Distributed Algorithms

Faults and Recovery

Ludovic Henrio
CNRS - projet OASIS
ludovic.henrio@inria.fr

Sources:
- A survey of rollback-recovery protocols in message-passing systems (Elnozahy, Alvisi, Wang and Johnson)
- Distributed systems (Tanenbaum and Van Steen)
Outline

- Background
- Generalities:
  - faults
  - redundancy
  - stable storage
  - Recovery principles
- Rollback-recovery protocols
  - checkpointing
  - message logging
- Additional exercises
BACKGROUND: MODELLING DISTRIBUTED EXECUTIONS
Execution representation: time diagram

- These executions are identical -> event representation
- Only the order of message reception matters, whatever the transmission and execution duration
**Happened-before relation:** →

- **When 2 events** $e_1$, $e_2$,
  - Are local to a process $P_i$, $e_1 \rightarrow e_2$
  - $e_1$: message send on $P_i$, $e_2$: corresponding message reception on $P_j$, $e_1 \rightarrow e_2$
- **Several events**, $e_1$, $e_2$, $e_3$ (transitivity)
  - If $e_1 \rightarrow e_2$, and $e_2 \rightarrow e_3$, then, $e_1 \rightarrow e_3$
- **Not all events are mandatorily related along** →
  - Incomparable, independent, concurrent: ||
    - Non transitivity of ||
- **Happened-before relation:** also named Causality (partial order)
GENERALITIES ABOUT FAULTS AND RECOVERY
Failure Models

- Different types of failures.

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td></td>
<td><em>Receive omission</em></td>
</tr>
<tr>
<td></td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td></td>
<td><em>Send omission</em></td>
</tr>
<tr>
<td></td>
<td>A server fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>The server's response is incorrect</td>
</tr>
<tr>
<td></td>
<td><em>Value failure</em></td>
</tr>
<tr>
<td></td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td></td>
<td><em>State transition failure</em></td>
</tr>
<tr>
<td></td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

- A system is **k-fault tolerant** if it can survive faults in k components and still meet its specification
Failure Masking by Redundancy, an example

(a)

(b) Voter
Stable storage - a prerequisite for recovery

- In a system that tolerates only *a single failure*, stable storage may consist of the volatile memory of another process.
- In a system that wishes to tolerate an arbitrary number of *transient failures*, stable storage may consist of a local disk in each host.
- In a system that tolerates *non-transient failures*, stable storage must consist of a persistent medium outside the host on which a process is running. A replicated file system is a possible implementation in such systems.
Recovery Stable Storage

(a) Stable Storage
(b) Crash after drive 1 is updated (drive 1 updated first)
(c) Bad spot
Cuts / consistent cuts

- Strongly consistent cut
- Not consistent cut
- Orphan message
- In transit messages
- A consistent cut

Diagram showing interactions between P0, P1, P2, and P3.
exercise

- Find a few consistent cuts in the figure below (passing by)
- Order the according to happened before
- Characterise a consistent cut based on the happened before relation
- How to characterize strongly consistent cuts?
Recovery: Principles:

A recoverable state contains enough information to replay an execution up to a coherent cut of the original execution (up to the failure point).

It is sufficient to reconstruct a state that could have occurred in a failure-free execution.
Recovery: Principles:
1 - Checkpointing

Checkpoint (stored local state)

Restart all, or almost all, processes from a consistent cut and let a new execution run
Only one (or a few) process recover and use message information to replay the previous execution until reaching the failure point.
In transit messages

- If message delivery is not guaranteed, they are not a problem!
- But if the communication protocol is reliable, they must be taken into account
  ➡ We have to store them (they are part of the recoverable state)
Orphan messages

- If P2 fails and restarts from the cut, the message will be re-emitted and received twice by P1
  - Either avoid using inconsistent cuts (in general for checkpointing)
  - Or avoid re-emitting the message (in general for message logging)
RECOVERY: CHECKPOINTING MECHANISMS
Checkpoint-based rollback recovery - Uncoordinated checkpointing

