Coordinated Checkpointing

- Uncoordinated checkpointing may lead to domino effect or to livelock
- Example:
  - $P$ wants to take a checkpoint at $CP_3$
  - This checkpoint will include the message $m$
  - Coordination needed to prevent $m$ from being orphaned - $Q$ must checkpoint at the same time
A Coordinated Checkpointing Algorithm

- Two types of checkpoints - tentative and permanent
- Process P records its state in a tentative checkpoint
- P then sends a message to set \( \hat{P} \) - all processes from whom it received a message since its last checkpoint
  - telling Q the last message, \( m_{qp} \), that P received from Q before the tentative checkpoint
- Q is asked to take a tentative checkpoint recording sending \( m_{qp} \) (if not already included in checkpoint)
- If all processes in \( \hat{P} \) that need to, confirm taking a checkpoint as requested, then all the tentative checkpoints are converted to permanent checkpoints
- Otherwise - P and all others in \( \hat{P} \) abandon their tentative checkpoints
- This process can set off a chain reaction of checkpoints among processes in \( \hat{P} \)

Time-Based Synchronization

- Orphan messages cannot happen if each process checkpoints at the same time
- Time-based synchronization - processes are checkpointed at previously agreed times
- Not enough to avoid orphan messages - clock skews and message communication times are not zero
- Example:
  - Each process is checkpointing at local time 1100
  - Skew between the two clocks - PO checkpoints much earlier (in real time) then P1
  - As a result, PO sends a message to P1 after its checkpoint, received by P1 before its checkpoint - message is potentially orphan
Preventing Creation of an Orphan Message

- Suppose skew between any two clocks is bounded by $\delta$
- Each process checkpoints when its local clock reads $\tau$
- Process remains silent during $[\tau, \tau+\delta]$ (local clock)
- Guarantees that all other processes took a checkpoint
- If inter-process message delivery time has a lower bound $\varepsilon$ - process needs to remain silent during a shorter interval $[\tau, \tau+\delta-\varepsilon]$
- If $\varepsilon > \delta$, this interval is of zero length - no need for process to remain silent

Different Method of Prevention

- Suppose $m$ received by $P_1$ when its clock reads $\tau$
- $m$ must have been sent (by $P_0$) no later than $\varepsilon$ earlier - before $P_1$'s clock read $\tau-\varepsilon$
- Since clock skew $\leq \delta$, at this time, $P_0$'s clock should have read at most $\tau-\varepsilon+\delta$
- If $\tau-\varepsilon+\delta < \tau$, sending of $m$ would be recorded in $P_0$'s checkpoint - $m$ cannot be an orphan
- A message $m$ received by $P_1$ when its clock reads at least $\tau-\delta+\varepsilon$ cannot be an orphan
- Orphan messages can be avoided by $P_1$ not using and not including in its checkpoint at $\tau$ any message received during $[\tau-\delta+\varepsilon, \tau]$ ($P_1$'s clock) until after taking its checkpoint at $\tau$
Diskless Checkpointing

- By avoiding disk writes, checkpointing can be faster
- Main memory is volatile and unsuitable for storing a checkpoint
- With extra processors, we can permit checkpointing in main memory
- Have redundant processors using RAID-like techniques to deal with failure
- Best used as one level in a two-level checkpointing

RAID-like Diskless Checkpointing

- Example: 5 executing and 1 extra, processors
- Executing processor stores checkpoint in memory; extra processor stores parity of these checkpoints
- If an executing processor fails, its checkpoint can be reconstructed from remaining five plus parity
- Inter-processor network must have enough bandwidth for sending checkpoints
- If all executing processors send checkpoints to checkpointing processor to calculate parity - potential hotspot
- Solution: Distribute the parity computations

C=Checkpointing Processor
Labels in non-root nodes indicate processor computing the parity of its descendants in the subtree.
Message Logging

♦ To continue computation beyond latest checkpoint, recovering process may need all messages it received since then, played back in original order

♦ In coordinated checkpointing - each process can be rolled back to its latest checkpoint and restarted: messages will be resent during reexecution

♦ To avoid overhead of coordination, logging messages is an option

♦ Two approaches to message logging:
  ♦ Pessimistic logging - ensures that rollback will not spread; if a process fails, no other process will need to be rolled back to ensure consistency
  ♦ Optimistic logging - a process failure may trigger rollback of other processes as well

Pessimistic Message Logging

♦ Simplest approach - receiver of a message stops whatever it is doing when it receives a message, logs it onto stable storage and resumes execution

♦ Recovering a process from failure - roll it back to latest checkpoint and play back messages it received since that checkpoint, in right order

