# Logging & Checkpointing
# Towards Recovery of Distributed Systems

## Distributed Systems / Case Study

Andrea Omicini

mailto:andrea.omicini@unibo.it
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Academic Year 2025/2026

1. Interaction, Dependencies, Causality, Time
2. Basic Techniques for Dependability
3. Models
4. Checkpoint-Based Protocols
5. Log-Based Protocols
6. Conclusion

# Next in Line...

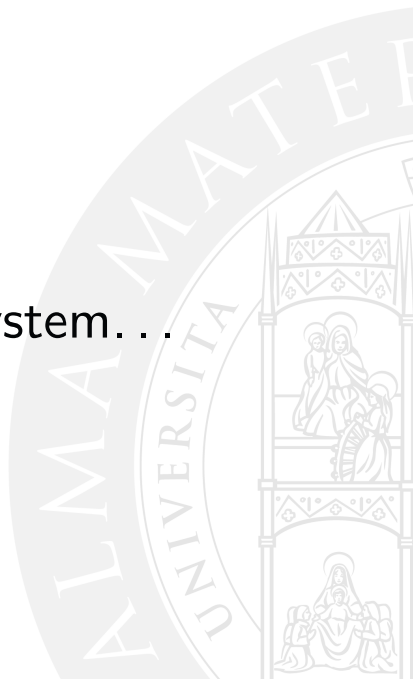
1. Interaction, Dependencies, Causality, Time
2. Basic Techniques for Dependability
3. Models
4. Checkpoint-Based Protocols
5. Log-Based Protocols
6. Conclusion

# Components

- components of a system play different roles in the overall system operation
- they have different
  - *functions* to perform
  - *services* to provide
  - *tasks* to execute
  - *goals* to achieve
  - ...

  mostly depending on the system paradigm of choice
  - of course, assuming some conceptual integrity for the system...

! yet, components of a system *do not work in isolation*



# Interaction

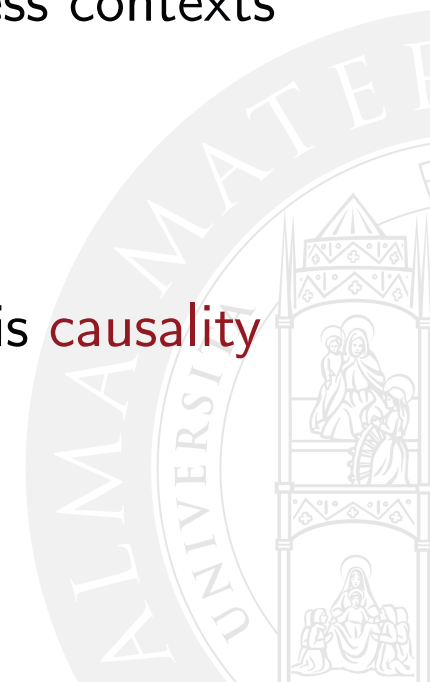
- components of a system interact with each other
  - components of a (concurrent) system interact with each other over *time*
  - components of a (distributed) system interact with each other over *space*
- ? why?
- first of all, because otherwise there would be no reason to build a system
- then, because *components depend on each other*

# (Inter)Dependencies

- components of a system depend on each other for whatever (function, service, task, goal) they are up to
  - (the successful completion of) their *operation* depends on (one or more of) the other components
- *inter-component dependencies* span over different process contexts
  - over time in concurrent systems
  - over space in distributed systems

? how do we model dependencies?

! one (mischievously) simple way to model dependencies is causality



# Causality I

## Cause as a cognitive tool

- the "cause-effect" link is a tool for humans to understand the dynamics of reality
- it is one of the basic mechanisms for us explain and predict the world around us
- we even use *motivations and goals* in order to understand, explain, and predict the course of actions of other humans we observe
  - which is called *mind reading*

# Causality II

## Cause in science

- things are not as simple as that
- two observations over physics scientific literature
  - the word "cause" is quite often used in the *title* of the physics papers of the nineteenth century
  - the word "cause" is almost never used in the *body* of the physics papers of the nineteenth century
- do we have a scientifically-viable notion of cause?
- is the "cause-effect" link a suitable tool for science to explain and predict the dynamics of reality?

# Causality III

## Correlation vs. causation: are there *actual* causes?

- *association* of effects might be a confounding factor
- e.g., let us hypothesises that some "carcinogenic genotype" in humans exists that leads to both lung cancer and predisposition to smoke
  - as a result, even a strong *correlation* in numbers between lung cancer and smokers would not allow us to point out any causal relationship
  - this was actually used by the tobacco industry when fighting back on the first research results to forestall antismoking legislation
- generally speaking, this is the main idea behind "correlation is not causation"

? does that mean that we cannot actually find "actual causes"?

# Causality IV

## Correlation vs. causation: detecting actual causes

? does this mean that we cannot actually find "actual causes"?

! of course we can, yet we need a precise conceptual and mathematical definition

- where *statistical* and *probabilistic* concepts and notions are suitably defined and distinguished

- "Causality: Models, Reasoning, and Inference", by Judea Pearl, is an essential treaty to understand the core problem at every level, and its available solutions[Pearl, 2009]
  - there, correlation is handled as a statistical concept
  - whereas causation is mathematically defined as a probabilistic one

# Causality V

## Causality and time

- once we have properly delimited the notion of cause (and causality), we can face some of the main problems in distributed systems
  - luckily for us, with no actual need of the full mathematical and philosophical apparatus
- we will use the notion of cause as strictly related to the abstraction of *system model*
  - in order to define dependencies between components—typically, (sequential) *processes*
  - and possibly to approximate some useful notion of (distributed) *time*

# Time

- *time* is an essential issue in distributed systems
- the basic idea here is that a causal link determines a *temporal relationship* between two *events*
- where, according to our intuition, a cause *temporally precedes* its effects
- possibly, an oversimplified notion—which however has its use in our specific context

# Next in Line...

1 Interaction, Dependencies, Causality, Time

2 Basic Techniques for Dependability

3 Models

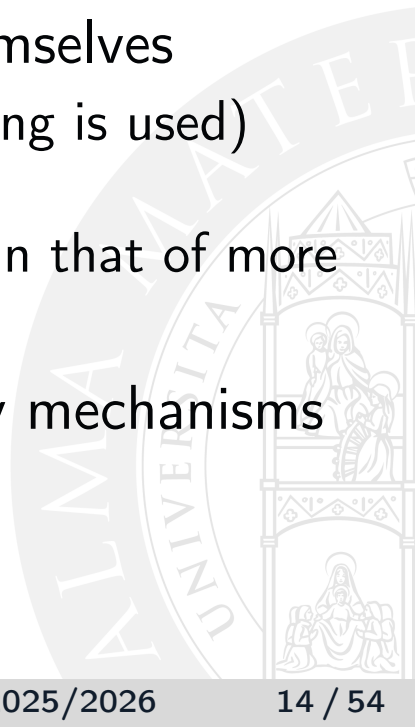
4 Checkpoint-Based Protocols

5 Log-Based Protocols

6 Conclusion

# Basic Mechanisms for Dependability

- checkpointing and logging are the most fundamental techniques to achieve dependability in distributed systems
  - by providing a path towards *distributed system recovery* after failure
- by themselves, they provide a form of fault tolerance that is relatively easy to implement and incurs low runtime overhead
- they have limitations, mostly when they are used by themselves
  - e.g., some information could be lost (if only checkpointing is used) when a fault occurs
  - and the recovery time after a fault is typically larger than that of more sophisticated fault tolerance approaches
- however, they are both used in *all levels* of dependability mechanisms