- **Hypothesis: Fail stop**
- Each process takes checkpoints from time to time
- Upon failure we compute the recovery line
  - A process (e.g., the failed one) initiates the process
  - Collects dependencies information from all the processes
  - Computes the recovery line and triggers recovery
Example: exercise 1

The algorithm used to compute the recovery line first marks the graph nodes corresponding to the states of processes $P_0$ and $P_1$ at the failure point (red ellipses). It then uses reachability analysis to mark all reachable nodes from any of the initially marked nodes. The union of the last unmarked nodes over the entire system forms the recovery line.

*rollback-dependency graph* [Bhargava and Lian 1988] in which each node represents a checkpoint and a directed edge is drawn from $c_{i,x}$ to $c_{j,y}$ if either:

1. $i \neq j$, and a message $m$ is sent from $I_{i,x}$ and received in $I_{j,y}$, or
2. $i = j$ and $y = x + 1$. 
Example: exercise 1

1 - build the rollback dependency graph
2 - What is the recovery line?
3 - What if P3 fails instead?
Exercise 1 contd

- Same exercise
- How can you extend the rules in order to also avoid in-transit message?
- What is the new recovery line?
Exercise 1 contd: the *domino effect*

- Find the recovery line

Conclusion: let us synchronize checkpoints !!!
Coordinated checkpointing

- There is an initiator process for the checkpointing
  - Only one (or 2) checkpoint per process (always consistent)
  - large latency: processed blocked until checkpoint is finished

Inconsistency if communications are not blocked until the end of the checkpointing phase.
Coordinated checkpointing (2)

Algorithm:
- block communications while the protocol executes
- An initiator takes a checkpoint and broadcasts a request message to all processes
- When a process receives this message, it
  - stops its execution,
  - flushes all the communication channels,
  - takes a tentative checkpoint, and
  - sends an acknowledgment message back
- the coordinator receives acknowledgments from all processes, and broadcasts a commit message
- After receiving the commit each process removes the old checkpoint, the new one becomes permanent
Coordintated Checkpointing (3)
Overall execution graph

P0: initiator
P1
P2
P3

checkpoint requests
acknowledgments
Solutions to avoid blocked states

- If communication channels are FIFO: propagate the checkpoint request before sending any other message.

- Or piggyback checkpoint request on first message => take the checkpoint before taking the message into account.

**Question:** is FIFO necessary when piggybacking?
Another version: Distributed Snapshot algo for FIFO channels [Chandy-Lamport]

- Channels are FIFO. Messages are not lost. No failure
- Snapshot algo. executes concurrently with the application
- Special “control” message
  - When receiving it for the 1st time through a channel:
    - Pi records its state, and channel state = empty
    - Pi forwards control message to all its outgoing neighbors
  - Messages received through the other incoming channels after a 1st received “control” msg are logged
  - When not the 1st time:
    - Pi adds to its state all logged msgs that came from this channel so far
- Any process may initiate the algo. at any time (triggers one control msg for itself), but concurrent execs must be distinguishable
- Terminated: all Pi received control msg from all incoming channels
- Logged msgs on P->Q, logged by Q=“msgs sent by P to Q while P and Q already logged their state, and Q waited the control msg from P” (m3 in the Ex.)

Ex.:

```
\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Example Diagram}
\end{figure}
```

```
Snapshot= \{Sp,Sq,Sr,m3\}
```
Exercise

- Why is FIFO necessary for Chandy-Lamport algorithm? How are orphan messages avoided?
- What about in transit messages: how are they managed with Chandy Lamport algorithm?
- Two processes P and Q are connected in a ring, they constantly rotate a message m (but might perform some local computation before re-sending the msg). At any time, there is only one copy of m in the system. Each process’s state consists of the number of times it has received m, P sends first. At a certain point, P has the message and its state is 101. Immediately after sending m, P initiates the snapshot algorithm. Explain the operations of the algorithm in this case and give the possible global state(s) reported by it.
Communication Induced Checkpointing

