Determining the cut

To have a valid application checkpoint we need at least consistent cut of checkpoints

How to make the cut?

Coordinated approach: Use marker messages to indicate that a checkpoint is being taken

Uncoordinated approach: Attempt to form a consistent global cut at recovery time

- Domino effect must be overcome

Coordinated assumptions

Here we'll make similar assumptions to [Koo and Toueg 1987]

- Processes communicate by exchanging messages through communication channels
- Channels are FIFO
- Communication failures do not partition the network
- A single process invokes the algorithm
- The checkpoint and the rollback recovery algorithms are not invoked concurrently

Blocking coordinated checkpointing

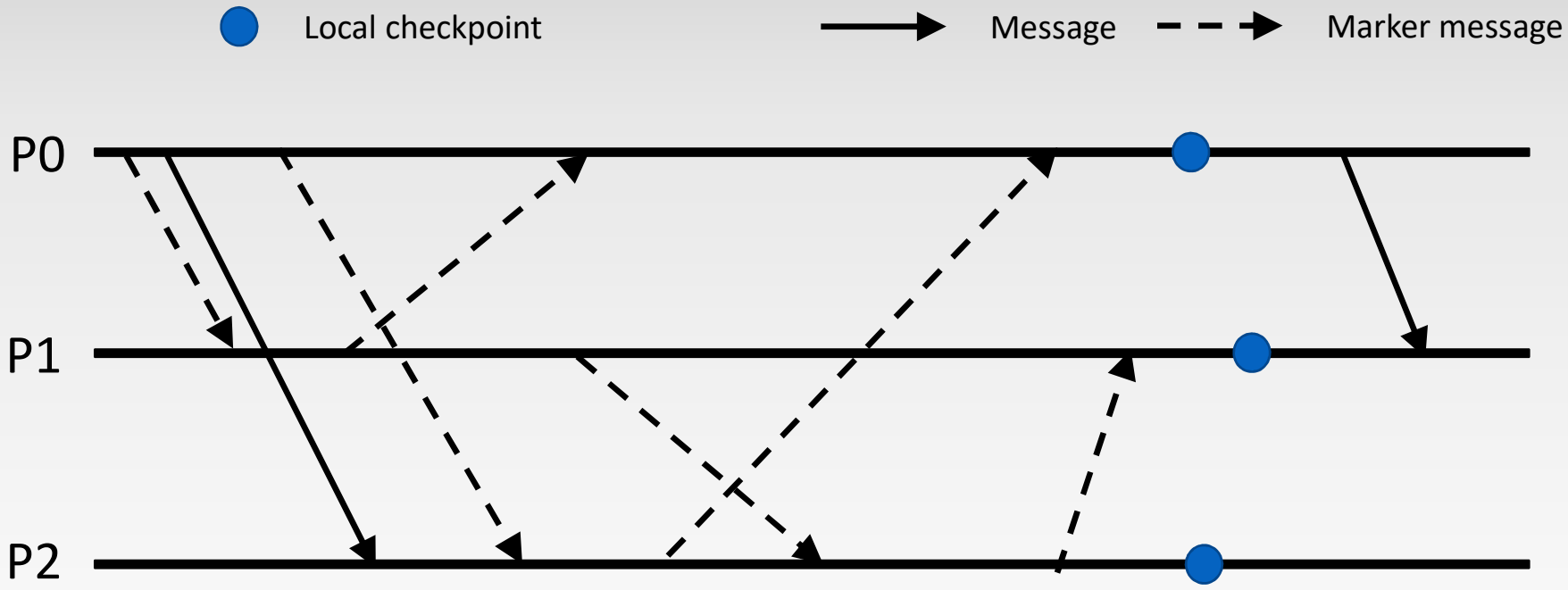
Marker message is an All-to-all operation

After receiving the first marker message the communication channel is silenced until the checkpoint is finished

Checkpoint taken after all marker messages are received

- Marker message from P_i means no more messages from P_i

Example – blocking



Why is there a message sent to P2 after P1 gets a marker message?