# Checkpointing: The Intuition

- a checkpoint of a distributed system refers to a copy of the system state
- if the checkpoint is available after the system fails, it can be used to *recover the system* to the state when the checkpoint was taken
- technically, checkpointing refers to the action of
  - (periodically) taking a *copy* of the system state, and
  - *saving* the checkpoint to a stable storage that can survive the faults tolerated

# Recovery: Spoiler Alert

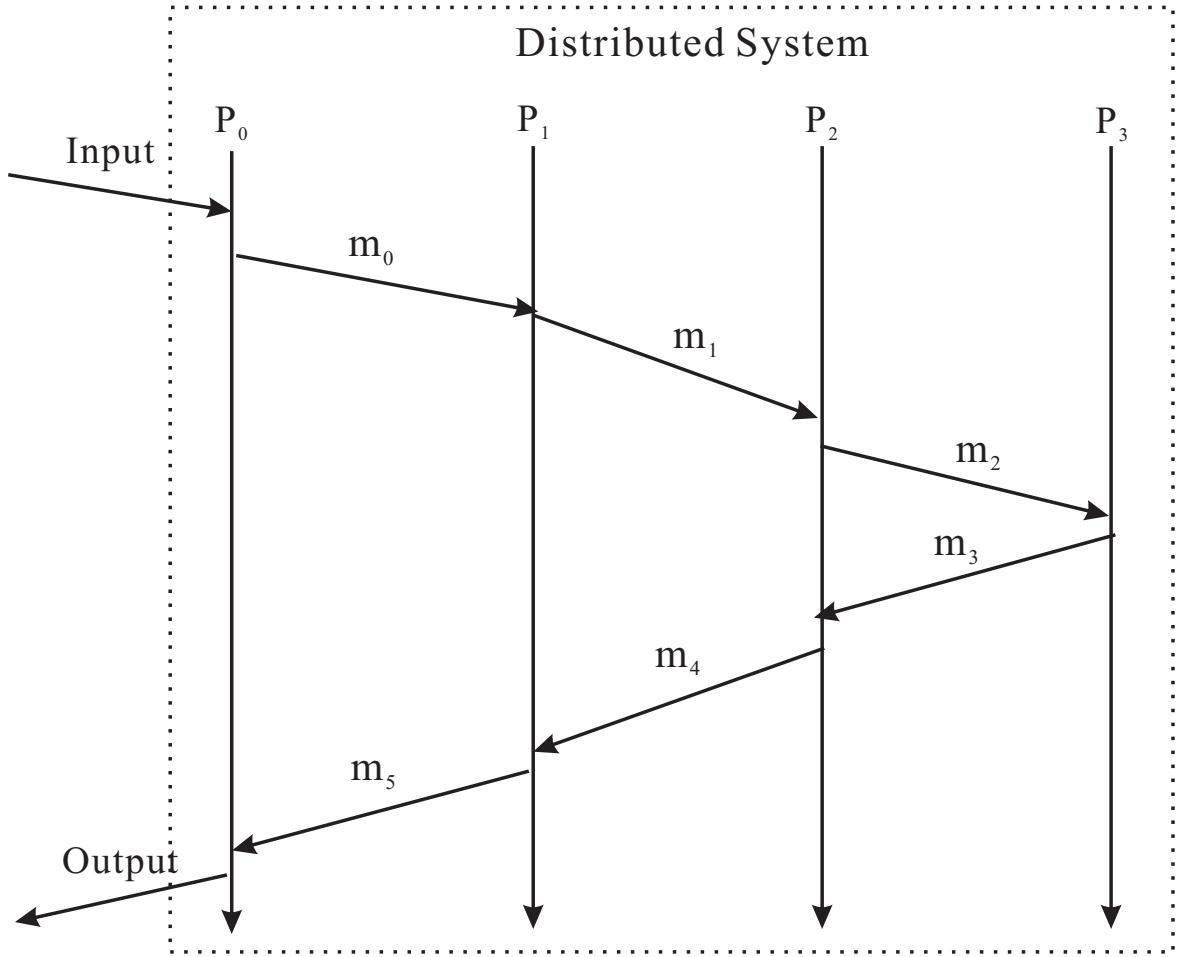
- to recover the system to the point right before it fails, other recovery information must be logged *in addition* to periodical checkpointing
- to this end, all incoming messages to the system are typically logged
- also, other nondeterministic events may have to be logged as well to ensure proper recovery
- checkpointing and logging provide a form of *rollback recovery*, since they can recover the system to a state *prior to the failure*

! mechanisms for *roll-forward recovery* exist, yet they typically incur higher runtime overhead and demand more physical resources

# Next in Line...


1. Interaction, Dependencies, Causality, Time
2. Basic Techniques for Dependability
3. Models
4. Checkpoint-Based Protocols
5. Log-Based Protocols
6. Conclusion

# System Model I




[Zhao, 2014]

# System Model II

- $N$ processes interacting through *message exchange*
- also interacting with the outside world via *input & output*

! the picture above also suggests some form of *layering* of processes—which however is not necessarily part of the model

# Fault Model

Assumptions

- failure occurs in a process
- when a process fails, it simply stops execution and loses all its volatile state—fail-stop
- communication is reliable – e.g., TCP – and FIFO

→ message ordering maintained

→ no network partition

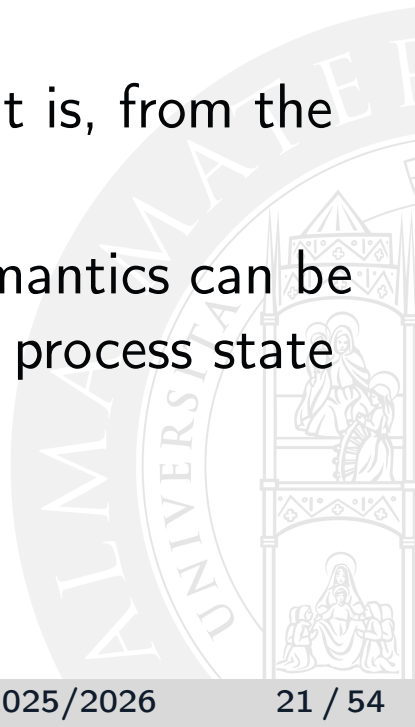
# Process State

Assumptions

- the state of each individual process is defined by its entire address space in the operating system

→ a generic checkpointing library just saves the entire address space as a checkpoint of the process

