Coordinated Checkpointing

Presented by Sarah Arnold
Agenda

- Goals
- Fault Tolerance
- Failure Recovery
- System Overview
- Coordinated Checkpointing
- Communication-Induced Checkpointing
- Logging
- Conclusions
Coordinated Checkpointing

Goals

- To recover the system after any type of fault has been introduced to the system and to minimize the amount of computation lost

- Hardware
- Software
- Processors
- Network
- Memory
- Disk

Diagram:
- Fault causes erroneous state
- Error leads to failure
- Recovery leads to valid state
Fault Tolerance

Fault Tolerance – a design that enables a system to continue operation, rather than failing completely, when some part of the system fails

- **Looking at problem from system perspective in terms of the state of the system being its “memory state”**
- **We know nothing of the application or outside world processes that may have introduced the error, but must still get the system back to a valid state**
Failure Recovery

- Failure Recovery – an attempt to put the system back into a valid state
  - **Backward Recovery:** Retreating back to an earlier state of the system
    - Operation-based: Logs of operations are maintained and replayed
    - State-based: Check-pointing particular states of the system as it evolves
  - **Forward Recovery:** Usually no previous state to retreat to; instead must fail into some forward condition
    - Messages sent to outside world are sent and cannot be retrieved: Imagine trying to recover Space Shuttle after liftoff!
System interacts with outside world as well as sends messages internally

- **System must be kept in a coherent state with the outside world process**
**Orphan Messages**

- **Orphan Message**: A message that is received but never sent (i.e. message \( m \) below); no sender can be identified
  - Due to the fact that, when restored back to their checkpoints, one part of the system is incoherent with another part of the system

- **Checkpoint**: Complete recorded state of the application

- **Failure**:

![Diagram](attachment:image.png)
Lost Messages

- If a process fails and has to recover to a previous state before it received a message, the message is lost.
  - **Sender might try and send again, but potential receiver doesn’t even know it had been sent already.**
In-/Consistent States

- When rolling back to a checkpoint, the system is in a **consistent state** if there are no orphan messages (see \( a \) below) and is in an **inconsistent state** if there are orphan messages (see \( b \) below).
In order to avoid orphan messages and rolling back to an inconsistent state, a failed process may trigger other processes to rollback as well – this is Domino Effect.

- **Goal is to checkpoint at most useful time/state**
- Consider the effect if Z failed after sending message $n$
Algorithm Considerations

- Output commit: when a message is sent to the outside world, there is no way to pull that message back; similarly, there may not be a way to reproduce a message from the outside world.
  - Therefore, the state of the system must be solid to ensure no failure past that point
  - Expense: Affects latency of message and additional checkpointing

- Garbage Collection: when can I get rid of older checkpoints?

- Stable Storage
  - All algorithms assume that the location of checkpointing data is on stable storage
Logging Elements

- **Determinant**: The information that must be logged that is needed to recover a message
  - How to record this depends on type of algorithm

- **Piecewise-Deterministic**
  - Postulates that all nondeterministic events that a process executes can be identified and the information needed to replay the events can be logged in its determinant
  - By logging and replaying the nondeterministic events in their exact order, a process can deterministically recreate its pre-failure state, even without a checkpoint
Recovery Algorithms

Rollback-Recovery

- checkpointing
  - uncoordinated
  - coordinated
  - communication-induced
    - blocking
    - non-blocking
    - model-based
    - index-based

- logging
  - pessimistic
  - optimistic
  - causal
Coordinated Checkpointing Protocol (Blocking)

- When a process takes a checkpoint, it engages a protocol to coordinate with other processes to also checkpoint:
  - Coordinator takes a checkpoint; broadcasts a message to all processes
  - Process receives this message and halts execution; takes tentative checkpoint
  - Coordinator receives acknowledgement from all processes; broadcasts commit message to end protocol
  - Process receives commit message, removes old permanent checkpoint and makes tentative checkpoint permanent
  - Processes resume execution
Coordinated Checkpointing Protocol (Blocking)

- Recovery line: guarantee that system will never have to go back to a state earlier than this line
  - \{x_1, y_1, z_1\} forms “recovery line”
  - Good for garbage collection
- Blocking: Application is paused and no messages can be in transit during checkpointing
Coordinated Checkpointing Notation

- Each message has a sequence number (an increasing counter) affixed to it by the system.
- When we checkpoint, we keep these vectors along with it.

