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

The Domino Effect

\[ \sum_{i=1}^{n} m_i \]

\[ m_2 \quad m_3 \quad m_4 \quad m_5 \quad m_6 \quad m_7 \quad m_8 \]

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

External Environment
The Output Commit Problem

External Environment
The Output Commit Problem

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

External Environment

Distributed Checkpointing at a Glance

- Consistent states
- Autonomy
- Scalability
- Domino effect

<table>
<thead>
<tr>
<th>Independent</th>
<th>Coordinated</th>
<th>Communication-induced</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Simplicity</td>
<td>+ Consistent states</td>
<td>+ Consistent states</td>
</tr>
<tr>
<td>+ Autonomy</td>
<td>+ Good performance</td>
<td>+ Autonomy</td>
</tr>
<tr>
<td>+ Scalability</td>
<td>+ Garbage Collection</td>
<td>+ Scalability</td>
</tr>
<tr>
<td>- Domino effect</td>
<td>- Scalability</td>
<td>- None is true</td>
</tr>
</tbody>
</table>

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

- 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:
1. Before sending \( m \), the sender logs its content: \( m \) is partially logged.
2. The receiver tells the sender the receive sequence number of \( m \), and the sender adds this information to its log: \( m \) is fully logged.

\[
\text{Sender Based Logging:}(m.\text{data}, m.\text{ssn}) \quad \text{or} \quad (\text{ACK}, m.\text{rsn})
\]

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.

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 \( m.\text{source} \) to \( m.\text{dest} \),

\[
\text{Depend}(m): \left\{ p \in P \mid \forall (p = m.\text{dest}) \text{ and } p \text{ delivered } m \right\}
\]

\[
\text{Log}(m): \text{ set of processes with a copy of the determinant of } m \text{ in their volatile memory}
\]

\( p \) orphan of a set \( C \) of crashed processes:

\[
(p \notin C) \land \exists m : (\text{Log}(m) \subseteq C \land p \in \text{Depend}(m))
\]

The “No-Orphans” Consistency Condition

No orphans after crash \( C \) if:

\[
\forall m : (\text{Log}(m) \subseteq C) \Rightarrow (\text{Depend}(m) \subseteq C)
\]

No orphans after any \( C \) if:

\[
\forall m : (\text{Depend}(m) \subseteq \text{Log}(m))
\]

The Consistency Condition

\[
\forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m)))
\]

Optimistic and Pessimistic

No orphans after crash \( C \) if:

\[
\forall m : (\text{Log}(m) \subseteq C) \Rightarrow (\text{Depend}(m) \subseteq C)
\]

Optimistic weakens it to:

\[
\forall m : (\text{Log}(m) \subseteq C) \Rightarrow \Diamond (\text{Depend}(m) \subseteq C)
\]

No orphans after any crash if:

\[
\forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m)))
\]

Pessimistic strengthens it to:

\[
\forall m : (\neg \text{stable}(m) \Rightarrow |\text{Depend}(m)| \leq 1)
\]

Causal Message Logging

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

\[
\forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m)))
\]

Causal strengthens it to:

\[
\forall m : \left( \neg \text{stable}(m) \Rightarrow \left( (\text{Depend}(m) \subseteq \text{Log}(m)) \land \Diamond (\text{Depend}(m) = \text{Log}(m)) \right) \right)
\]
An Example

Causal Logging:

\[ \forall m : (-\text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m))) \]

If \( f = 1 \), \( \text{stable}(m) \equiv |\text{Log}(m)| \geq 2 \)

\[
\begin{array}{c}
\text{P1} \\
\text{m_2} \\
\text{m_4} \\
\text{P2} \\
\text{m_3 \#m_3, \#m_2} \\
\text{m_5 < \#m_5} \\
\text{P3}
\end{array}
\]

Family-Based Logging

Each \( p \) maintains \( D_p \equiv \{ \#m : p \in \text{Depend}(m) \} \) 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.

\[ |\text{Log}(m)|_p \leq f \land (q \notin \text{Log}(m)_p) \]

On receiving a message \( m' \):
- adds to \( D_p \) any new determinant piggybacked on \( m' \)
- adds \( \#m' \) to \( D_p \)
- updates its estimate of \( |\text{Log}(m)|_p \) for all determinants \( \#m \in D_p \)

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 \( \forall m : (-\text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m))) \)
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.

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

**Weak Dependency Vectors**

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

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

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

**Dependency Matrix**

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

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

- \(WDV_p[m.dest] \geq m.rsn \Rightarrow p \in \text{Depend}(m)\)

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

**Message Logging at a Glance**

<table>
<thead>
<tr>
<th>Pessimistic</th>
<th>Optimistic</th>
<th>Causal</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ No orphans</td>
<td>+ Non-blocking</td>
<td>+ Non-blocking</td>
</tr>
<tr>
<td>+ Easy recovery</td>
<td>- Orphans</td>
<td>+ No orphans</td>
</tr>
<tr>
<td>- Blocks</td>
<td>- Complex recovery</td>
<td>- Complex recovery</td>
</tr>
</tbody>
</table>

Log\((m)_p = \{p, q, s\}\)
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
- Checkpoint to prevent dangerous patterns
- Avoid useless checkpoints

They are equivalent

Example of Index Based

After Briatico, Ciuffoletti & Simoncini 84
**Z-Paths**

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

- $i < j$ and $x = y$
- There exists a path $[m_0, m_1, \ldots, m_n]$ such that:
  - $C_{xi} \rightarrow send_x(m_0)$
  - $\forall i < n$ either $deliver_x(m_i) \rightarrow send_k(m_{i+1})$ or $send_k(m_{i+1})$, $deliver_k(m_{i+1})$ 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.