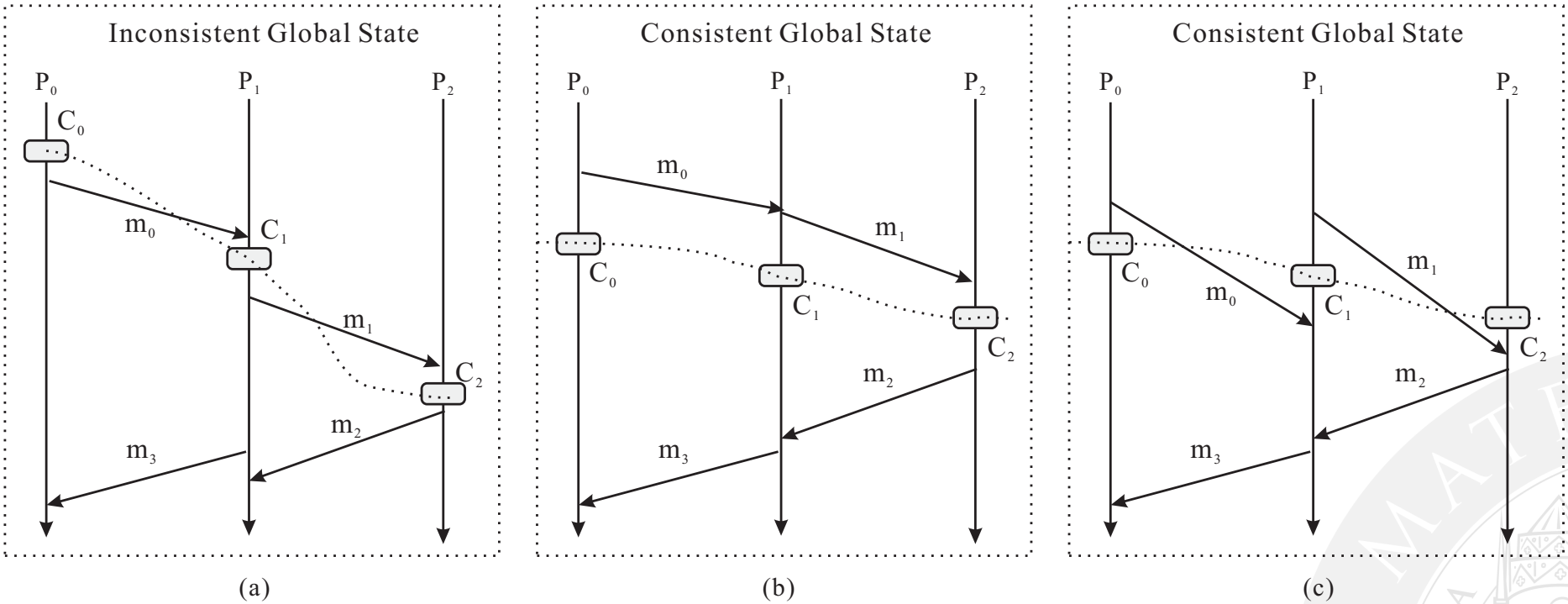
- yet, that is the most general notion of process state, that is, from the operating system viewpoint
- the specific nature of the process and the application semantics can be exploited to define a smaller and more specific notion of process state



# Global State I

- the state of a distributed system is usually referred to as the global state of the system
  - which is meant to include the state of every process in the system
- aggregation of process state is not enough: the states of different processes in a distributed system are *related*
  - because *message* exchanges, moving information between processes
  - information exchange *causing* change of state
  - processes causally depend on other processes
  - process states are *interdependent*
  - ! dependencies cannot be lost in the global state

# Global State II




[Zhao, 2014]

# Global State III

## Scenario ($a$): inconsistent global state

- checkpoints taken by different processes are *incompatible*

→ they cannot be used to recover the system upon failure

- in fact
  - $C_1$ reflects reception of $m_0$, $C_0$ does not
- e.g.,
  - $P_0$ and $P_1$ represent two bank accounts $A$ and $B$, respectively
  - $m_0$ represents a \$100 deposit from $A$ to $B$
  - if $P_0$ crashes after $m_0$, and $P_1$ after checkpoint $C_1$, recovery from $C_0$ and $C_1$ would make \$100 appear from nowhere
- global state from the wrong set of checkpoints is not *admissible*
  - i.e., it is not *reachable* from the initial state of the system
- this is what is called an inconsistent global state

# Global State IV

## Scenario ($b$): consistent global state

- checkpoints taken by different processes are *compatible*
- they could be used to recover the system upon failure
- in fact
  - $C_1$ reflects reception of $m_0$, so does $C_0$
  - the same holds for $C_1$, $C_2$, and $m_1$
- e.g.,
  - $P_0$ and $P_1$ represent two bank accounts $A$ and $B$, respectively
  - $m_0$ represents a \$100 deposit from $A$ to $B$
  - if $P_0$ crashes after $C_0$, and $P_1$ after $C_1$, recovery from $C_0$ and $C_1$ would make \$100 correctly move from $A$ to $B$

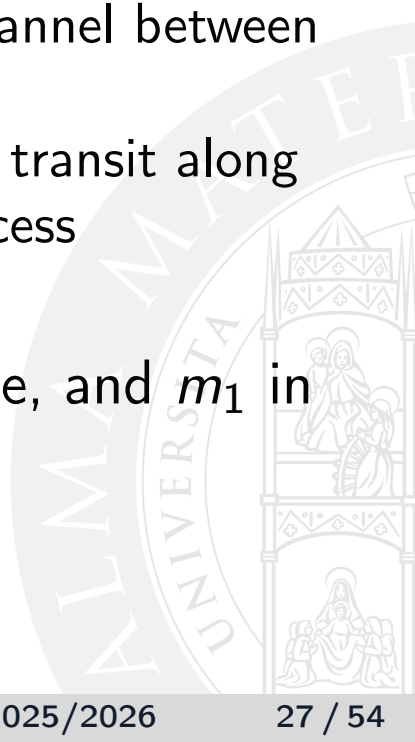
# Global State V

## Scenario ($c$): consistent yet unrecoverable global state

- checkpoints taken by different processes are *compatible*
- yet, they could *not* be used to recover the system upon failure
- in fact
  - checkpoints are consistent, since they represent a reachable state for the system
  - yet, *dependencies* on $m_0$ and $m_1$ would be *lost* on recovery
- e.g.,
  - in the example, \$100 would possibly disappear
- the problem is the loss of the messages in transit
- in order to accommodate this scenario, a further sort of state is required: channel state

# System Model Refined I

- a set of $N$ processes
  - a process consists of a set of *states* and a set of *events*
  - one state is the *initial state*
  - events trigger the change of state of a process
- a set of channels
  - a channel is a uni-directional *reliable* communication channel between two processes
  - the state of a channel is the set of messages that are in transit along the channel—that is, not yet received by the target process
  - e.g., a TCP connection is two channels
- e.g., in scenario $(c)$, if $m_0$ is saved in $C_0$ as channel state, and $m_1$ in $C_1$ the same way, recovery from $C_0$ and $C_1$ is possible



# Piecewise Deterministic Assumption

- checkpoint-based protocols only ensure to recover the system up to the most recent consistent global state that has been recorded, and all executions happened afterwards are lost
- instead, logging (log-based protocols) can be used to recover the system to the state right before the failure, provided that all events are logged, and the log is available upon recovery

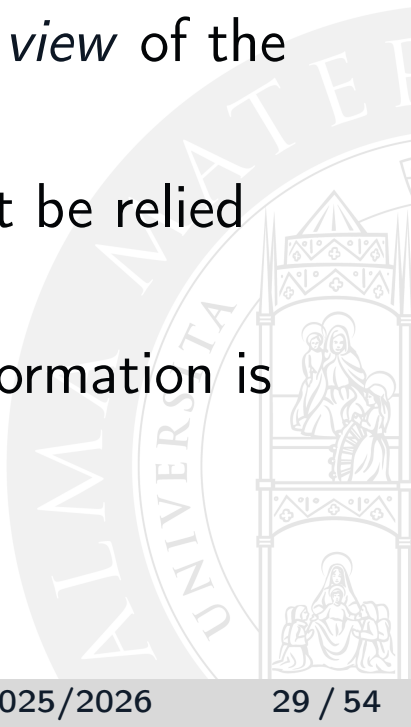
! they works based on a *model assumption*, referred to as the piecewise deterministic assumption, that is

