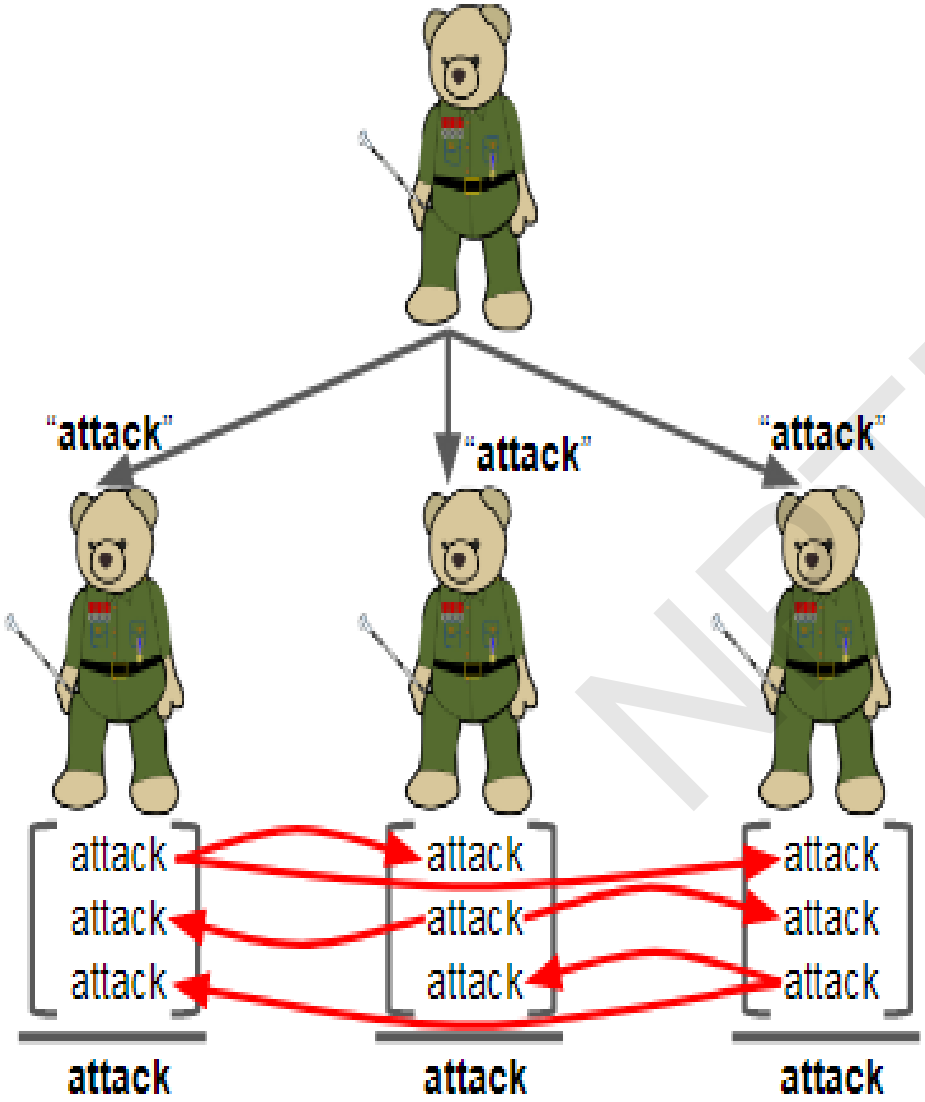


OM(0):

C: Sends order.

L: Follows if received.

Inductive Solution for Oral Messages



OM(m), m>0:

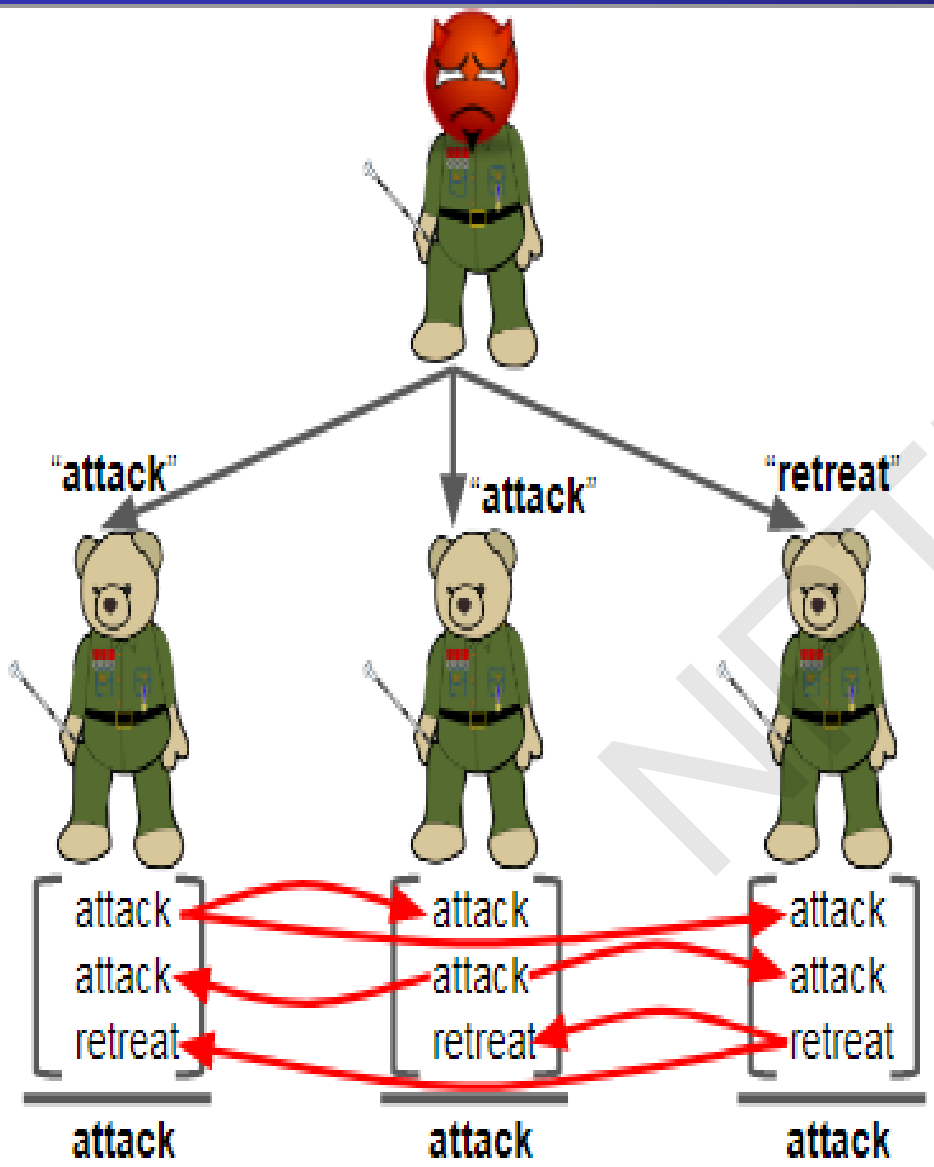
C: Sends order.

L: 1. Records if received.

2. Use OM(m-1) to tell others.

3. Follows majority() order.

How this works with m=1



OM(m), m>0:

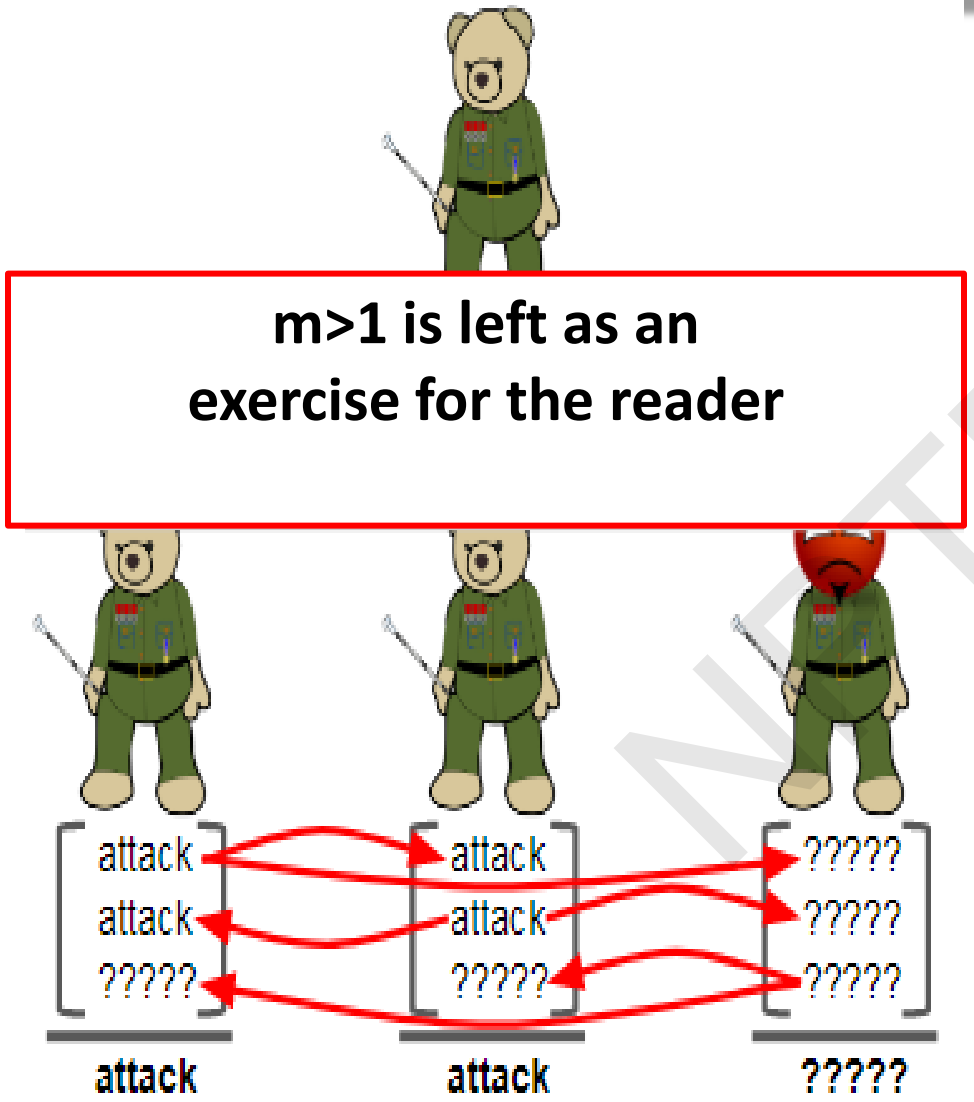
C: Sends order.

L: 1. Records if received.

2. Use OM(m-1) to tell others.

3. Follows majority() order.

How this works with m=1



OM(m), m>0:

C: Sends order.

L: 1. Records if received.

2. Use OM(m-1) to tell others.

3. Follows majority() order.

Running Time

Expensive

m	Message Sent
0	O(n)
1	O(n^2)
2	O(n^3)
3	O(n^4)

So:

- Don't solve BGP;
- Use someone else's solution; or
- Keep n & m small

2. Consensus Algorithm for Byzantine Failure (Message Passing, Synchronous Systems)

Model :

– Total of *n* processes, at most *f* of which can be faulty
– Reliable communication medium
– Fully connected
– Receiver always knows the identity of the sender of a message
– Byzantine faults
– **Synchronous system:** In each round, a process receives messages, performs computation, and sends messages.

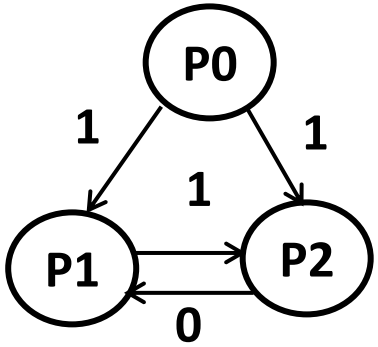
Solution for Byzantine Agreement Problem

- The solution of Byzantine Agreement Problem is first defined and solved by ***lamport***.