♦ No orphan messages will exist - every message will either have been received before the latest checkpoint or explicitly saved in message log

♦ Rolling back one process will not trigger rollback of any other process
Sender-Based Message Logging

- Logging messages into stable storage can impose a significant overhead
- Against isolated failures sender-based message logging can be used
  * Sender of a message records it in a log - when required, log is read to replay the message
  * Each process has send- and receive-counters, incremented every time the process sends or receives a message
  * Each message has a Send Sequence Number (SSN) - value of send-counter when it is transmitted
  * A received message is allocated a Receive Sequence Number (RSN) - value of receive-counter when it was received
  * Receiver also sends out an ACK to sender, including RSN allocated to message
  * Upon receiving this ACK, sender acknowledges the ACK in a message to receiver

Sender-Based Message Logging - Cont'd

- Between time receiver receives message and sends its ACK, and when it receives sender's ACK of its own ACK, receiver forbidden to send messages to other processes - essential to maintaining correct functioning upon recovery
- A message is said to be fully-logged when sending node knows both its SSN and its RSN; it is partially-logged when it does not yet know its RSN
- When a process rolls back and restarts computation from latest checkpoint, it sends out to other processes a message listing SSN of their latest message that it recorded in its checkpoint
- When this message is received by a process, it knows which messages are to be retransmitted, and does so
- Recovering process has to use these messages in same order as they were used before it failed - easy to do for fully-logged messages, since their RSNs are available, and they can be sorted by this number
Partially-logged Messages

- Remaining problem - partially-logged messages, whose RSNs are not available
- Sent out, but their ACK never received by sender
- Receiver failed before message could be delivered to it, or it failed after receiving message but before it could send out ACK
- Receiver forbidden to send out messages of its own to other processes between receiving message and sending ACK
- As a result, receiving partially-logged messages in a different order the second time cannot affect any other process in the system - correctness is preserved
- This approach is only guaranteed to work if there is at most one failed node at any time

Optimistic Message Logging

- Lower overhead than pessimistic logging
- Recovery from failure is much more complex
- Optimistic logging is of theoretical interest only
- Messages are written into a volatile buffer which, at a suitable time, is copied into stable storage
- Process execution is not disrupted - logging overhead is very low
- Upon failure, contents of buffer can be lost
- Multiple processes will have to be rolled back
  * Need a scheme to handle this situation
Staggered Checkpointing

♦ Some algorithms can cause large number of processes to take checkpoints at nearly same time - can cause congestion at disks or network or both
♦ Two approaches to solve problem:
  * (1) Write checkpoint into a local buffer, then stagger writes from buffer to stable storage
    » Assuming a buffer of sufficiently large capacity
  * (2) Try staggering checkpoints in time
♦ Consistency not guaranteed - orphan messages possible
♦ Can be avoided by a coordinating phase - each process logs in stable storage all messages it sent since its previous checkpoint - message-logging phase of processes will overlap in time
♦ If volume of messages is less than size of individual checkpoints - disks and network will see reduced surge

Recovery From Failure

♦ If a process fails, it can be restarted after rolling it back to its last checkpoint and all messages stored in log played back
♦ This combination of checkpoint and message log is called a logical checkpoint
♦ Staggered checkpointing algorithm guarantees that all logical checkpoints form a consistent recovery line
♦ Algorithm for a distributed system with n processors P0,P1,...,Pn-1 consists of two phases:
  * Checkpointing phase, and
  * Message-logging phase
Two Phases of Staggering Algorithm

/* Checkpointing Phase */

for (i=0; i<=m-1; i++) {
    P_i takes a checkpoint.
    P_i sends a message to P_{i+1} ordering the latter to take a checkpoint.
}

/* Message Logging Phase */

if (no previous marker message was received in this route by P_i) then {
    P_i sends a marker message on each of its outgoing channels.
    P_i logs all messages received by it after the preceding checkpoint
    and before the marker was received.
} else

P_i updates its message log by adding all the messages received by it
since the last message log and before the marker was received.

