Table of Contents

- Introduction and Motivation
- Theoretical Foundations
- Distributed Programming Languages
- Distributed Operating Systems
- Distributed Communication
- Distributed Data Management
- Reliability
- Applications
- Conclusions
- Appendix
Distributed Data Management

- Data objects
  - Files
  - Directories
- Data objects are dispersed and replicated
  - Unreplicated
  - Fully replicated
  - Partially replicated
Serializability Theory

Atomic execution

- A transaction is an "all or nothing" operation.
- The concurrent execution of several transactions affects the database as if executed serially in some order.
- The interleaved order of the actions of a set of concurrent transactions is called a schedule.
Example 22: Concurrent Transactions

$T_1$ begin
1 read A (obtaining $A\_balance$)
2 read B (obtaining $B\_balance$)
3 write $A\_balance$-$10$ to A
4 write $B\_balance$+$10$ to B
end

$T_2$ begin
1 read B (obtaining $B\ balance$)
2 write $B\_balance$-$5$ to B
end
Three types of conflict: \( r-w \) (read-write), \( w-r \) (write-read), and \( w-w \) (write-write).

\( r_j[x] \) reads from \( w_i[x] \) iff

\( w_i[x] < r_j[x] \).

There is no \( w_k[x] \) such that \( w_i[x] < w_k[x] < r_j[x] \).

Two schedules are equivalent iff

Every read operation reads from the same write operation in both schedules.

Both schedules have the same final writes.

When a non-serial schedule is equivalent to a serial schedule, it is called **serializable schedule**.
A nonserializable schedule (a) and serializable schedule (b) for Example 22.

<table>
<thead>
<tr>
<th>Transaction (step)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1(1)</td>
<td>read A(obtaining A_balance)</td>
</tr>
<tr>
<td>T1(2)</td>
<td>read B(obtaining B_balance)</td>
</tr>
<tr>
<td>T1(3)</td>
<td>write A_balance-$10 to A</td>
</tr>
<tr>
<td>T2(1)</td>
<td>read B(obtaining B_balance)</td>
</tr>
<tr>
<td>T2(1)</td>
<td>write B_balance-$5 to B</td>
</tr>
<tr>
<td>T2(4)</td>
<td>write B_balance+$10 to B</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Transaction (step)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1(1)</td>
<td>read A(obtaining A_balance)</td>
</tr>
<tr>
<td>T2(1)</td>
<td>read B(obtaining B_balance)</td>
</tr>
<tr>
<td>T2(1)</td>
<td>write B_balance-$5 to B</td>
</tr>
<tr>
<td>T1(2)</td>
<td>read B(obtaining B_balance)</td>
</tr>
<tr>
<td>T1(3)</td>
<td>write A_balance-$10 to A</td>
</tr>
<tr>
<td>T1(4)</td>
<td>write B_balance+$10 to B</td>
</tr>
</tbody>
</table>

(b)
Concurrency Control

Optimistic (assuming conflicts are less frequent)
- Optimistic concurrency control
  First tentatively perform updates locally
  Then are made permanent and propagated if there are no conflicts

Conservative (assuming conflicts are frequent)
- Locking scheme
- Timestamp-based scheme
A transaction is **well-formed** if it
- locks an object before accessing it,
- does not lock an object that is already locked, and
- before it completes, unlocks each object it has locked.

A schedule is **two-phase** if no object is unlocked before all needed objects are locked.
Example 23: Well-Formed, Two-Phase Transactions

$T_1$: `begin`
  lock A
  `read` A (obtaining A balance)
  lock B
  `read` B (obtaining B balance)
  `write` A_balance-$10 to A
  unlock A
  `write` B_balance+$10 to B
  unlock B
`end`

$T_2$: `begin`
  lock B
  `read` B (obtaining B balance)
  `write` B_balance-$5 to B
  unlock B
`end`
Different Looking Schemes

- **Centralized locking algorithm**: distributed transactions, but centralized lock management.

- **Primary-site locking algorithm**: each object has a single site designated as its primary site (as in INGRES).

- **Decentralized locking**: The lock management duty is shared by all the sites.
Focus 19: Timestamp-based Concurrency Control

Each request (transaction) is associated with timestamp: ts

$\text{Time}_r(x) \ (\text{Time}_w(x))$: the largest timestamp of any read (write) processed thus far for object x.

Timestamp-based concurrency control:

- **(Read)** If $ts < \text{Time}_w(x)$ then the read request is rejected and the corresponding transaction is aborted; otherwise, it is executed and $\text{Time}_r(x)$ is set to $\max\{\text{Time}_r(x), ts\}$.