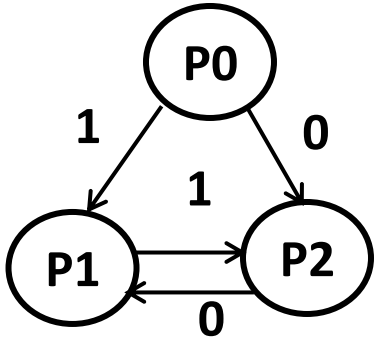
- ***Pease*** **showed** that in a fully connected network, **it is impossible to reach an agreement if number of faulty processes *'f'* exceeds *(n-1)/3* where *n* is number of processes**

Byzantine agreement can not be reached among three processes if one process is faulty

P_0 is Non-Faulty



P_0 is Faulty



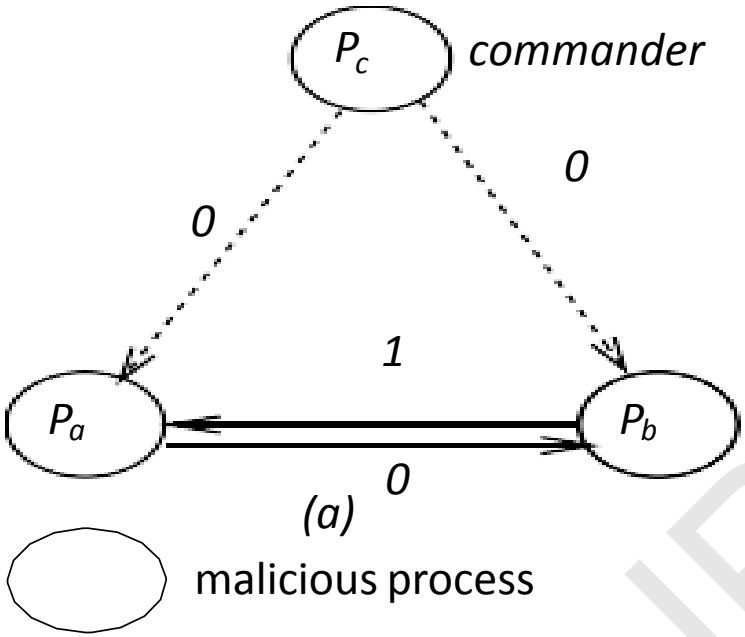
Note: Here P_0 is a source process

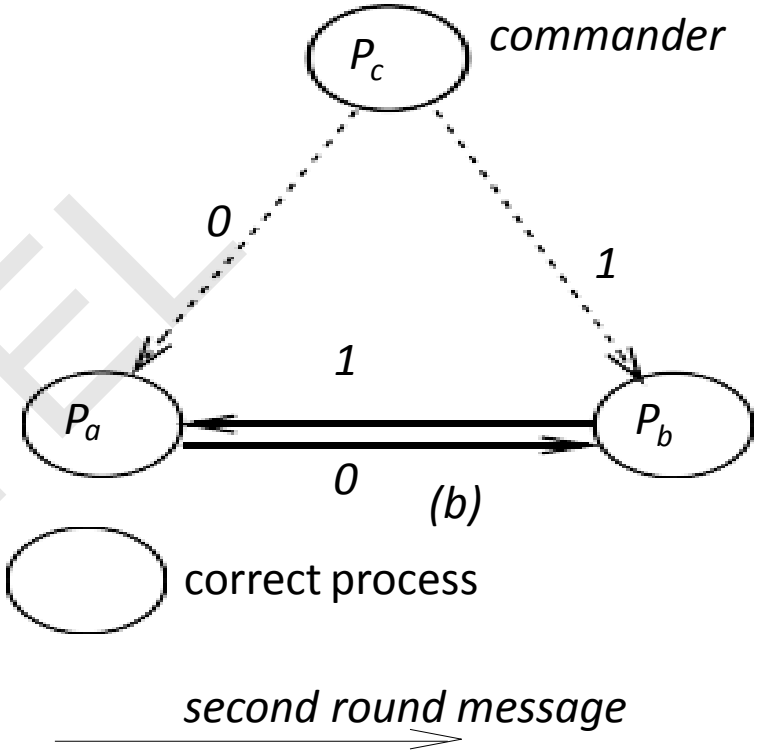
Upper bound on Byzantine processes

In a system of *n* processes, the **Byzantine agreement problem** (as also the other variants of the agreement problem) can be solved in a synchronous only if the number of Byzantine processes *f* is such that $f \leq \lfloor((n-1)/3\rfloor$

Upper Bound on Byzantine Processes

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

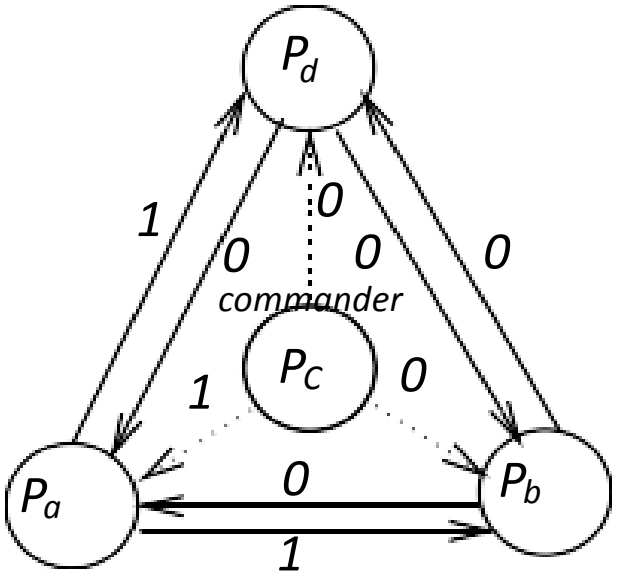


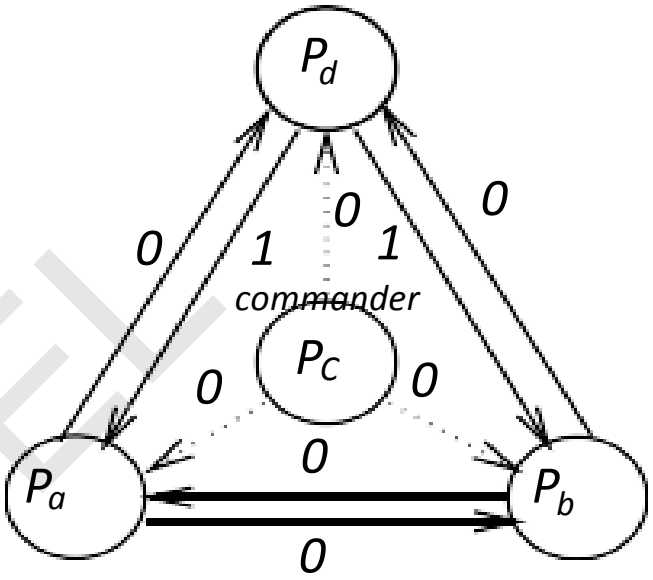


Taking simple majority decision does not help because loyal commander P_a cannot distinguish between the possible scenarios (a) and (b); hence does not know which action to take.

Proof using induction that problem solvable if $f \leq \lfloor((n-1)/3\rfloor$

Consensus Solvable when $f = 1, n = 4$





first round exchange

second round exchange

malicious process

correct process

- 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.

Lamport-Shostak-Pease Algorithm

This algorithm also known as ***Oral Message Algorithm OM(f)***
where *f* is the number of faulty processes
'*n*' = Number of processes and ***n >= 3f + 1***

Algorithm is Recursively defined as follows:

Algorithm OM(0)

1. Source process sends its values to each other process
2. Each process uses the value it receives from the source.
   [If no value is received default value 0 is used]

Contd...

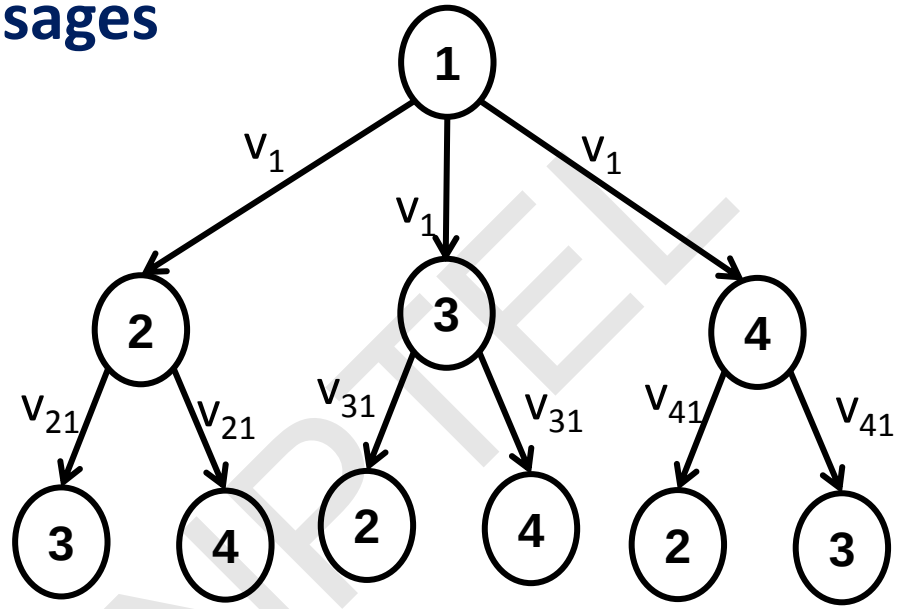
Algorithm OM(*f*), *f* > 0

1. The source process sends its value to each other process.
2. For each *i*, let v_i be the value process *i* receives from source. [Default value 0 if no value received]
3. Process *i* acts as the new source and initiates **Algorithm *OM(f - 1)*** where it sends the value v_i to each of the *n-2* other processes.
4. For each *i* and *j* (not *i*), let v_j be the value process *i* received from process *j* in STEP 3. Process *i* uses the value **majority (v_1, v_2 …. v_{n-1}).**

"The function majority(v_1, v_2…. v_{n-1}) computes the majority value if exists otherwise it uses default value 0."

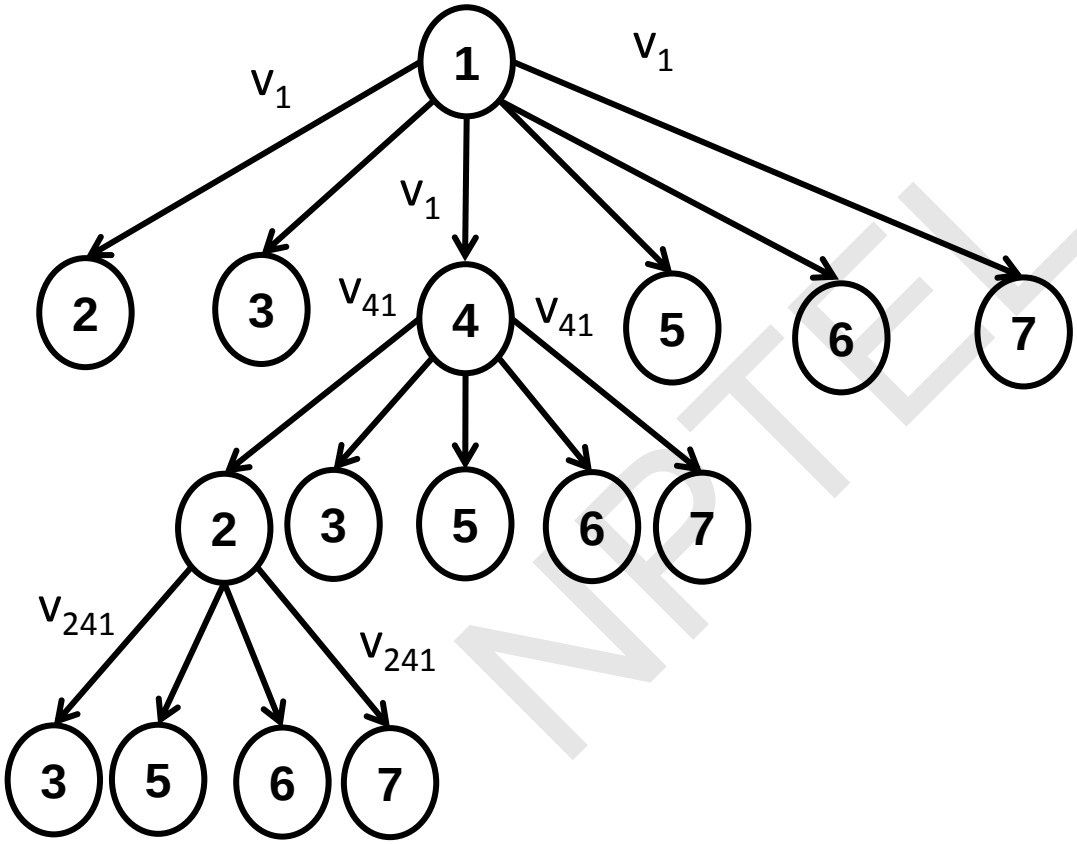
(i) Solving the Byzantine agreement, for $f = 1$ and $n = 4$

- **Number of messages for agreement on one value is: 3+2.3 = 9 messages**



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*	*1*	*4 – 1= 3*	*4 – 1 = 3*
2	*2*	*1– 1 = 0*	*4 – 2= 2*	*(4 – 1) · (4 – 2) = 3.2*

(ii) Solving the Byzantine agreement, for $f = 2$ and $n = 7$



First Round

Second Round

Third Round

Contd...