Non-blocking coordinated checkpointing

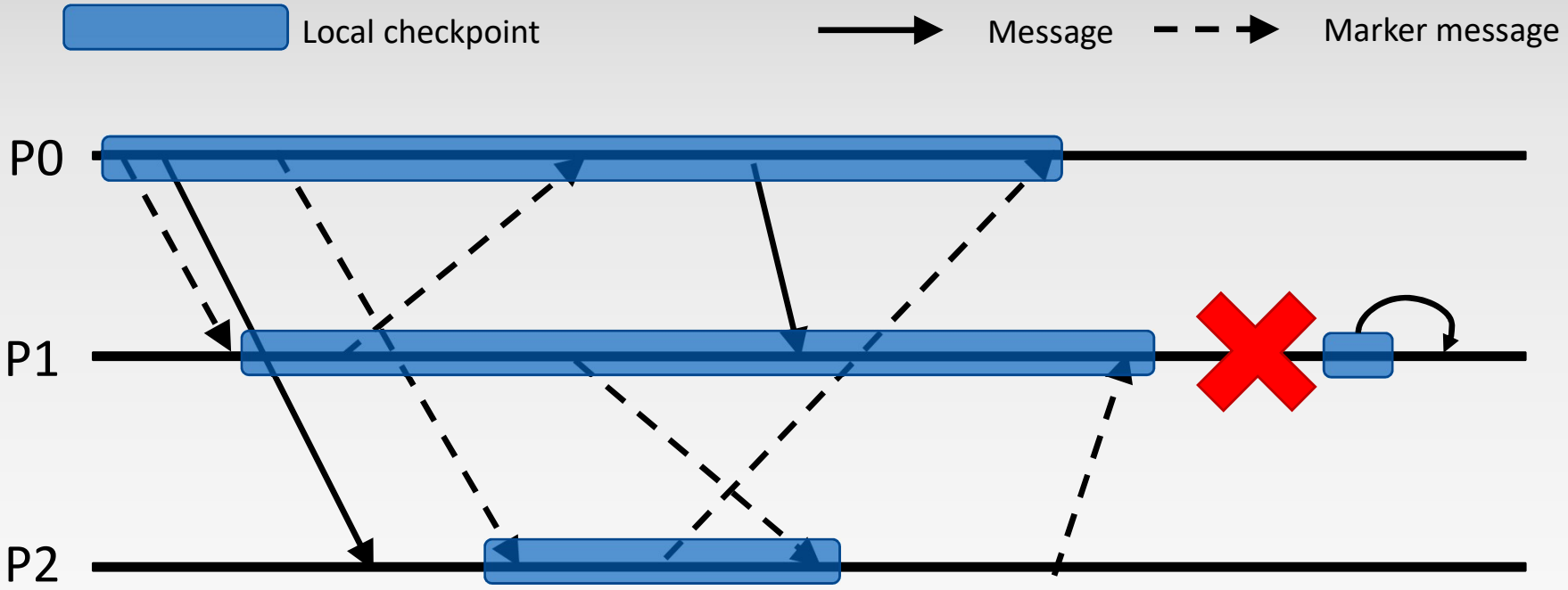
Marker message is an All-to-all operation

After receiving the first marker message

- Checkpoint local state
- Log all incoming messages into the checkpoint until checkpoint is complete
  - Received all marker messages

Checkpoint finishes when all marker messages are received

Example – nonblocking



Message from P0 to P1 stores in P1's checkpoint (receiver logging)

Disadvantages of coordinated checkpointing

What are some limitations of coordinated checkpointing?

- Marker messages must be exchanged to coordinate checkpointing
- Marker messages are a form of synchronization
- No application messages until the checkpoint is complete
- If failures are rare, overhead can impact performance

How to make improve coordinated checkpointing?