- **(Write)** If $ts < \text{Time}_w(x)$ or $ts < \text{Time}_r(x)$, then the write request is rejected; otherwise, it is executed and $\text{Time}_w(x)$ is set to $ts$. 
Example 24

- \( \text{Time}_r(x) = 4 \) and \( \text{Time}_w(x) = 6 \) initially.
- Sample:
  - \( \text{read}(x, 5), \text{write}(x, 7), \text{read}(x, 9), \text{read}(x, 8), \text{write}(x, 8) \)
- First and last are rejected and \( \text{Time}_r(x) = 7, \text{Time}_w(x) = 9 \) when completed.
Conservative Timestamp Ordering

- Each site keeps a write queue (W-queue) and a read queue (R-queue).
  - A read \((x, ts)\) request is executed if all W-queues are nonempty and the first write on each queue has a timestamp greater than \(ts\); otherwise, the read request is buffered in the R-queue.
  - A write \((x, ts)\) request is executed if all R-queues and W-queues are nonempty and the first read (write) on each R-queue (W-queue) has a timestamp greater than \(ts\); otherwise, the write request is buffered in the W-queue.
Strict Consistency

- Any read returns the result of the most recent write.
- Impossible to enforce, unless
  - All writes are instantaneously visible to all processes.
  - All reads get the then-current values, no matter how quickly next writes are done.
  - An absolute global time order is maintained.
Weak Consistency

- **Sequential consistency**: All processes see all shared accesses in the same order.
- **Causal consistency**: All processes see causally-related shared accesses in the same order.
- **FIFO consistency**: All processes see writes from each process in the order they were issued.
**Example 25: Sample Consistent Models**

<table>
<thead>
<tr>
<th></th>
<th>W(x,a)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td>R(x,b)</td>
<td>R(x,a)</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td>R(x,b)</td>
<td>R(x,a)</td>
</tr>
</tbody>
</table>

**sequentially-consistent**

<table>
<thead>
<tr>
<th></th>
<th>W(x,a)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td>R(x,b)</td>
<td>R(x,a)</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td>R(x,a)</td>
<td>R(x,b)</td>
</tr>
</tbody>
</table>

**non-sequentially-consistent**

**Linearizable:** sequentially-consistent, but taking ordering based on synchronized clocks
Example 25: Sample Consistent Models

<table>
<thead>
<tr>
<th></th>
<th>W(x,a)</th>
<th>W(x,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(x,a)</td>
<td>W(x,b)</td>
</tr>
<tr>
<td>P3</td>
<td>R(x,a)</td>
<td>R(x,c)</td>
</tr>
<tr>
<td>P4</td>
<td>R(x,a)</td>
<td>R(x,b)</td>
</tr>
</tbody>
</table>

causally-consistent

<table>
<thead>
<tr>
<th></th>
<th>W(x,a)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(x,a)</td>
<td>W(x,b)</td>
</tr>
<tr>
<td>P3</td>
<td>R(x,b)</td>
<td>R(x,a)</td>
</tr>
<tr>
<td>P4</td>
<td>R(x,a)</td>
<td>R(x,b)</td>
</tr>
</tbody>
</table>

non-causally-consistent
Example 25 (Cont’d)

<table>
<thead>
<tr>
<th></th>
<th>W(x,a)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(x,a)</td>
<td>W(x,b)</td>
<td>W(x,c)</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>R(x,b)</td>
<td>R(x,a)</td>
<td>R(x,c)</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td>R(x,a)</td>
<td>R(x,b)</td>
<td>R(x,c)</td>
</tr>
</tbody>
</table>

FIFO-consistent
Weak Consistency (Cont’d)

■ **Weak consistency**: Enforces consistency on a group of operations, not on individual reads and writes.

■ **Release consistency**: Enforces consistency on a group of operations enclosed by acquire and release operations.

■ **Eventual consistency**: All replicas will gradually become consistent. (Web pages with dominated read operations.)
Update Propagation for Multiple Copies

- **State versus Operations**
  - Propagate a notification of an update (such as invalidate signal)
  - Propagate data
  - Propagate the update operation

- **Pull versus Push**
  - Push-based approach (server-based)
  - Pull-based approach (client-based)
  - Lease-based approach (hybrid of push and pull)

- Consistency of duplicated data
  - **Write-invalidate** vs. **write-through**
  - **Quorum-voting** as an extension of single-write/multiple-read
Focus 20: Quorum-Voting

\[ w > \frac{v}{2} \text{ and } r + w > v \]

where \( w \) and \( r \) are write and read quorum and \( v \) is the total number of votes.

E.g., suppose \( v=9 \), there are the following possibilities:

\[(r, w): (5, 5), (4, 6), (3, 7), (2, 8), (1, 9)\]
Hierarchical Quorum Voting

A 3-level tree in the hierarchical quorum voting with read quorum= 2 and write quorum = 3.
Network Partition

Optimistic approaches: version vectors used in LOCUS

\( V=(v_1, v_2, \ldots, v_n) \), where \( n \) is the number of sites at which the file is stored. Version number \( v \) increases at each update.
Network Partition (cont’d)

Pessimistic approaches: Each site maintains a pair of
(version number, cardinality)

Majority-based dynamic voting: a majority of the most recent update

Example: \{A: (6, 5), B: (6, 5), C: (6, 5), D: (6, 5), E: (6, 5)\} before partition

A partition: \{A, B, C\} and \{D, E\} and two updates at majority \{A, B, C\}
Another partition: \{A: (8,3), D: (6,5)\}, \{B: (8,3), C: (8,3), E: (6,5)\}
{B, C, E} has majority, but E needs a catch up (which is a new update)

Use dynamic vote reassignment to find a majority: different weights
Brewer’s CAP Theorem (2000):
It is impossible for a web service to provide all three: consistency, availability, and partition tolerance (CAP)

Consistency: atomicity of transactions
Availability: any request to a non-faulty service leads to a response
Partition tolerance: service will be available during a partition
Distributed Atomic Transactions

Jim Gray’s Distributed Two-Phase Commitment:
One coordination with multiple participants

Two-step commit for one transaction:
(1) violate to stable and (2) stable
(2) to database

If an error occurs in (1), T is aborted; or in (2), it is rewritten to database and T is committed.

