10 - Fault Tolerance, Logging and Recovery
Recap: Last Lecture

- ACID Properties
  - Atomicity, Consistency, Isolation, Durability

- 2-Phase Commit for distributed transactions

- 2PC assumptions:
  - Coordinator
  - Ability to recover state, persistence after “DoCommit”
Today's Lecture Outline

- Motivation – Fault Tolerance
- Fault Tolerance using Checkpoints
- Fault Tolerance using Logging and Recovery
- Logging and Recovery in Practice: ARIES
What is Fault Tolerance?

- Dealing successfully with partial failure within a distributed system
- Fault tolerant \(\rightarrow\) dependable systems
- Dependability implies the following:
  1. Availability
  2. Reliability
  3. Safety
  4. Maintainability
Dependability Concepts

- **Availability** – the system is ready to be used immediately.
  - “High availability”: system is ready at any given time, with high probability.

- **Reliability** – the system runs continuously without failure.
  - “High reliability”: system works without interruption during a long period of time.

Subtle difference. Consider:
- Random but rare failures (one millisecond every hour)
- Predictable maintenance (two weeks every year)
Dependability Concepts

- **Safety** – if a system fails, nothing catastrophic will happen. (e.g. process control systems)

- **Maintainability** – when a system fails, it can be repaired easily and quickly (sometimes, without its users noticing the failure). Also called **Recovery**.
  - What’s a failure? : System that cannot meet its goals => faults
  - Recover from all kinds of faults:
    - Transient: appears once, then disappears
    - Intermittent: occurs, vanishes, reappears
    - Permanent: requires replacement / repair
## Failure Models

<table>
<thead>
<tr>
<th>Type of Failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Crash</strong> failure</td>
<td>Server halts, working correctly before.</td>
</tr>
<tr>
<td><strong>Omission</strong> failure</td>
<td>Server fails to respond to request</td>
</tr>
<tr>
<td>Receive omission</td>
<td>Server fails to receive incoming msg</td>
</tr>
<tr>
<td>Send omission</td>
<td>Server fails to send msg</td>
</tr>
<tr>
<td><strong>Timing</strong> failure</td>
<td>Server’s response outside specified time interval</td>
</tr>
<tr>
<td><strong>Response</strong> failure</td>
<td>Incorrect server response</td>
</tr>
<tr>
<td>Value failure</td>
<td>Wrong value of response</td>
</tr>
<tr>
<td>State transition failure</td>
<td>Deviation from flow of control</td>
</tr>
<tr>
<td><strong>Arbitrary</strong> (Byzantine) failure</td>
<td>Server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>
1. *Information Redundancy* – add extra bits to allow for error detection/recovery
   (Hamming codes: detect 2-bit errors, correct 1-bit errors)

2. *Time Redundancy* – perform operation and, if needs be, perform it again.
   (Purpose of transactions: BEGIN/END/COMMUT/ABORT)

3. *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

   Can you think of Physical redundancy in Nature?
Redundancy in Electronics

Triple modular redundancy in a circuit (b) A, B, C are circuit elements and V* are voters
Redundancy is Expensive

- But without redundancy, we need to recover after a crash.
Today's Lecture Outline

- Motivation – Fault Tolerance
- Fault Tolerance using Checkpoints
- Fault Tolerance using Logging and Recovery
- Logging and Recovery in Practice: ARIES
Recovery Strategies

When a failure occurs, we need to bring the system into an error free state (recovery).

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
   - Packet retransmit in case of lost packet

2. **Forward Recovery**: bring the system into a correct new state, from which it can then continue to execute.
   - Erasure coding → Forward Error Correction
Forward and Backward Recovery

• **Major disadvantage of Backward Recovery:**
  • Checkpointing can be very expensive (especially when errors are very rare).

• **Major disadvantage of Forward Recovery:**
  • In order to work, all potential errors need to be accounted for *up-front*.
  • “Harder” the recovery mechanism need to know how do to bring the system *forward* to a correct state.

• In practice: backward recovery common
Backward Recovery

- Checkpoint: snapshot the state of the DS
  - Transactions
  - Messages received / sent
  - Roles like coordinator, …

- Frequent checkpoints are expensive
  - Requires writing to stable storage
  - Very slow if checkpoint after every event!

