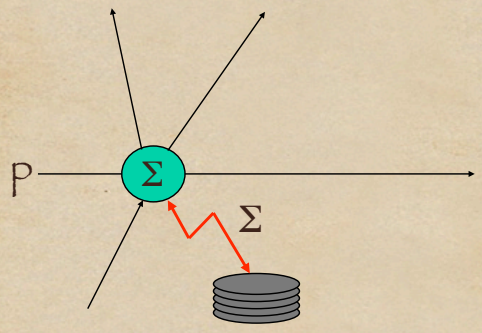
Rollback-Recovery

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

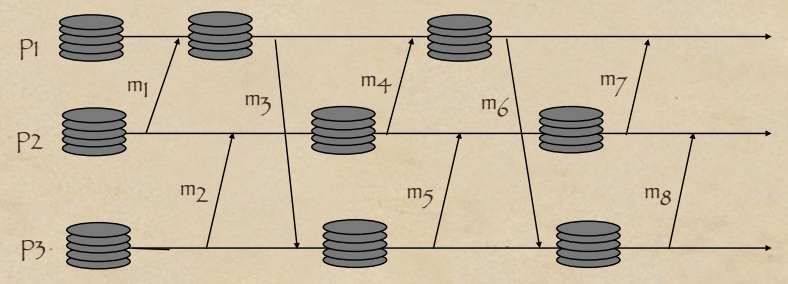
How (not) to take a checkpoint

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

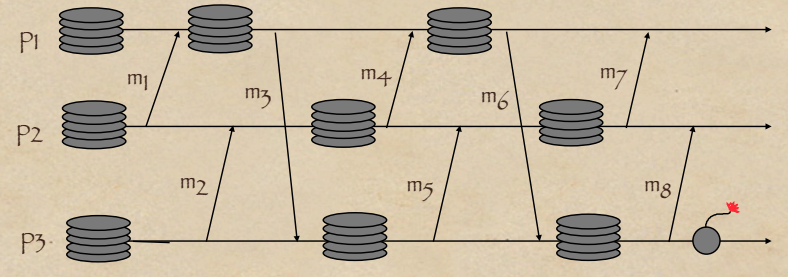
How to take a checkpoint

- 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

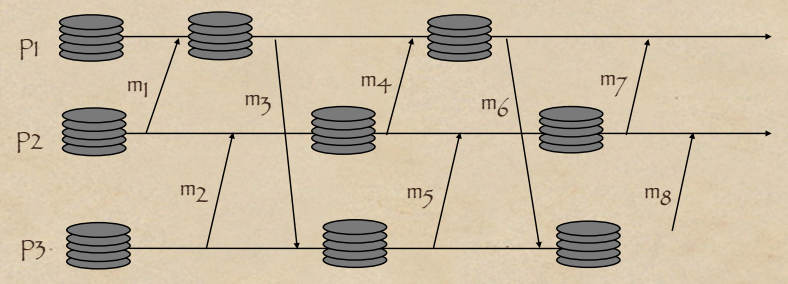
The Domino Effect



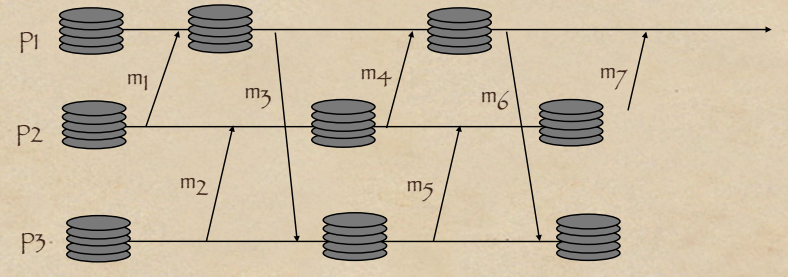
The Domino Effect



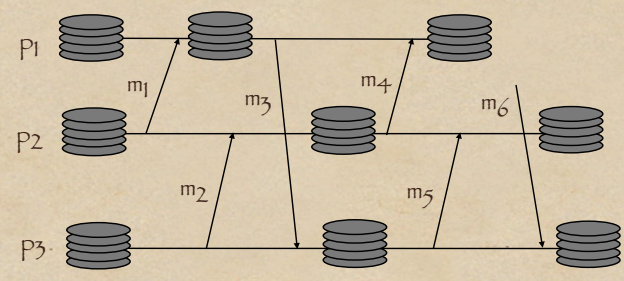
The Domino Effect



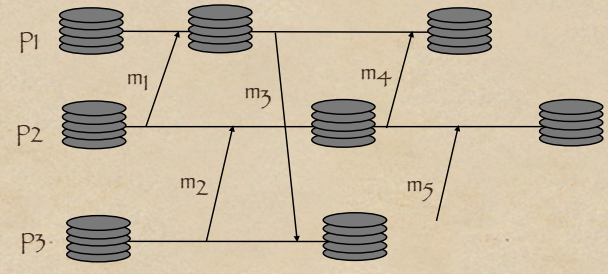
The Domino Effect



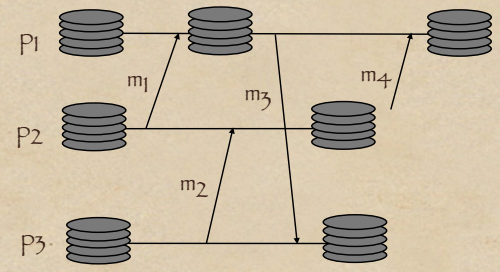
