ECE 6930-004
HPC Fault Tolerance

BASICS OF CHECKPOINT RESTART

DR. JON CALHOUN

Schadenfreude! Late October 2012

Peer 1 hosting located in lower Manhattan had backup generators located on the 18th floor

Hurricane Sandy came ashore flooding the building's lobby and basement

Unfortunately, the emergency generator fuel pumping system was located in the basement

Due to post 9/11 restrictions sufficient supplies of fuel were not kept on site

Schadenfreude! Late October 2012

To prevent downtime, a bucket brigade was formed to carry fuel to the 17th floor where the fuel tanks were located

Peer 1 engineers and customers worked tirelessly to fill the fuel tanks

No downtime was reported

Oddly enough bucket brigade was not part of the Peer 1 disaster recovery plan

We NEED volunteers

Date	Paper/Topic	Presenter
8/23	Introduction/Syllabus/What is HPC	Calhoun
8/28	Basic Fault Tolerance Concepts	Calhoun
8/30	Modeling Reliability / Basic Failure Detection	Calhoun
9/4	Toward Exascale Resilience	Calhoun
9/6	Lessons Learned From the Analysis of System Failures at Petascale: The Case of Blue Waters	Omar
9/11	Basics of Checkpoint-restart	Calhoun
9/13	Evaluation of Simple Causal Message Logging for Large-Scale Fault Tolerant HPC Systems	
9/28	Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System	
9/20	MCRENGINE: A Scalable Checkpointing System Using Data-Aware Aggregation and Compression	
9/25	What is a soft error?	Calhoun

When an application fails

How to detect failures?

Heart beats

Detect + Recover =

Fail-stop failures often result in the crash of the application

Without some mechanism to save and restart the application during its execution, it must restart from the beginning

Methods of recovery

How to recover from fail-stop failures?

Redundancy | **Replication** | **Checkpointing**

Due to resource overheads HPC primarily leverages checkpointing to recover applications

Checkpoint-restart

Let’s lets explore how checkpoint-restart and variations on it are used in HPC

What should we checkpoint?

Checkpointing can be used at many different levels (e.g., instruction, function, application)

Here we will look at checkpointing the application

System level

Serializes the entire process's image
- Memory allocations
- Processor registers
- File handles

Can be done by:
- OS (BLCR)
- Compiler (C3)
- External library (libckpt)

System level checkpointing requires little modifications to source code
- Can use preemption

System level

What is there not to like about system level checkpointing?

Although it requires minimal modifications to the application, system level checkpoint has several key drawbacks:

- Not portable
- Job not malleable
- Size of checkpoint is large

Let's leverage user-level knowledge to address these issues

Application level

The user serializes the state of the process

PROS

Smaller checkpoint sizes

Portability

Custom file formats

May support job malleability

CONS

Difficult to implement if preemption is required

- Random location in call graph
- Keep all the local temporary variables

More work on the programmer

When to checkpoint?

Determining checkpointing time can be based on

- Elapsed time
- Number of messages received or sent count

Synchronous/Blocking:

- All computation stops and all pending communications must finish
- Application is fully serialized before computation and communication can resume

Asynchronous:

- Application does not need to halt when it is time to checkpoint
- System level can use copy-on-write with new process
- Application level saves state around key locations

Where to checkpoint?

The checkpoint must survive the failure of system components; therefore, it must reside in non-volatile storage

The need for true persistence does not preclude the use of volatile memories

- RAM
- Neighbor nodes

The checkpoints taken in volatile memories are periodically written to non-volatile storages

What does checkpointing look like in practice?

To understand what happens when checkpointing, let’s look at the interactions of 3 processes though time

On restart, computation halts, rolls back to the previous checkpoint, and computation resumes

- All processes or just failed?

Initially let’s look at the case when every process must restart

- Mitigate risks of the domino effect, missing messages, orphan messages

Checkpointing notation

Each process P_i has local state denoted $LS_{i,k}$ at some time k

Let:

$send(m_{ij})$ be a send event of a message m_{ij} by P_i to P_j

$\mathrm{recv}(m_{ij})$ be a receive event of message m_{ij} by P_j

$time(x)$ be the time that event x occurs

Furthermore,

$$send(m_{ij}) \in LS_{i,k} \ \boldsymbol{iff} \ time\left(send(m_{ij})\right) < time(LS_{i,k})$$

$$recv(m_{ij}) \notin LS_{j,k} \ \boldsymbol{iff} \ time\left(recv(m_{ij})\right) < time(LS_{j,k})$$

Checkpointing notation

For processes P_i and P_j let's define two message sets to denote a message's status

Transit:

$$transit(LS_{i,k}, LS_{i,k}) = \{m_{ij} \mid send(m_{ij}) \in LS_{i,k} \wedge recv(m_{ij}) \notin LS_{j,k}\}$$