- What can we do to make checkpoints cheaper?
  - Less frequent checkpoints, e.g., “every 10 seconds’
Independent Checkpointing

A recovery line to detect the correct distributed snapshot. This becomes challenging if checkpoints are un-coordinated.
The Domino Effect

The domino effect – Cascaded rollback

P2 crashes, roll back, but 2 checkpoints inconsistent (P2 shows m received, but P1 does not show m sent)
Coordinated Checkpointing

• Key idea: each process takes a checkpoint after a globally coordinated action. (Why?)

• Simple Solution: 2-phase blocking protocol
  • Coordinator multicast checkpoint_REQUEST message
  • Participants receive message, takes a checkpoint, stops sending (application) messages and queues them, and sends back checkpoint_ACK
  • Once all participants ACK, coordinator sends checkpoint_DONE to allow blocked processes to go on

• Optimization: consider only processes that depend on the recovery of the coordinator (those it sent a message since last checkpoint)
Successful Coord. Checkpoint

Blue: application messages
Red: checkpoint messages
Unsuccessful Coord. Checkpoint

Checkpoints can fail, if participant sent msg before `checkpoint_REQUEST` and receiver gets msg after `checkpoint_REQUEST`. Then: abort and do try another coordinated checkpoint soon (compare to 2PC).
Today's Lecture Outline

• Motivation – Fault Tolerance

• Fault Tolerance using Checkpoints

• Fault Tolerance using Logging and Recovery

• Logging and Recovery in Practice: ARIES
Goal: Make transactions Reliable

...in the presence of failures
- Machines can crash: disk contents (OK), memory (volatile)
- Assume that machines don’t misbehave
- Networks are flaky, packet loss, handle using timeouts

If we store database state in memory, a crash will cause loss of “Durability”.

May violate atomicity, i.e. recover such that uncommitted transactions COMMIT or ABORT.

General idea: store enough information to disk to determine global state (in the form of a LOG)
Challenges:

- Disk performance is poor (vs memory)
  - Cannot save all transactions to disk
  - Memory typically several orders of magnitude faster

- Writing to disk to handle arbitrary crash is hard
  - Several reasons, but HDDs and SSDs have buffers

- Same general idea: store enough data on disk so as to recover to a valid state after a crash:
  - Shadow pages and Write-ahead Logging (WAL)
  - Idea is to provide Atomicity and Durability
Shadow Paging Vs WAL

- Shadow Pages
  - Provide Atomicity and Durability, “page” = unit of storage
  - Idea: When writing a page, make a “shadow” copy
    - No references from other pages, edit easily!
  - ABORT: discard shadow page
  - COMMIT: Make shadow page “real”. Update pointers to data on this page from other pages (recursive). Can be done atomically
  - Essentially “copy-on-write” to avoid in-place page update
Shadow Paging vs WAL

• Write-Ahead-Logging
  • Provide Atomicity and Durability
  • Idea: create a log recording every update to database
  • Updates considered reliable when stored on disk
  • Updated versions are kept in memory (page cache)
  • Logs typically store both REDO and UNDO operations
  • After a crash, recover by replaying log entries to reconstruct correct state

• WAL is more common, fewer disk operations, transactions considered committed once log written.
Today's Lecture Outline

• Motivation – Fault Tolerance

• Fault Tolerance using Checkpoints

• Fault Tolerance using Logging and Recovery

• Logging and Recovery in Practice: ARIES
ARIES Recovery Algorithms

- **ARIES**: Algorithms for Recovery and Isolation Exploiting Semantics
- Used in major databases
  - IBM DB2 and Microsoft SQL Server
  - Deals with many practical issues
- **Principles**
  - Write-ahead logging
  - Repeating history during Redo
  - Logging changes during Undo
Write-Ahead Logging

- View as sequence of entries, sequential number
  - Log-Sequence Number (LSN)
  - Database: fixed size PAGES, storage at page level
- Pages on disk, some also in memory (page cache)
  - “Dirty pages”: page in memory differs from one on disk
- Reconstruct global consistent state using
  - Log files + disk contents + (page cache)
- Logs consist of sequence of records
  - What do we need to log?
    - Seq#, which transaction, operation type, what changed…
Write-Ahead Logging

• Logs consist of sequence of records
  • To record an update to state
  • LSN: [prevLSN, TID, “update”, pageID, new value, old value]
    • PrevLSN forms a backward chain of operations for each TID
    • Storing “old” and “new” values allow REDO operations to bring a page up to date, or UNDO an update reverting to an earlier version

• Transaction Table (TT): All TXNS not written to disk
  • Including Seq Num of the last log entry they caused

