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 $\in D_p$ 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 $\in D_p$

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):

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

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

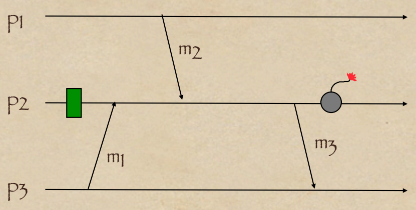
Each p keeps a
Dependency Matrix (DM_p)

Given $\#m = <\overset{source}{u}, \overset{des}{s,_t} \overset{ssn}{14}, \overset{rsn}{15}>$,

$$DM_p = \begin{array}{c|ccc} & & s & \\ p & & 21 & \\ q & & 16 & \\ r & & 8 & \\ s & & 21 & \\ t & & 12 & \\ u & & 7 & \end{array}$$

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

Message Logging at a Glance

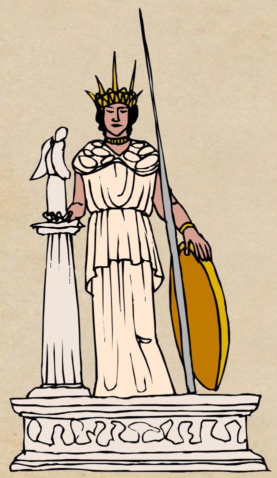


Pessimistic	Optimistic	Causal
+ No orphans	+ Non-blocking	+ Non-blocking
+ Easy recovery	- Orphans	+ No orphans
- Blocks	- Complex recovery	- Complex recovery

Rollback Recovery Protocols:
A Success Story?

- Over 300 papers in the area
- Relatively few implementations
- Why?
  - Performance issues not understood
  - Hard to integrate recovery protocol with application
  - One size doesn't fit all

Egida



- Trasparent
  - seamless integration with applications
- Extensible
  - easily handles new sources of non-determinism
- Flexible
  - allows to select best protocol for application
- Smart
  - don't want to implement 300 protocols
- Powerful
  - a "microscope" to understand rollback recovery

The Unifying Theme

- All rollback recovery protocols enforce the no-orphans consistency condition
- The challenge is handling non determinism
  - a process may execute non-deterministic events
  - a process may interact with other processes or with the environment and generate dependencies on these events
- Characterize a protocol according to how it handles non-determinism
  - identify relevant events
  - specify which actions to take when event occurs

Relevant Events

- Non-deterministic events
  - message delivery, file read, clock read, lock acquire
- Failure-detection events
  - time-outs, message delivery
- Internal dependency-generating events
  - message send, file write, lock release
- External dependency-generating events
  - output to printer or screen, file write
- Checkpointing events
  - time-outs, explicit instruction, message delivery

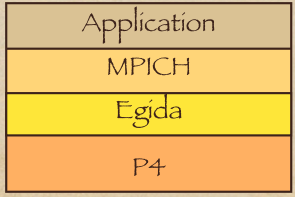
The Architecture

- Event handlers invoked on relevant events
- Library of modules
  - implement core functionalities
  - (checkpointing, creating determinants, logging, piggybacking, detecting orphans, restarting a faulty process
  - provide basic services
  - (stable storage, failure detection, etc
  - single interface–multiple implementations
- Specification language to select desired modules and corresponding implementations
- Synthesize protocol automatically from specification

Integration with MPICH

MPICH

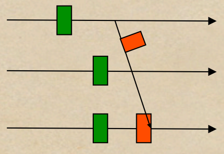
- 2 layers architecture
- upper layer exports MPI to application
- lower layer performs data transfer using application-specific libraries (e.g. P4)



- Modifications to MPICH:
  - Replace calls to P4 with call to Egida's API
- Modifications to P4:
  - Handle socket-level errors
  - Allow reconnection of recovering process
- Modification to applications:

NONE

Communication Induced Checkpointing



+ Consistent states
+ Autonomy
+ Scalability
+ No useless checkpoints

Really?