❖ A message from P_{n-1} asking P_0 to checkpoint is the
cue for P_0 to initiate second phase - it sends out a
marker message on each of its outgoing channels
❖ A process receiving a marker message starts phase 2

Example of Staggering Algorithm - Phase One

♦ P_0 takes a checkpoint and sends take_checkpoint
   order to P_1
♦ P_1 sends such an order to P_2 after taking its own
   checkpoint
♦ P_2 sends a take_checkpoint order back to P_0
♦ At this point, each process has taken a checkpoint
   and second phase can begin
Example - Phase 2

- **P0** sends **message_log** to **P1** and **P2** - logging messages they received since last checkpoint
- **P1** and **P2** send out similar **message_log** orders
- Each time such a message is received - the process logs the messages
- If it is the first time such a **message_log** order is received by it - the process sends out marker messages on each of its outgoing channels

Recovery

- **Assumption** - given checkpoint and messages received, a process can be recovered
- We may have orphan messages with respect to the physical checkpoints taken in first phase
- Orphan messages will not exist with respect to the latest (in time) logical checkpoints that are generated using the physical checkpoint and the message log
Checkpointing in Shared-Memory Systems

♦ A variant of CARER for shared-memory bus-based multiprocessors - each processor has its own cache
♦ Change algorithm to maintain cache coherence among multiple caches
♦ Instead of single bit marking a line as unchangeable, we have a multi-bit identifier:
♦ A checkpoint identifier, $C_{id}$ with each cache line
♦ A (per processor) checkpoint counter, $C_{count}$, keeping track of current checkpoint number

Shared Memory - Cont.

♦ To take a checkpoint, increment the counter
♦ A line modified before will have its $C_{id}$ less than the counter
♦ When a line is updated, set $C_{id} = C_{count}$
♦ If a line has been modified since being brought into cache and $C_{id} < C_{count}$, the line is part of checkpoint state, and is unwritable
♦ Any writes into such a line must wait until line is first written into main memory
♦ If counter has $k$ bits, it rolls over to 0 after reaching $2^{k-1}$
Bus-Based Coherence Protocol

♦ A cache coherence algorithm which does not take account of checkpointing:
♦ All traffic between caches and memory must use bus - all caches can watch traffic on bus
♦ A cache line can be in one of following states: invalid, shared unmodified, exclusive modified, and exclusive unmodified
♦ Exclusive - this is the only valid copy in any cache
♦ Modified - line has been modified since it was brought into cache from memory

Bus-Based Coherence Protocol - Cont’d

♦ If processor wants to update a line in shared unmodified state, it moves into exclusive modified state
♦ Other caches holding same line must invalidate their copies - no longer current
♦ When in exclusive modified or exclusive unmodified states, another cache puts out a read request on bus, this cache must service that request (only current copy of that line)
♦ Byproduct - memory is also updated if necessary
♦ Then, move to shared unmodified
♦ Write miss, line into cache - exclusive modified
Bus-Based Coherence and Checkpointing Protocol

♦ Modifying for checkpointing:
  ♦ Original exclusive modified state now splits into two states:
    ♦ Exclusive modified
    ♦ Unmodifiable
  ♦ When a line becomes part of the checkpoint, it is marked unmodifiable to keep it stable
  ♦ Before it can be changed, it must be copied to memory for use in the event of a rollback

Directory-Based Protocol

♦ A directory is maintained centrally which records status of each line
♦ Regard this directory as being controlled by some shared-memory controller
♦ This controller handles all read and write misses and all other operations which change line state
♦ Example: If a line is in exclusive unmodified state and cache holding that line wants to modify it, it notifies controller of its intention
♦ Controller can change state to exclusive modified
♦ Very simple to implement this checkpointing scheme atop such a protocol
Checkpointing in Real-Time Systems

♦ A real-time system has deadlines
  * In a hard real-time systems, missed deadlines can be costly
  * In a soft real-time systems, missed deadlines lower quality of service but are not catastrophic
  * Application determines whether system is hard or soft
♦ Performance of a real-time system is related to probability that system meets all its critical deadlines
♦ Goal of checkpointing in a real-time system is to maximize this probability
  * Not to minimize mean execution time
♦ Checkpointing in real-time systems may even increase average execution time while decreasing probability of missing a deadline

Checkpointing in Real-Time Systems – Analytical Model

♦ Calculate density function of task execution time
♦ Place a checkpoint after every $T_{\text{ex}}$ seconds of useful work
♦ Each checkpoint takes $T_{\text{ov}}$ seconds in overhead
♦ Simple system – $T_{\text{lt}} = T_{\text{ov}}$
♦ Transient faults – constant rate $\lambda$ per second
♦ Repair time of a transient failure – $T_{\text{r}}$
♦ $f_{\text{int}}(t)$ – probability density function of time between successive initiations of checkpoints
♦ Proceed as before by considering two cases:
  ♦ Case I: No failures during interval $T_{\text{ex}}+T_{\text{ov}}$
  ♦ Case II: There is at least one failure
Analytical Model – Inter-Checkpoint Interval

♦ Case I: interval is of length $T_{ex} + T_{ov}$; probability of
$e^{-\lambda(T_{ex} + T_{ov})}$