- each state evolve deterministically until a nondeterministic event happens
  - such as the reception of a message
- all nondeterministic events can be identified
- enough information is logged for each event

# Output Commit

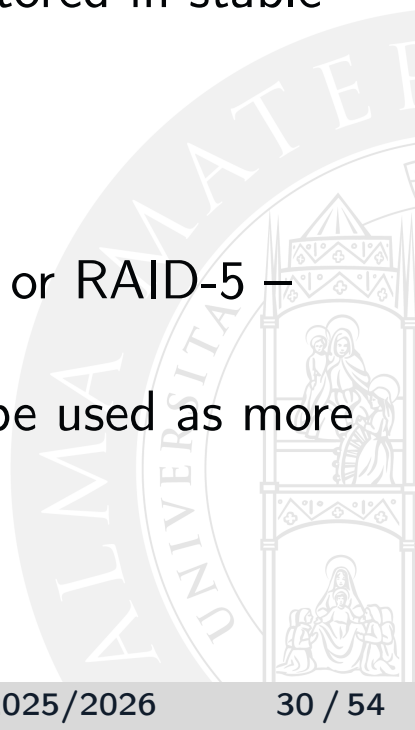
- a distributed system interacts with the outside world, such as the clients of the services provided by the distributed system
- once an output is emitted, a portion of the state of the distributed system becomes *observable* by the outside world—as a sort of commitment, or achievement
- recovery should also account for a *observably-consistent view* of the system from the outside
- even though, if a failure occurs, the outside world cannot be relied upon for recovery: this is the output commit problem

→ *observable consistency* requires that enough recovery information is logged before the system commits to an output message



# Stable Storage

- an essential requirement for logging and checkpointing protocols is the availability of stable storage
- stable storage can survive *process failures* in that upon recovery, the information stored in the stable storage is readily available to the recovering process
  - as such, all checkpoints and messages logged must be stored in stable storage
- there are various forms of stable storage
  - to tolerate only process failures, *local disks* are enough
  - to tolerate disk failures, *redundant disks* – e.g., RAID-1 or RAID-5 – could be used as stable storage
  - *replicated file systems*, or *cloud-based file systems* can be used as more robust stable storage



# Next in Line...

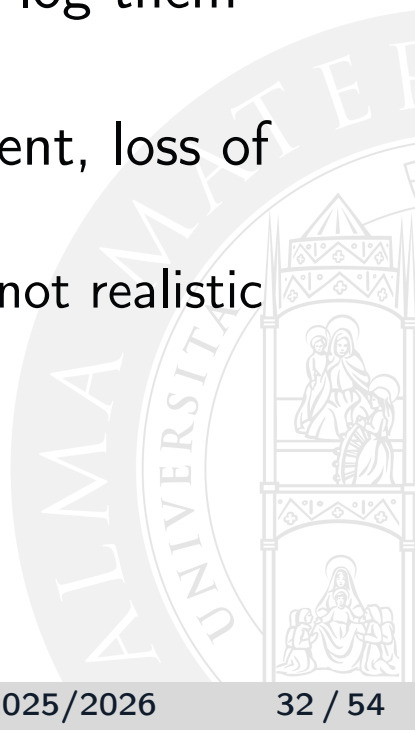

1. Interaction, Dependencies, Causality, Time
2. Basic Techniques for Dependability
3. Models
4. Checkpoint-Based Protocols
5. Log-Based Protocols
6. Conclusion

# Generality

- focussing on *state*, rather than on events, checkpoint-based protocols do not rely on the piecewise deterministic assumption

→ they are simpler to implement and less restrictive—since there is no need to identify all forms of nondeterministic events and log them properly

- however, if we cannot take a checkpoint before *every* event, loss of execution has to be tolerated
  - and, taking a checkpoint before every event is typically not realistic



# Uncoordinated Checkpointing

- in *uncoordinated checkpointing*, each process in the distributed system is autonomously decide when to checkpoints
- the intuition is that, when a failure occurs, the recovery processes goes backwards to select the most recent set of consistent checkpoints for the recovery of the global state
- the main problem is that the checkpoints taken by the processes might not allow reconstruction of a consistent global state
- to enable enable the selection of a set of consistent checkpoints during recovery, the *dependency* between the checkpoints has to be determined and recorded together with each checkpoint

→ this incurs additional overhead and increases the complexity of the implementation

# Tamir and Sequin Global Checkpointing Protocol I

- the Tamir and Sequin global checkpointing protocol[Tamir and Sequin, 1984] is a *coordinated checkpointing protocol*, where
  - one of the processes is designated as the coordinator, and knows all the other processes in the system
  - the remaining processes are participants
  - the coordinator uses a two-phase commit protocol to ensure that
    - the checkpoints taken at individual processes are consistent with each other
    - the global checkpointing operation is either carried out as an atomic transaction or aborted—that is, either all processes successfully create a new set of checkpoints or they abandon the current round and revert back to their previous set of checkpoints
  - first phase aims at creating a *quiescent* point of the distributed system state
  - second phase aims at atomically moving from the old checkpoint to the new one
  - checkpointing round is aborted when a participant fails to respond

# Tamir and Sequin Global Checkpointing Protocol II

## Control messages

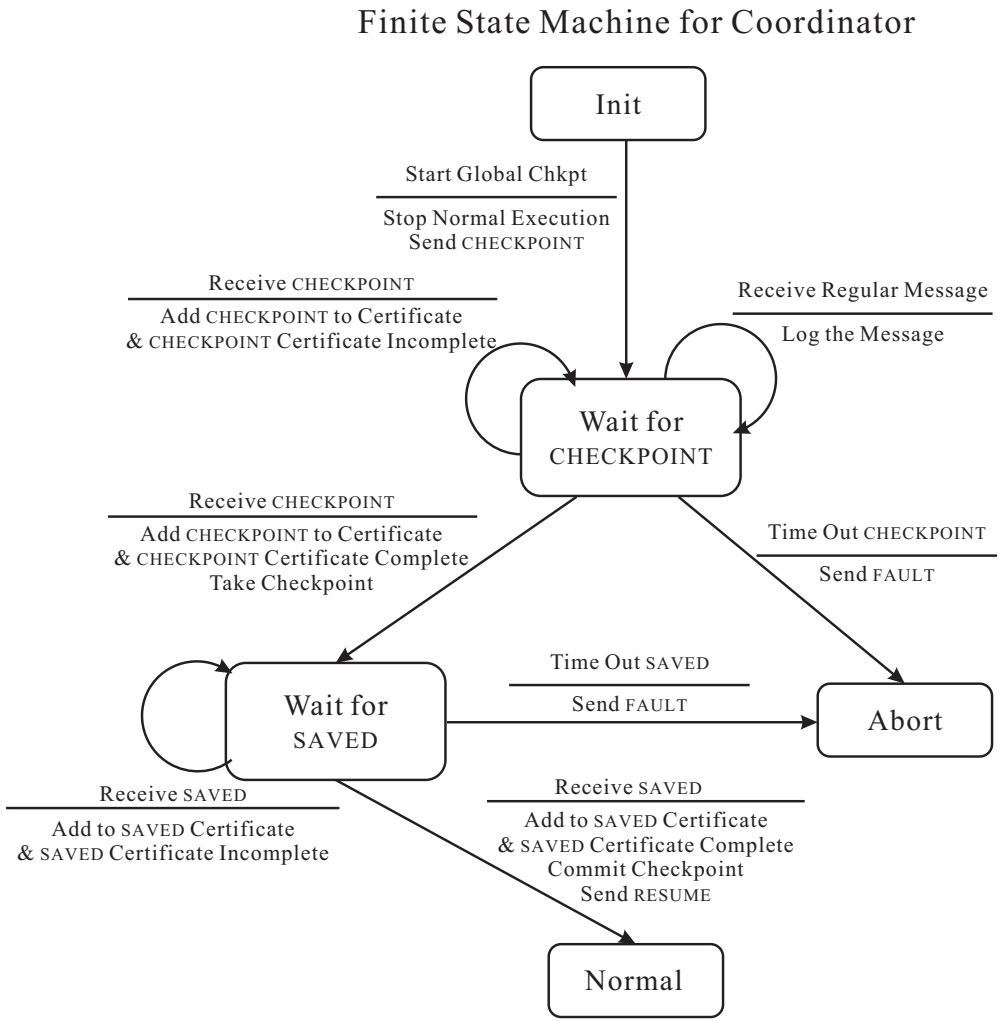
**CHECKPOINT** message — used to initiate a global checkpoint, also used to establish a quiescent point of the distributed system where all processes have stopped normal execution