The Domino Effect



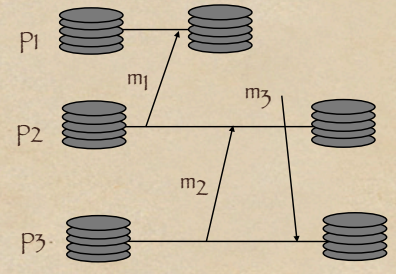
The Domino Effect



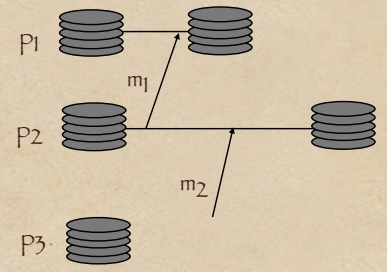
The Domino Effect



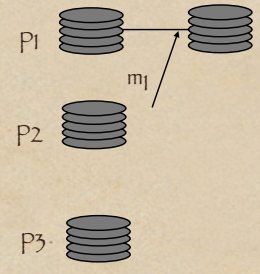
The Domino Effect



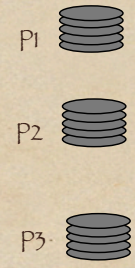
The Domino Effect



The Domino Effect



The Domino Effect



How to Avoid the Domino Effect

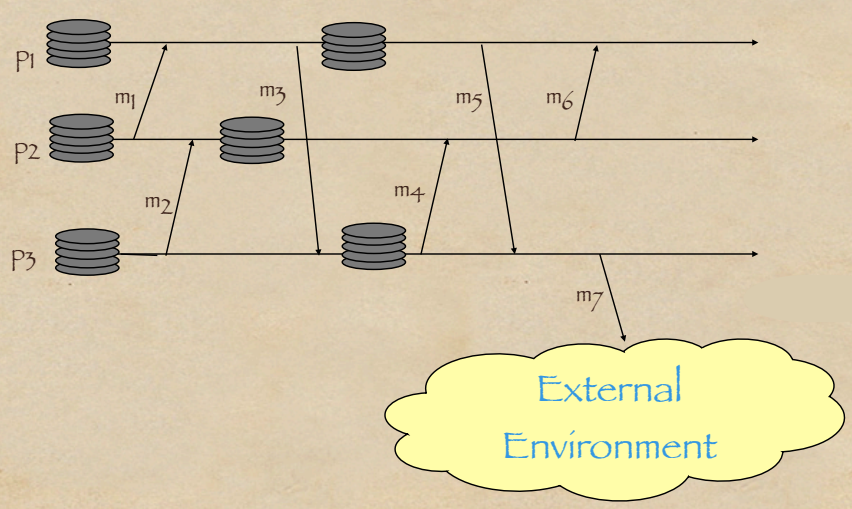
Coordinated Checkpointing

- No independence
- Synchronization Overhead
- Easy Garbage Collection

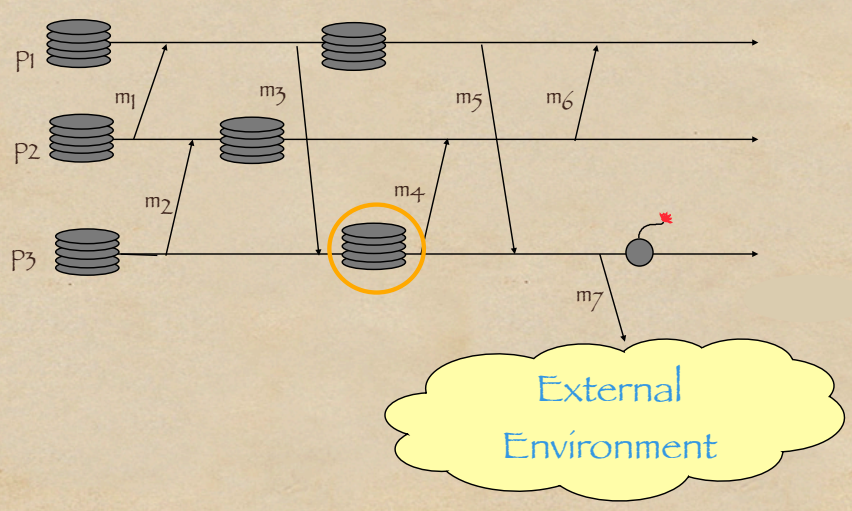
Communication Induced Checkpointing : detect dangerous communication patterns and checkpoint appropriately