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

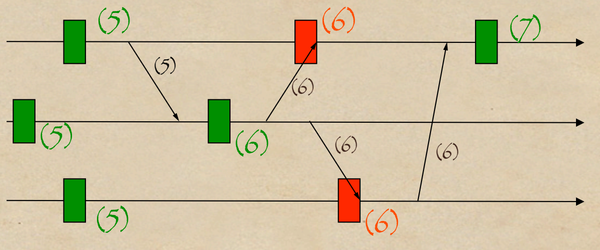
- explicit coordination
- protocol-specific messages

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 • Take checkpoints to prevent dangerous patterns • Avoid useless checkpoints

They are equivalent

Example of Index Based

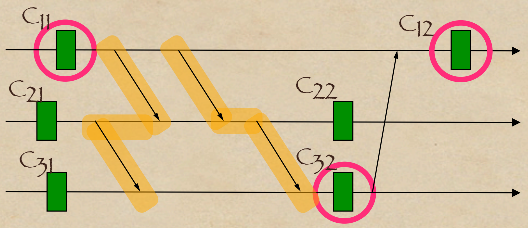


🟩 Local checkpoint

🟥 Forced checkpoint

After Briatico, Ciuffoletti & Simoncini 84

Z-Paths

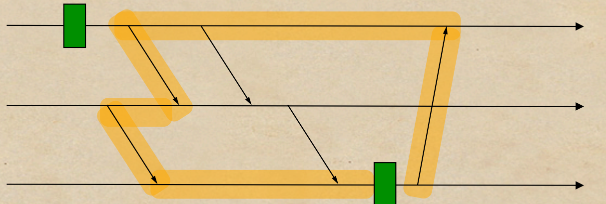


A Z-Path exists between C_{xi} and C_{yj} iff [Netzer & Xu 95]:

$i < j$ and $x = y$ or There exists $[m_0, m_1, \ldots m_n]$ such that:

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

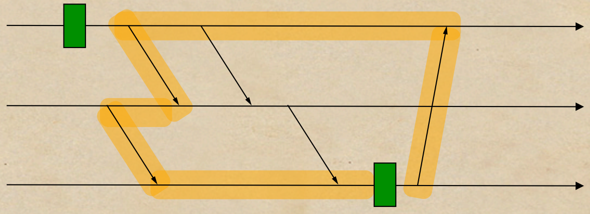
Z-Cycles



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

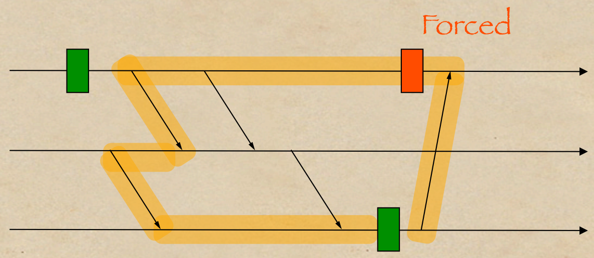
---

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:

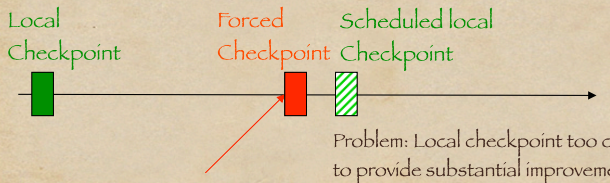
- Baldoni, Quaglia & Ciciani '98,
  - BQC, $O(n^2)$/message

Autonomy?

Processes take independent checkpoints

But:

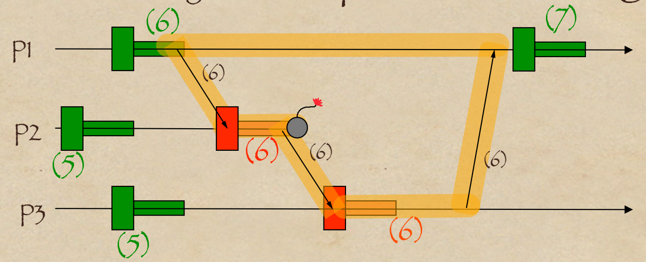
- Selecting a checkpointing placement policy is hard
- A process has no control over forced checkpoints



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

No Useless Checkpoints?

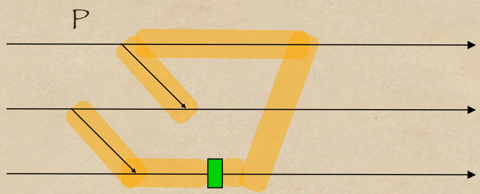
- Yes, but only if checkpoints are blocking!



