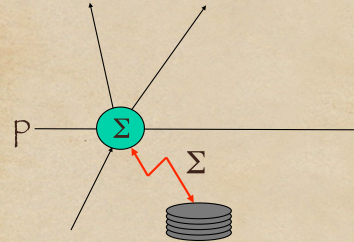
Rollback-Recovery

Uncoordinated Checkpointing



- Easy to understand
- No synchronization overhead
- Flexible
  - can choose when to checkpoint
- To recover from a crash:
  - go back to last checkpoint
  - restart

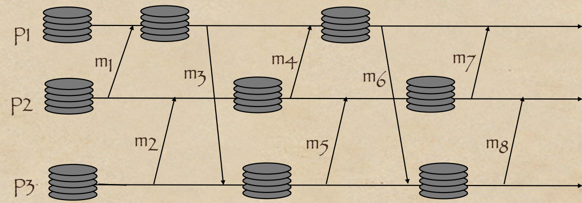
How (not) to take a checkpoint

- Block execution, save entire process state to stable storage
  - very high overhead during failure-free execution
  - lots of unnecessary data saved on stable storage

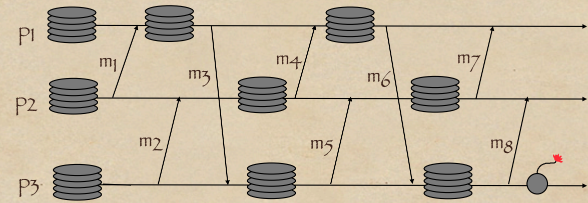
How to take a checkpoint

- Take checkpoints incrementally
  - save only pages modified since last checkpoint
  - use "dirty" bit to determine which pages to save
- Save only "interesting" parts of address space
  - use application hints or compiler help to avoid saving useless data (e.g. dead variables)
- Do not block application execution during recovery
  - copy-on-write