- 2 kinds of checkpoints: *local* and *forced*
- prevent the creation of useless checkpoints
- no coordination message: only piggybacks information
- Simplest = index-based:
  - processes piggyback timestamps (increasing timestamps for a given process)
  - For example [Briatico et al.] forces a checkpoint upon receiving a message with a greater index than the local index
  - A recovery line consists of checkpoints with the same index
Communication Induced Checkpointing (2)

0<1: in transit messages

P2(at 0) receives 1: take a checkpoint before reception
forced checkpoint

A consistent cut

a local checkpoint
Exercise

- show that the domino effect of exercise 1 is not possible anymore: assign index to checkpoints, add forced checkpoints and give piggybacked indexes on messages (black boxes are the local checkpoints)

- check with different failure points
exercise contd.

- what to do if more than 1 number of difference between indices?
- What does it mean when the piggybacked index is smaller than the current checkpoint? What can be done / can we use this information?
Question

- Suppose each process takes a local checkpoint regularly (e.g. every 10 minutes), the preceding protocol multiplies (forced) checkpoints, how to keep a linear number of checkpoints (forced+local)? (forced CP are blue ones) while having always a checkpoint (forced or local)
Note: using clocks

- One could also use synchronized clocks to “synchronize” checkpoints: wait long enough to avoid possible clock deviation
In transit messages

- Remember that if the communication protocol is reliable, they must be stored
  - It is easy to store them with the next checkpoint of the message sender (sender-based) or receiver.
  - Receiver-based: checkpoint already stored
  - Sender-based: messages are sent again upon recovery

Question:
Can we optimize the recovery process and avoid re-sending in-transit messages to processes that have not failed?
RECOVERY: MESSAGE LOGGING MECHANISMS
Message Logging

- Hypothesis: *piecewise determinism* = all non-deterministic events can be identified and their determinants can be stored on stable storage.

- An execution is a sequence of deterministic events (replayed) and non-deterministic events (logged and simulated from log)

- Determinants of non-deterministic events are stored during failure-free execution

- + checkpoints to avoid recovering from the start

- Additional hypothesis: It is possible to prevent a message from being sent or received
Message Logging

- A process is orphan if it depends on the execution of a non-logged non-deterministic event
- Always no orphan process
  - Log(e) = set of processes locally storing the event e
  - Stable(e) if e’s determinant is logged on stable storage
  - Depend(e) processes affected by a non-deterministic event e

\[ \forall e : \neg Stable(e) \Rightarrow \text{Depend}(e) \subseteq \text{Log}(e) \]

else the process is said orphan
Tiny exercise

- Question: what is \text{depend}(e) in the example below?
- What about \text{depend}(e')
Pessimistic message logging

- orphan processes are never created but requires a lot of synchronizations with the stable storage
- Logs the determinant of ND events before executing them

\[ \forall e : \neg Stable(e) \Rightarrow |Depend(e)| = 0 \]

Only 1 process restarts

message resent during recovery (might have to be cancelled)  
logged message reception
Pessimistic message logging (2)

- only the failed processes recover
- simple
- restart from last checkpoint, recovery simple and very fast
- garbage collection simple
- Easier to take into account outside world
- performance penalty due to synchronous logging
- NB: if message delivery is not guaranteed then logging does not have to be synchronous, it is only necessary to log a reception before sending the next message -> exercise
**Optimistic message logging (principles)**

- Determinant kept locally, and sometimes stored on global storage
- Track causal dependencies between messages
- Synchronous recovery: compute the maximum recoverable state
- Asynchronous: trigger recovery of causally related processes during the recovery process
- Risk of exponential rollbacks
Summary

- In fault tolerance strong (interesting) results require strong assumptions, or a lot of redundancy and inefficiency.
- Fortunately in practice most system are reliable enough.
- What was not presented:
  - safe communications
  - details of optimistic message logging
  - causal logging
  - complex protocols in general
  - redundancy and basic coherence, safety algorithm (course placed on a higher protocol level)
<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>PWD assumed?</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>Domino 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>
Advantages and drawbacks of ML/CP (simplified!)