- Less synchronization
- Less independence
- Complex

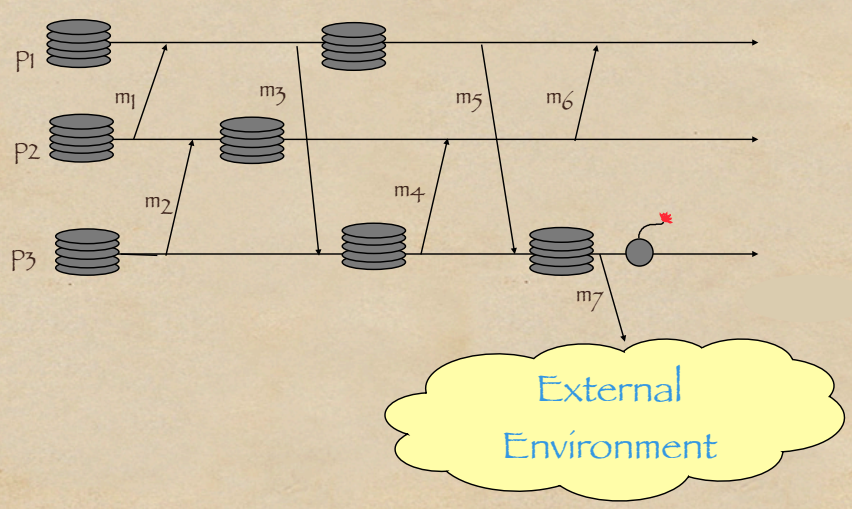
The Output Commit Problem



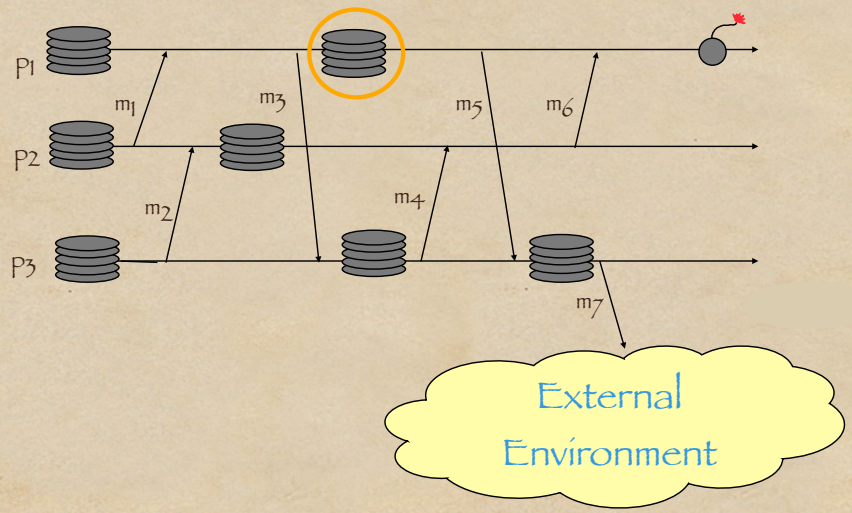
The Output Commit Problem



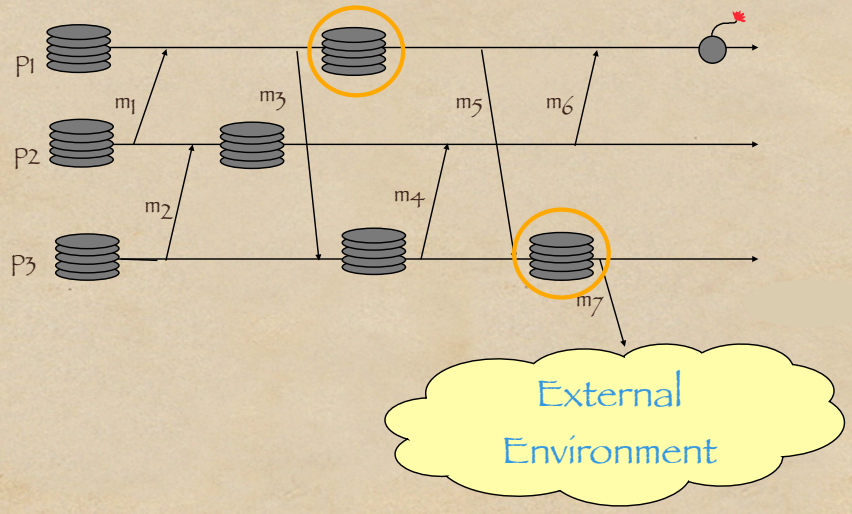
The Output Commit Problem



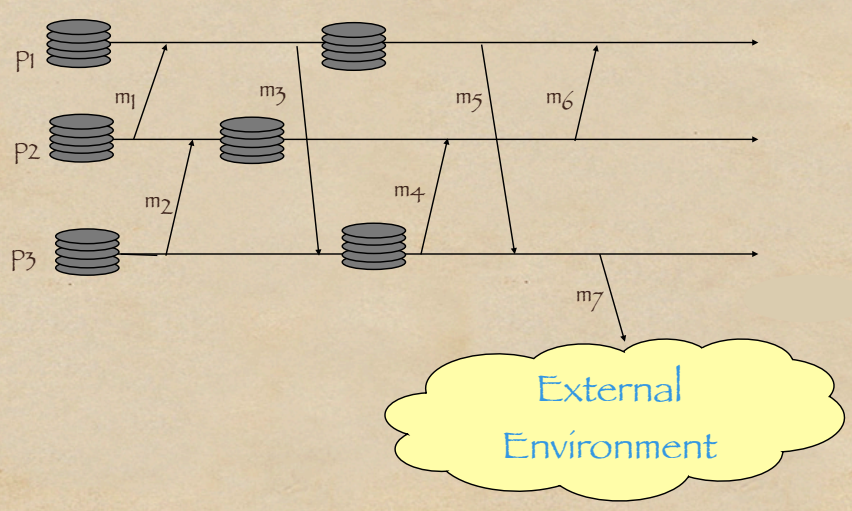
The Output Commit Problem



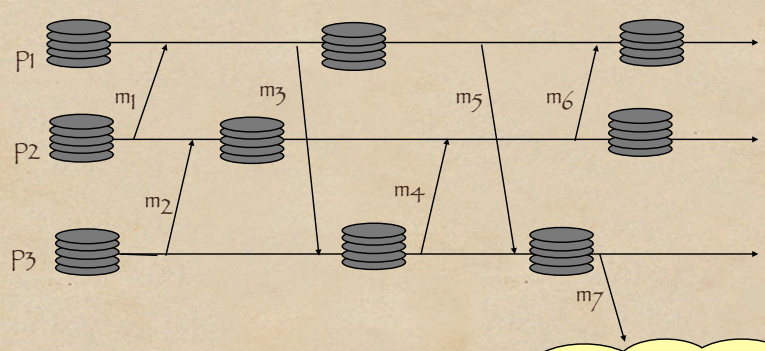
The Output Commit Problem



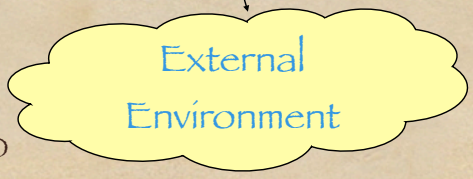
The Output Commit Problem



The Output Commit Problem



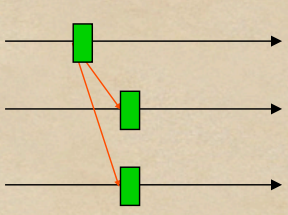
- Coordinated checkpoint for every output commit
- High overhead if frequent I/O with external environment



Distributed Checkpointing at a Glance

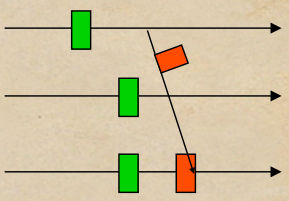
Independent

+ Simplicity
+ Autonomy
+ Scalability
- Domino effect



Coordinated

+ Consistent states
+ Good performance
+ Garbage Collection