The finite state machine model for the two-phase commit protocol.
Phase 1

At the coordinator:

/*prec: initiate state (q) */
1. The coordinator sends a commit_request message to every participant and waits for replies from all the participants.

/*postc: waiting state (w) */

At participants:

/*prec: initiate state (q)*/
1. On receiving the commit_request message, a participant takes the following actions. If the transaction executing at the participant is successful, it writes undo and redo log, and sends a yes message to the coordinator; otherwise, it sends a no message.

/*postc: wait state (w) if yes or abort state (a) if no*/
Phase 2

At the coordinator

/*prec: wait state (w)*/
1. If all the participants reply yes then the coordinator writes a commit record into the log and then sends a commit message to all the participants. Otherwise, the coordinator sends an abort message to all the participants.

/*postc: commit state (c) if commit or abort state (a) if abort */

2. If all the acknowledgments are received within a timeout period, the coordinator writes a complete record to the log; otherwise, it resends the commit/abort message to those participants from which no acknowledgments were received.
Phase 2 (Cont’d)

At the participants

/*prec: wait state (w) */
1. On receiving a commit message, a participant releases all the resources and locks held for executing the transaction and sends an acknowledgment.

/*postc: commit state (c) */
/*prec: abort state (a) or wait state (w) */

2. On receiving an abort message, a participant undoes the transaction using the undo log record, releases all the resources and locks held by it, and sends an acknowledgment.

/*postc: abort state (a) */
### Site/Message Failures and Recovery Actions

<table>
<thead>
<tr>
<th>Location</th>
<th>Time of failure</th>
<th>Actions at coordi.</th>
<th>Actions at parti.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordi.</td>
<td>Before <em>commit</em></td>
<td>Broadcasts <em>abort</em> on recovery</td>
<td>Committed parti. Undo the trans.</td>
</tr>
<tr>
<td>Coordi.</td>
<td>Before <em>complete</em> after <em>commit</em></td>
<td>Broadcasts <em>commit</em> on recovery</td>
<td>___</td>
</tr>
<tr>
<td>Coordi.</td>
<td>After complete</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Parti.</td>
<td>In Phase 1</td>
<td>Coordi. aborts the transaction</td>
<td>___</td>
</tr>
<tr>
<td>Parti.</td>
<td>In Phase 2</td>
<td>___</td>
<td>Commit/abort on recovery</td>
</tr>
</tbody>
</table>
Two Types of Logs

- *undo* log allows an uncommitted transaction to record in stable storage values it wrote. (T₁, T₄, and T₅ in the example)
- *redo* log allows a transaction to commit before all the values written have been recorded in stable storage. (T₂ and T₇)

A recovery example.
If a participant fails in state $w$ and its vote is $yes$, it can be either commit or abort, waiting is needed to contact coordinator. Therefore, two-phase commitment is blocking.

Three-phase (non-blocking) commitment inserts a new state (precommit) to avoid a state containing both abort and commit options.
Skeen's Three-Phase Commitment Protocol

F: failure transition
T: timeout transition
Exercise 6

1. For the following two transactions:

   T1 begin
   1 read A (obtaining A balance)
   2 write A balance- $10 to A
   3 read B (obtaining B balance)
   4 write B balance+$10 to B
   end

   T2 begin
   1 read A (obtaining A balance)
   2 write A balance+$5 to A
   end

(a) Provide all the interleaved executions (or schedules).
(b) Find all the serializable schedules among the schedules obtained in (a).
Exercise 6 (Cont’d)

2. Point out serializable schedules in the following

L1 = w2(y)w1(y)r3(y)r1(y)w2(x)r3(x)r3(z)r2(z)
L2 = r3(z)r3(x)w2(x)r2(z)w1(y)r3(y)w2(y)r1(y)
L3 = r3(z)w2(y)w2(x)r1(y)r3(y)r2(z)r3(x)w1(y)
L4 = r2(z)w2(y)w2(x)w1(y)r1(y)r3(y)r3(z)r3(x)

3. A voting method called \textit{voting-with-witness} replaces some of the replicas by witnesses. Witnesses are copies that contain only the version number but no data. The witnesses are assigned votes and will cast them when they receive voting requests. Although the witnesses do not maintain data, they can testify to the validity of the value provided by some other replica. How should a witness react when it receives a read quorum request? What about a write quorum request? Discuss the pros and cons of this method.