<table>
<thead>
<tr>
<th></th>
<th>Target system</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpointing</td>
<td>small and medium size</td>
<td>Rather low</td>
</tr>
<tr>
<td>Message logging</td>
<td>large scale</td>
<td>Medium or high</td>
</tr>
</tbody>
</table>

EXERCISES
Exercise: Improving pessimistic message logging

- The performance overhead of pessimistic logging can be reduced by delivering a message or an event and deferring its logging until the receiver communicates with any other process,

\[ \forall e : \neg Stable(e) \Rightarrow |Depend(e)| \leq 1 \]
Exercise: Improving optimistic message logging

- Below, which messages have to be logged? when?
- Can there be orphan processes?
- What can happen at recovery?
- Illustrate it on example and if P2 crashes:
- What if P1 and P2 crashes “simultaneously”
Additional exercise: an hybrid protocol

- Supposing processes are split into different groups
- We want to implement CIC inside a group, and pessimistic message logging between groups
- Specify an hybrid protocol that implements such an hybrid algorithm:
  - what happens when we send/receive a com
    - inside a group
    - between groups
  - when is a forced checkpoint taken
- Specify the recovery mechanism (NB a whole group recovers)
Additional exercise

- execution example 2 groups x 3 processes
  - place forced messages?
  - which messages are logged?
  - what happens at recovery?
For next week

- paper:
  A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability

- Or Questions:
  - Is PWD necessary for CIC protocols, why? And for other coordinated/non-coordinated checkpointing?
  - PLUS exercise on the next 3 slides
Exercise: Z-paths

Given two checkpoints $c_{i,x}$ and $c_{j,y}$, a Z-path exists between $c_{i,x}$ and $c_{j,y}$ if and only if one of the following two conditions holds:

1. $x < y$ and $i = j$; or
2. There exists a sequence of messages $[m_0, m_1, \ldots, m_n]$, $n \neq 0$, such that:
   - $c_{i,x} \leftrightarrow send_i(m_0)$;
   - $\forall l < n$, either $deliver_k(m_l)$ and $send_k(m_{l+1})$ are in the same checkpoint interval, or $deliver_k(m_l) \leftrightarrow send_k(m_{l+1})$; and
   - $deliver_j(m_n) \leftrightarrow c_{j,y}$

where $send_i$ and $deliver_i$ are communication events executed by process $P_i$. In
Exercise: link between Z-paths and checkpoint dependencies

1 - draw the checkpoint dependency graph for the execution of the previous slide
2 - find Z-path in the executions mentioned in the course:
Equivalence?

- Zpaths have been used to prove correctness of some CIC protocols, because a checkpoint in a Z-cycle is not useful:
  
  On the preceding examples, show that the checkpoints in Z-cycles would not be used upon recovery according to the checkpoint dependency graph.

  NB: this is not a proof of equivalence!
AN HYBRID PROTOCOL
In-transit Messages

- Delayed checkpoints

Logged requests and replies
In-transit Messages (recovery)

- Delayed checkpoints

  Resend Logged messages upon recovery
Orphan Requests
Orphan Requests

- Reception is removed from the checkpoint
- Requests removed from the Requests queue
Orphan Requests (recovery)

- Orphan request is also received during recovery
Orphan Replies

- Replies cannot be removed from the internal state
Orphan Replies (recovery)

- Replies resent during recovery must be ignored.
- The protocol must guarantee that these replies are identical in the two executions.
Equivalent Executions

I

J

K

Serv(Q1) Serv(Q2)
Equivalent Executions

- repeated replies must be equal:
  Q1 must be served before Q2 sur J
Equivalent Executions

- Twice replies must be equal:
  Q1 must be served before Q2 sur J
Causal relations:
Q1 must be received before Q3 on J
Causal relations:
Q1 must be received before Q3 on J