- \( \text{last\_label\_rcvd}_x[Y] \) \hspace{1cm} Last label \( X \) received before checkpoint was from \( Y \)
- \( \text{last\_label\_sent}_x[Y] \) \hspace{1cm} Last label \( X \) sent before checkpoint was to \( Y \)
- \( \text{first\_label\_sent}_y[X] \) \hspace{1cm} First label \( Y \) sent after checkpoint was to \( X \)

\[ m.l \] (a message \( m \) and its label \( l \))
Coordinated Checkpointing Questions

- When to take a checkpoint?
  - Application specific
  - Balance the cost of taking the checkpoint against the amount of computation that you’re going to lose by not taking one and having to use an earlier one

- Checkpoint protocol
  - When should I do a checkpoint?
  - If I take a checkpoint, who else do I have to ensure also takes a checkpoint?
    - and...
      - When must I rollback?
      - If I rollback, who else must rollback?

- Answers are based on label vectors!
Coordinated Checkpointing Algorithm

(1) When must I take a checkpoint?
(2) Who else has to take a checkpoint when I do?

(1) When I (Y) have sent a message to the checkpointing process, X, since my last checkpoint:
\[ \text{last\_label\_rcvd}_X[Y] \geq \text{first\_label\_sent}_X[X] > s1 \]

(2) Any other process from whom I have received messages since my last checkpoint.
\[ \text{ckpt\_cohort}_X = \{Y \mid \text{last\_label\_rcvd}_X[Y] > s1 \} \]
Coordinated Checkpointing Algorithm

(1) When must I rollback?
(2) Who else might have to rollback when I do?

(1) When I, Y, have received a message from the restarting process, X, since X's last checkpoint.

$$last_{-}label_{-}rcvd_Y(X) > last_{-}label_{-}sent_X(Y)$$

(2) Any other process to whom I can send messages.

$$roll_cohort_Y = \{Z \mid Y \text{ can send message to } Z\}$$
Coordinated Checkpointing: Non-blocking Protocol

- Key issue with coordinated checkpointing:
  - Being able to prevent a process from receiving application messages that could make the checkpoint inconsistent

- Problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint upon receiving the first checkpoint-request message
Communication-Induced Checkpointing

- Avoids domino effect without coordinated checkpoints

- Processes take two kinds of checkpoints
  - Local: can be taken independently
  - Forced: must be taken to guarantee progress of recovery line
    - Piggyback protocol-specific information on each application message

- Follow application trends to make sure checkpoint is necessary
  - Z-paths and Z-cycles form patterns
Coordinated Checkpointing

Communication-Induced Checkpointing

- **Z-path**: sequence of messages in the interval between two checkpoints
  - \([m_1, m_2], [m_1, m_4], [m_3, m_2]\) and \([m_3, m_4]\)

- **Z-cycle**: Z-path that begins and ends within the same interval
  - \([m_5, m_3, m_4]\)
  - Makes checkpoint \(c_{2,2}\) useless
Logging

- Goal: Capture messages that are received and avoid orphan processes
  - **Always-no-orphans condition:** If any surviving processes depend on an event e, either the event is logged on stable storage or the process has a copy of e’s determinant.