- **Number of messages for agreement on one value is: 6+6.5+6.5.4 = 156 messages**

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*	*2*	*7 – 1= 6*	*7 – 1 = 6*
2	*2*	*2– 1 = 1*	*7 – 2= 5*	*(7 – 1) · (7 – 2) = 6.5*
3	*3*	*2 - 2 = 0*	*7 – 3 = 4*	*(7 – 1) · (7 – 2) · (7 – 3) = 6.5.4*

(iii) Solving the Byzantine agreement, for $f = 3$ and $n = 10$

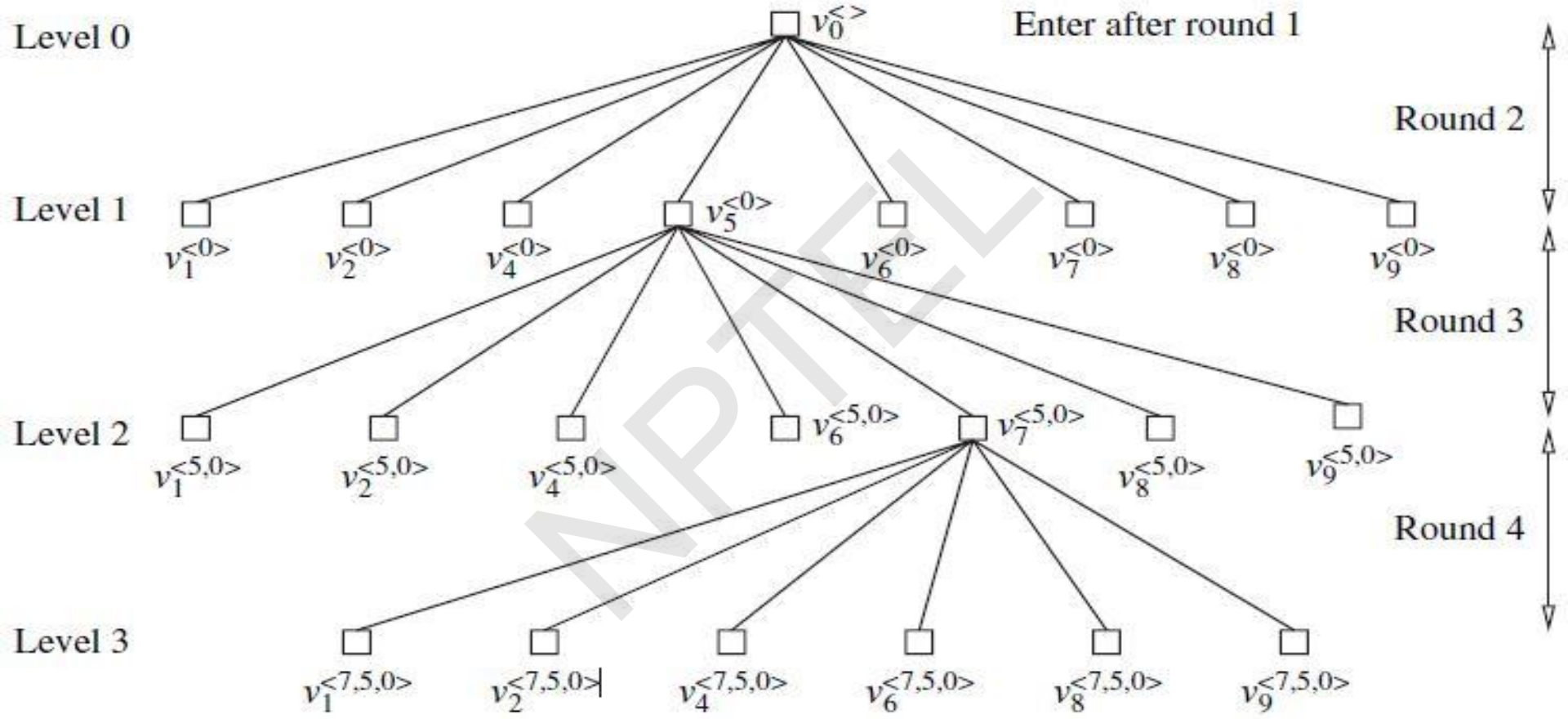


Fig. Local tree at P_3 for solving the Byzantine agreement, for $n = 10$ and $f = 3$, commander is P_0. Only one branch of the tree is shown for simplicity.

Contd...

- **Number of messages for agreement on one value is: 9+9.8+9.8.7+9.8.7.6 = 3609 messages**

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*	*3*	*10 – 1= 9*	*10 – 1 = 9*
2	*2*	*3– 1 = 2*	*10 – 2= 8*	*(10 – 1) · (10 – 2) = 9.8*
3	*3*	*3 - 2 = 1*	*10 – 3 = 7*	*(10 – 1) · (10 – 2) · (10 – 3) = 9.8.7*
4	*4*	*3 – 3 = 0*	*10 – 4 = 6*	*(10 – 1) · (10 – 2) · (10 – 3) · (10 – 4) = 9.8.7.6*

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	$n-f-1$	$(n-1)(n-2)\ldots(n-f-1)$

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

Complexity: $f+1$ rounds, exponential amount of space, and $(n-1)+(n-1)(n-2)+\ldots+(n-1)(n-2)..(n-f-1)$ messages

Agreement in Asynchronous Message-Passing Systems with Failures

Impossibility Result for the Consensus Problem

Fischer-Lynch-Paterson (FLP) Impossibility Result (By M. Fischer, N. Lynch, and M. Paterson, April 1985)

- **Fischer et al.** showed a fundamental result on the impossibility of reaching agreement in an asynchronous (message-passing) system.
- It states that it is "***Impossible to reach consensus in an asynchronous message passing system even if a single process has a crash failure***"
- This result, popularly known as the **FLP impossibility result**, has a significant impact on the field of designing distributed algorithms in a failure-susceptible system.

Weaker Versions of Consensus Problem

Consensus Problem	Description
Terminating reliable broadcast	It states that a correct process always gets a message even if the sender crashes while sending. If the sender crashes while sending the message, the message may be a null message but it must be delivered to each correct process.
k-set consensus	It is solvable as long as the number of crash failures f is less than the parameter k. The parameter k indicates that the non-faulty processes agree on different values, as long as the size of the set of values agreed upon is bounded by k.
Approximate agreement	Like k-set consensus, approximate agreement also assumes the consensus value is from a multi-valued domain. However, rather than restricting the set of consensus values to a set of size k, ϵ-approximate agreement requires that the agreed upon values by the non-faulty processes be within ϵ of each other.
Renaming problem	It requires the processes to agree on necessarily distinct values.
Reliable broadcast	A weaker version of reliable terminating broadcast(RTB), namely reliable broadcast, in which the termination condition is dropped, is solvable under crash failures.

(i) Terminating Reliable Broadcast (TRB) Problem

- A correct process always gets a message, even if sender crashes while sending (in which case the process gets a null message).

- **Validity:** If the sender of a broadcast message *m* is non-faulty, then all correct processes eventually deliver *m*.
- **Agreement:** If a correct process delivers a message *m*, then all correct processes deliver m.
- **Integrity:** Each correct process delivers at most one message. Further, if it delivers a message different from the null message, then the sender must have broadcast m.
- **Termination:** Every correct process eventually delivers some message.

Contd...

- The reduction from consensus to terminating reliable broadcast is as follows:
- A commander process broadcasts its input value using the terminating reliable broadcast. A process decides on a “0” or “1” depending on whether it receives “0” or “1” in the message from this process.
- However, if it receives the null message, it decides on a default value. As the broadcast is done using the terminating reliable broadcast, it can be seen that the conditions of the consensus problem are satisfied.
- **But as consensus is not solvable, an algorithm to implement terminating reliable broadcast cannot exist.**

(ii) Reliable Broadcast Problem

- **Reliable Broadcast (RB)** is RTB without terminating condition.
- RTB requires eventual delivery of messages, even if sender fails before sending. In this case, a null message needs to get sent. In RB, this condition is not there.
- RTB requires recognition of a failure, even if no msg is sent
- **Crux:** RTB is required to distinguish between a failed process and a slow process.
- **RB is solvable under crash failures; $O(n^2)$ messages**

Algorithm: Protocol for reliable broadcast

(1)Process P_0 initiates Reliable Broadcast:

(1a) **broadcast** message M to all processes.

(2) A process P_i, $1 \leq i \leq n$, receives message M:

(2a) **if** M was not received earlier **then**

(2b) **broadcast** M to all processes;

(2c) deliver M to the application.

Applications of Agreement Algorithms

1) Fault-Tolerant Clock Synchronization

- Distributed Systems require physical clocks to synchronized
- Physical clocks have drift problem
- Agreement Protocols may help to reach a common clock value.

2) Atomic Commit in Distributed Database System (DDBS)

- DDBS sites must agree whether to commit or abort the transaction
- Agreement protocols may help to reach a consensus.

Conclusion

- **Consensus problems are fundamental aspects** of distributed computing because they **require inherently distributed processes** to reach agreement.

- This lecture first covers:

  -Different forms of the **consensus problem**,

  -Then gives an overview of what forms of consensus are solvable under different failure models and different assumptions on the synchrony/asynchrony.

Contd...

- Then we have covered agreement in the following categories:

(i) Synchronous message-passing systems with failures:

-Used Fault Models: fail-stop model and the Byzantine model.

(ii) Asynchronous message-passing systems with failures:

-Impossible to reach consensus in this model.

-Hence, several weaker versions of the consensus problem, **i.e. terminating reliable broadcast, reliable broadcast** are considered.

Failures & Recovery Approaches in Distributed Systems

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

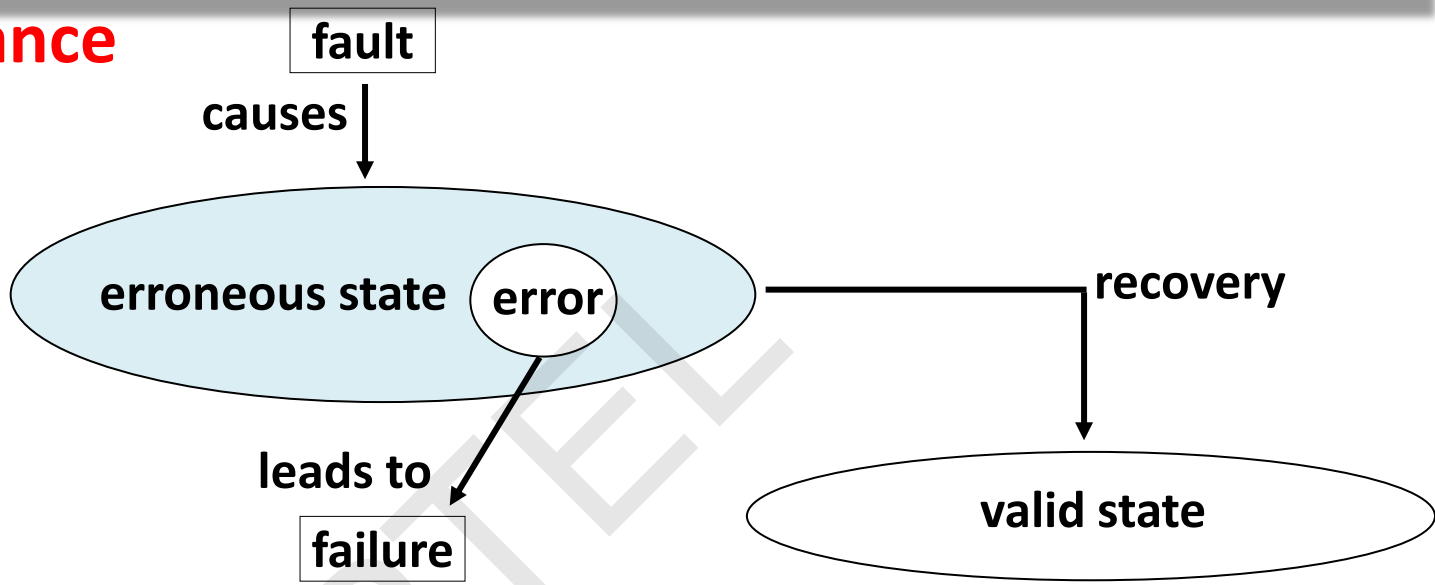
- In this lecture, we will discuss about basic fundamentals and underlying concepts of '**Failure and Rollback Recovery**' and

- Also discuss **Checkpointing & Rollback Recovery Schemes** in distributed systems i.e.