- Scalability



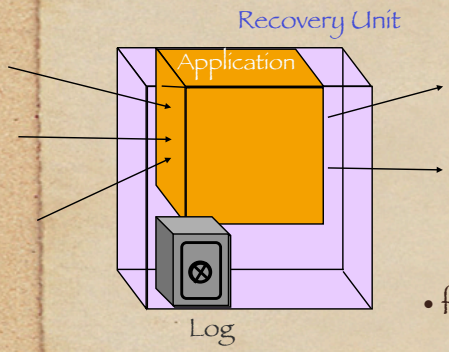
Communication-induced

+ Consistent states
+ Autonomy
+ 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:

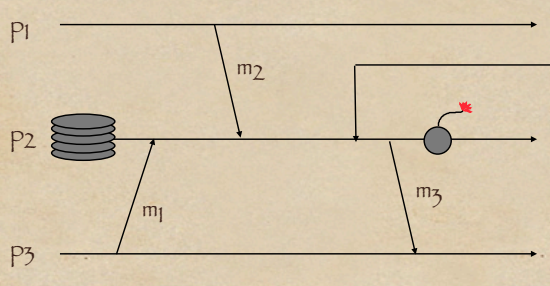
- periodically checkpoint application state;
- log on stable storage determinants of non-deterministic events executed after checkpointed state.

• 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

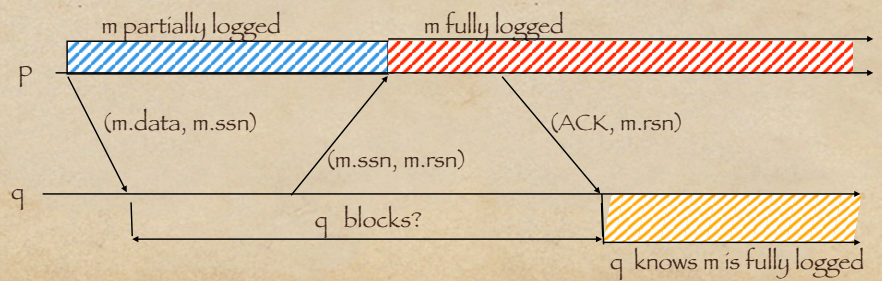
(Johnson and Zwaenepoel, FTCS 87)

Message log is maintained in volatile storage at the sender.

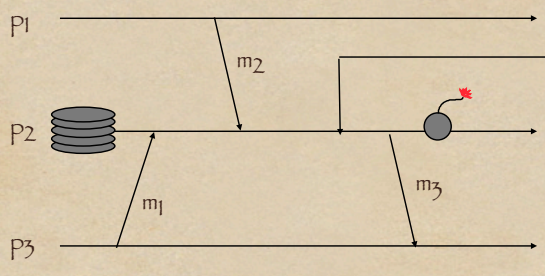
A message m is logged in two steps:

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



- P_2 sends m_3 without first logging determinants.
- If P_2 fails before logging the determinants of m_1 and m_2, P_3 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:

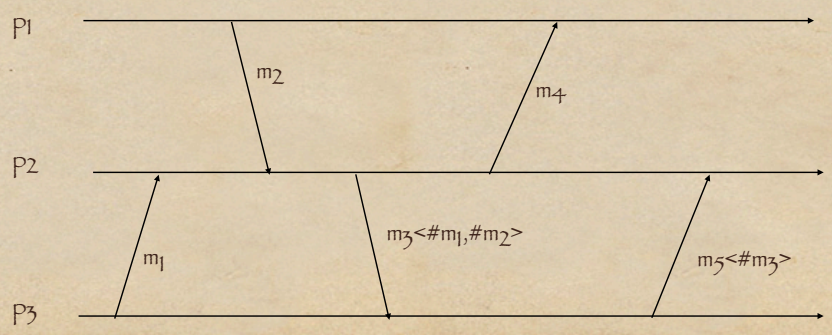
$$\forall m : \left(\neg stable(m) \Rightarrow \left(\begin{array}{l} \wedge(Depend(m) \subseteq Log(m)) \\ \wedge\Diamond(Depend(m) = Log(m)) \end{array}\right)\right)$$

An Example

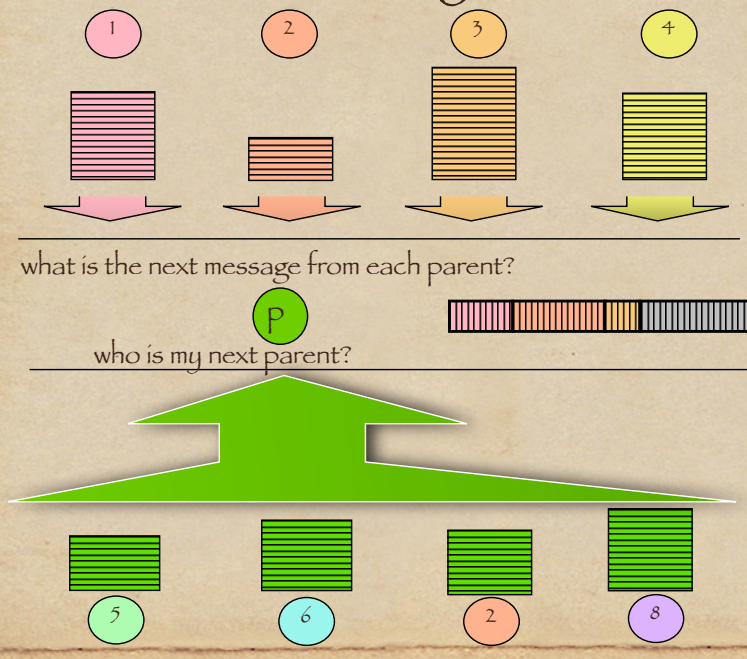
Causal Logging:

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

If f = 1, stable(m) ≡ |Log(m)| ≥ 2



Recovery for f = 1



parents of p

Messages previously sent to p by its parents

SSN order

Determinants of messages delivered by parents

RSN order

Determinants of messages delivered by p

children of p

Family-Based Logging

Each p maintains $D_p \equiv \{\#m : p \in Depend(m)\}$ in volatile memory

On sending a message m'

- adds m' to volatile send log
- piggybacks on messages to q all determinants #m ∈ D_p s.t.

$$|Log(m)|_p \leq f \wedge (q \notin Log(m)_p)$$

On receiving a message m'

- adds to D_p any new determinant piggybacked on m'
- adds #m' to D_p
- updates its estimate of $|Log(m)|_p$ for all determinants #m ∈ D_p

Estimating Log(m) and |Log(m)|

Each process p maintains estimates of

$$Log(m)_p \quad \text{and} \quad |Log(m)|_p$$

p piggybacks #m on m' to q if

$$|Log(m)|_p \leq f \wedge (q \notin Log(m)_p)$$

- How can p estimate $Log(m)_p$ and $|Log(m)|_p$?
- How accurate should these estimates be?

– inaccurate estimates cause useless piggybacking

– keeping estimates accurate requires extra piggybacking

The Idea

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

we can approximate Log(m) from below with:

and then use vector clocks to track Depend(m)!

$$Log(m) = \begin{cases} Depend(m) & \text{if } |Depend(m)| \leq f \\ \text{Any set } S : |S| > f & \text{otherwise} \end{cases}$$

Dependency Vectors

Dependency Vector (DV): vector clock that tracks causal dependencies between message delivery events.

$$deliver_p(m) \rightarrow deliver_q(m') \equiv$$
$$DV_p(deliver_p(m))[p] \leq DV_q(deliver_q(m'))[p]$$

Weak Dependency Vectors

Weak Dependency Vector (WDV):
track causal dependencies on deliver(m) as long as

$$(|Depend(m)| \leq f)$$

$$(deliver_p(m) \rightarrow deliver_q(m')) \wedge (|Depend(m)| \leq f) \Rightarrow WDV_p(deliver_p(m))[p] \leq WDV_q(deliver_q(m'))[p]$$

$$WDV_p(deliver_p(m))[p] \leq WDV_q(deliver_q(m'))[p] \Rightarrow deliver_p(m) \rightarrow deliver_q(m')$$

Dependency Matrix

Use WDVs to determine if p ∈ Log(m):

$$p \in Depend(m) \wedge |Depend(m)| \leq f \Rightarrow WDV_p[m.dest] \geq m.rsn$$

$$WDV_p[m.dest] \geq m.rsn \Rightarrow p \in Depend(m)$$

Each p keeps a Dependency Matrix (DM_p)