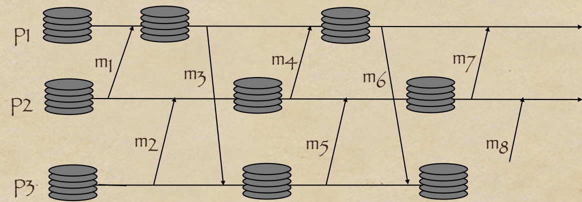
The Domino Effect



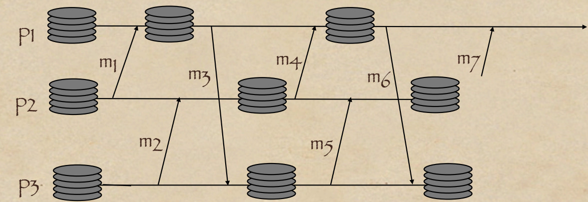
The Domino Effect



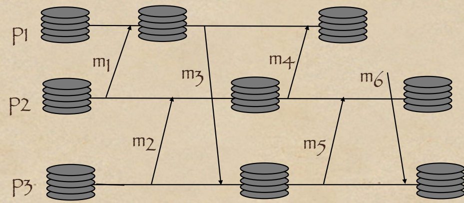
The Domino Effect



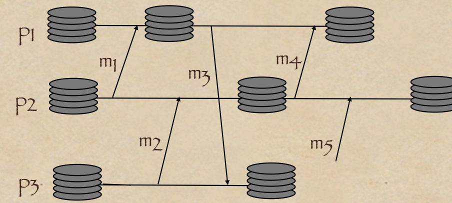
The Domino Effect



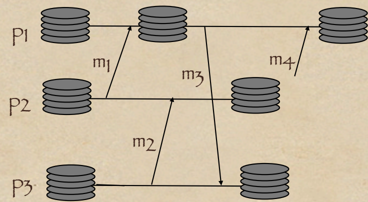
The Domino Effect



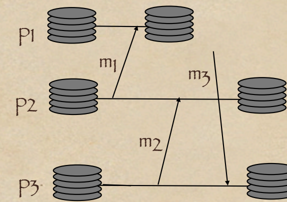
The Domino Effect



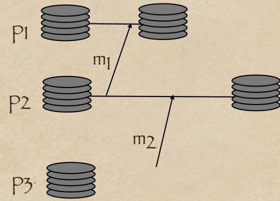
The Domino Effect



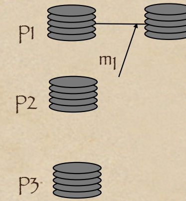
The Domino Effect



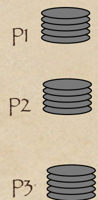
The Domino Effect



The Domino Effect



The Domino Effect



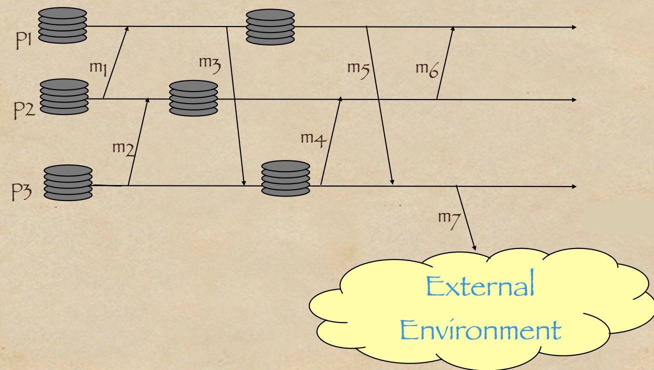
How to Avoid the Domino Effect

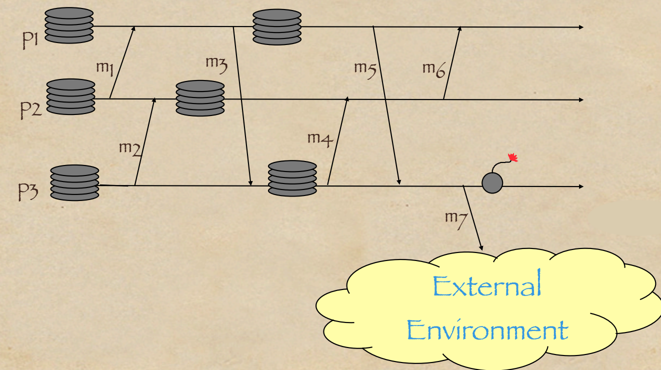
Coordinated Checkpointing

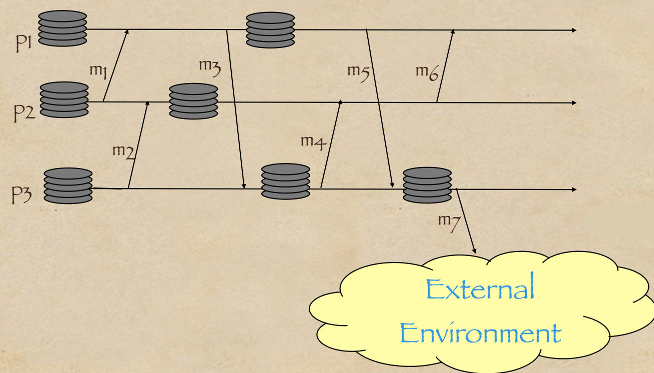
- No independence
- Synchronization Overhead
- Easy Garbage Collection

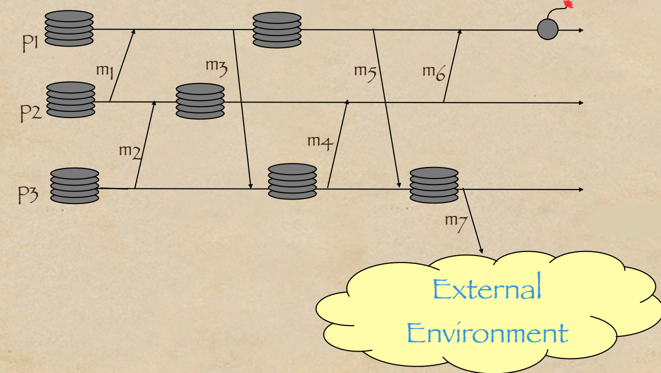
Communication Induced Checkpointing : detect dangerous communication patterns and checkpoint appropriately