  (i) Checkpoint based and
  (ii) Log based

Introduction

- **Fault Tolerance**



- An error is a manifestation of a fault that can lead to a failure.
- **Failure Recovery:**
  - Backward recovery- Restore system state to a previous error-free state
    - **operation-based** (do-undo-redo logs)
    - **state-based** (checkpointing/logging)
  - Forward recovery- Repair the erroneous part of the system state

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***"

Contd…

- 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

Preliminaries

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

Assumptions:

-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 kth local checkpoint at process P_i

- $C_{i,0}$

  -A process P_i takes a checkpoint $C_{i,0}$ before it starts execution

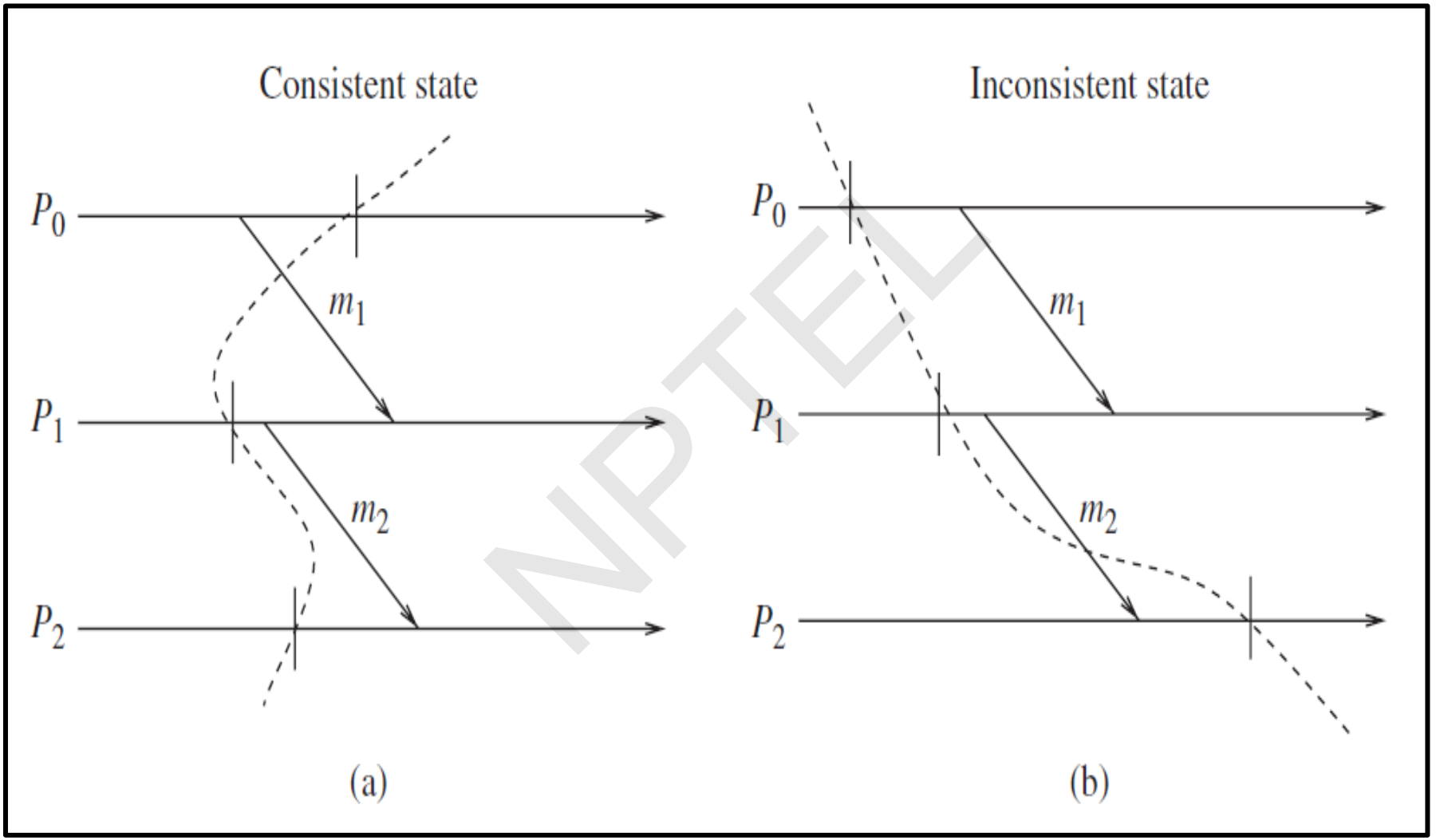
Consistent states

- **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 distribution of distributed computation
  - if a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message

- **A global checkpoint**
  - 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

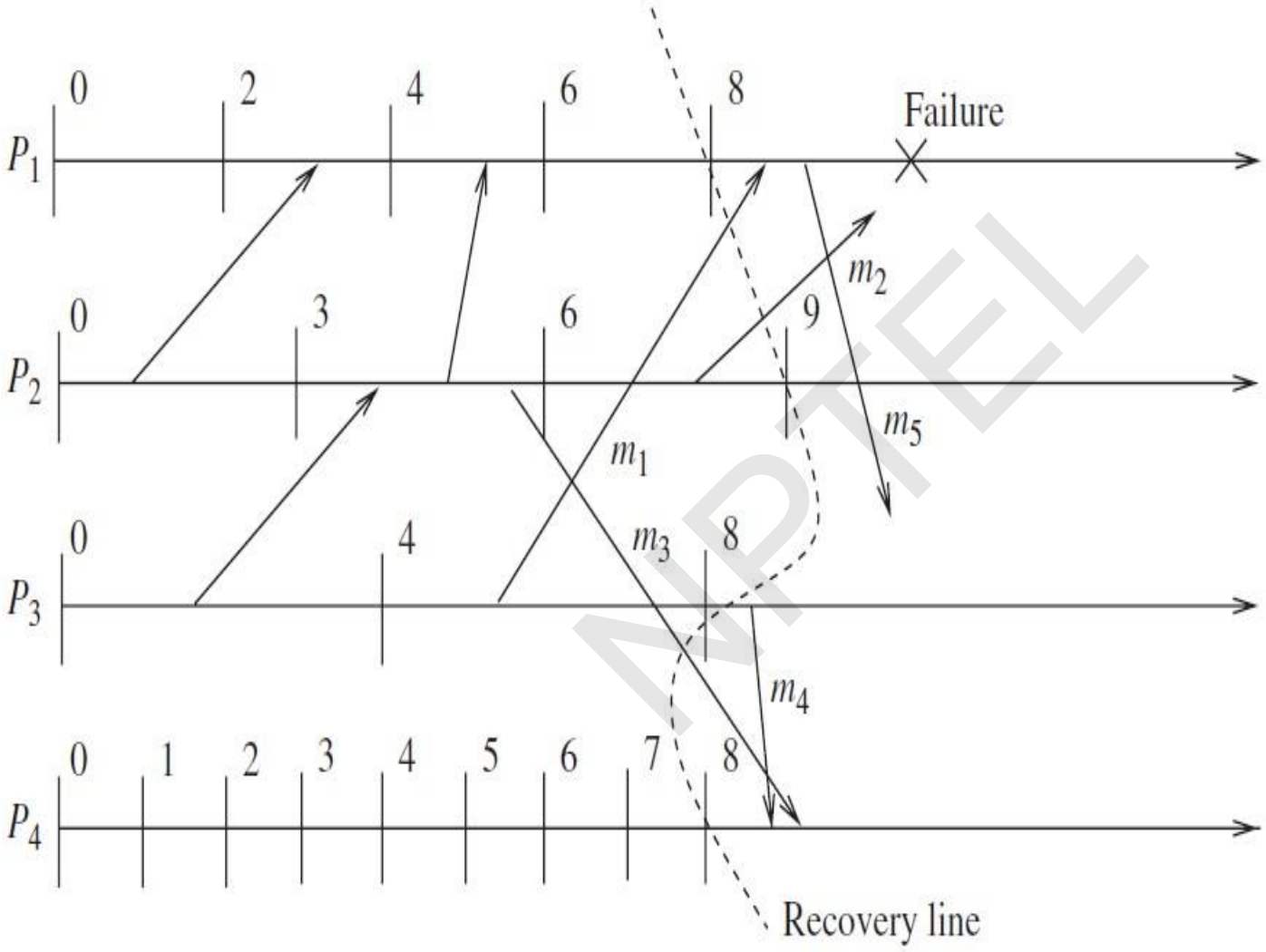
Orphan messages

- messages with 'receive' recorded but message 'send' not recorded
- do not arise if processes roll back to a consistent global state

Duplicate messages

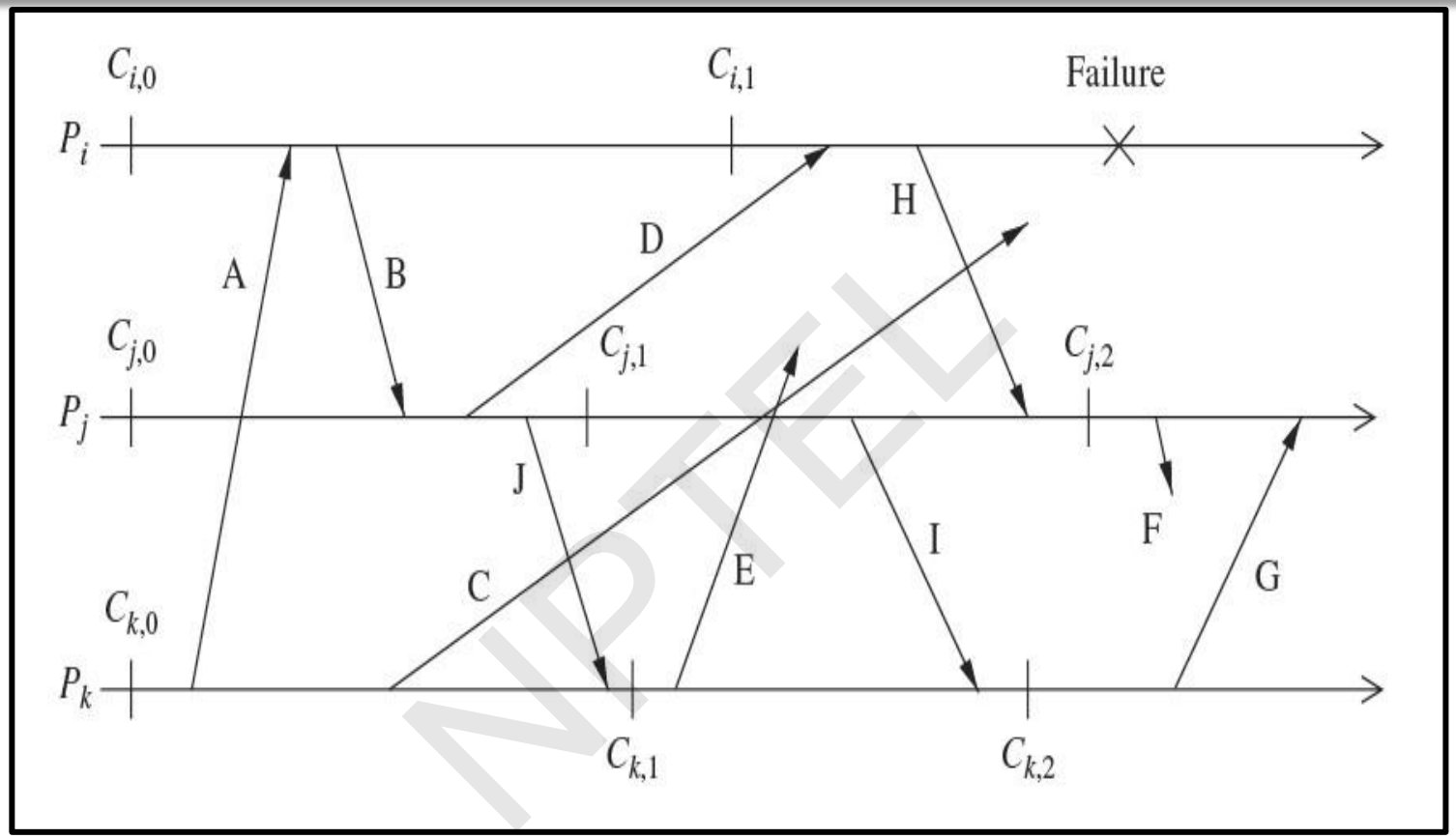
- arise due to message logging and replaying during process recovery