- Uses checkpointing and logs

- Useful with applications that interact frequently with the outside world
  - **Enables process to repeat its execution without having to take expensive checkpoints before sending messages**

- Not susceptible to domino effect

- Piecewise determinism
  - **Rollback recovery protocol can identify all nondeterministic events (messages received, input from outside world, etc.) executed and logs the determinant; can recover a failed process and replay its execution as it occurred before the failure**
Logging

- Recoverable: a state interval is recoverable if there is sufficient information to replay the execution up to that point despite any future failures.

- Stable: a state interval is stable if the determinant of the nondeterministic event that started it is logged on stable storage.
  - **Recoverable is always stable, but opposite is not always true.**

- If P₁ and P₂ fail before logging m₅ and m₆? M₇ becomes an orphan message → Maximum Recoverable State: X, Y, Z
Coordinated Checkpointing

Pessimistic Logging

- Designed under assumption that a failure can occur after any nondeterministic event
  - Protocol logs determinant to stable storage before event is allowed to affect computation
- Periodic checkpoints are taken to aid in repeating execution
  - Application is restarted from most recent checkpoint and the logged determinants are used to recreate execution

Pros:
- Immediate output commit
- Restart from most recent checkpoint
- Recovery limited to failed processes
  - Always-no-orphans: if a surviving process depends on an event, either the event is logged or that process has a copy of the event’s determinant
- Simple garbage collection

Con:
- Performance Penalty due to synchronous logging
Optimistic Logging

- Log determinants asynchronously
  - Optimistic assumption that logging will complete before a failure occurs
- Determinants are kept in a volatile log that is periodically flushed to stable storage
  - No blocking necessary (less overhead)
  - More complicated recovery, garbage collection, and slower output commit
- Does not implement always-no-orphans
  - Permits temporary creation of orphan processes
  - Upon a failure, dependency information is used to recover latest global state of pre-failure execution in which no process is an orphan
- Great for failure free executions
Causal Logging

- Failure-free performance from optimistic + allowing processes to commit output independently and always-no-orphans from pessimistic
- Determinants of all causally preceding events are logged to stable storage or are available locally
- Limits rollback to most recent checkpoint
  - Reduces overhead of storage and work at risk
- Piggybacks on each message information about preceding messages
## Rollback-Recovery Protocols

<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><strong>PWD assumed?</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Checkpoint/process</strong></td>
<td>Several</td>
<td>1</td>
<td>Several</td>
<td>1</td>
<td>Several</td>
<td>1</td>
</tr>
<tr>
<td><strong>Domino effect</strong></td>
<td>Possible</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Orphan processes</strong></td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
<td>No</td>
</tr>
<tr>
<td><strong>Rollback extent</strong></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><strong>Recovery data</strong></td>
<td>Distributed</td>
<td>Distributed</td>
<td>Distributed or local</td>
<td>Distributed or local</td>
<td>Distributed</td>
<td>Distributed</td>
</tr>
<tr>
<td><strong>Recovery protocol</strong></td>
<td>Distributed</td>
<td>Distributed</td>
<td>Distributed</td>
<td>Local</td>
<td>Distributed</td>
<td>Distributed</td>
</tr>
<tr>
<td><strong>Output commit</strong></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>
Conclusions

- Issues at hand: Piecewise determinism, performance overhead, storage overhead, ease of output commits, ease of garbage collection, ease of recovery, avoiding domino effect and orphan processes

- Checkpointing:
  - Coordinated: simplifies recovery and garbage collection, overall good performance
  - Uncoordinated: suffers from potential domino effects and complicates recovery
  - Communication-Induced: no domino effect or coordination, but nondeterministic nature complicates garbage collection and degrades performance

- Logging: Natural choice for applications that often interact with outside world
  - Pessimistic: simplifies recovery and output commit; simple and robust
  - Causal: reduces overhead, fast output commit and orphan-free recovery
  - Optimistic: reduces overhead more than Causal, but complicates recovery by increasing extent of future rollbacks
Questions?

Thank you!
References

- Checkpointing-Recovery, Dr. Dennis Kafura