CSCE 5760: Design or Fault Tolerance

NO CLASS on Nov 26
Two reading assignments for Dec 3
Project Presentations Dec 10
Final ? (Dec 12, 1:30-3:30pm)

Review

Checkpointing
Who does the checkpointing
system (OS)
Application
What to checkpoint
total systems
data
state
Where to store the checkpoints
local or remote
How often to checkpoint
cost of checkpointing
cost of recovery
cost due to loss on a fault

CSCE5760  Nov 19, 2019

How often should we checkpoint?
How much to checkpoint?
How long it takes to recover?

A simple model

Checkpoint represents state of system at \( t_0, t_3, t_6 \)
If a failure occurs in \([t_3, t_5]\) - checkpoint is useless - system must roll back to previous checkpoint \( t_0 \)
\( T_r \) - average recovery time - time spent in a faulty state plus time to recover to a functional state
\( E_{int} \) - amount of time between completions of two consecutive checkpoints
\( T_{ex} \) - amount of time spent executing application during this time
\( T \) – program execution time ; \( N \) uniformly placed checkpoints
\( T_{ex} = \frac{T}{N+1} \)
CSCE 5760: Design or Fault Tolerance

CASE 1: If no errors: \( E_{\text{int}} = T_{\text{ex}} + T_{\text{ov}} \)

We need to derive \( E_{\text{int}} \) when there are errors – we will assume at most one failure between checkpoints

Case II: Failure occurs \( \tau \) hours into \( T_{\text{ex}} + T_{\text{ov}} \)

We lose all work done after preceding checkpoint was taken = \( T_{\text{lt}} - T_{\text{ov}} + \tau \)

It takes an average of \( T_{\text{r}} \) hours to recover

Total amount of additional time = \( \tau + T_{\text{lt}} - T_{\text{ov}} + T_{\text{r}} \)

Average value of \( \tau = \frac{(T_{\text{ex}}+T_{\text{ov}})}{2} \)

Average additional time = \( \frac{(T_{\text{ex}}+T_{\text{ov}})}{2} + T_{\text{lt}} - T_{\text{ov}} + T_{\text{r}} \)

If we assume errors are based on exponential distribution with a rate = \( \lambda \)

Probability of (Case 1) no error during \( T_{\text{ex}}+T_{\text{ov}} \) is given by

\[ e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})} \]

So, the contribution to execution time in between checkpoints due to no errors is given by

\[ (T_{\text{ex}} + T_{\text{ov}})e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})} \]

What is the probability of 1 error (we will assume the probability of more than one error is negligible)

\[ (1 - e^{-\lambda(T_{\text{ex}}+T_{\text{ov}})}) \]

Let us compute the contribution to execution time, in between checkpoints when one error
To evaluate sensitivity of the execution time between two checkpoints with respect to $T_{lt}$ and $T_{ov}$, we find first order derivatives.

Combining the two cases, execution time in between checkpoints is

$$E_{int} \approx \frac{3}{2} T_{ex} + \frac{T_{ov}}{2} + T_{r} + T_{lt} - \left(\frac{T_{ex}}{2} + T_{r} + T_{lt} - \frac{T_{ov}}{2}\right) e^{-\lambda(T_{ex} + T_{ov})}$$

To evaluate sensitivity of the $E_{int}$ with respect to $T_{lt}$ and $T_{ov}$, we find first order derivatives.

$$\frac{dE_{int}}{dT_{ov}} > \frac{dE_{int}}{dT_{lt}}$$

$T_{ov}$ plays more significant role.

Note that if $T$ is the total execution time, $T_{ex} = T/(N+1)$

And we derived $E_{int}$ in terms of $T_{ex}$.

So we need to find how many checkpoints? What is the optimal value for $N$?

Let us define a figure of merit which is $(E_{int}/T_{ex}) - 1$.

Let us simplify the following – since we assume only one failure in between checkpoints.

$$e^{-\lambda(T_{ex} + T_{ov})} = 1 - \lambda(T_{ex} + T_{ov})$$

Error probability is proportional to time.

Using this value in $E_{int}$ and solving for figure of merit,

$$\eta = \frac{\frac{3}{2} T_{ex} + \frac{T_{ov}}{2} + T_{r} + T_{lt} - \left(\frac{T_{ex}}{2} + T_{r} + T_{lt} - \frac{T_{ov}}{2}\right) (1 - \lambda(T_{ex} + T_{ov}))}{T_{ex}} - 1$$

$$= \frac{(T_{ex} + T_{ov}) \left[1 + \lambda(\frac{T_{ov}}{2} + T_{r} + T_{lt} - \frac{T_{ex}}{2})\right]}{T_{ex}} - 1$$

CSCE5760 Nov 19, 2019
CSCE 5760: Design or Fault Tolerance

To find optimal figure of merit we find derivative with respect to $T_{ex}$
And then solving for $T_{ex}$

$$ T_{ex}^{opt} = \sqrt{\frac{2T_{ov}}{\lambda} + 2T_{ov} \left( T_r + T_{lt} - \frac{T_{ov}}{2} \right)} $$

$N_{opt}$ is the optimum number of checkpoints

Checkpointing in distributed systems
Why is this any different from regular systems?