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



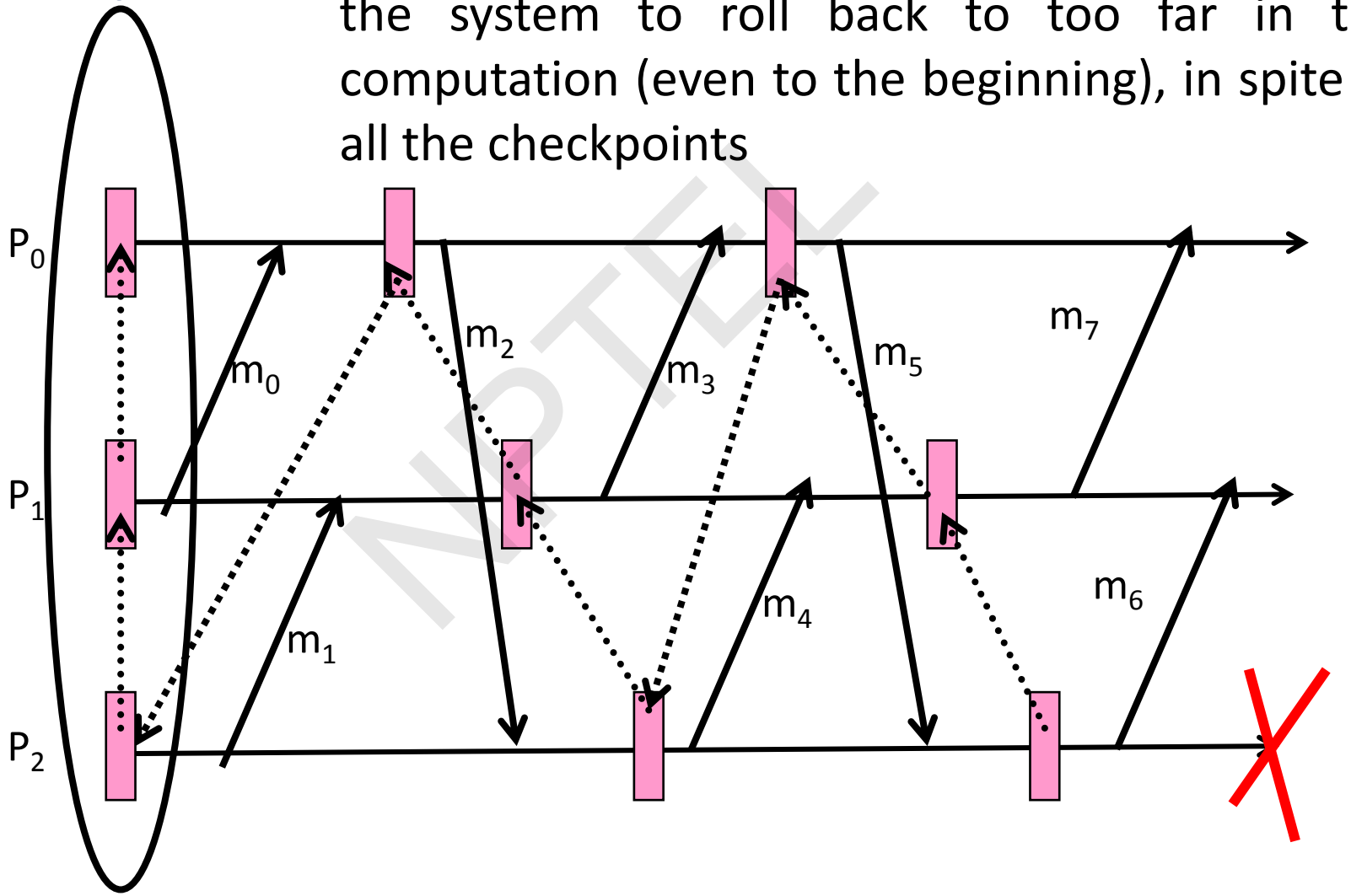
- **Checkpoints :** $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}, C_{k,2}\}$
- **Messages :** A -J
- **The restored global consistent state :** $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

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

Domino effect: example

Domino Effect: Cascaded rollback which causes the system to roll back to too far in the computation (even to the beginning), in spite of all the checkpoints



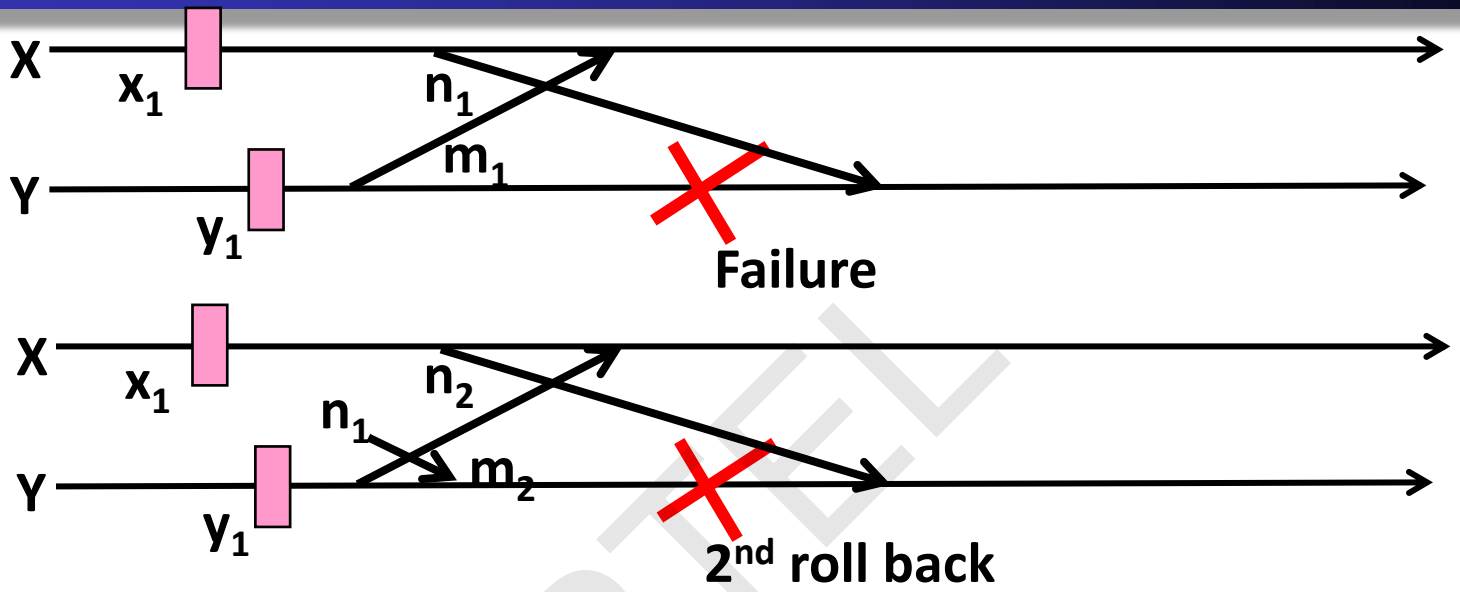
Problem of Livelock

- **Livelock:** case where a single failure can cause an infinite number of rollbacks.

- **The Livelock problem** may arise when a process rolls back to its checkpoint after a failure and requests all the other affected processes also to roll back.

- In such a situation if the roll back mechanism has no synchronization, it may lead to the livelock problem.

Livelock: Example



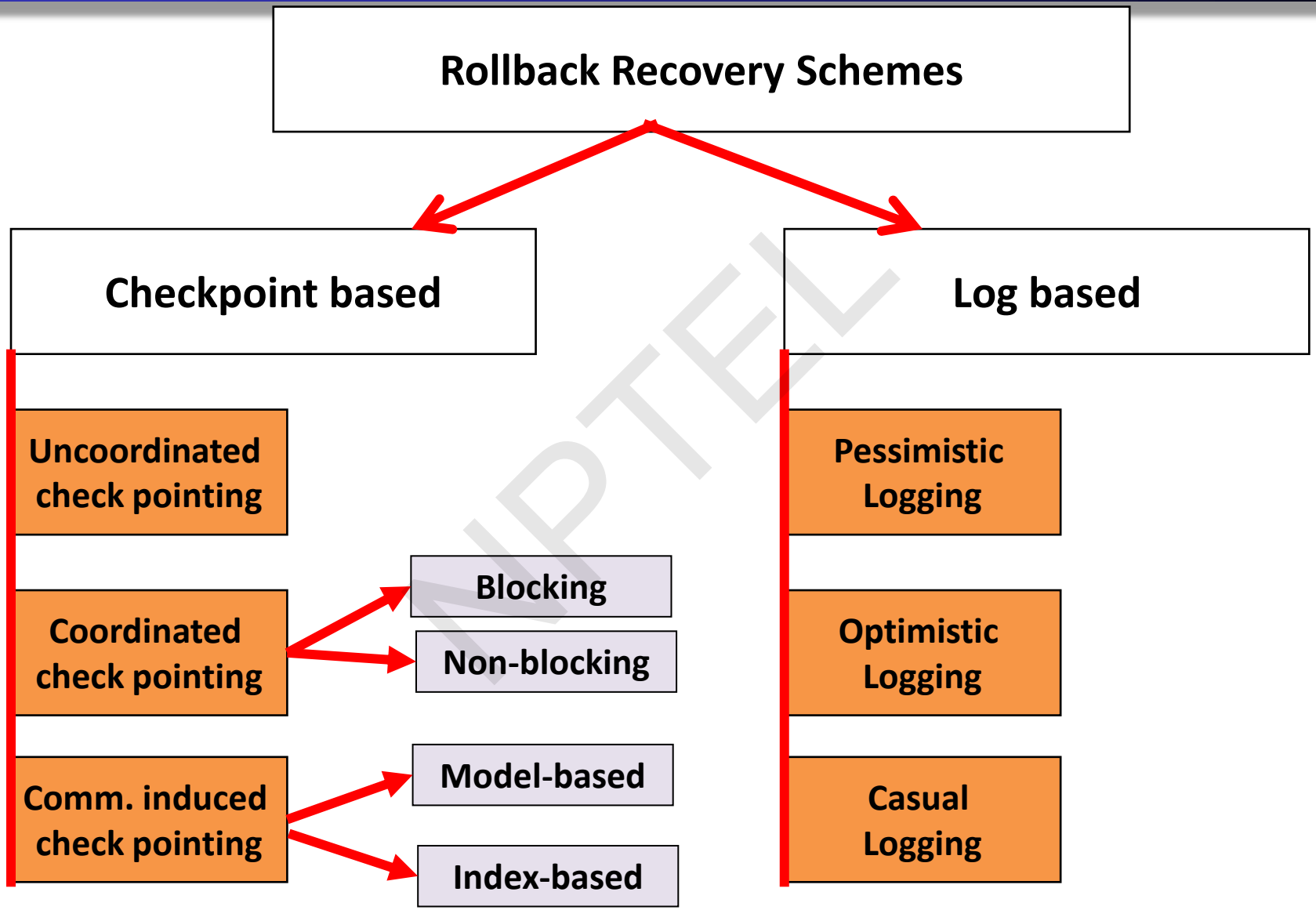
Case-I

- Process Y fails before receiving message 'n1' sent by X
- Y rolled back to y1, no record of sending message 'm1', causing X to rollback to x1