- Message is in flight

Inconsistent:

$$inconsistent(LS_{i,k}, LS_{i,k}) = \{m_{ij} \mid send(m_{ij}) \notin LS_{i,k} \wedge recv(m_{ij}) \in LS_{j,k}\}$$

Checkpointing notation

Let the global state of the application (p processes) be defined as

$GS = \{LS_0, LS_1, \ldots LS_p,\}$

The global state is said to be consistent iff

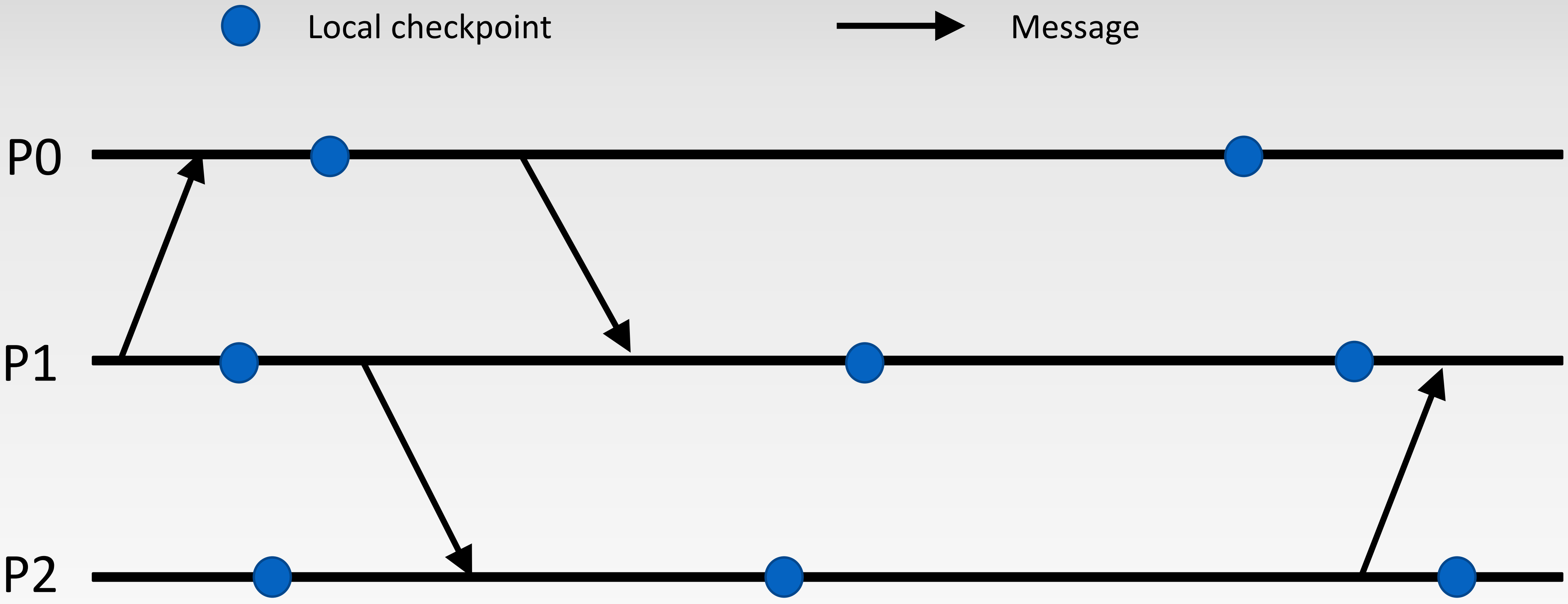
$$\forall\, i, \forall\, j : 0 \leq i, j \leq p \;\; :: inconsistent\left(LS_i,\ LS_j\right) = \emptyset$$

The global state is said to be transitless iff

$$\forall\, i, \forall\, j : 0 \leq i, j \leq p \;\; :: transit\left(LS_i,\ LS_j\right) = \emptyset$$