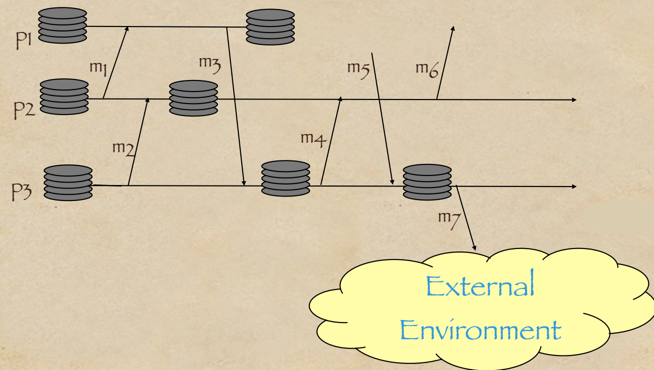
- Less synchronization
- Less independence
- Complex

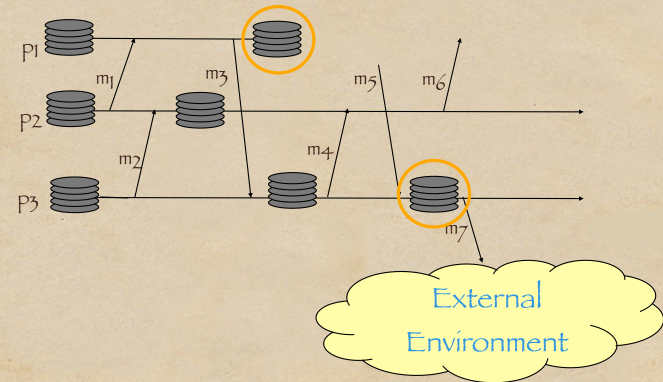

The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment

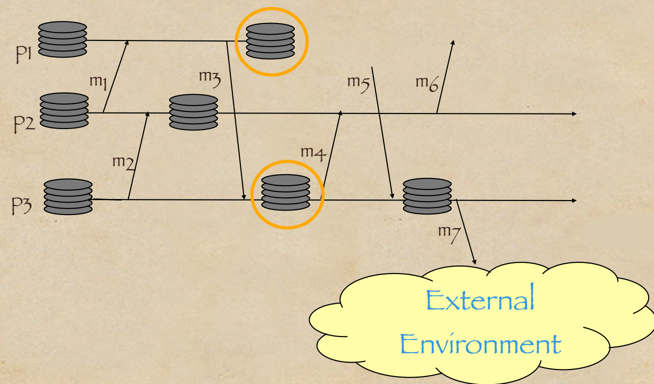

The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment

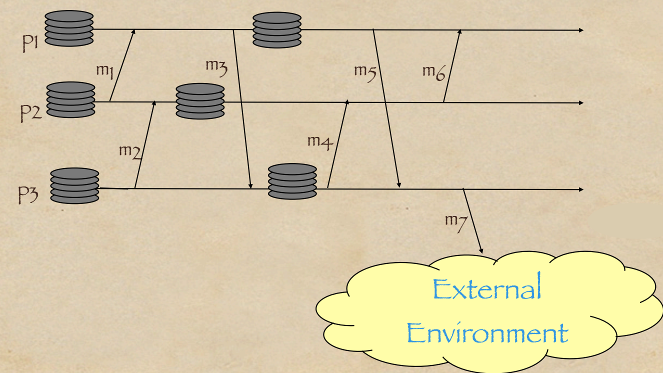

The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment


The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment

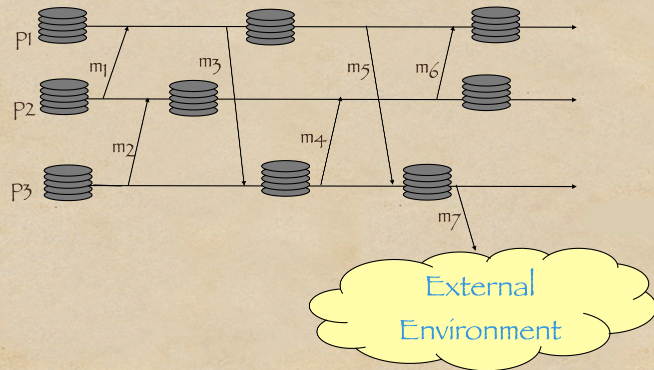

The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment


The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment

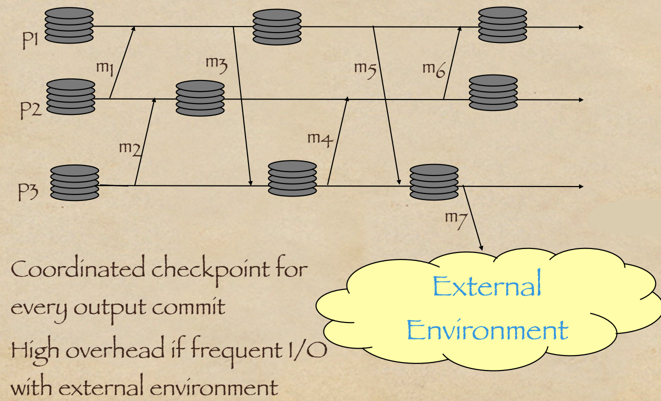

The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment


The Output Commit Problem
p1
p2
p3
m1
m2
m3
m4
m5
m6
m7
External
Environment

The Output Commit Problem



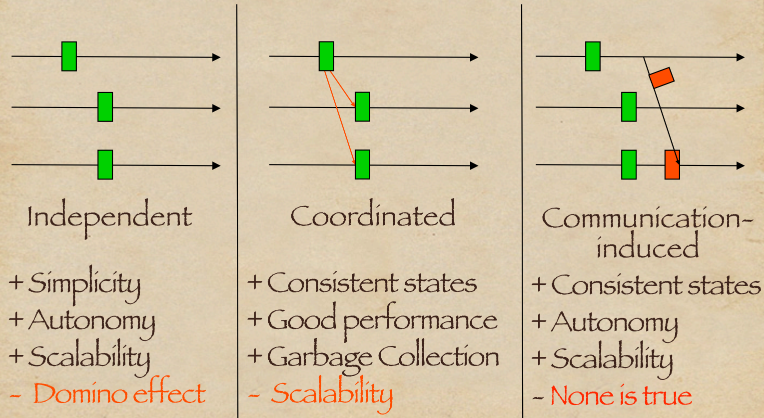
The Output Commit Problem



Coordinated checkpoint for every output commit

High overhead if frequent I/O with external environment