♦ Case II: suppose 1st failure occurs $\tau$ seconds into interval
  * Lose all $\tau$ seconds of computation
  * It takes $T_r$ seconds to recover
  * Following $\tau + T_r$, interval starts again

♦ Probability of 1st failure happening during $[\tau, \tau + dt]$ is
$\lambda e^{-\lambda \tau} dt$

♦ Execution time can never be less than $T_{ex} + T_{ov}$
♦ Cannot fall in interval $(T_{ex} + T_{ov}, T_{ex} + T_{ov} + T_r)$ because a failure takes $T_r$ seconds to recover from
♦ It will be exactly equal to $T_{ex} + T_{ov}$ in the (common) case that there is no failure

Density Function of Inter-Checkpoint Interval

$$f_{int}(t) = \begin{cases} e^{-\lambda(T_{ex} + T_{ov})/2} (t - (T_{ex} + T_{ov})) & \text{if } t = T_{ex} + T_{ov} \\ 0 & \text{if } t \neq T_{ex} + T_{ov} \text{ and } T_{ex} + T_{ov} + t < T_{ex} + T_{ov} + T_r \\ \int_{T_{ex} + T_{ov}}^{\infty} e^{-\lambda t} \delta(t - (t + T_r)) dt & \text{if } t > T_{ex} + T_{ov} + T_r \end{cases}$$

♦ Probability of common case $t = T_{ex} + T_{ov}$ represented by a Dirac Delta function at that point of magnitude $e^{-\lambda(T_{ex} + T_{ov})}$

♦ $\delta(t)$ has the property that for any density function $f(t)$ and some constant $a$, $\int_{-\infty}^{\infty} f(t) \delta(t - a) dt = f(a)$

♦ Equation for $f_{int}(t)$ can be solved numerically

♦ The density function of the total execution time for $N$ checkpoints is the $(N+1)$-fold convolution of $f_{int}(t)$

♦ For a deadline $t_d$, probability of missing it is
$$P_{missing} = \int_{0}^{t_d} f_{exec}(t) dt$$
Example - Miss Probability

- $T = 0.15$ ; $\lambda = 0.001$ ; $Tr = 0.1$
- Probability of missing the deadline for different numbers of checkpoints ($n$) :

![Graph showing the probability of missing the deadline for different numbers of checkpoints.]

$T_{ov}; 0.015$ $T_{ov}; 0.025$

Example - Execution Time

- $T = 0.15$ ; $\lambda = 0.001$ ; $Tr = 0.1$
- Average execution time as a function of number of checkpoints

<table>
<thead>
<tr>
<th>Number of checkpoints $n$</th>
<th>$T_{ov} = 0.015$</th>
<th>$T_{ov} = 0.025$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.180</td>
<td>0.200</td>
</tr>
<tr>
<td>2</td>
<td>0.195</td>
<td>0.225</td>
</tr>
<tr>
<td>3</td>
<td>0.210</td>
<td>0.250</td>
</tr>
<tr>
<td>4</td>
<td>0.225</td>
<td>0.275</td>
</tr>
<tr>
<td>5</td>
<td>0.240</td>
<td>0.300</td>
</tr>
<tr>
<td>6</td>
<td>0.255</td>
<td>0.325</td>
</tr>
<tr>
<td>7</td>
<td>0.270</td>
<td>0.350</td>
</tr>
<tr>
<td>8</td>
<td>0.285</td>
<td>0.375</td>
</tr>
</tbody>
</table>

- Average execution time as a function of number of checkpoints
Trade-off

♦ In the example:
♦ Expected execution time gets worse as number of checkpoints increases – results from very low probability of failure during execution
♦ Probability of missing a deadline:
  ♦ For tight deadlines, having more checkpoints is worse
  ♦ For farther away deadlines, a greater number of checkpoints is better
♦ For T_{ov}=0.015:
  ♦ If deadline is 0.5, 6 checkpoints are better than 3
  ♦ If deadline is 0.3, 3 checkpoints are better than 6
♦ Exercise: Explain difference in results between T_{ov}=0.015 and T_{ov}=0.025

Other Uses of Checkpointing

♦ (1) Process Migration: Migrating a process from one processor to another means moving checkpoint, and resuming computation on new processor - can be used to recover from permanent or intermittent faults
  ♦ Nature of checkpoint determines whether new processor must be same model and run same operating system
♦ (2) Load-balancing: Better utilization of a distributed system by ensuring that the computational load is appropriately shared among the processors
♦ (3) Debugging: Core files are dumped when a program exits abnormally – these are essentially checkpoints - debuggers use core files in the debugging process
♦ (4) Snapshots: Observing the program state at discrete epochs - deeper understanding of program behavior