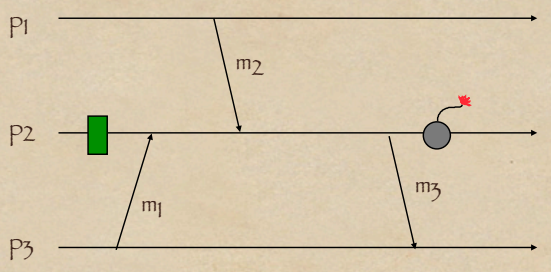
Given #m = <u, s, 14, 15>,

DM_p =

	s
p	21
q	16
r	8
s	21
t	12
u	7

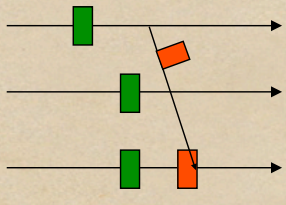
$Log(m)_p$ = {p, q, s}

Message Logging at a Glance



Pessimistic	Optimistic	Causal
+ No orphans	+ Non-blocking	+ Non-blocking
+ Easy recovery	- Orphans	+ No orphans
- Blocks	- Complex recovery	- Complex recovery

Communication Induced Checkpointing



Really?

+ Consistent states
+ Autonomy
+ Scalability
+ No useless checkpoints

CIC Protocols

- Independent local checkpoints
- Forced checkpoints before processing some messages
- Piggyback information about checkpoints on application messages

Always a consistent set of checkpoints without

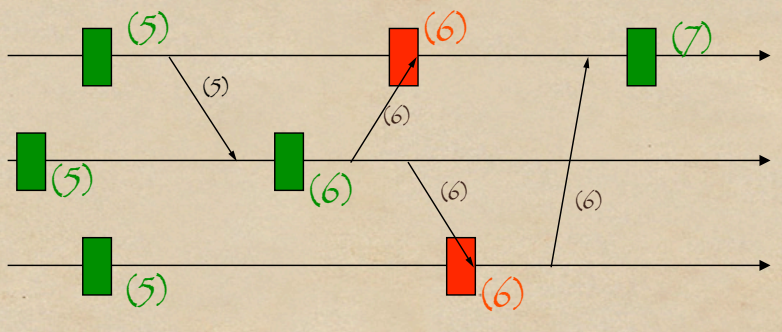
- explicit coordination
- protocol-specific messages

CIC Protocol Families

Index-Based	Pattern-Based
• Each checkpoint has an index • Indices piggybacked on application messages • Checkpoints with same index are consistent	• Detect communication patterns • Checkpoint to prevent dangerous patterns • Avoid useless checkpoints

They are equivalent

Example of Index Based

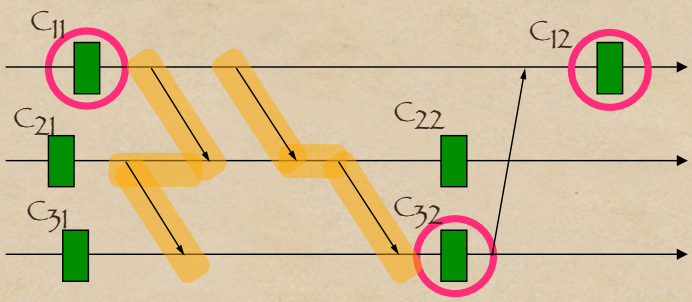


Local checkpoint

Forced checkpoint

After
Briatico, Ciuffoletti & Simoncini 84

Z-Paths



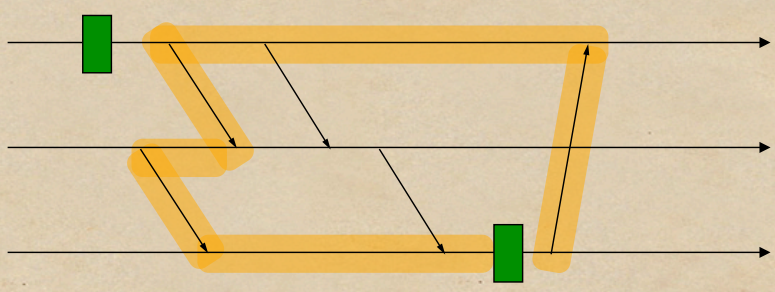
A Z-Path exists between C_{xi} and C_{yj} iff [Netzer & Xu 95]:

$i < j$ and $x = y$ or

There exists a path $[m_0, m_1, \ldots, m_n]$ such that:

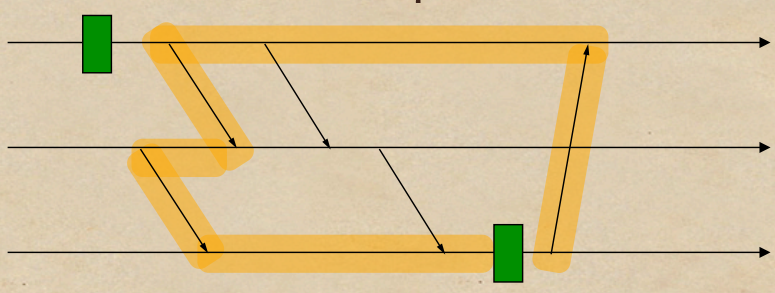
- $C_{xi} \rightarrow send_x(m_0)$
- $\forall l < n$ either $deliver_k(m_l) \rightarrow send_k(m_{l+1})$ or $send_k(m_{l+1}), deliver_k(m_l)$ are in the same checkpoint interval
- $deliver_y(m_n) \rightarrow C_{yj}$