Uncoordinated checkpointing

Each process P_i takes a checkpoint without coordination

- No marker messages
- Checkpointing is faster

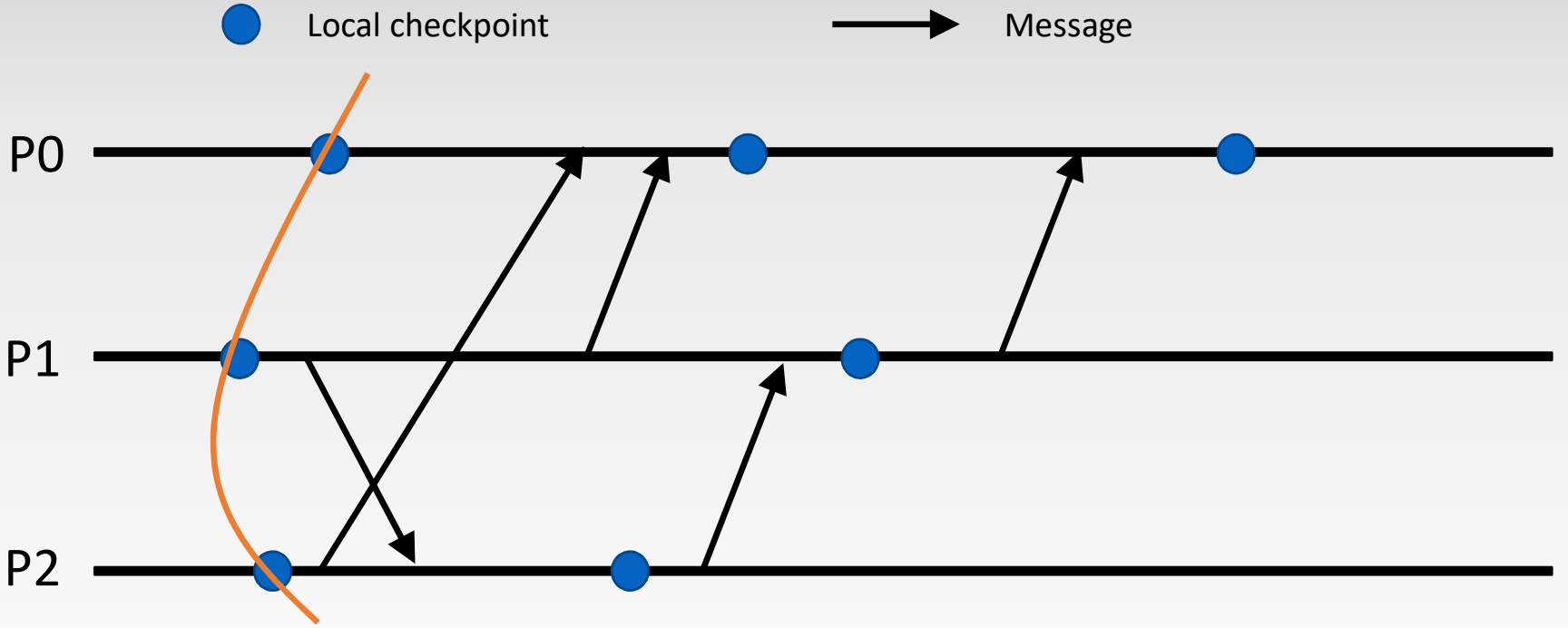
Challenge in constructing a consistent cut with the checkpoints

- Consistent cuts may require handling of missing messages by logging each message
- Strongly consistent cuts require no special handling

Need to keep older checkpoints to get consistent cut

Uncoordinated checkpointing may suffer domino effect

Domino effect



Rollback of some process forces rollback of other processes beyond the most recent checkpoint

Logging messages

Logging incoming messages on each process helps minimize the amount of computation during a restart

Pessimistic:

- Log incoming message before it is processed
  - Slows down computation

Optimistic:

- Processes does not stop computing
- Incoming messages are stored in volatile storage and logged at certain intervals
  - Messages that have yet to be logged to stable storage are lost if the process fails
  - Does not slow down computation

