Family-Based Logging

Each process $p$ maintains $D_p = \{ #m : p \in \text{Depend}(m) \}$ 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. $|\text{Log}(m)|_p \leq f \land (q \notin \text{Log}(m)_p)\) for all determinants $#m \in D_p$.

On receiving a message $m'$:
- Adds $D_p$ any new determinant piggybacked on $m'$.
- Adds $#m'$ to $D_p$.
- Updates its estimate of $|\text{Log}(m)|_p$.

Estimating $\text{Log}(m)$ and $|\text{Log}(m)|$

Each process $p$ maintains estimates of $\text{Log}(m)_p$ and $|\text{Log}(m)|_p$.

$p$ piggybacks $#m$ on $m'$ to $q$ if $|\text{Log}(m)|_p \leq f \land (q \notin \text{Log}(m)_p)$.

- How can $p$ estimate $\text{Log}(m)_p$ and $|\text{Log}(m)|_p$?
- How accurate should these estimates be?
  - Inaccurate estimates cause useless piggybacking.
  - Keeping estimates accurate requires extra piggybacking.

The Idea

Because $\forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m)))$, we can approximate $\text{Log}(m)$ from below with:

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

Dependency Vectors

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

$\text{deliv}_{p}(m) \rightarrow \text{deliv}_{q}(m') \equiv DV_p(\text{deliv}_{p}(m))[p] \leq DV_q(\text{deliv}_{q}(m'))[p]$
Weak Dependency Vectors

**Weak Dependency Vector (WDV):**
track causal dependencies on `deliver(m)` as long as
\[(|Depend(m)| \leq f)\]

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

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

**Dependency Matrix**

Use WDV to determine if \(p \in \text{Log}(m)\):

\[p \in \text{Depend}(m) \land (|Depend(m)| \leq f) \Rightarrow WDV_{p}[m,\text{dest}] \geq m,rsn\]

Each \(p\) keeps a Dependency Matrix (DM\(_p\))

\[DM_p = \begin{array}{cccc}
\text{source} & \text{dest} & \text{src} & \text{rsn} \\
\hline
a & b & 1 & 1 \\
\end{array}\]

Given \#m = \langle u, s, 14, 15 \rangle,

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

**Message Logging at a Glance**

- **Pessimistic**
  - No orphans
  - Easy recovery
  - Blocks

- **Optimistic**
  - Non-blocking
  - Complex recovery

- **Causal**
  - Non-blocking
  - No orphans
  - 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**
- Transparent
  - 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 acquire
- 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)

<table>
<thead>
<tr>
<th>Application</th>
<th>MPICH</th>
<th>Egida</th>
<th>P4</th>
</tr>
</thead>
</table>

- Modifications to MPICH:
  - Replace calls to P4 with call to Egida’s API

- Modifications to P4:
  - Handle socket-level errors
  - Allow reconnection of recovering processes

- 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
- Each checkpoint has an index
- Indices piggybacked on application messages
- Checkpoints with same index are consistent

Pattern-Based
- Detect communication patterns
- Take checkpoints to prevent dangerous patterns
- Avoid useless checkpoints

They are equivalent
Example of Index Based

Z-Paths

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

- $i < j$ and $x = y$
- There exists $[m_0, m_1, ..., m_n]$ such that:
  - $C_{xi} \rightarrow \text{send}_{k}(m_0)$
  - $\forall l < n$, either $\text{deliver}_{k}(m_l) \rightarrow \text{send}_{k}(m_{l+1})$ or $\text{send}_{k}(m_{l+1}), \text{deliver}_{k}(m_l)$ in same ckpt interval
  - $\text{delivery}(m_n) \rightarrow C_{yj}$

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
- 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

BQC’s Behavior

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
Missing information leading P to suspect a Z-cycle

Process Q has already broken the suspect Z-cycle

Forced checkpoint not really necessary
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 communications
- Fewer forced checkpoints with index-based than eager pattern-based protocols