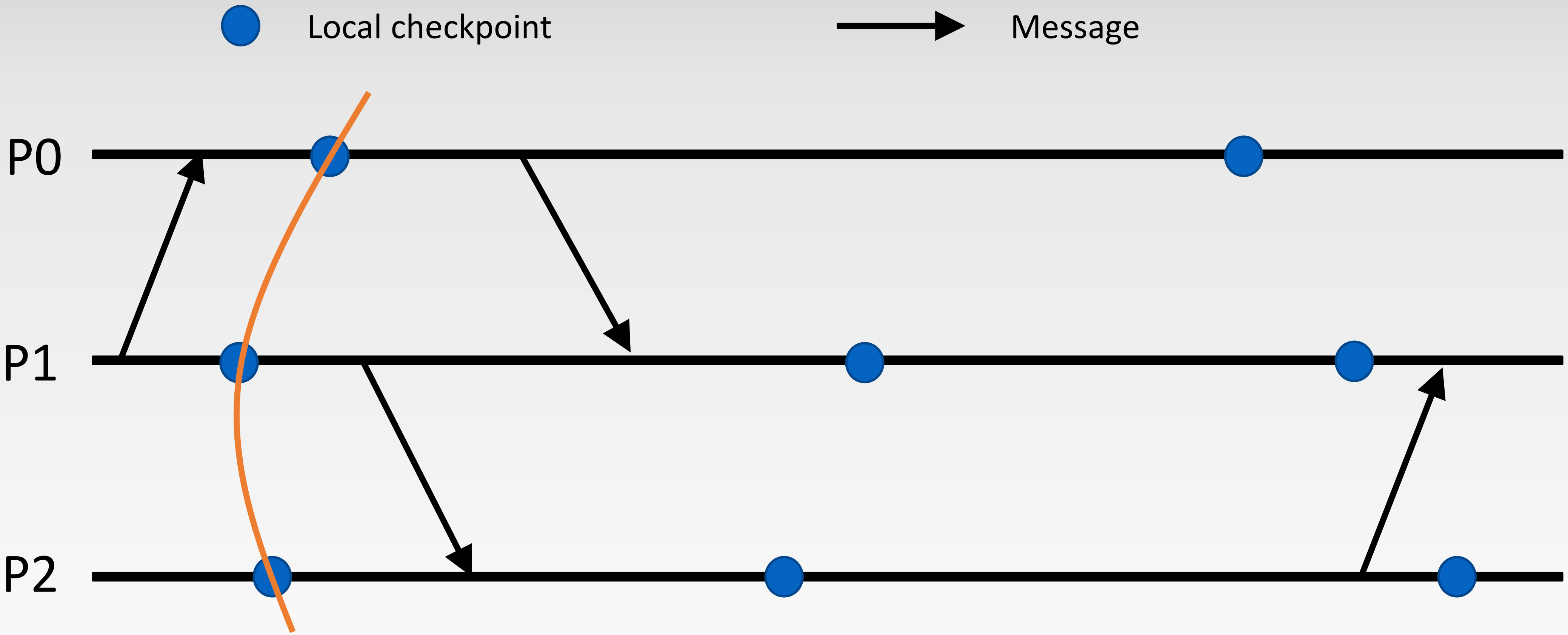
The global state is strongly consistent if it is both consistent and transitless

Example



What global state make sense?