Distributed Checkpointing at a Glance

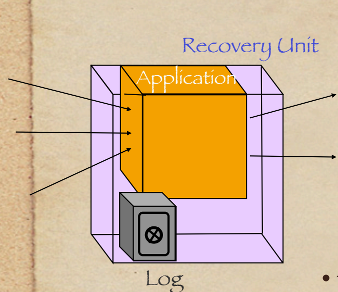


Independent	Coordinated	Communication-induced
+ Simplicity	+ Consistent states	+ Consistent states
+ Autonomy	+ Good performance	+ Autonomy
+ Scalability	+ Garbage Collection	+ Scalability
- Domino effect	- Scalability	- None is true

Message Logging

- Can avoid domino effect
- Works with coordinated checkpoint
- Works with uncoordinated checkpoint
- Can reduce cost of output commit
- More difficult to implement

How Message Logging Works



To tolerate crash failures:

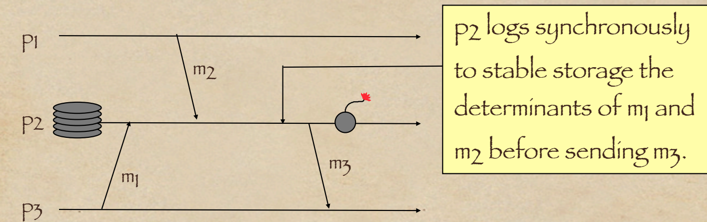
- periodically checkpoint application state;
- log on stable storage determinants of non-deterministic events executed after checkpointed state.
- for message delivery events:

  #m = (m.dest, m.rsn, m.source, m.ssn)

Recovery:

- restore latest checkpointed state;
- replay non-deterministic events according to determinants

Pessimistic Logging



p2 logs synchronously to stable storage the determinants of m_1 and m_2 before sending m_3.