• Dirty Page Table (DPT): all dirty pages in memory
  • Modified pages, but not written back to disk.
  • Includes recoveryLSN: first log entry to make page dirty
Recovery using WAL – 3 passes

• Analysis Pass
  • Reconstruct TT and DPT (from start or last checkpoint)
  • Get copies of all pages at the start

• Recovery Pass (redo pass)
  • Replay log forward, make updates to all dirty pages
  • Bring everything to a state at the time of the crash

• Undo Pass
  • Replay log file backward, revert any changes made by transactions that had not committed (use PrevLSN)
  • For each write Compensation Log Record (CLR)
  • Once reach entry without PrevLSN → done
ARIES (WAL): Data Structures

**TT: Transaction Table**

<table>
<thead>
<tr>
<th>TID</th>
<th>LastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>567</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

**DPT: Dirty Page Table**

<table>
<thead>
<tr>
<th>pageID</th>
<th>recoveryLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>567</td>
</tr>
<tr>
<td>46</td>
<td>568</td>
</tr>
<tr>
<td>77</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>42</td>
</tr>
</tbody>
</table>

**TID: Transaction ID**

LastLSN: LSN of the most recent log record seen for this Transaction. i.e. latest change

**pageID: key/ID of a page**

recoveryLSN: LSN of first log record that made page dirty i.e. earliest change to page
Example Log File

DB Buffer

Page 42

<table>
<thead>
<tr>
<th>LSN</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>77</td>
<td>55</td>
</tr>
</tbody>
</table>

Page 46

<table>
<thead>
<tr>
<th>LSN</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>22</td>
</tr>
</tbody>
</table>

LOG

<table>
<thead>
<tr>
<th>TID</th>
<th>LastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DPT</th>
<th>recoveryLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, "update", pageID, redo, undo]  # Update

DB Buffer
Page 42
LSN=1
a=78
b=55

Page 46
LSN=-
c=22

LOG
1: [-,1,"update",42,a+=1, a-=1]
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, "update", pageID, redo, undo]  # Update

DB Buffer
Page 42
LSN=2
a=78
b=58

Page 46
LSN=-
c=22

LOG
1: [-,1,"update",42,a+=1, a-=1
2: [-,2,"update", 42,b+=3, b-=3]
Example Log File

LSN: \([\text{prevLSN}, \text{TID}, \text{type}]\)  \# All
LSN: \([\text{prevLSN}, \text{TID}, \text{“update”}, \text{pageID}, \text{redo}, \text{undo}]\)  \# Update

**DB Buffer**

- **Page 42**
  - LSN=4
  - a=78
  - b=59

- **Page 46**
  - LSN=3
  - c=24

**LOG**

1: \([-1,”update”,42,a+=1, a=1]\)
2: \([-2,”update”, 42,b+=3, b=3]\)
3: \([2,2,”update”,46,c+=2, c=2]\)
4: \([1,1,”update”,42, b+=1, b=1]\)

**TT**

<table>
<thead>
<tr>
<th>TID</th>
<th>LastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**DPT**

<table>
<thead>
<tr>
<th>pageID</th>
<th>recoveryLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>3</td>
</tr>
</tbody>
</table>
Example Log File

LSN: [prevLSN, TID, type] # All
LSN: [prevLSN, TID, "update", pageID, redo, undo] # Update

DB Buffer
Page 42
LSN=4
a=78
b=59
Page 46
LSN=3
c=24

LOG
1: [-,1,"update",42,a+=1, a=1
2: [-,2,"update", 42,b+=3, b=3]
3: [2,2,"update",46,c+=2, c=2]
4:[1,1,"update",42, b+=1, b=1]
5:[3,2,"commit"]

TT
TID | LastLSN
1   | 4

DPT
pageID | recoveryLSN
42    | 1
46    | 3
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, “update”, pageID, redo, undo]  # Update
LSN: [prevLSN, TID, “comp”, redoTheUndo, undoNextLSN]  #compensation

On Disk

Page 42
LSN=-
a=77
b=55

Page 46
LSN=-
c=22

LOG

1: [-,1,”update”,42,a+=1, a=1
2: [-,2,”update”, 42,b+=3, b=3]
3: [2,2,”update”,46,c+=2, c=2]
4:[1,1,”update”,42, b+=1, b=1]
5:[3,2,”commit”]
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, "update", pageID, redo, undo]  # Update
LSN: [prevLSN, TID, "comp", redoTheUndo, undoNextLSN]  # Compensation