**SAVED** message — used for a participant to inform the coordinator that it has done a local checkpoint

**FAULT** message — used to indicate that a timeout has occurred and the current round of global checkpointing should be aborted

**RESUME** message — used by the coordinator to inform the participants that they now can resume normal execution

# Tamir and Sequin Global Checkpointing Protocol III



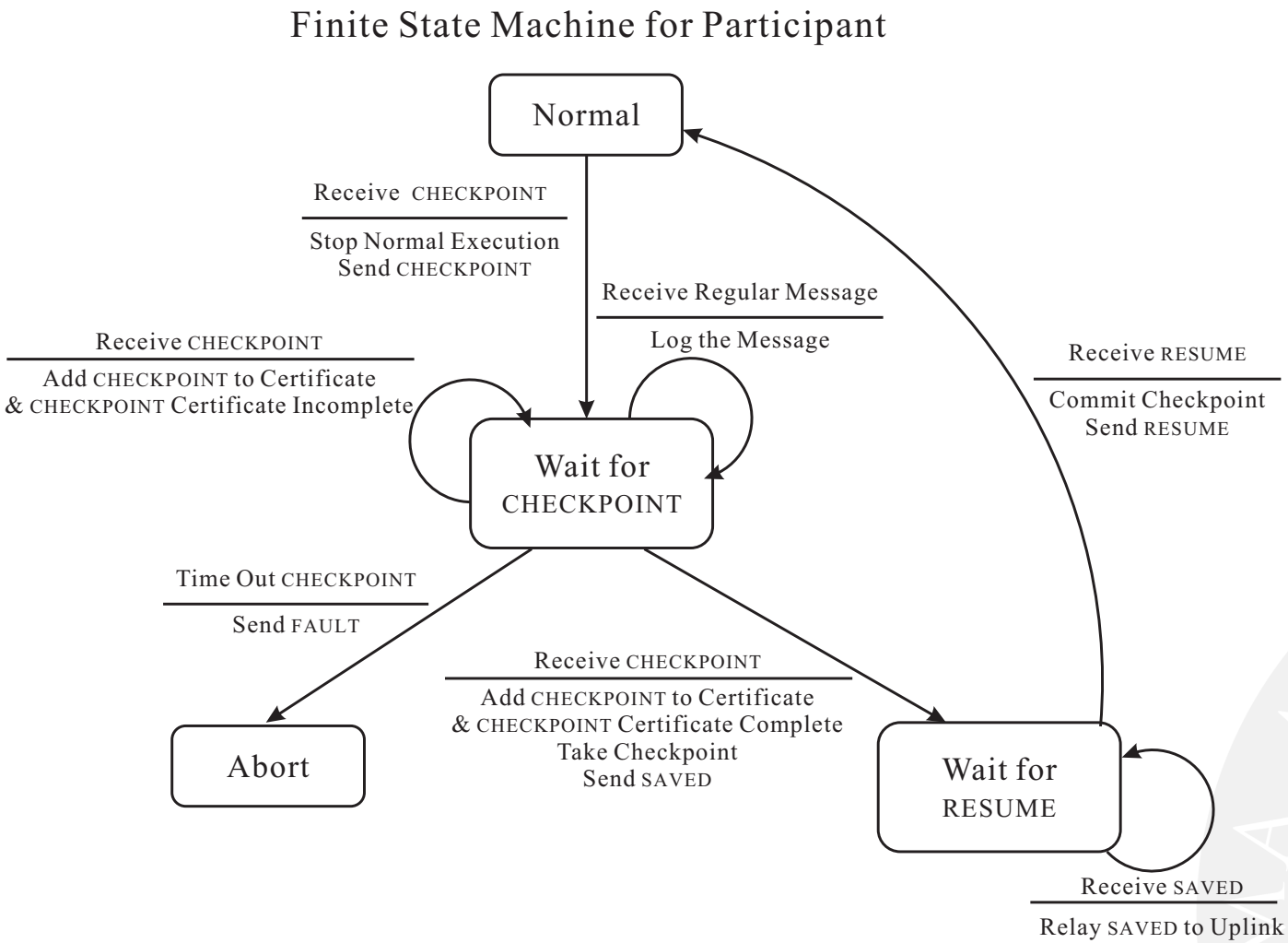

[Zhao, 2014]

# Tamir and Sequin Global Checkpointing Protocol IV

## Coordinator behaviour

- starts first phase by stopping normal execution and sending CHECKPOINT message to every process
- waits for CHECKPOINT incoming messages from all processes—aborts when some are missing
- checkpoints its state
- waits for SAVED messages from every process—aborts when some are missing, sending FAULT messages
- switches to the new checkpoint, and sends RESUME message to every process
- resumes normal execution

# Tamir and Sequin Global Checkpointing Protocol V



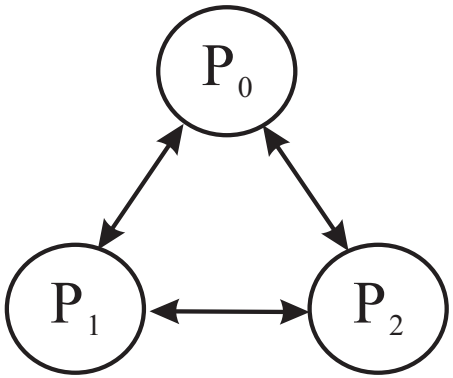

[Zhao, 2014]