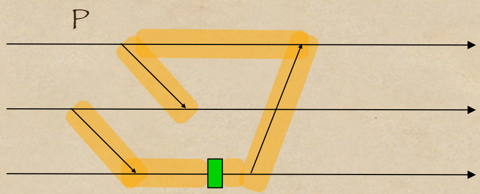
Checkpoint (6) of p3
can become useless

p1 may run garbage collection
and discard checkpoint (6)

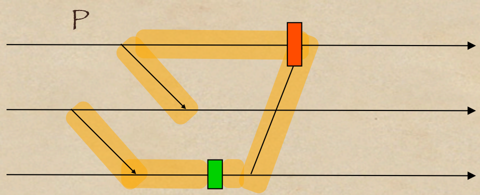
BQC's Behavior



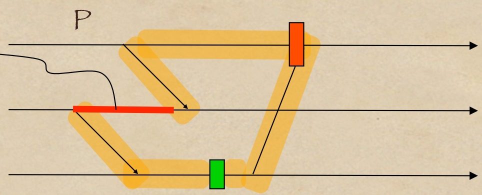
BQC's Behavior



BQC's Behavior

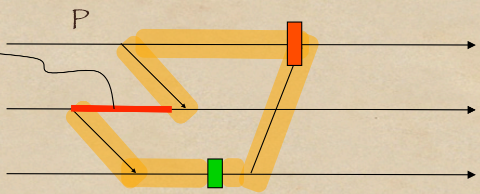


BQC's Behavior



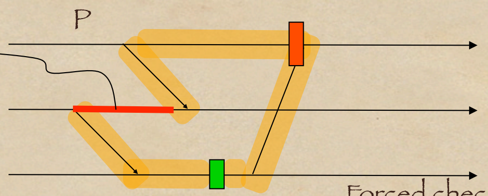
Missing information leading P to suspect a Z-cycle

BQC's Behavior

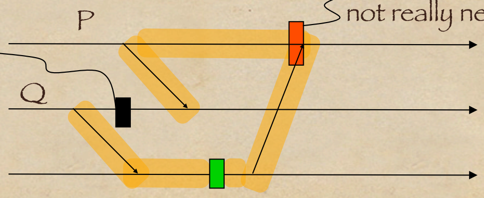


Missing information leading P to suspect a Z-cycle

BQC's Behavior



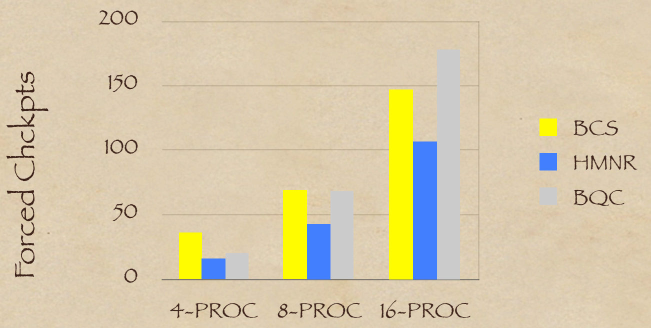
Missing information leading P to suspect a Z-cycle



Forced checkpoint not really necessary

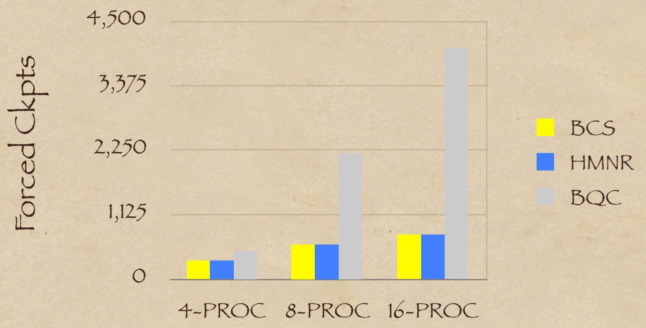
Process Q has already broken the suspect Z-cycle

Scalability: random pattern



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

Scalability: uniform pattern



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 performan ce
- Unpredictable behavior:
    - Difficult to plan resources, decide on local ckpts, or estimate overhead
- Performs well for random pattern, low-load communications
- Fewer forced checkpoints with index-based than eager pattern-based protocols