On Disk

Page 42
LSN=-
a=77
b=55

Page 46
LSN=-
c=22

LOG
1: [-,1,"update",42,a+=1, a=1]
2: [-,2,"update", 42,b+=3, b=3]
3: [2,2,"update",46,c+=2, c=2]
4:[1,1,"update",42, b+=1, b=1]
5:[3,2,"commit"]

1. Analysis
1. Analysis to figure out the start of the redo.
=> start from 1

TT
TID | LastLSN
---|---
1 | 4

DPT
pageID | recoveryLSN
---|---
42 | 1
46 | 3
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, “update”, pageID, redo, undo]  # Update
LSN: [prevLSN, TID, “comp”, redoTheUndo, undoNextLSN]  # Compensation

1. Analysis
2. Redo

1: [-1,”update”,42,a+=1, a=-1
2: [-2,”update”, 42,b+=3, b=-3]
3: [2,2,”update”,46,c+=2, c=-2]
4:[1,1,”update”,42, b+=1, b=-1]
5:[3,2,”commit”]

DB Buffer
Page 42
| LSN=4 |
| a=78 |
| b=59 |

Page 46
| LSN=3 |
| c=24 |

LOG

<table>
<thead>
<tr>
<th>TT</th>
<th>TID</th>
<th>LastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DPT</th>
<th>pageID</th>
<th>recoveryLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>46</td>
<td>3</td>
</tr>
</tbody>
</table>
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, "update", pageID, redo, undo]  # Update
LSN: [prevLSN, TID, "comp", redoTheUndo, undoNextLSN]  # compensation

1. Analysis
2. Redo
3. Undo

DB Buffer
Page 42
LSN=6
a=78
b=58
Page 46
LSN=3
c=24

LOG
1: [-,1,"update",42,a+=1, a-=1
2: [-,2,"update", 42,b+=3, b-=3]
3: [2,2,"update",46,c+=2, c-=2]
4:[1,1,"update",42, b+=1, b-=1]
5:[3,2,"commit"]
6: [4,1,"comp",42,b-=1, b+=1]
Example Log File

LSN: [prevLSN, TID, type]  # All
LSN: [prevLSN, TID, “update”, pageID, redo, undo]  # Update
LSN: [prevLSN, TID, “comp”, redoTheUndo, undoNextLSN]  # compensation

1. Analysis
2. Redo
3. Undo

1: [-,1,”update”,42,a+=1, a=-1
2: [-,2,”update”, 42,b+=3, b=-3]
3: [2,2,”update”,46,c+=2, c=-2]
4:[1,1,”update”,42, b+=1, b=-1]
5:[3,2,”commit”]

6: [4,1,”comp”,42,b-=1, b+=1]
7: [6,1,”comp”, 42, a-=1, b+=1]
2PC works great with WAL/ARIES

- WAL can integrate with 2PC
  - Have additional log entries that capture 2PC operation
  - **Coordinator:** Include list of participants
  - **Participant:** Indicates coordinator
  - Votes to commit or abort
  - Indication from coordinator to Commit/Abort

![Diagram showing 2PC process with Coordinator and Participant nodes connected to Server nodes.]
Optimizing WAL

- As described earlier:
  - Replay operations back to the beginning of time
  - Log file would be kept forever (→ entire Database)

- In practice, we can do better with CHECKPOINT
  - Periodically save DPT, TT
  - Store any dirty pages to disk, indicate in LOG file
  - Prune initial portion of log file: All transactions upto checkpoint have been committed or aborted.
Summary

• Basic concepts for Fault Tolerant Systems
  • Properties of dependable systems
  • Redundancy, process resilience (see T8.2)
  • Reliable RPCs (see T8.3)

• Fault Tolerance – Backward recovery using checkpoints.
  • Tradeoff: independent vs coordinated checkpointing

• Fault Tolerance – Recovery using Write-Ahead-Logging
  • Balances the overhead of checkpointing and ability to recover to a consistent state
Additional Material in the Book

- Process Resilience (when processes fail) T8.2
  - Have multiple processes (redundancy)
  - Group them (flat, hierarchically), voting

- Reliable RPCs (communication failures) T8.3
  - Several cases to consider (lost reply, client crash, …)
  - Several potential solutions for each case

- Distributed Commit Protocols T8.5
  - Perform operations by all group members, or not at all
  - 2 phase commit, … (last lecture)

- Logging and recovery T8.6