# Tamir and Sequin Global Checkpointing Protocol VI

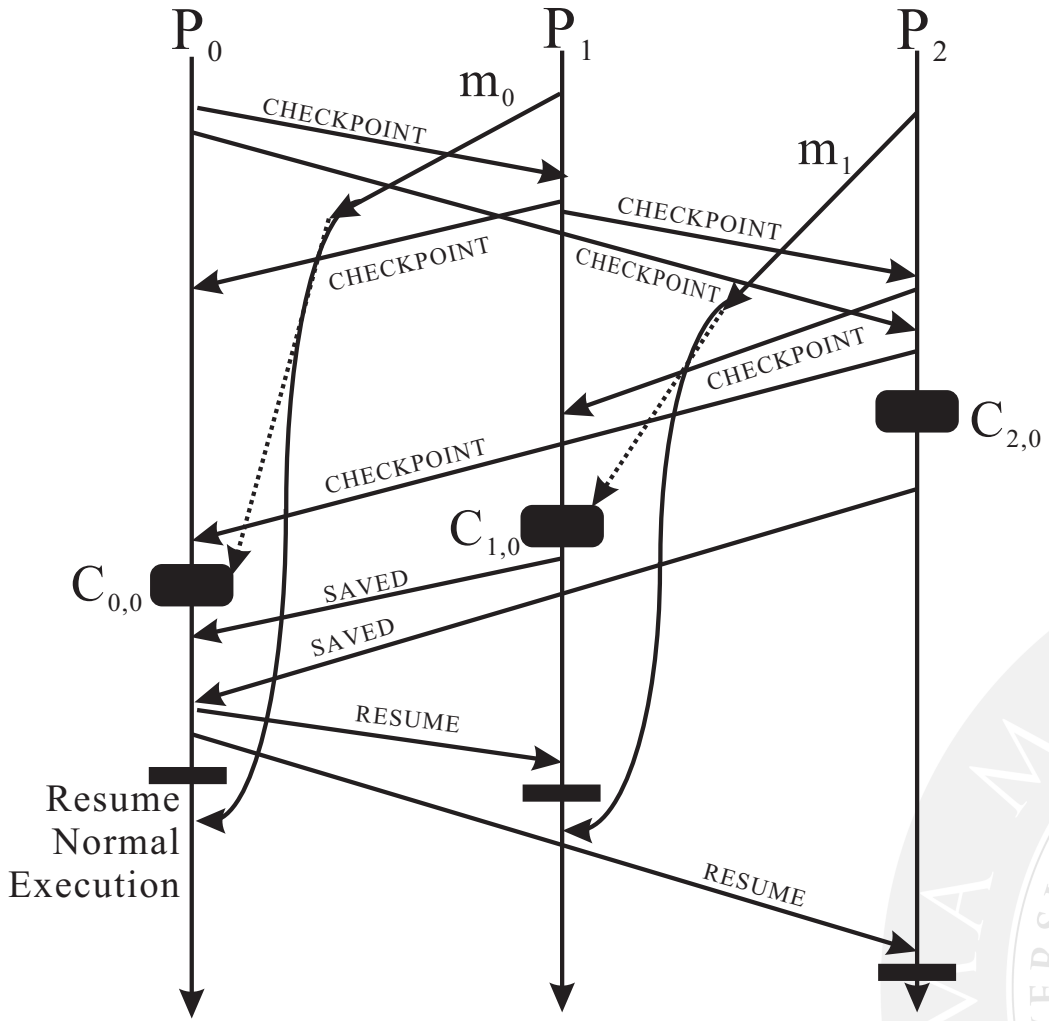
## Participant behaviour

- stops normal execution upon CHECKPOINT message reception
- sends CHECKPOINT messages through all outgoing channels and waits for CHECKPOINT from all incoming channels—aborts when some are missing
- checkpoints its state
- passes SAVED message from one of its downstream neighbour processes up to its upstream neighbour
- propagates RESUME message through every outgoing channels—except the one that just sent RESUME
- resumes normal execution

