A Survey of Rollback-Recovery Protocols in Message-Passing Systems

Paper by Elnozahy, Alvisi, Wang, and Johnson
Presented By Bryan Jacobs

Background
- Message-passing system with N processes
  - Possible to model shared memory using message passing
- Fail-stop failures possible at any time
- GOAL: recover all processes to some “consistent” state after one or more nodes have failed and then recovered (or been replaced)

Definitions
- Piecewise Deterministic (PWD) Assumption
  - Any nondeterministic element of the system may be captured in such a way as to allow replaying it in a deterministic fashion at a later time
- State Interval – the period between nondeterministic events
- OWP – A “process” whose incoming messages represent program output and whose incoming messages are program input
- “In Transit” - a message sent but not received

More Definitions!
- “Orphan” - a process whose checkpoint depends on a state that cannot be recreated
- “no-orphans” - good property which means orphans cannot be formed
- “output commit problem” - You can't un-print something, and it's not nice to ask the user to type in the exact same information they entered before
Checkpointing

- Basic idea: If something goes wrong, we just pick up the pieces and put everything back where it was.
- At regular intervals, dump all memory and resources associated with a process to persistent storage.
- When a process fails, we reload the stored file into memory, restore all state, and restart the process from where it was when the checkpoint was taken.

Concurrent Checkpointing

- Idea: trap writes to pages you haven't checkpointed, and copy the original pages to a buffer before allowing the write.
- In filesystems, this is called “snapshotting.”
- Advantage: the process may continue computing while the checkpointing takes place.

Incremental Checkpointing

- Idea: save only MODIFIED pages.
- Best when used together with concurrent checkpointing.
- Could even use differential storage to reduce space overhead of multiple checkpoints for a single process.

Consistency Issue: if P1 checkpoints and then sends a message to P2, you can no longer roll P1 back to before the message was sent unless P2 rolls back to before the message’s receipt.

Checkpointing in a Distributed Environment

- Issue: Checkpointing can be very expensive; you don't want to checkpoint more than necessary, but you also don't want to have to roll back too far.
- Issue: Communication with the outside world doesn't get undone.
Illustration of Consistency Problems

- Every once in a while, a process will checkpoint its own state.
- For recovery, each process sends information on the checkpoints it has to some central point, which calculates the "recovery line" - the last set of checkpoints to create a globally consistent state.
- Problem: the "domino effect".
- Problem: garbage collection of old checkpoints.
- Problem: many useless checkpoints.
- Problem: communication with OWP requires global information.

Uncoordinated Checkpointing

Domino Effect

- Idea: all message-passing calls block while checkpointing takes place at the same time across all processes.
- Checkpoints are only valid after it is known that all processes have successfully completed their own copy.
- Advantage: only one checkpoint needs to be kept.
- Advantage: recovery is just rolling back everything to THE checkpoint.
- Disadvantage: high overhead.

Coordinated Checkpointing
Non-blocking Coordinated Checkpointing

- Idea: processes are allowed to continue communicating, but if the recipient of one of their messages is behind the times with respect to its checkpoint, it is forced to catch up before it may use the message
- Requirement: communications must be reliable
  - Or you may incur a time overhead on each message sent during checkpointing
- Still a problem: scalability of global checkpoints

Minimal Checkpoints

- When a process initiates a checkpoint, it goes through a traversal to find the processes which depend on some message it has sent since the last checkpoint
- All those processes then checkpoint together
- Advantage: Not global, but still has the benefits of coordinated checkpointing

Communication-Induced Checkpointing

- Two different types of checkpoints: local and forced
- Idea: each process sends some information about the state of its (and others') checkpoints along with each message it sends
- Receiver uses information to decide when it needs an additional “forced” checkpoint
- Very similar to vector-timestamp algorithms
- Advantage: avoids the “domino effect” without global coordination
- Disadvantage: there are too many checkpoints!