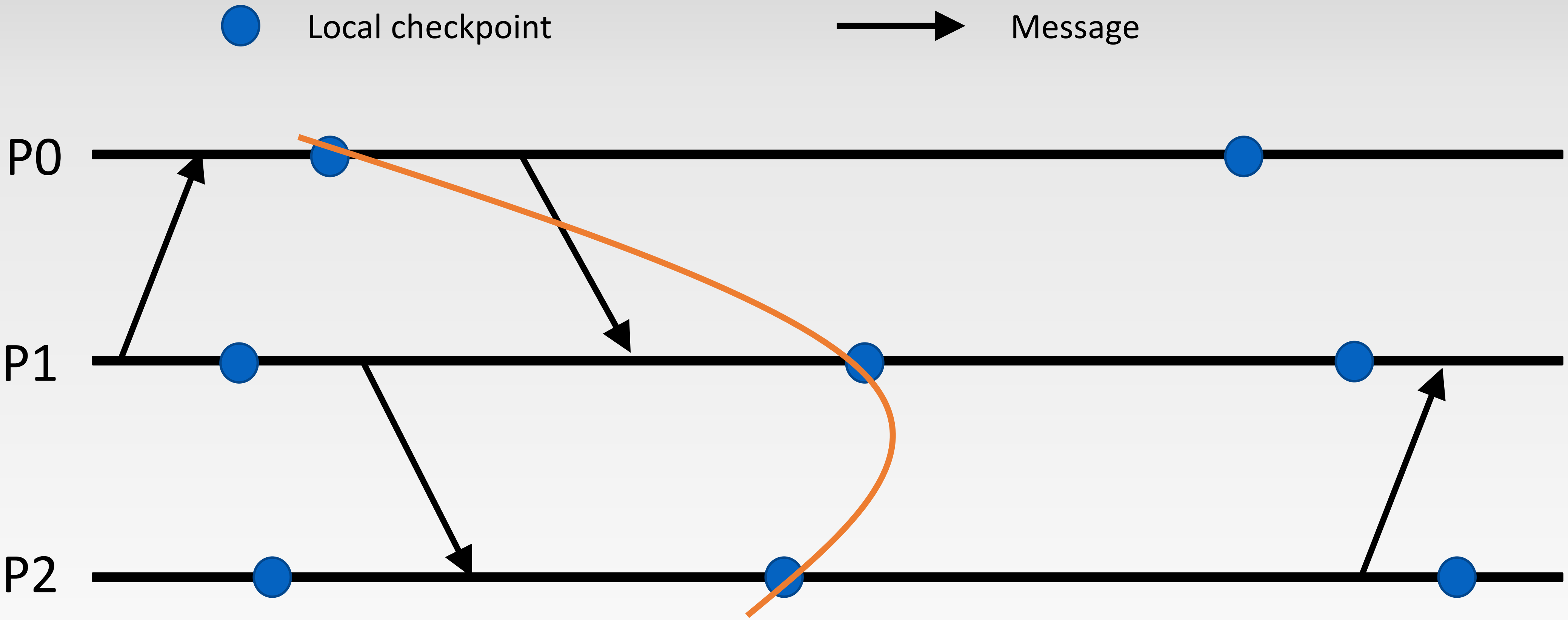
Example



$GS = \{LS_{0,0}, LS_{1,0}, LS_{2,0}\}$

Strongly Consistent

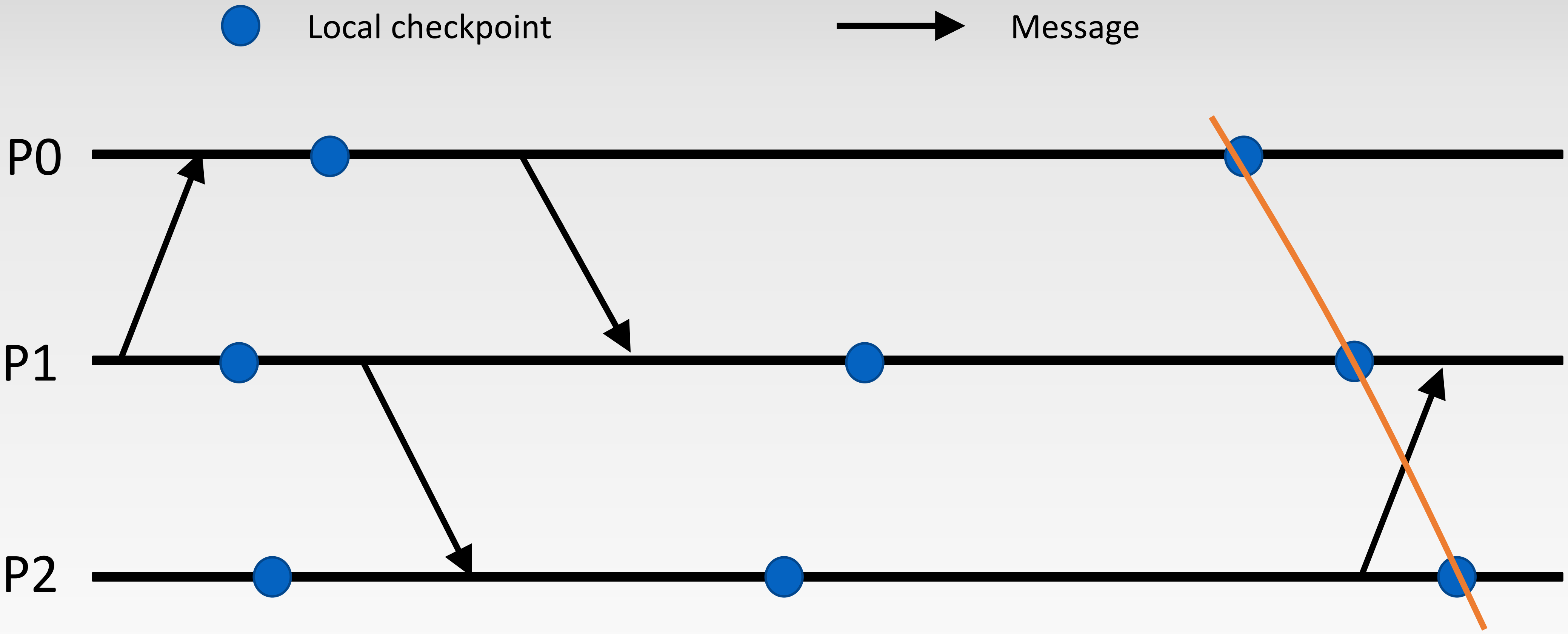
Example



$$GS = \{LS_{0,0}, LS_{1,1}, LS_{2,1}\}$$

Inconsistent

Example



$$GS = \{LS_{0,1}, LS_{1,2}, LS_{2,2}\}$$

Consistent