# Tamir and Sequin Global Checkpointing Protocol VII



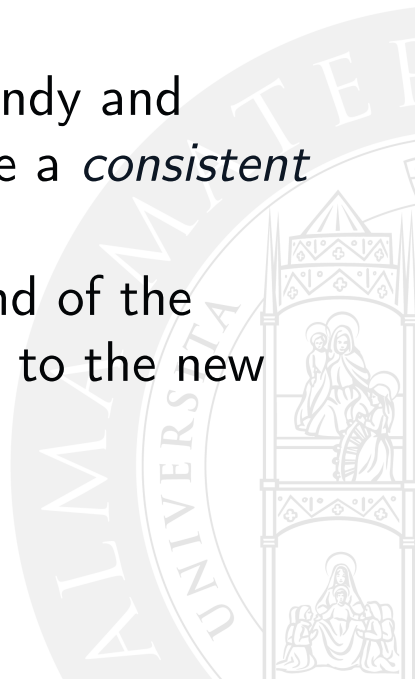

System Topology



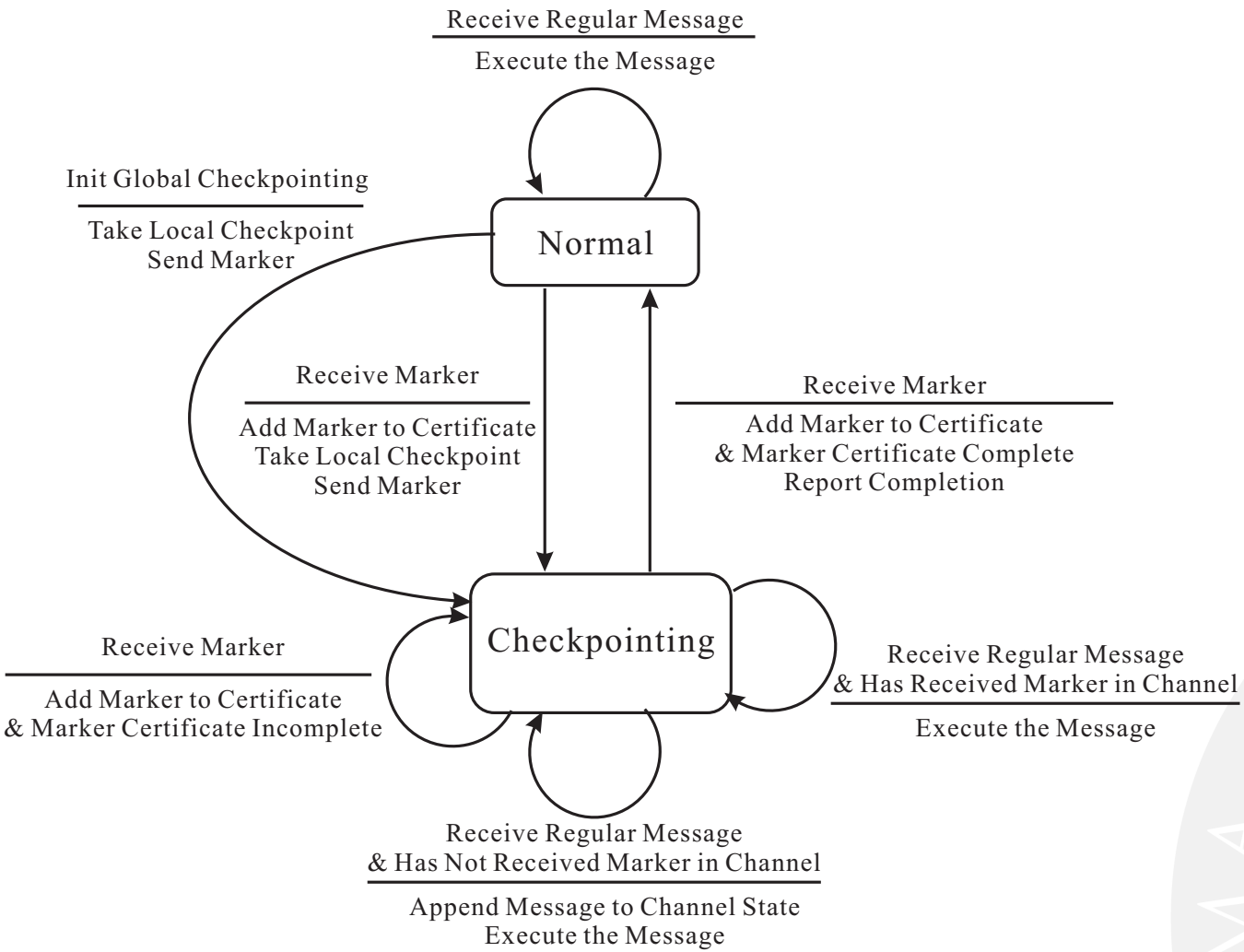

[Zhao, 2014]

# Chandy and Lamport Distributed Snapshot Protocol I

- the Tamir and Sequin global checkpointing protocol is a *blocking* protocol: normal execution is suspended during each round of global checkpointing
- the Chandy and Lamport distributed snapshot protocol[Chandy and Lamport, 1985] is a nonblocking protocol: normal execution is *not* interrupted by the global checkpointing
  - however, unlike the Tamir and Sequin protocol, the Chandy and Lamport protocol is only concerned with how to produce a *consistent global checkpoint*
  - it prescribes no mechanisms on how to determine the end of the checkpointing round, and how to atomically switch over to the new global checkpoint



# Chandy and Lamport Distributed Snapshot Protocol II



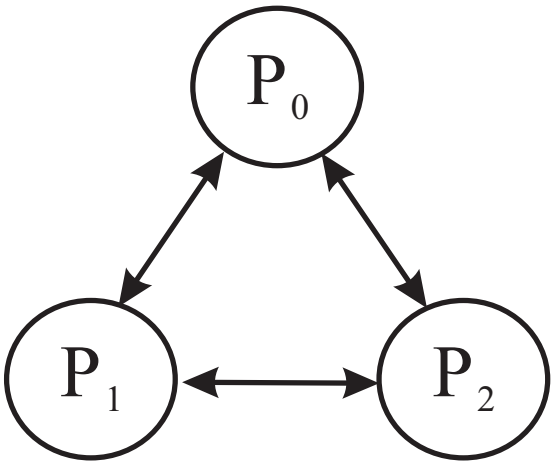

[Zhao, 2014]

# Chandy and Lamport Distributed Snapshot Protocol III

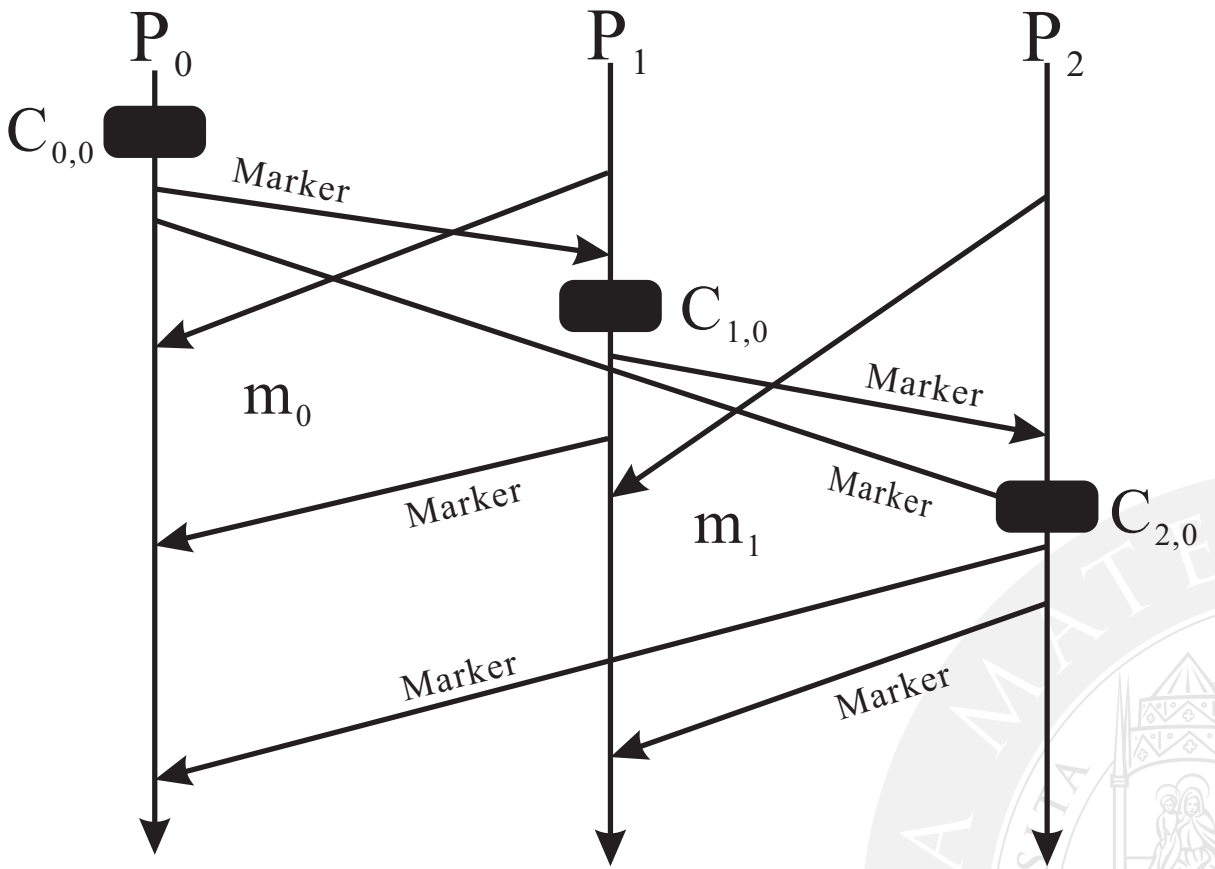
## Protocol description

- processes can be in either *Normal* or *Checkpointing* state
- any process can initiate global checkpointing, taking a local checkpoint and sending a Marker message to all outgoing channels—state to Checkpointing
- upon Marker message reception, a process moves from Normal to Checkpointing state, does the same as above plus records the Marker Certificate—which needs to be completed
- when the Marker Certificate is completed, a process goes back into Normal state

# Chandy and Lamport Distributed Snapshot Protocol IV




System Topology




[Zhao, 2014]

# Checkpointing Protocols: Discussion I

## Analogies

- both protocols rely on basically the same system model, and use a special control message to propagate and coordinate the global checkpointing
- they both recognise the need to capture the channel state to ensure the recoverability of the system
- the mechanism to capture the channel state is virtually the same for both protocols
- the communication overhead of the two protocols is identical