Case-II

- When Y restarts, sends out 'm2' and receives 'n1' (delayed)
- Y has to roll back again, since there is no record of 'n1' being sent
- This cause X to be rolled back again, since it has received 'm2' and there is no record of sending 'm2' in Y
- The above sequence can repeat indefinitely

Different Rollback Recovery Schemes



Checkpoint Based Recovery Schemes

Checkpoint Based Recovery: Overview

1. Uncoordinated Checkpointing: Each process takes its checkpoints independently

2. Coordinated Checkpointing: Process coordinate their checkpoints in order to save a system-wide consistent state. This consistent set of checkpoints can be used to bound the rollback

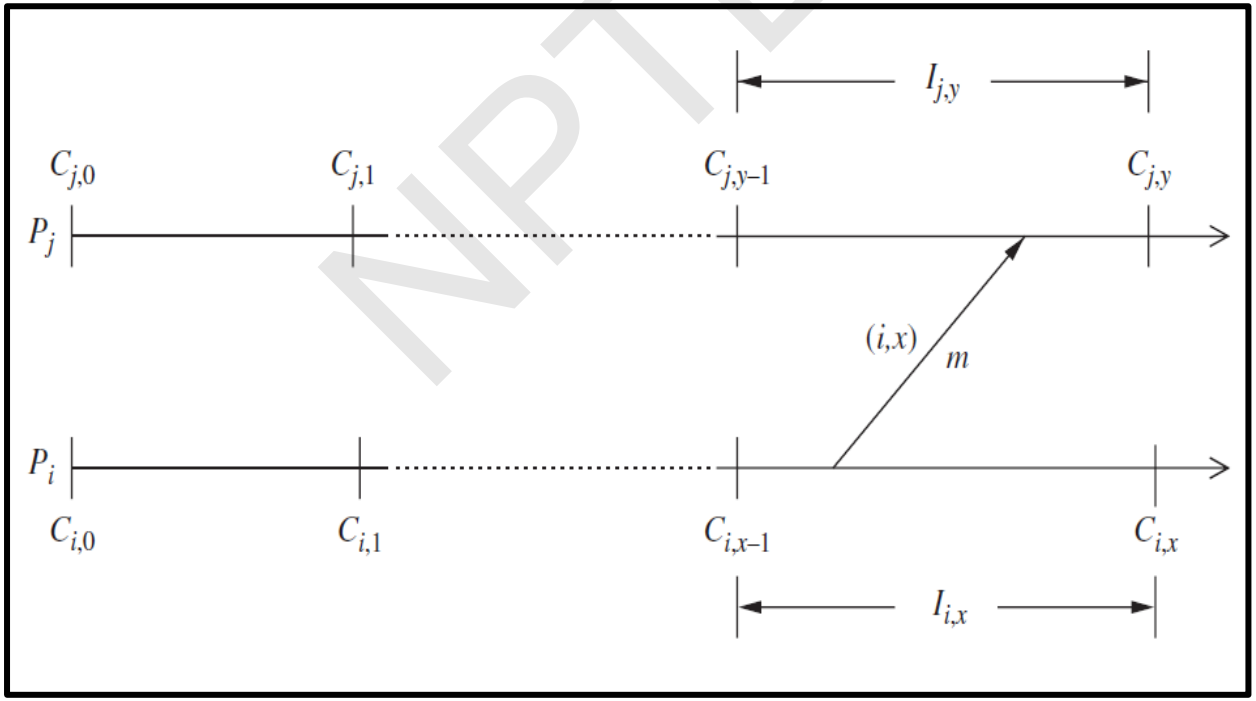
3. Communication-induced Checkpointing: It forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes.

1. 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 output commits
- The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation

Example: 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 $C_{j,y}$



2. Coordinated Checkpointing

- **Blocking Checkpointing**
  - After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete
  - **Disadvantages**
    - The computation is blocked during the checkpointing

- **Non-blocking Checkpointing**
  - The processes need not stop their execution while taking checkpoints
  - A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

Example

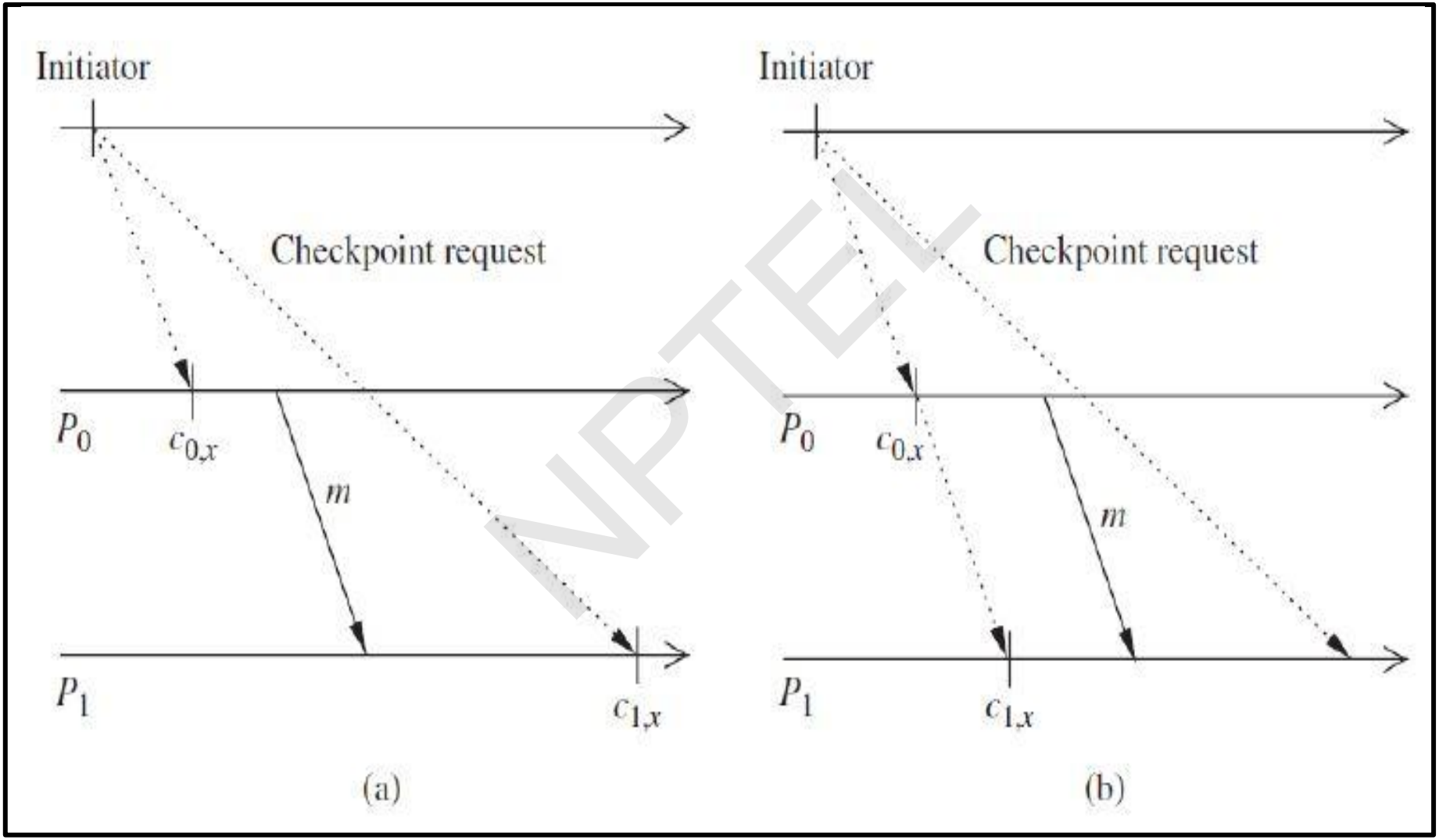
Example (a) : checkpoint inconsistency

- 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 the first post-checkpoint message.

Contd…


Initiator
Checkpoint request
P_0
$c_{0,x}$
m
P_1
$c_{1,x}$
(a)
Initiator
Checkpoint request
P_0
$c_{0,x}$
m
P_1
$c_{1,x}$
(b)

3. Communication-induced Checkpointing

- **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

Contd…

- **Two types of communication-induced checkpointing**
  - model-based checkpointing and
  - index-based checkpointing.

In '***model-based checkpointing',*** the system **maintains checkpoints and communication structures** that prevent the domino effect or achieve some even stronger properties.

In '***index-based checkpointing',*** *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.

Log-based Rollback Recovery Schemes

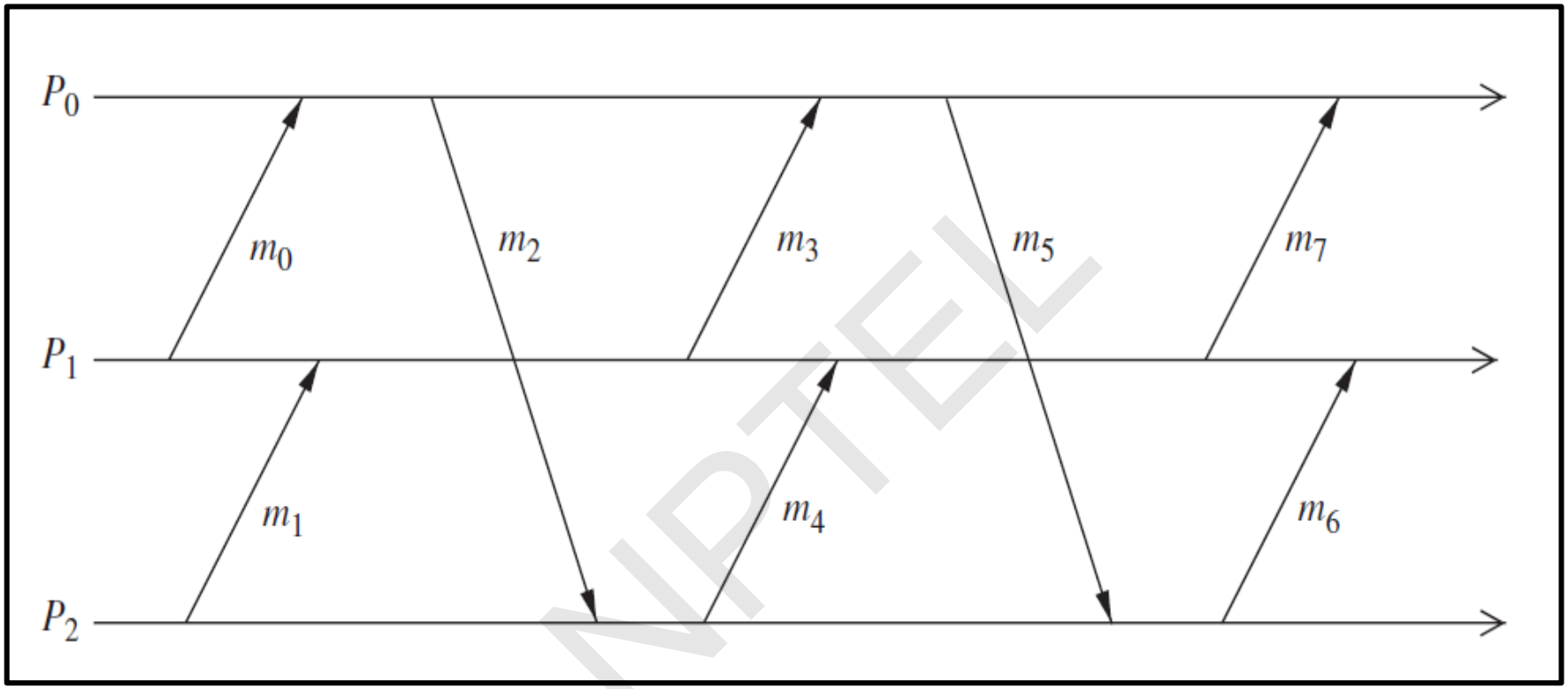
Log-based Rollback Recovery: Overview

- **It combines checkpointing with logging of nondeterministic events**.
- It relies on the **piecewise deterministic (PWD) assumption**, which postulates that all nondeterministic 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 (all info. necessary to replay the event).
- By logging and replaying the nondeterministic 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 is in general attractive for applications that frequently interact with the outside world which consists of input and output logged to stable storage.