How do we coordinate checkpoints at all nodes?
May not be possible to create a correct global time

If different nodes checkpoint at (slightly) different times
$\Rightarrow$ may cause problems with messages sent/received between checkpoints

Consistent Checkpoints
Checkpoint sets representing consistent system states:

Consider the following checkpoint pairs

$\{CP_1, CQ_1\}$ neither processor has any knowledge of the message
$\{CP_2, CQ_1\}$ P has sent the message but Q did not receive
$\Rightarrow$ may require P to send the message again or assume message will arrive later
$\{CP_2, CQ_2\}$ Both P and Q have knowledge of the message
$\{CP_1, CQ_2\}$ P does no have any knowledge of sending message but Q received it inconsistent (message is considered an orphan message)
CSCE 5760: Design or Fault Tolerance

Recovery point
⇒ feasible recovery points

Consistent set of checkpoints forms a recovery line
- can roll system back to them and restart from there

Example: \{CP1, CQ1\}
  - Rolling back P to CP1 undoes sending of m
  - Rolling back Q to CQ1 means: Q has no record of m
  - Restarting from CP1, CQ1, P will again send m

Example: \{CP2, CQ1\}
  - Rolling back P to CP2 means: it will not retransmit m
  - Rolling back Q to CQ1: Q has no record of receiving m
  - Recovery process has to be able to play back m to Q
    - Adding it to checkpoint of P, or
    - Have a separate message log which records everything received by Q

Recovery in distributed systems is complicated by message exchanges
- If a message is sent after a checkpoint ⇒ the message will needs retransmission
- If a message is received before checkpointing ⇒ new messages after recovery may lead to “orphan” message

When a process takes a checkpoint, it request other processors from which it received a message to checkpoint also – COORDINATED checkpoints

The checkpoint initially is marked as “tentative” until all other processors also checkpointed and acknowledged.

Until the acknowledgement is received, do not send any new messages.

This may lead to cascading checkpoints.
CSCE 5760: Design or Fault Tolerance

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 P - all processes from whom it received a message since its last checkpoint
  - informing Q the last message, m_{pq}, that P received from Q before the tentative checkpoint
  - Q is asked to take a tentative checkpoint recording sending m_{pq} (if not already included in checkpoint)
- If all processes in 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 P abandon their tentative checkpoints
- This process can set off a chain reaction of checkpoints among processes in

CSCE 5760: Design or Fault Tolerance

Time based coordination

Agree on times when a checkpoint is taken
Say on the hour. But we may have differences in the local clocks

All processors checkpoint at 1100. P_0 checkpointed and sends a message
P_1 clock has not reached 1100 when the message is received
Can we timestamp message with sender time?
CSCE 5760: Design or Fault Tolerance

Preventing creation of orphaned messages

- 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

Message logging

Store all messages you sent so that they can be played back

Two types of logging

- Pessimistic – ensure that a rollback will not cause domino effect
  - if a processor fails, no other processor needs to rollback
- Optimistic – a failure may trigger rollback in other processors
  - not practical since only limited amount is logged and requires complex rollback

Simplest pessimistic logging

- A processor immediately logs the messages it receives
  - all processing must stop when a message is received until logged
- If the processor fails, it checks the message logs and replays them in right order (during logs, need to record order of messages)
- No other processor needs to rollback
  - But may incur overhead since need to stop processing on each message receipt
Another method
Each processor maintains two counters
Send count: incremented every time a new message is sent
Receive count: incremented every time a new message is received

Sender includes the send count with each new message

When a processor receives a message, it acknowledges the sender with its “Receive Count” for the message
And sender acknowledges the acknowledgement

A receiver will NOT send new messages until it receives the ack of the ack.

Sender logs all sent messages with send count and receive counts

So when a process rolls back, it knows the sent number of the last message received from all processors and notifies senders
Other processors check their message logs, and send any messages along with receive counts

Staggered checkpointing – to schedule checkpoints to shared disk

When a processor Pᵢ takes a checkpoint, it also sends messages to other processors from which it received a message to checkpoint.
This continues – the other processors in turn send messages to checkpoint

After all processors checkpoint, messages received since last checkpoint are logged
Log messages are sent to processors

Checkpointing using caches in shared memory systems

Read: “Fault-tolerance using cache coherent distributed shared memory systems” by D. Hecht, K. Kavi, R. Gaede
CSCE 5760: Design or Fault Tolerance

Rollback vs Roll Forward

Normally, when we detect a failure, we rollback to previous checkpoint and reexecute

What does it mean Roll Forward?