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

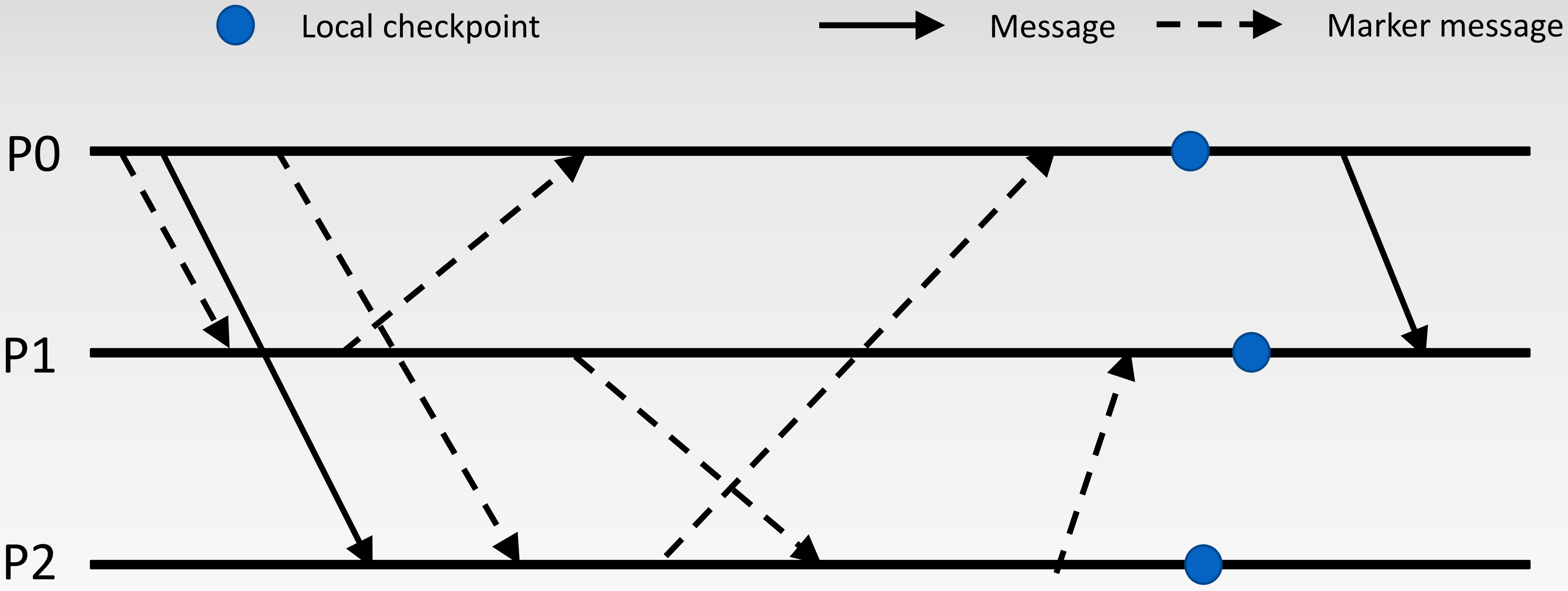
Marker message is an All-to-all operation

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

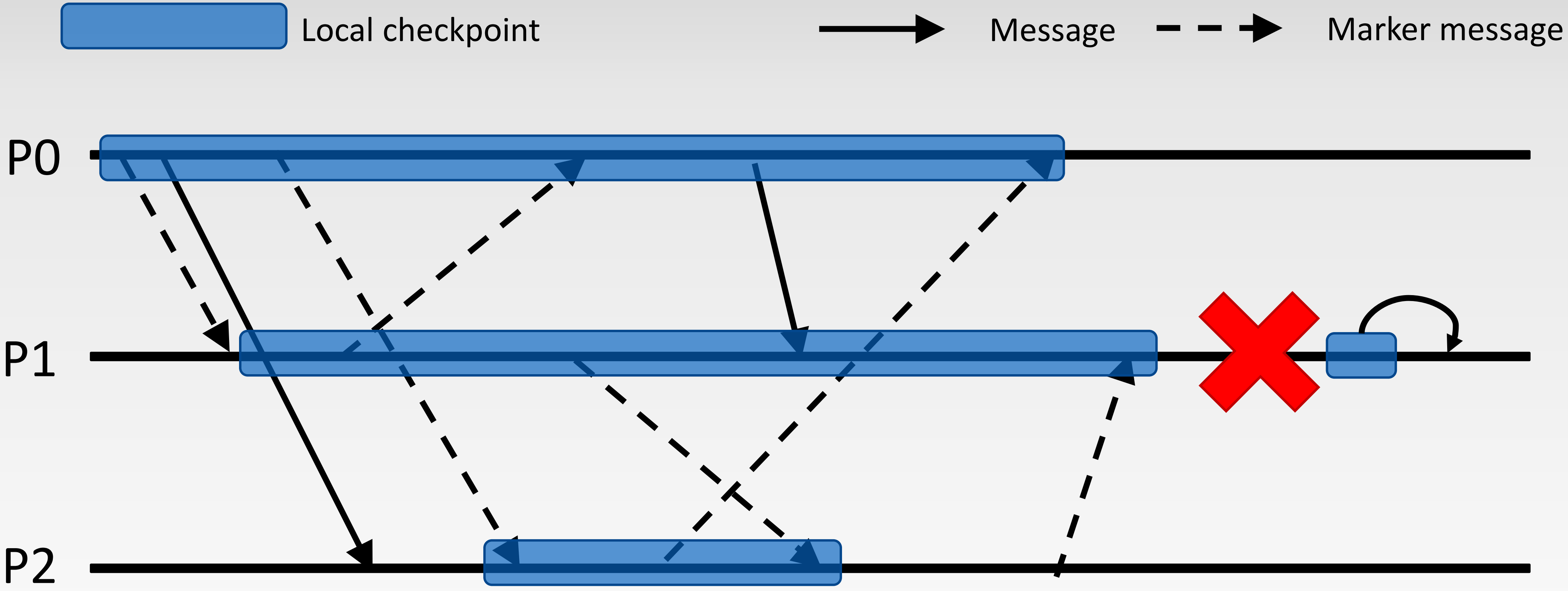
Marker message is an All-to-all operation