Contd…

- 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
  - 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

Deterministic and Non-deterministic events: Example



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.

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**

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

Log-based recovery schemes

- Schemes differ in the way the determinants are logged into the stable storage.

1. **Pessimistic Logging:** The application has to block waiting for the determinant of each nondeterministic event to be stored on stable storage before the effects of that event can be seen by other processes or the outside world. It simplifies recovery but hurts the failure-free performance.
2. **Optimistic Logging:** The application does not block, and the determinants are spooled to stable storage asynchronously. It reduces failure free overhead, but complicates recovery.
3. **Casual Logging:** Low failure free overhead and simpler recovery are combined by striking a balance between optimistic and pessimistic logging.

1. Pessimistic Logging

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

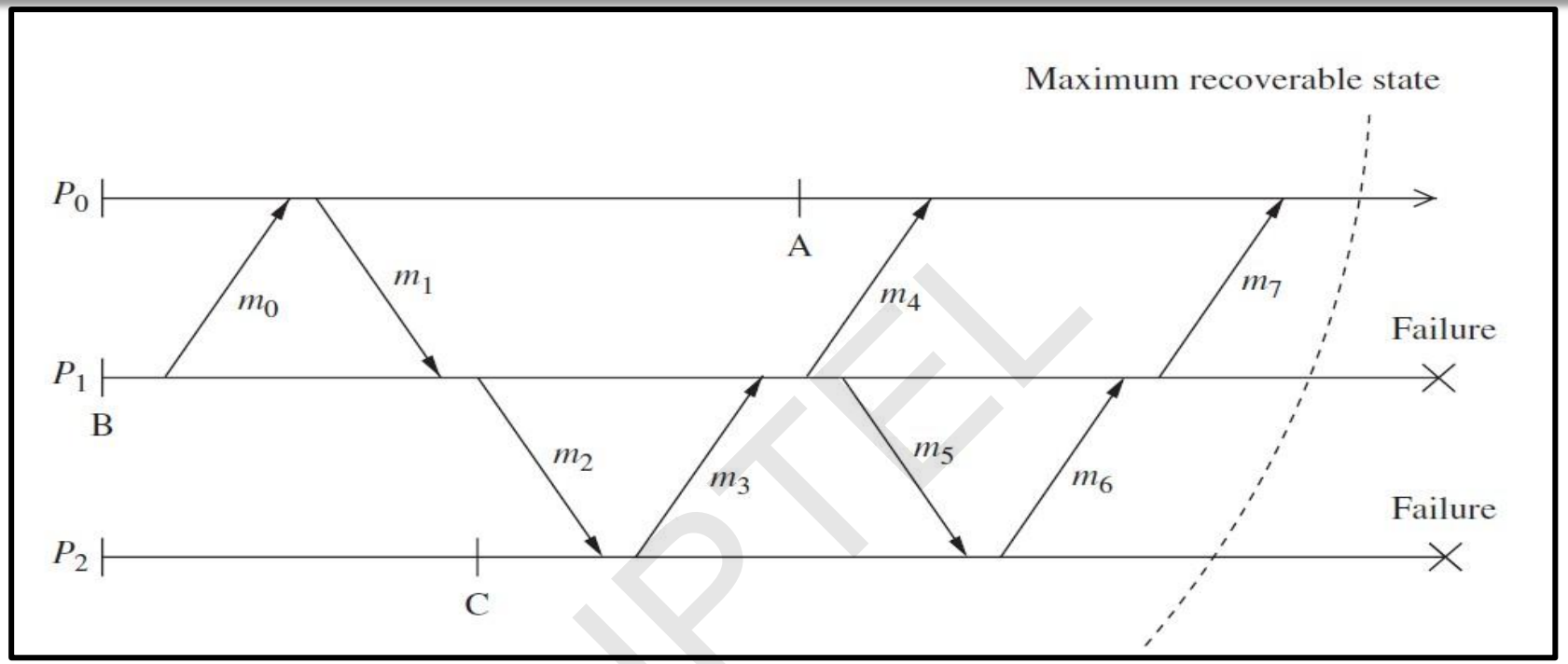
- However, in reality failures are rare

Synchronous logging:

  - $\forall e: \neg Stable(e) \Rightarrow |Depend(e)| = 0$
  - 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

Pessimistic Logging: Example



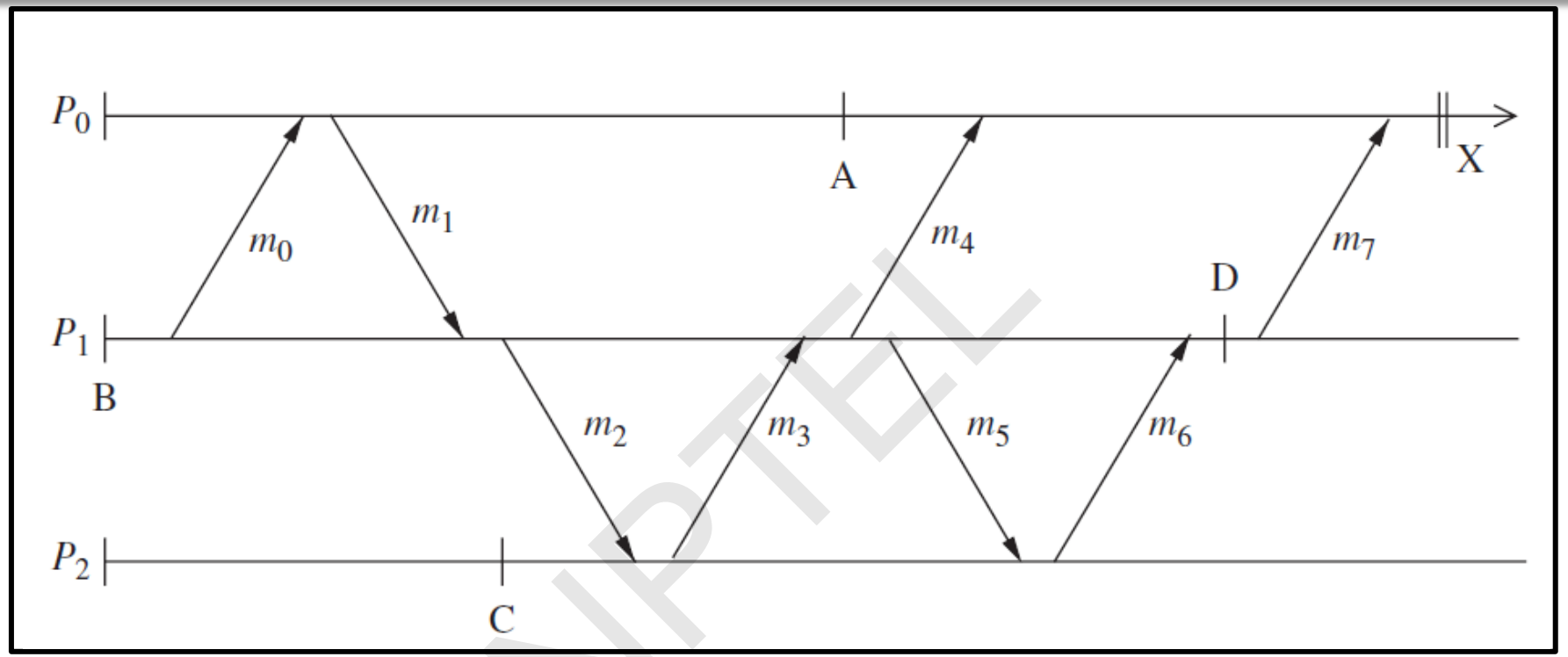
- Suppose processes P_1 and P_2 fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to deliver again the same sequence of messages as in the pre-failure execution

- Once the recovery is complete, both processes will be consistent with the state of P_0 that includes the receipt of message m_7 from P_1

2. Optimistic Logging

- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a failure occurs
- Do not implement the ***always-no-orphans*** condition
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution
- Optimistic logging protocols require a non-trivial garbage 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

Optimistic Logging: Example

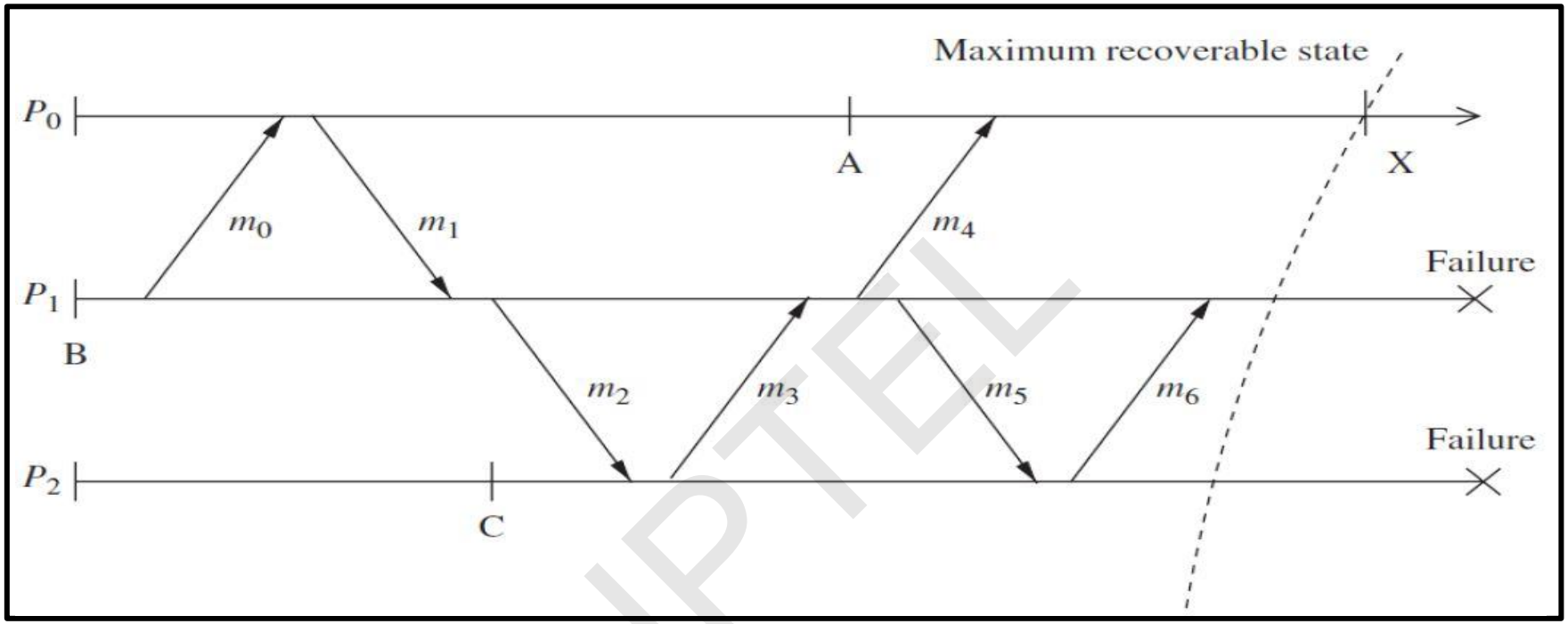


Consider the example shown in figure. Suppose process P_2 fails before the determinant for m_5 is logged to the stable storage. Process P_1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message m_6. The rollback of P_1 further forces P_0 to roll back to undo the effects of receiving message m_7.

3. Causal Logging

- **Combines the advantages of both pessimistic and optimistic logging** at the expense of a more complex recovery protocol
- **Like optimistic logging**, it does not require synchronous access to the stable storage except during output commit
- **Like pessimistic logging**, it allows each process to commit output independently and never creates orphans, thus isolating processes from the effects of failures at other processes
- Make sure that the always-no-orphans property holds
- Each process maintains information about all the events that have causally affected its state

Causal Logging: Example



Messages m_5 and m_6 are likely to be lost on the failures of P_1 and P_2 at the indicated instants. Process P_0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's happened-before relation. These events consist of the delivery of messages m_0, m_1, m_2, m_3, and m_4. The determinant of each of these non-deterministic events is either logged on the stable storage or is available in the volatile log of process P_0.