# Checkpointing Protocols: Discussion II

## Differences

- the blocking protocol is more conservative—but, yeah, blocking
- the blocking protocol is more complete and robust
- the non-blocking protocol provides no mechanism for the atomicity of the global checkpointing round
- the blocking protocol leverages on a coordinated checkpointing protocol, whereas the non-blocking one promotes autonomy of the protocol initiation—which may fit differently diverse sorts of distributed systems

# Next in Line...

1 Interaction, Dependencies, Causality, Time

2 Basic Techniques for Dependability

3 Models

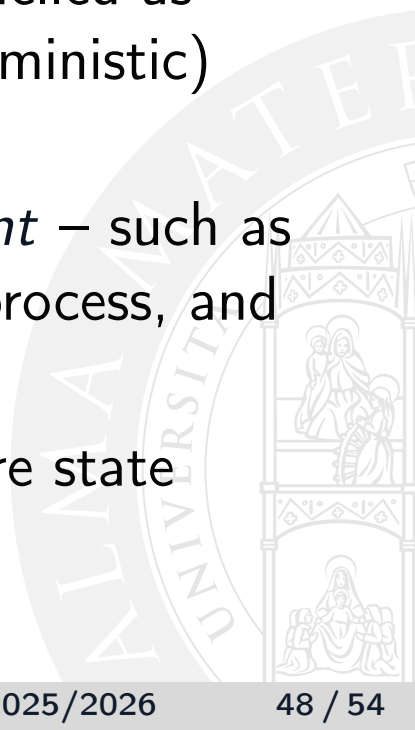
4 Checkpoint-Based Protocols

5 Log-Based Protocols

6 Conclusion

# Generality

- checkpoint-based protocols leverage on saving *state* and recovery from there, at the possible expense of some loss of execution
- logging leverage on saving *events*, which can be used to recover the system to the state right before the failure, provided that the *piecewise deterministic assumption* is valid
- in log based protocols, the execution of a process is modelled as *consecutive state intervals*—intervals between (nondeterministic) events
- each state interval is initiated by a *nondeterministic event* – such as the receiving of a message – or the initialisation of the process, and followed by a sequence of *deterministic state changes*
- as long as the nondeterministic event is logged, the entire state interval can be replayed



# Sorts of Logging Protocols[Alvisi and Marzullo, 1998]

**pessimistic** logging — message received is *synchronously* logged prior to its execution

**optimistic** logging — to reduce the latency overhead, the nondeterministic events are first stored in volatile memory, then logged asynchronously to stable storage

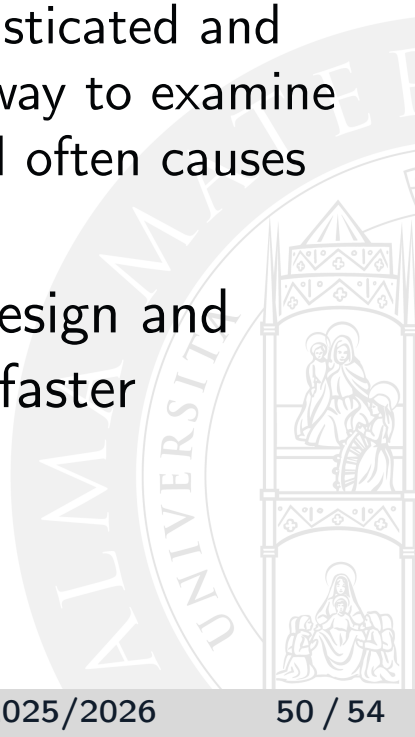
→ the failure of a process might result in permanent loss of some messages, which would force a rollback to a state earlier than the state when the process fails

**causal** logging — the nondeterministic events not yet logged to stable storage are *piggybacked* with each message sent

→ with the piggybacked information, a process can have access all the nondeterministic events that may have causal effects on its state, thereby enabling a consistent recovery of the system upon a failure

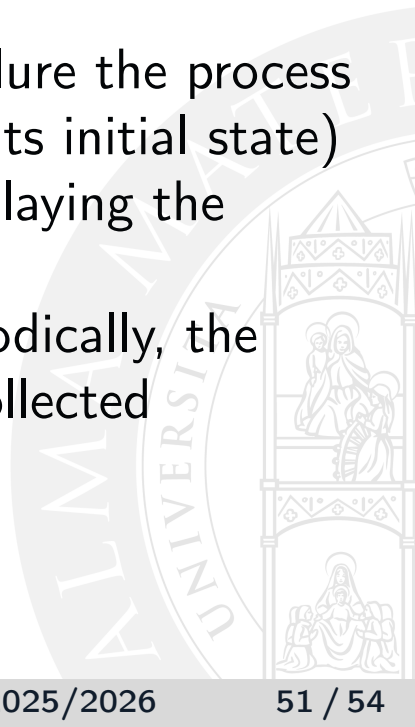
# Long Story Short

- in both optimistic logging and causal logging protocols, the dependency of the processes has to be tracked and sufficient dependency information has to be piggybacked with each message sent
  - this not only increases the complexity of the logging mechanisms, but most importantly, makes the failure recovery more sophisticated and expensive because the recovering process has to find a way to examine its logs and determines if it is missing any messages and often causes cascading recovery operations at other processes
- pessimistic logging protocols are much simpler in their design and implementation and failure recovery can be made much faster



# Logging & Checkpointing

- for all practical purposes, *logging is always used in conjunction with checkpointing*
- so that states are saved as checkpoints, and events are logged from one checkpoint to the next one
- so as to enjoy two benefits
  - limiting the recovery time because to recover from a failure the process can be restarted from its last checkpoint (instead from its initial state) and its state can be recovered prior to the failure by replaying the logged nondeterministic events
  - limiting the size of the log: by taking a checkpoint periodically, the logged events prior to the checkpoint can be garbage collected



# Next in Line...


1. Interaction, Dependencies, Causality, Time
2. Basic Techniques for Dependability
3. Models
4. Checkpoint-Based Protocols
5. Log-Based Protocols
6. Conclusion

# Lessons Learnt

- dependability requires a coherent notion of dependencies and causality
- simple dependability techniques such as checkpointing and logging can be used to improve the chance of recovery of distributed systems in case of failures

# Logging & Checkpointing
# Towards Recovery of Distributed Systems

## Distributed Systems / Case Study

Andrea Omicini

mailto:andrea.omicini@unibo.it
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Academic Year 2025/2026

# References

[Alvisi and Marzullo, 1998] Alvisi, L. and Marzullo, K. (1998).
Message logging: pessimistic, optimistic, causal, and optimal.
*IEEE Transactions on Software Engineering*, 24(2):149–159
DOI:10.1109/32.666828

[Chandy and Lamport, 1985] Chandy, K. M. and Lamport, L. (1985).
Distributed snapshots: determining global states of distributed systems.
*ACM Transactions on Computer Systems*, 3(1):63–75
DOI:10.1145/214451.214456

[Pearl, 2009] Pearl, J. (2009).
*Causality: Models, Reasoning, and Inference, 2nd Edition*.
Cambridge University Press
DOI:10.1017/CBO9780511803161

[Tamir and Sequin, 1984] Tamir, Y. and Sequin, C. H. (1984).
Error recovery in multicomputers using global checkpoints.
In *13th International Conference on Parallel Processing (ICPP '84)*, pages 32–41, Bellaire, MI, USA

[Zhao, 2014] Zhao, W. (2014).
*Building Dependable Distributed Systems*.
Wiley
DOI:10.1002/9781118912744