Optimistic message logging

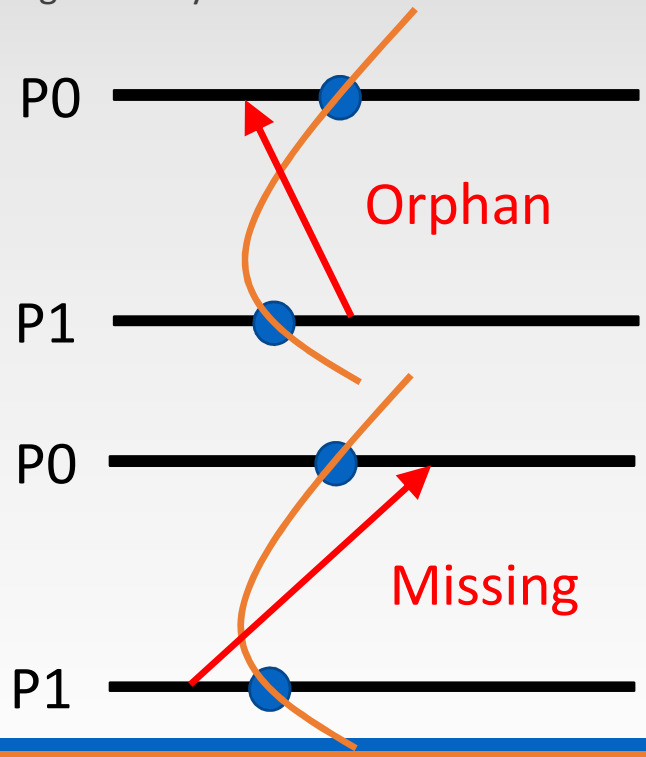
Messages that don't reside in non-volatile storage are not available during recovery

Other process state may causally depend on lost messages

- Creates orphan messages
- Creates missing messages

Orphan and missing messages are determined by tracking event state

- Process consists of sequence of events
- Receipt of message starts a new event
- Outgoing messages dependent upon current event state of a process



Tracking dependencies

Each process keeps a dependency vector with one entry per process

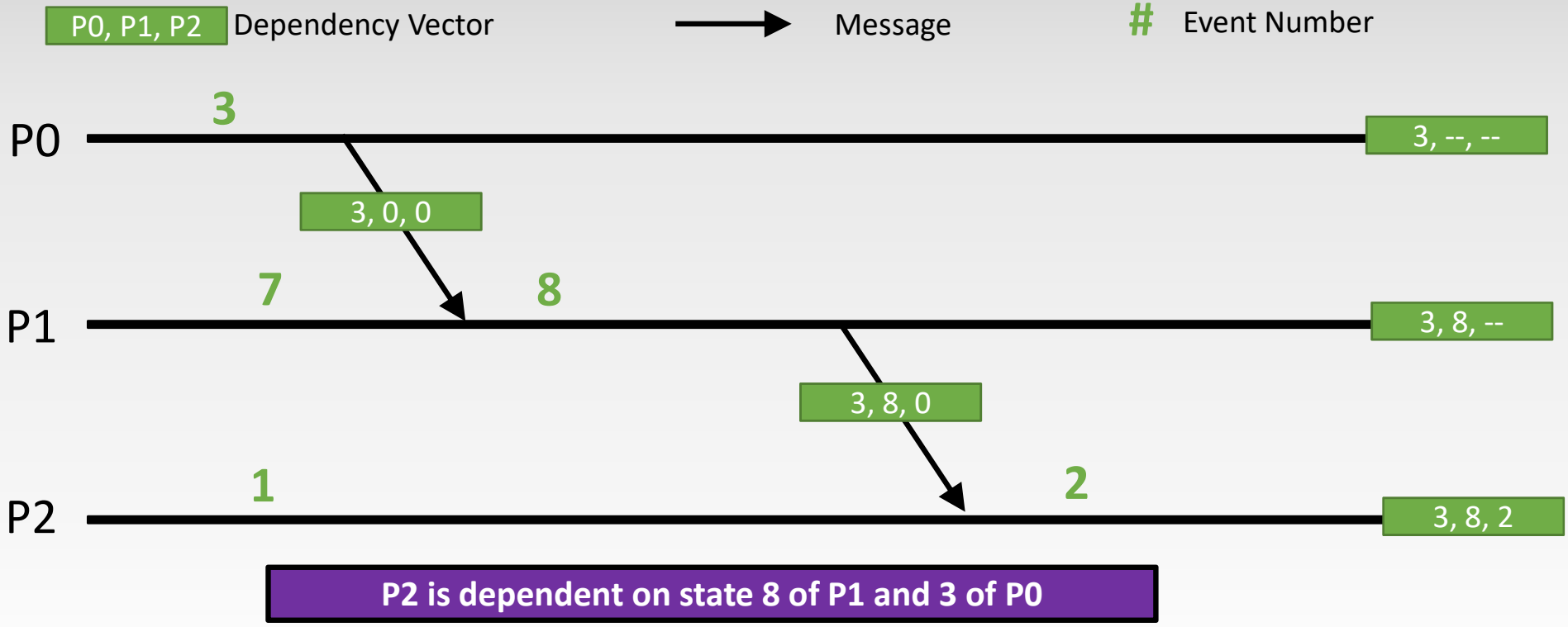
- v[j] denotes the most recent state event of P_j that this process depends on

