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 - P0 checkpoints much earlier (in real time) than P1
  - As a result, P0 sends a message to P1 after its checkpoint, received by P1 before its checkpoint - message is potentially orphan

Different Method of Prevention

- Suppose \( m \) received by P1 when its clock reads \( \tau \)
- \( m \) must have been sent (by P0) no later than \( \varepsilon \) earlier - before P1's clock read \( t - \varepsilon \)
- Since clock skew \( \leq \delta \), at this time, P0's clock should have read at most \( t - \varepsilon + \delta \)
- If \( t - \varepsilon + \delta < \tau \), sending of \( m \) would be recorded in P0's checkpoint - \( m \) cannot be an orphan
- A message \( m \) received by P1 when its clock reads at least \( \tau - \delta + \varepsilon \) cannot be an orphan
- Orphan messages can be avoided by P1 not using and not including in its checkpoint at \( \tau \) any message received during \( [\tau - \delta + \varepsilon, \tau] \) (P1'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 method.

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 hotspots.
- Solution: Distribute the parity computations.

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.

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 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
  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:
  1. Checkpointing phase, and
  2. Message-logging phase

Example of Staggering Algorithm

- PO takes a checkpoint and sends take_checkpoint order to P1
- P1 sends such an order to P2 after taking its own checkpoint
- P2 sends a take_checkpoint order back to PO
- At this point, each process has taken a checkpoint and second phase can begin
**Example - Phase 2**

- PO 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, Cid with each cache line
   - A (per processor) checkpoint counter, Ccount, keeping track of current checkpoint number

**Shared Memory - Cont.**

- To take a checkpoint, increment the counter
- A line modified before will have its Cid less than the counter
- When a line is updated, set Cid = Count
- If a line has been modified since being brought into cache and Cid < 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
- 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

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