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

Uncoordinated checkpointing

Each process P_i takes a checkpoint without coordination

- No marker messages
- Checkpointing is faster

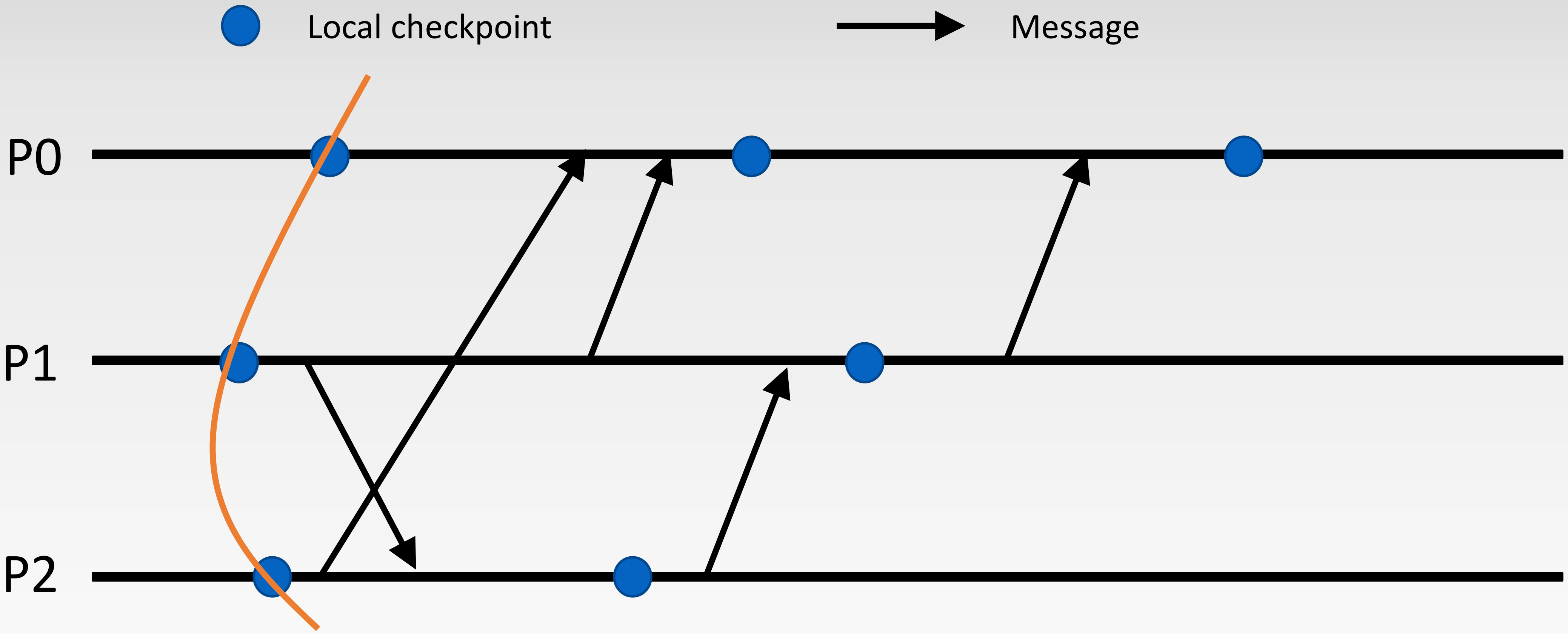
Challenge in constructing a consistent cut with the checkpoints

- 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

- 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:

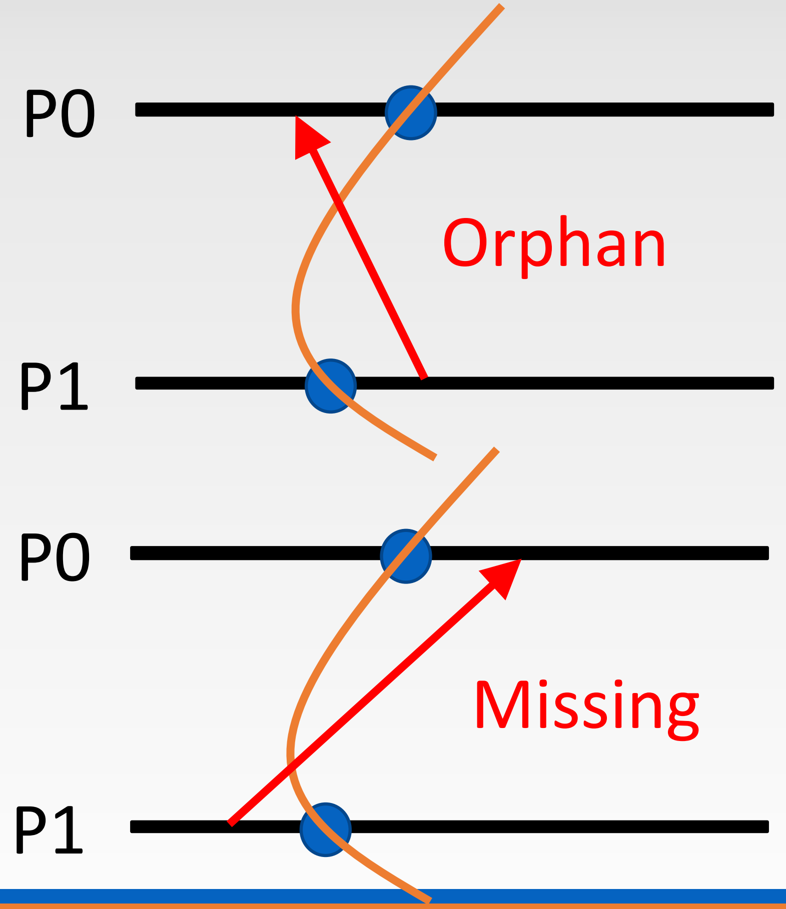
- Processors does not stop computing
- Incoming messages are stored in volatile storage and logged at certain intervals
  - Messages that have yet to be logged to stable storage are lost if the process fails
  - Does not slow down computation

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

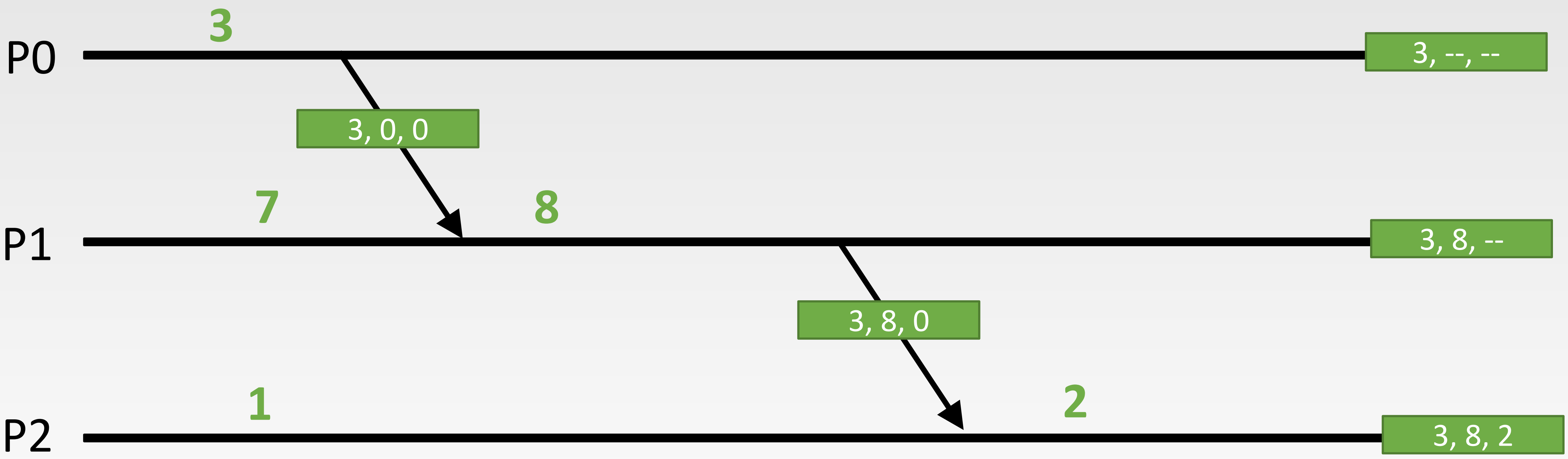
Dependency vector piggybacked on outgoing messages