Checkpointing Overview

<table>
<thead>
<tr>
<th></th>
<th>Uncoordinated</th>
<th>Coordinated</th>
<th>Communication-Induced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery</td>
<td>Unbounded</td>
<td>From last checkpoint</td>
<td>Bounded</td>
</tr>
<tr>
<td>Point Overhead</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Message Overhead</td>
<td>None</td>
<td>None</td>
<td>High</td>
</tr>
<tr>
<td>Complexity</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>
Logging as a Compliment to Checkpointing

- Logging is only viable where PWD assumption holds
- We don't need to log all computation, only the "determinants" of the nondeterministic events since the last checkpoint
- Advantage of logging systems: ideally, you lose NO progress after the recovery is complete!
- Logging-based solutions are not susceptible to the "domino effect"

Pessimistic Logging

- No process may use the result of a nondeterministic event until the determinant is committed to stable storage
- Result: 100% recovery!
- Disadvantage: Potentially very high overhead even in the absence of a fault
- This should NOT replace checkpointing completely; if you only do logging this way, then log replays would take almost as long as your original program run until the fault!

Mitigating the Downsides of Pessimistic Logging

- Non-volatile fast storage for the logs
  - Flash is getting cheaper...
- Hardware support for logging messages
  - Nondeterministic events must be converted to messages
- Store determinants in other processes' memory instead of on disk
- Defer logging until the process communicates with somebody else (so it doesn't matter if the process performs differently on replay, because nobody saw it and so it didn't really happen!)

Optimistic Logging

- As pessimistic logging, but processes are allowed to use the result of the nondeterministic operation before the determinant has been committed to stable storage
- Gain: performance
- Loss: now our garbage collection and recovery are much, much more complicated
Causal Logging

- Pass ALL determinants preceding the current state of a process to that process
- This means that SOMEWHERE, the information necessary to recover from a crash exists – everyone affected by an event knows its determinants, so if the knowledge does not exist it was not crucial anyhow
- Advantages: Performance of optimistic logging and no-orphans guarantee of pessimistic logging
- Disadvantage: Very complicated recovery and potentially high overhead

<table>
<thead>
<tr>
<th></th>
<th>Uncoordinated Checkpointing</th>
<th>Coordinated Checkpointing</th>
<th>Comm. Induced Checkpointing</th>
<th>Pessimistic Logging</th>
<th>Optimistic Logging</th>
<th>Causal Logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>P/ID saved?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Checkpoint process</td>
<td>Several</td>
<td>1</td>
<td>Several</td>
<td>1</td>
<td>Several</td>
<td>1</td>
</tr>
<tr>
<td>Domain effect</td>
<td>Possible</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Orphan processes</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
<td>No</td>
</tr>
<tr>
<td>Rollback extent</td>
<td>Unbounded</td>
<td>Last global checkpoint</td>
<td>Possibly several checkpoints</td>
<td>Last checkpoint</td>
<td>Possibly several checkpoints</td>
<td>Last checkpoint</td>
</tr>
<tr>
<td>Recovery data</td>
<td>Distributed</td>
<td>Distributed</td>
<td>Distributed</td>
<td>Distributed or local</td>
<td>Distributed or local</td>
<td>Distributed</td>
</tr>
<tr>
<td>Recovery protocol</td>
<td>Distributed</td>
<td>Distributed</td>
<td>Distributed</td>
<td>Local</td>
<td>Distributed</td>
<td>Distributed</td>
</tr>
<tr>
<td>Output commit</td>
<td>Not possible</td>
<td>Global coordination required</td>
<td>Global coordination required</td>
<td>Local decision</td>
<td>Global coordination required</td>
<td>Local decision</td>
</tr>
</tbody>
</table>

Table 1: A comparison between various flavors of rollback-recovery protocols.

Overview

- Pessimistic Logging is the surest way to avoid loss of progress, but it has high performance overheads which can best be mitigated by fast stable storage
- Coordinated Checkpointing USED TO be impractical because of low network bandwidths, but is actually rather attractive in the modern environment – it adds little overhead

Recovery Protocols

- Synchronous Recovery
  - All computation is stopped, information is collected and analyzed, and then finally rollback is performed across all processes
- Asynchronous Recovery
  - Processes branch into “instances” when they receive a rollback announcement; each instance then insures that processes communicating with it do not rely on the other instances’ determinants
Is a real system PWD?

- Three classes of system call:
  - Deterministic (getuid)
  - Nondeterministic once (gettimeofday)
  - Nondeterministic reexecute (setenv)
- Instruction counters used to find exactly when interrupts fired