Dependency vector piggybacked on outgoing messages

Receivers update their own dependency vector from piggybacked vector

Causal dependencies propagated through piggybacked vector

Dependency example

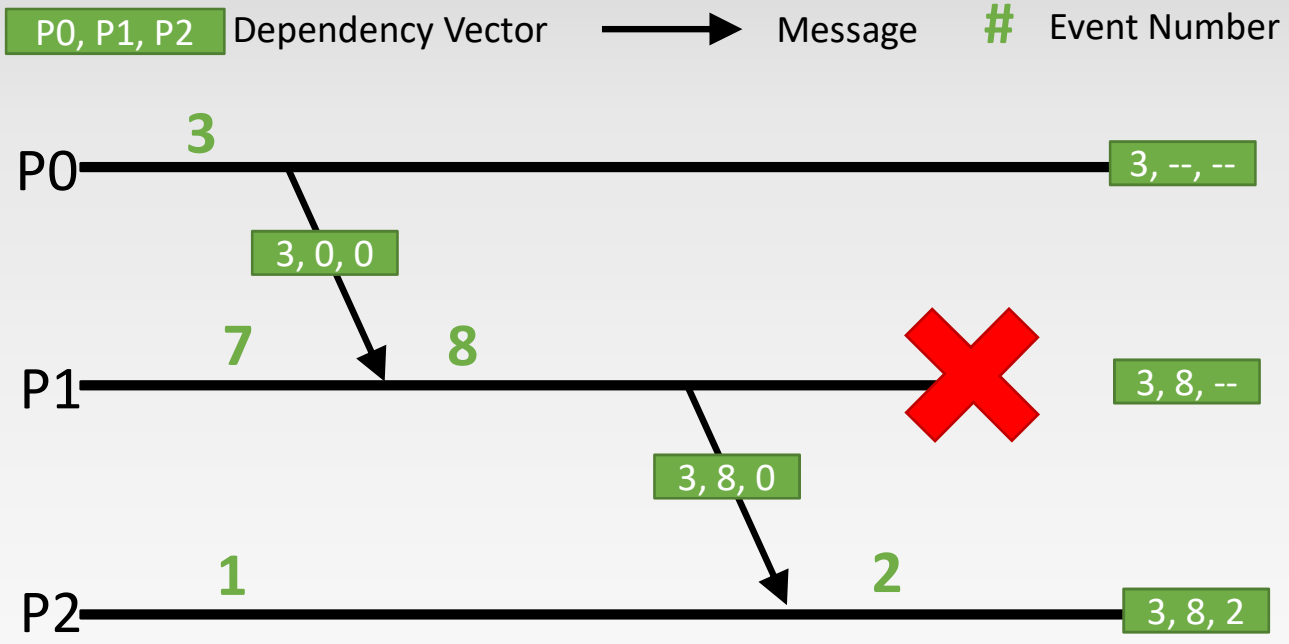


Recovery

If P1 has not logged message from P0, then state **8** is not valid

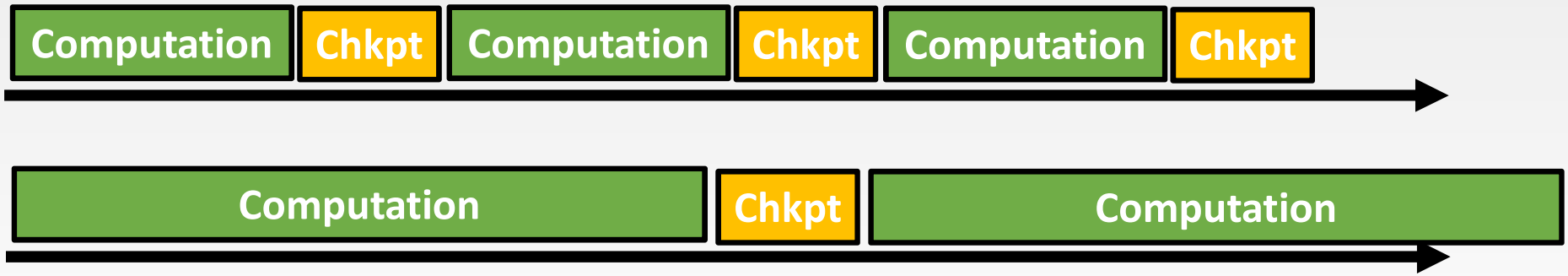
Restore from state 7

Any process where v[1] >7 must rollback

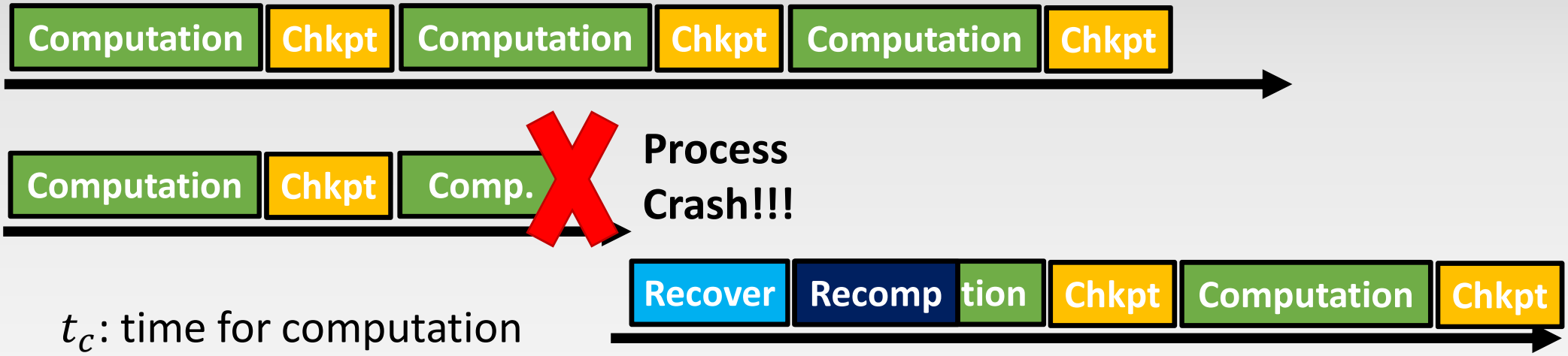