Z-Cycles



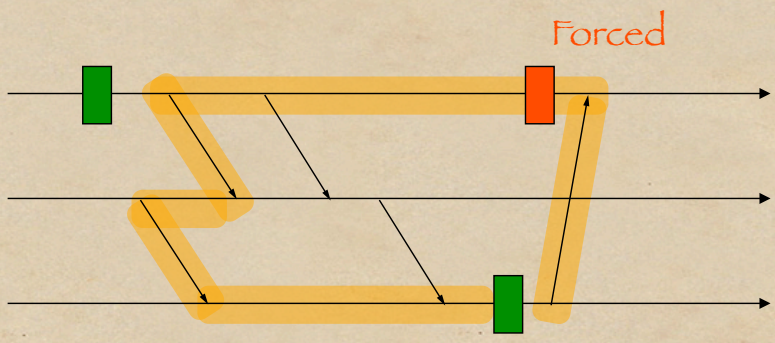
A Z-Cycle is a Z-path that begins and ends at the same checkpoint

Z-Cycles & Useless Checkpoints



A checkpoint in a Z-cycle can never be part of a consistent state

Example of Pattern-Based



The forced checkpoint breaks the Z-cycle, preventing the local checkpoint from becoming useless

(Baldoni, Quaglia & Ciciani 98)

Experiment Goals

- How to implement CIC protocols?
- What is the performance?
- How do they scale?
- Which is better, index-based or pattern-based?

Outline

- Implemented 3 CIC protocols in Egida
- Used NASA NPB 2.3 benchmark applications

Appl.	Communication Rate		Communication Pattern	Exec. Time (sec)
	Mess/sec	Size(KB)		
bt	6	50.7	All processes	1530
cg	20	60.7	Two neighbors	1516
lu	62	3.7	Two neighbors	975
sp	22	44.4	All processes	1222

- For most experiments, direct measures
- Simulation to extrapolate for scale
  - Used implementation to validate simulator

The Three Protocols

Index-Based:

- Briatico, Ciuffoletti & Simoncini '84,
  - BCS, O(1)/message
- Hélary, Mostefaoui, Netzer & Raynal '97,
  - HMNR, O(n)/message

Pattern-based:

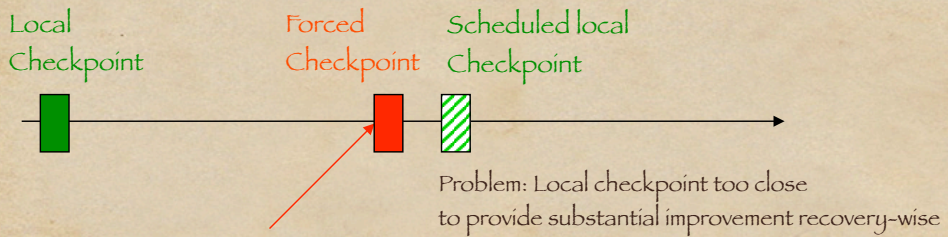
- Baldoni, Quaglia & Ciciani '98,
  - BQC, $O(n^2)$/message

Autonomy?

Processes take independent checkpoints

But:

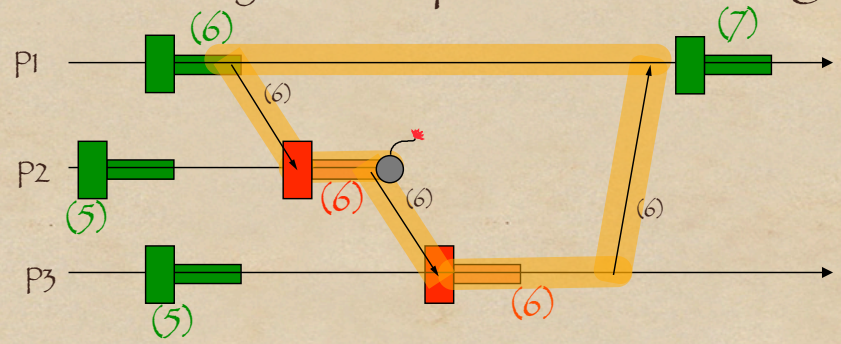
- Selecting a checkpointing placement policy is hard
- A process has no control over forced checkpoints



Problem: Local checkpoint too close to provide substantial improvement recovery-wise

No Useless Checkpoints?