Checkpointing and Recovery Algorithms

Koo-Toueg Coordinated Checkpointing Algorithm

- **Koo and Toueg (1987)** proposed a coordinated checkpointing and recovery technique that takes a consistent set of checkpointing and **avoids 'domino effect' and 'livelock problems'** during the recovery

- It Includes 2 parts:

(i) The checkpointing algorithm and

(ii) The recovery algorithm

Contd…

(i) The Checkpointing Algorithm

- **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

- Two kinds of checkpoints: permanent and tentative
  - **Permanent checkpoint:** local checkpoint, part of a consistent global checkpoint
  - **Tentative checkpoint:** temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully

Contd…

Checkpointing Algorithm:

- **2 phases**
  1. 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
  2. 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)

Contd…

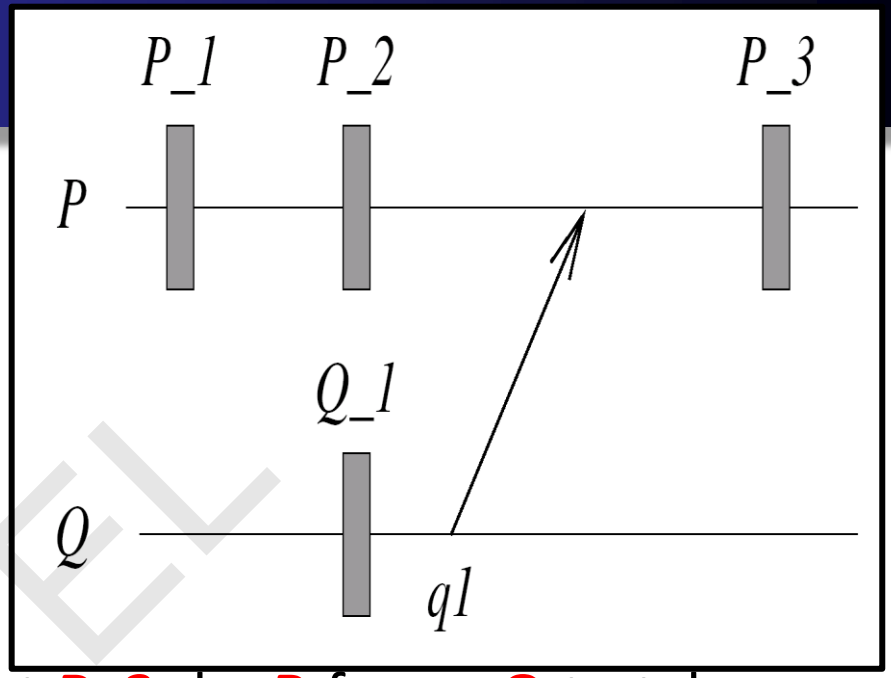
(ii) The Rollback Recovery Algorithm:

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

- **2 phases**
  1. The initiating process send a message to all other processes and ask for the preferences – restarting to the previous checkpoints. All need to agree about either do or not.
  2. The initiating process send the final decision to all processes, all the processes act accordingly after receiving the final decision.

Example

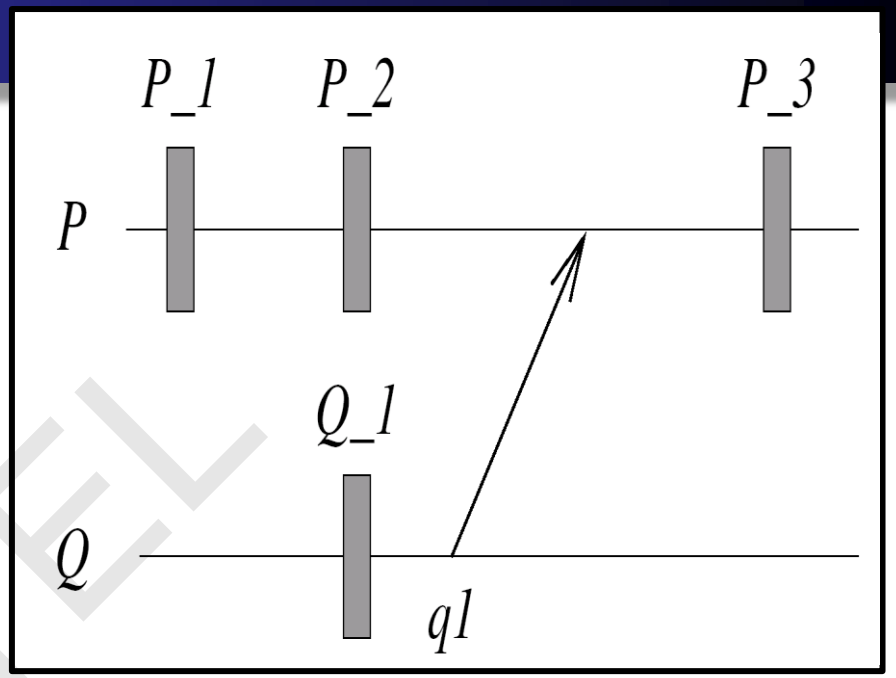
Suppose **P** wants to establish a checkpoint at **P_3** This will record that **q1** was received from **Q** - to prevent q1 from being orphaned, Q must checkpoint as well



- Thus, establishing a checkpoint at **P_3** by **P** forces **Q** to take a checkpoint to record that **q1** was sent
- An algorithm for such coordinated checkpointing has two types of checkpoints - tentative and permanent
- **P** first records its current state in a tentative checkpoint, then sends a message to all other processes from whom it has received a message since taking its last checkpoint
- Call the set of such processes Π

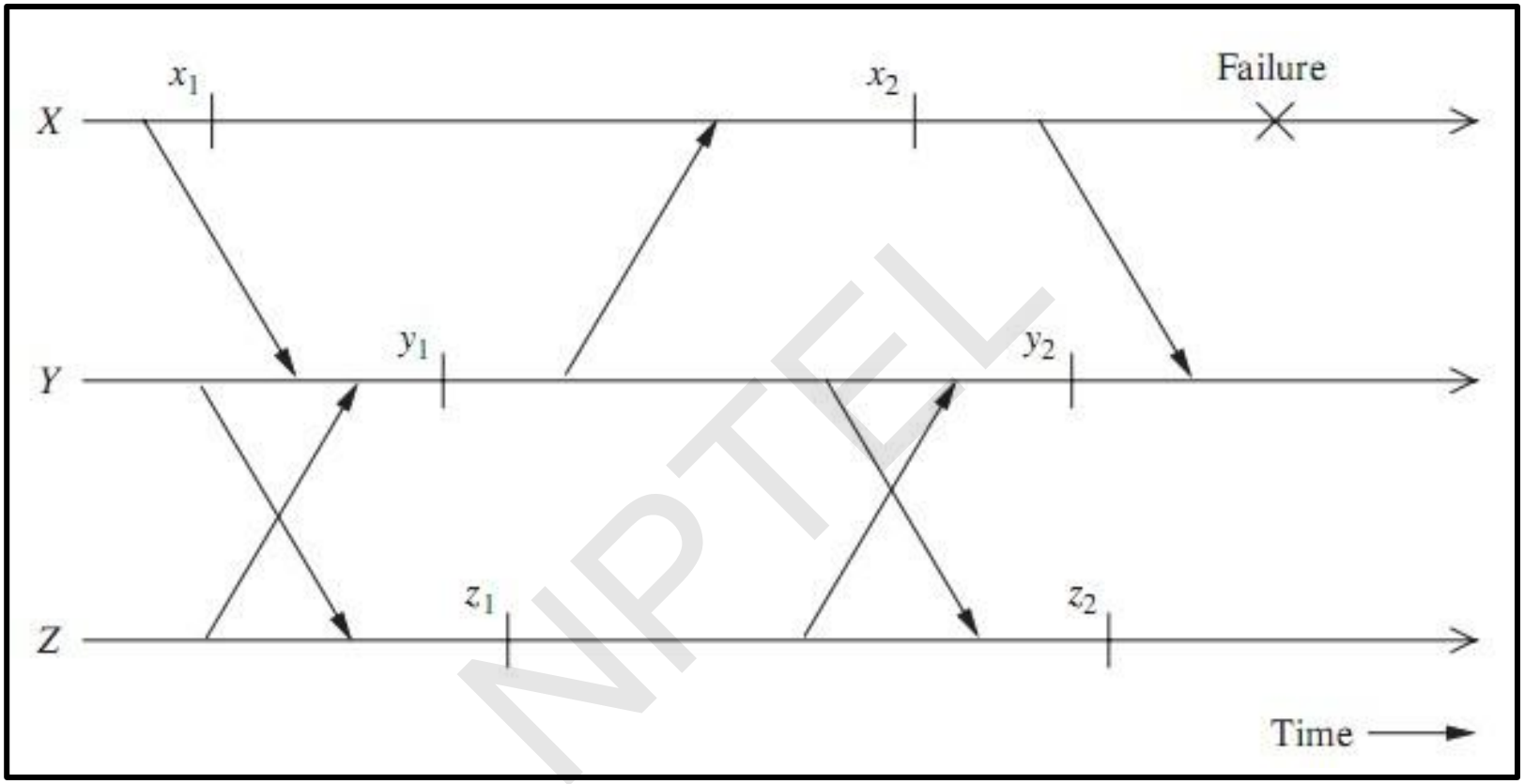
Contd…

The message tells each process in Π (e.g., Q), the last message, **m_qp**, that **P** has received from it before the tentative checkpoint was taken If **m_qp** was not recorded in a checkpoint by **Q**: to prevent **m_qp** from being orphaned, **Q** is asked to take a tentative checkpoint to record sending **m_qp**



- If all processes in Π, that need to, confirm taking a checkpoint as requested, then all tentative checkpoints can be converted to permanent
- If some members of Π, are unable to checkpoint as requested, **P** and all members of Π abandon the tentative checkpoints, and none are made permanent
- This may set off a chain reaction of checkpoints
- Each member of Π can potentially spawn a set of checkpoints among processes in its corresponding set

Contd…



- **Correctness:** resume from a consistent state
- **Optimization:** may not to recover all, since some of the processes did not change anything

Other Algorithms for Checkpointing and Recovery

Algorithm	Basic Idea
Juang–Venkatesan (1991) algorithm for asynchronous checkpointing and recovery	Since the algorithm is based on asynchronous checkpointing, the main issue in the recovery is to find a consistent set of checkpoints to which the system can be restored. The recovery algorithm achieves this by making each processor keep track of both the number of messages it has sent to other processors as well as the number of messages it has received from other processors.
Manivannan–Singhal (1996) quasi-synchronous checkpointing algorithm	The Manivannan–Singhal quasi-synchronous checkpointing algorithm Improves the performance by eliminating useless checkpoints. The algorithm is based on communication-induced checkpointing, where each process takes basic checkpoints asynchronously and independently, and in addition, to prevent useless checkpoints, processes take forced checkpoints upon the reception of messages with a control variable.
Peterson–Kearns (1993) algorithm based on vector time	The Peterson–Kearns checkpointing and recovery protocol is based on the optimistic rollback. Vector time is used to capture causality to identify events and messages that become orphans when a failed process rolls back.
Helary–Mostefaoui–Netzer–Raynal (2000, 1997) communication-induced protocol	The Helary–Mostefaoui–Netzer–Raynal communication-induced checkpointing protocol prevents useless checkpoints and does it efficiently. To prevent useless checkpoints, some coordination is required in taking local checkpoints.

Conclusion

- **Rollback recovery achieves fault tolerance** by periodically saving the state of a process during the failure-free execution, and restarting from a saved state on a failure to reduce the amount of lost computation.

- There are three basic approaches for checkpointing and failure recovery: (**i) uncoordinated, (ii) coordinated, and (iii) communication induced checkpointing** and for **log based: (i) Pessimistic, (ii) Optimistic and (iii) Casual Logging**

- Over the last two decades, **checkpointing and failure recovery** has been a very active area of research and several checkpointing and failure recovery algorithms have been proposed. In this lecture, we described '**Koo-Toueg Coordinated Checkpointing Algorithm'** and given an overview of other algorithms.