Optimal checkpointing period

How often should we take a checkpoint?



Optimal checkpointing period



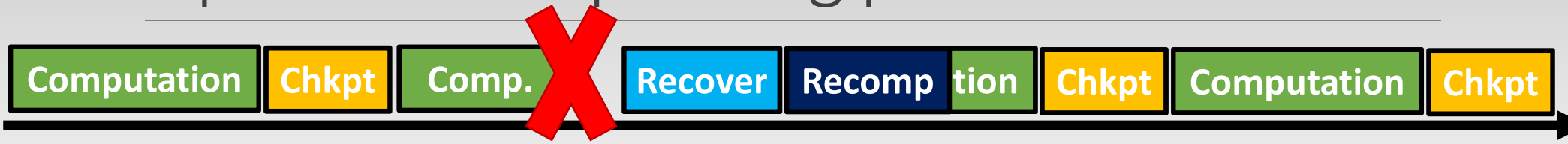
t_c: time for computation

t_s: time for checkpoint

t_r: time for recover

t_l: time for recompute (lost)

Optimal checkpointing period



t_c: time for computation
t_l: time for recompute (lost)

t_r: time for recover
t_s: time for checkpoint

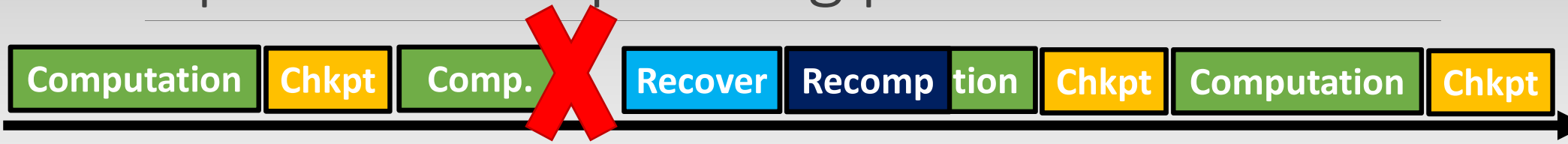
$T_{FF} = N(t_c + t_s)$: Failure free execution