Receivers update their own dependency vector from piggybacked vector

Causal dependencies propagated through piggybacked vector

Dependency example

P0, P1, P2 Dependency Vector — Message # Event Number



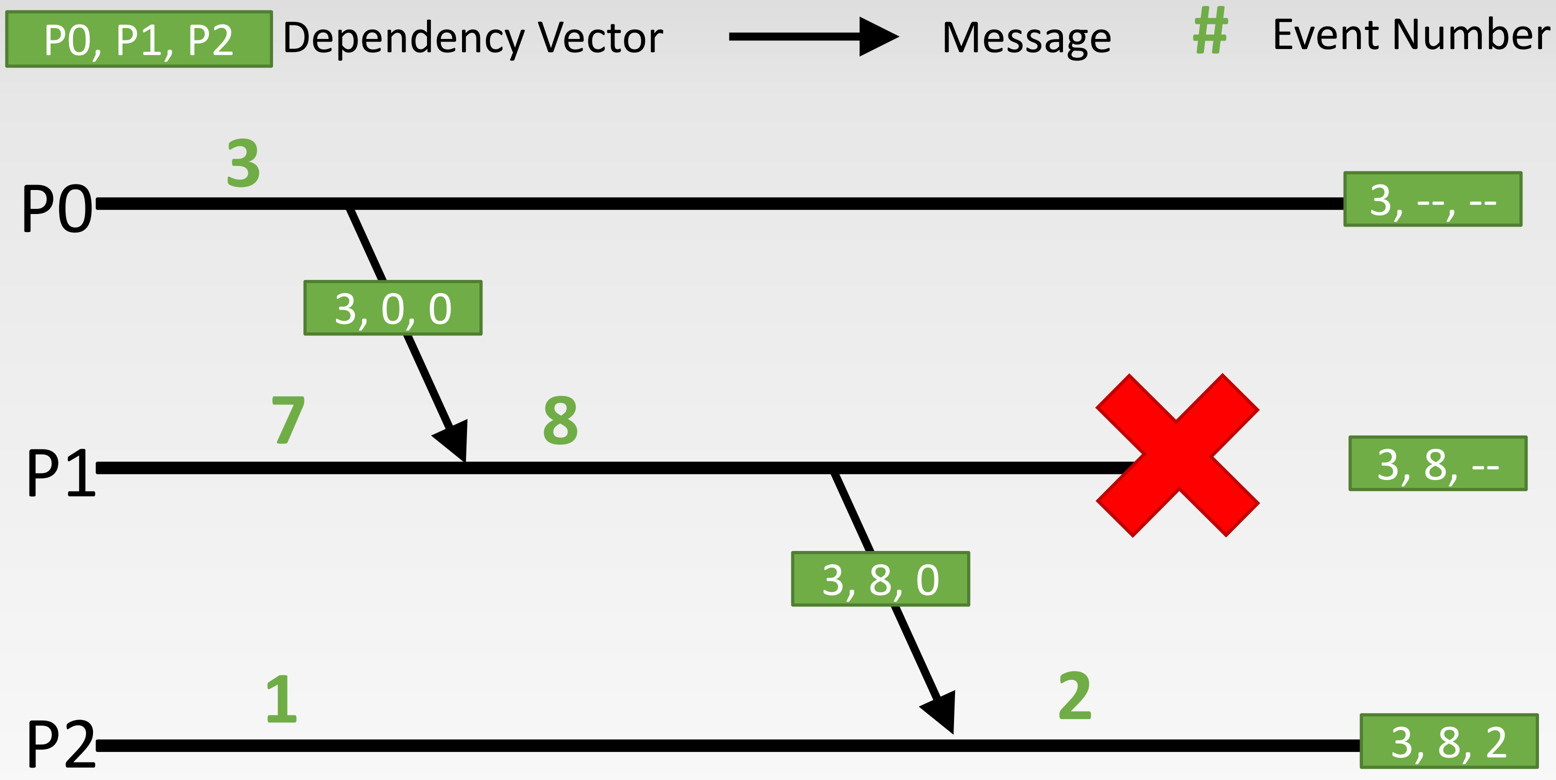
P2 is dependent on state 8 of P1 and 3 of P0

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

[Young 1974] derives the optimal checkpointing period in the case that failures occur based on a Poisson process

- Easy 1 page read

$$\tau_{opt} = \lambda\sqrt{\frac{2C}{\lambda}}$$

Where

$\lambda = MTBF$

$\mathrm{C} = Time\ to\ checkpoint$

Valid only when C $\ll \lambda$

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

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