Consider a fault tolerant system with two processors

![Diagram](image)

B failed before checkpoint \( t_1 \)
A is fault free

Should we rollback both A and B?
Or let A proceed but rollback B?
Consider a variation

Consider a system with a spare or backup
The spare will use B’s checkpoint at \( t_0 \) and re-execute B’s computation
Let A and B continue execution, checkpoint at time \( t_1 \)
S also computes a checkpoint corresponding to \( t_1 \)
Compare S’s checkpoint with those of A and B
If S detects that B’s checkpoint did not match with that of B
So, we detect a failure of B. We can swap B out and continue Spare
with A’s checkpoint at time \( t_2 \)
Checkpointing in real-time systems
Meeting deadline vs error recovery
Probability of successfully completing a task within a deadline
How do we define deadlines that include checkpoint overhead?

Textbook analyzes real-time systems to obtain the probability of success vs number of checkpoints
The math is very complicated and in many cases we use numerical solutions
Example with the following: Execution time = 0.15, error rate (λ) = 0.001
Recovery time $T_r=0.1$
$T_{ov}$ 0.015 or 0.025

For different deadline values, plot probability of successful completion

$T_{ov} = 0.015$
$T_{ov} = 0.025$
Somewhat related to this

Performability measures
Degraded performance in presence of failures

For real-time systems, a failure may cause delays
retry, checkpoint etc

In some cases, a task may produce different levels of quality solutions with different amounts of execution

We can increase the probability of success when the difference between execution time $T$ and Deadline $D$ is large. We call this difference as "slack time"

We can derive formulas to relate how much slack time we need for different probabilities of successes, for different failure rates.

Consider the probability distribution for a task to complete in time $T$ under different failure rates.
Now, if we have a set of tasks comprising a real-time system we define task graphs to show dependency among the tasks.

Each task would have a probability of completion behavior

Then we can construct a model to find how much slack time do we need for the entire set of task can complete within a deadline

Petri nets and stochastic Petri nets

- A Petri net (PN) is a bipartite directed graph consisting of two kinds of nodes: places and transitions

  - Places typically represent conditions within the system being modeled
  - Transitions represent events occurring in the system that may cause change in the condition of the system
  - Arcs connect places to transitions and transitions to places. But never an arc from a place to a place or from a transition to a transition

A PN is a 5-tuple \((P,T,I,O,M)\)

\[
\begin{align*}
P & : \text{set of places} \\
T & : \text{set of transitions} \\
I & : \text{input arcs} \\
O & : \text{output arcs} \\
M & : \text{initial marking}
\end{align*}
\]
CSCE 5760: Design or Fault Tolerance

- Input arcs are directed arcs drawn from places to transitions, representing the conditions that need to be satisfied for the event to be activated.

- Output arcs are directed arcs drawn from transitions to places, representing the conditions resulting from the occurrence of the event.

- Input places of a transition are the set of places that are connected to the transition through input arcs.

- Output places of a transition are the set of places to which output arcs exist from the transition.

Tokens are dots (or integers) associated with places; a place containing tokens indicates that the corresponding condition holds.

Marking of a Petri net is a vector listing the number of tokens in each place of the net: \( M = (n_1, n_2, \ldots, n_P) \), where \( P \) is the number of places.

When input places of a transition have the required number of tokens, the transition is enabled.

An enabled transition may fire (event occurs) taking a specified number of tokens from each input place and depositing a specified number of tokens in each of its output places.
Example of a PN

- p1 – resource idle
- p2 – resource busy
- t1 – task arrives
- t2 – task completes

A 2-processor failure/repair model
Petri nets can be used to describe concurrency, mutual execution.

A token in P1 leads to t1 firing and places tokens in both p2 and p3.

Another token in t5 requires tokens in both p4 and p5.

\[
P = \{p_1, p_2, p_3, p_4, p_5\} \\
T = \{t_1, t_2, t_3, t_4, t_5\} \\
I(t_1) = \{p_1\} \quad \text{\text{O}}(t_1) = \{p_2, p_3\} \\
I(t_2) = \{p_2\} \quad \text{\text{O}}(t_2) = \{p_4\} \\
I(t_3) = \{p_3\} \quad \text{\text{O}}(t_3) = \{p_4\} \\
I(t_4) = \{p_4\} \quad \text{\text{O}}(t_4) = \{p_3\} \\
I(t_5) = \{p_4, p_5\} \quad \text{\text{O}}(t_5) = \{p_1\} \\
M_0 = (1, 0, 0, 0, 0)
\]
Petri nets can be used to Producer with buffers

Mutual Exclusion

Markings, Transitions and Reachability Analysis

- A marking is reachable from another marking if there exists a sequence of transition firings starting from the original marking that result in the new marking.

- The reachability set of a PN is the set of all markings that are reachable from its initial marking.
An Example of Markings and Reachability

How to generate the markings
Stochastic Petri Nets

- Petri nets are extended by associating time with the firing of transitions, resulting in timed Petri nets.

- A special case of timed Petri nets is stochastic Petri net (SPN) where the firing times are considered to be random variables with exponential distributions.

The marking process is mapped into a continuous time Markov chain (CTMC).

The state space of the CTMC is isomorphic to the reachability graph of the PN.
From Stochastic Petri net to Continuous Markov Process

From SPN to CTMC  Mutual Exclusion Exclusion
CSCE 5760: Design or Fault Tolerance