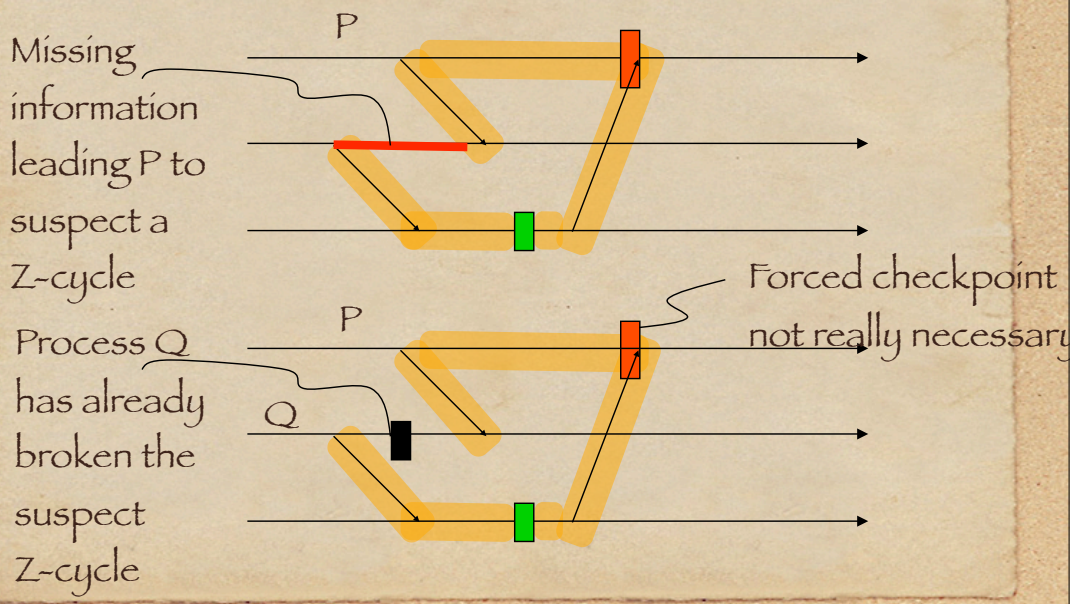
- Yes, but only if checkpoints are blocking!



Checkpoint (6) of p3 can become useless

p1 may run garbage collection and discard checkpoint (6)

Analysis of BQC's



Non-Blocking Checkpointing & CIC

Solutions:

Blocking checkpoints, as assumed by protocol

- Performance problems

Be optimistic and don't block

- Violates protocol's invariant

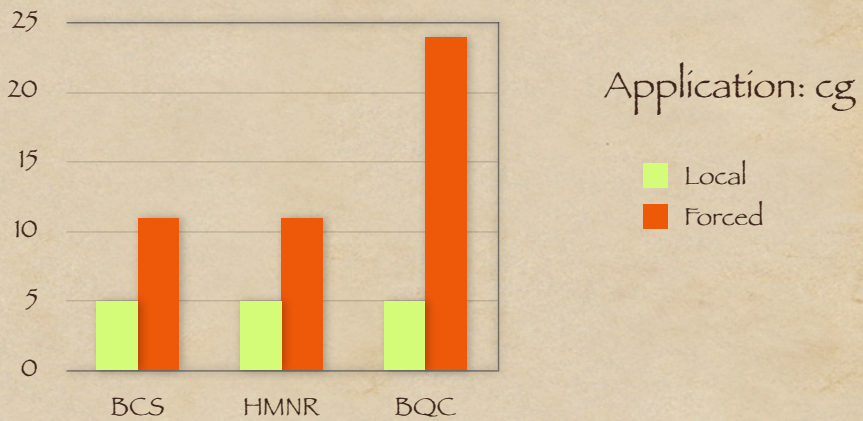
we take non-blocking checkpoints

Execution Overhead

Local checkpoints taken with exponential distribution, mean 6 minutes

- 4 Pentium-based workstations, 300 MHz
- Lightly-loaded 100Mb/s Ethernet
- Stable storage on local disk

Forced vs. Local

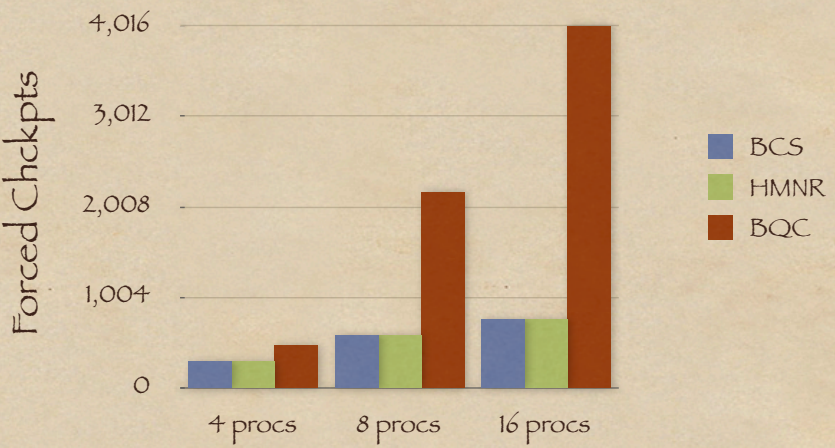


Local checkpoints taken with exponential distribution, mean 6 minutes

Scalability

Simulation with 119 local ckp's/proc
Low comm. load of 10 msg/ckp, random pattern

Scalability (cont'd)



Simulation with 118 local ckp's/proc
High comm. load of 500 msg/ckp, uniform pattern

Summary

- Scalability? Not exactly…
- Autonomy in checkpointing? Not exactly…
  - # of forced ckp's is often greater than twice the # of local ones
  - adaptation necessary for good performance
- Unpredictable behavior:
  - Difficult to plan resources, decide on local ckpts, or estimate overhead
- Performs well for random pattern, low-load