Never creates orphans

- may incur blocking
- straightforward recovery

Sender Based Logging

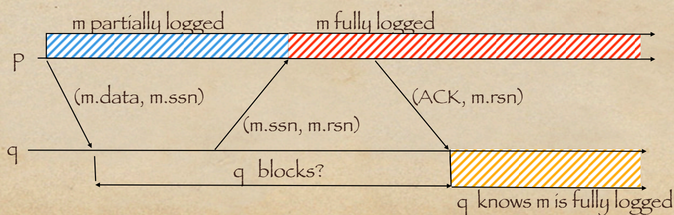
(Johnson and Zwaenepoel, FTCS 87)

Message log is maintained in volatile storage at the sender.

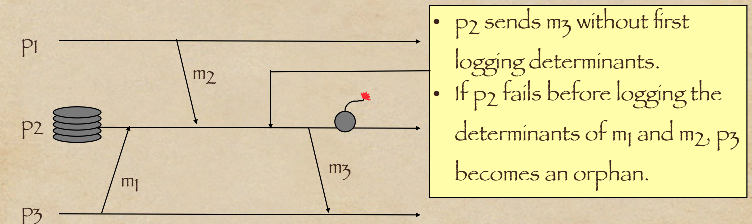
A message m is logged in two steps:

i) before sending m, the sender logs its content: m is partially logged

ii) the receiver tells the sender the receive sequence number of m, and the sender adds this information to its log: m is fully logged .



Optimistic Logging



- p2 sends m_3 without first logging determinants.
- If p2 fails before logging the determinants of m_1 and m_2, p3 becomes an orphan.

Eliminates orphans during recovery

- non-blocking during failure-free executions
- rollback of correct processes
- complex recovery

Causal Logging

- ☑ No blocking in failure-free executions
- ☑ No orphans
- ☑ No additional messages
- ☑ Tolerates multiple concurrent failures
- ☑ Keeps determinant in volatile memory
- ☑ Localized output commit

Preliminary Definitions

Given a message m sent from m.source to m.dest,

Depend(m): $\left\{ p \in P \;\middle|\; \begin{array}{l} \vee(p = m.dest) \text{ and } p \text{ delivered } m \\ \vee(\exists e_p : (deliver_{m.dest}(m) \rightarrow e_p)) \end{array} \right\}$

Log(m): set of processes with a copy of the determinant of m in their volatile memory

p orphan of a set C of crashed processes:

$$(p \notin C) \wedge \exists m : (Log(m) \subseteq C \wedge p \in Depend(m))$$

The "No-Orphans" Consistency Condition

No orphans after crash C if:

$$\forall m : (Log(m) \subseteq C) \Rightarrow (Depend(m) \subseteq C)$$

No orphans after any C if:

$$\forall m : (Depend(m) \subseteq Log(m))$$

The Consistency Condition

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Optimistic and Pessimistic

No orphans after crash C if:

$$\forall m : (Log(m) \subseteq C) \Rightarrow (Depend(m) \subseteq C)$$

Optimistic weakens it to:

$$\forall m : (Log(m) \subseteq C) \Rightarrow \Diamond(Depend(m) \subseteq C)$$

No orphans after any crash if:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Pessimistic strengthens it to:

$$\forall m : (\neg stable(m) \Rightarrow |Depend(m)| \leq 1)$$

Causal Message Logging

No orphans after any crash of size at most f if:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Causal strengthens it to:

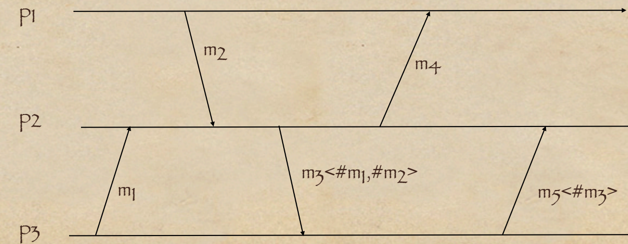
$$\forall m : \left(\neg stable(m) \Rightarrow \left(\begin{array}{l} \wedge(Depend(m) \subseteq Log(m)) \\ \wedge\Diamond(Depend(m) = Log(m)) \end{array} \right)\right)$$

An Example

Causal Logging:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

If $f = 1$, $stable(m) \equiv |Log(m)| \geq 2$



Recovery for f = 1