$T_{Failure} = N(t_c + t_s) + t_r + t_l$: Failure execution

Since all checkpoints prior to the failure are not used they contribute waste

$$t_w = n(t_s) + t_l$$

Optimal checkpointing period



t_c: time for computation
t_l: time for recompute (lost)

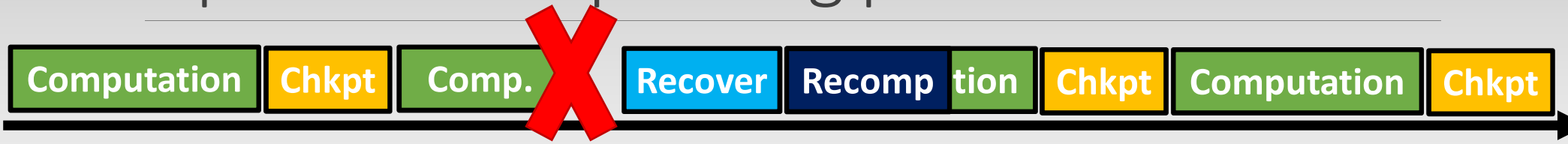
t_r: time for recover
t_s: time for checkpoint

$t_w = n(t_s) + t_l$

Assume that failures are from a Poisson distributed with failure rate λ:

$T_w = \sum_{n=0}^{\infty} \int_{n(t_c+t_s)}^{(n+1)(t_c+t_s)} [t - nt_c]\left(\lambda e^{-\lambda t}\right) dt$: Total waste (failure + prior)

Optimal checkpointing period



t_c: time for computation
t_l: time for recompute (lost)

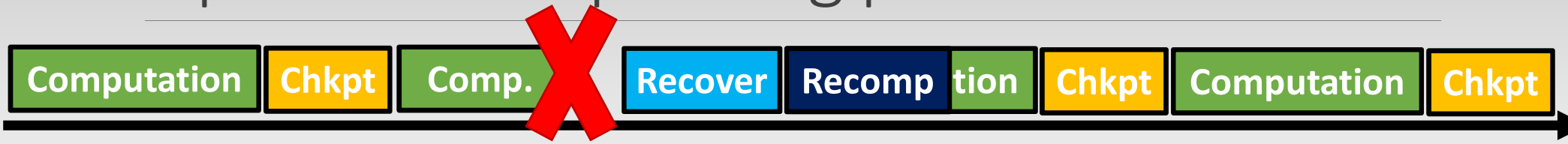
t_r: time for recover
t_s: time for checkpoint

$t_w = n(t_s) + t_l$

Integrate and simplify yields:

$$T_w = \frac{1}{\lambda} + \frac{t_c}{1 - e^{-\lambda(t_c + t_s)}}$$

Optimal checkpointing period



t_c: time for computation
t_l: time for recompute (lost)

t_r: time for recover
t_s: time for checkpoint

$t_w = n(t_s) + t_l$

Differentiating with respect to t_c , set $T'_w = 0$, and solve for t_c :

$$t_c = \sqrt{\frac{2t_s}{\lambda}}$$

Valid only when $t_s \ll \lambda$

Summary

Discussed merits and draw backs of system and application based checkpointing

Explored coordinated and uncoordinated checkpointing

Improved uncoordinated checkpointing by using message logging and dependency vectors

Derived a formula to compute the optimal checkpointing formula

References

ECE 542 University of Illinois Lecture Slides
- https://courses.engr.illinois.edu/ece542/sp2015/

CS 425 University of Illinois Lecture Slides
- https://courses.engr.illinois.edu/cs425/fa2016/

“Fault-tolerant Techniques for HPC: Theory and Practice” - George Bosilca, Aurelien Bouteiller, Thomas Herault, and Yves Robert

http://fault-tolerance.org/downloads/sc14tutorial.pdf

Schadenfreude
- https://www.informationweek.com/cloud/7-data-center-disasters-youll-never-see-coming/d/d-id/1320702?image_number=5