Distributed Transactions

Arvind Krishnamurthy

University of Washington
Today’s Topics

• What is correctness or different forms of consistency?
• Distributed transactions
• Transaction chains
Consistency and Performance

• Weaker consistency often means better performance
  • But harder for programmers to reason about system behavior
Sequential Consistency

• “The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program” (Lamport, 1979)

• Coherence is the correctness criteria when restricted to a single memory location
**Sequential Consistency**

- Is the following execution sequentially consistent?

<table>
<thead>
<tr>
<th>$p_1$: $W(x)a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_2$: $W(x)b$</td>
</tr>
<tr>
<td>$p_3$: $R(x)b$ 1</td>
</tr>
<tr>
<td>$p_4$: $R(x)b$ 1</td>
</tr>
<tr>
<td>$p_5$: $R(x)a$ 2</td>
</tr>
<tr>
<td>$p_6$: $R(x)a$ 2</td>
</tr>
</tbody>
</table>
Sequential Consistency

• Is the following execution sequentially consistent?

<table>
<thead>
<tr>
<th>$p_1$:</th>
<th>$W(x)a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_2$:</td>
<td>$W(x)b$</td>
</tr>
<tr>
<td>$p_3$:</td>
<td>$R(x)b$</td>
</tr>
<tr>
<td>$p_4$:</td>
<td>$R(x)a$</td>
</tr>
<tr>
<td></td>
<td>$R(x)a$</td>
</tr>
<tr>
<td></td>
<td>$R(x)b$</td>
</tr>
</tbody>
</table>
What could cause this behavior?

- Multiple caches & non-serialized updates
- Non-blocking operations
- Compiler rewrites
Linearizability

• “The result of any execution is the same as if the operations of all the processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program” (Lamport, 1979)

• In addition, if $ts(A) < ts(B)$, then operation $A$ should precede $B$ in this sequence (Herlihy & Wing, 1991)
Causal Consistency

• Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines. (Hutto and Ahamad, 1990)

Is this data store sequentially consistent? Causally consistent?
Causal Consistency

• Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines. (Hutto and Ahamad, 1990)

\[
\begin{align*}
p_1 &: W(x)a & W(x)c \\
p_2 &: R(x)a \\
p_3 &: R(x)a & W(x)b & R(x)c & R(x)b \\
p_4 &: R(x)a & R(x)b & R(x)c
\end{align*}
\]

Is this data store sequentially consistent? Causally consistent?
Transactions

• Operations sometimes need to atomically update multiple data items
  • Transactions help cope with crashes and concurrency

• Example: calendar system, each user has a calendar
• Sched(u1, u2, t):
  begin_transaction()
  ok1 = reserve(u1, t)
  ok2 = reserve(u2, t)
  if ok1 and ok2:
    commit_transaction()
  else abort_transaction()
Formalizing Correctness (Serializability)

• **Atomic**: state shows either all the effects of a transaction or none of them

• **Consistent**: transaction moves only between states where integrity holds

• **Isolated**: effects of transactions is the same as transactions running one after another

• **Durable**: once a transaction has committed, its effects remain in the database
Serializability

- Conflicting operations:
  - two updates to the same location
  - an update and an access to the same location
- Serializability check:
  - Order conflicting operations from different transactions
  - All ordering constraints between two transactions should go in the same direction
    - i.e., T1’s operations happened before T2’s operations or the other way around
Distributed Transactions

- Data distributed across multiple nodes
- How to provide serializable transactions in a distributed setting?
Idea #1

• tentative changes, later commit or undo (abort)

reserve_handler(u, t):
  if u[t] is free:
    temp_u[t] = taken  // A TEMPORARY VERSION
    return true
  else:
    return false

commit_handler():
  copy temp_u[t] to real u[t]
abort_handler():
  discard temp_u[t]
Idea #2

- Single entity decides whether to commit to ensure agreement
  - let's call it the Transaction Coordinator (TC)
- Client sends RPCs to nodes A, B
- Client's commit_transaction() sends "go" to TC
- TC/A/B execute distributed commit protocol
- TC reports "commit" or "abort" to client
2-Phase Commit

Coordinator $c$

I. sends VOTE-REQ to all participants

III. if (all votes YES) then
    decide := COMMIT
    send COMMIT to all
else
    decide := ABORT
    send ABORT to all who voted YES
halt

Participant $p_i$

II. send vote to Coordinator
    if vote = NO then
        decide := ABORT
        halt

IV. if received COMMIT then
    decide := COMMIT
else
    decide := ABORT
halt
Dealing with failures

- Failures result in timeouts
- What should the coordinator do when it times-out on a participant?
- What should the participant do when it times-out on the coordinator?
  - For the vote request?
  - For the decision?
- What state should the coordinator/participant maintain in stable storage?
Transaction Chains

• Lynx system for geo-distributed data
  • Low latency without needing to contact all shards
  • Serializable semantics if transactions have a certain structure
• Takes advantage of the fact that:
  • Most transaction systems have a fixed and known set of transactions
Why transaction chains?

Auction service

<table>
<thead>
<tr>
<th>Bidder</th>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Book</td>
<td>$100</td>
</tr>
<tr>
<td>Bob</td>
<td>Book</td>
<td>$20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seller</th>
<th>Item</th>
<th>Highest bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>iPhone</td>
<td>$20</td>
</tr>
<tr>
<td>Bob</td>
<td>Camera</td>
<td>$100</td>
</tr>
</tbody>
</table>
Why transaction chains?

Operation: Alice bids on Bob’s camera

1. Insert bid to Alice’s Bids
2. Update highest bid on Bob’s Items

Alice

Datacenter-1

Alice’s Bids

| Alice | Book  | $100 |

Bob

Datacenter-2

Bob’s Items

| Bob  | Camera | $100 |
Why transaction chains?

Operation: Alice bids on Bob’s camera

1. Insert bid to Alice’s Bids
2. Update highest bid on Bob’s Items
Low latency with first-hop return

Alice's Bids

<table>
<thead>
<tr>
<th>Alice</th>
<th>Book</th>
<th>$100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Camera</td>
<td>$500</td>
</tr>
</tbody>
</table>

Bob’s Items

| Bob | Camera | $500 |

Alice bid on Bob’s camera
Problem: what if chains fail?

1. What if servers fail after executing first-hop?

2. What if a chain is aborted in the middle?
**Solution: provide all-or-nothing atomicity**

1. Chains are durably logged at first-hop
   - Logs are replicated to another closest data center
   - Chains are re-executed upon recovery

2. Chains allow user-aborts only at first hop
   - First hop commits $\rightarrow$ all hops eventually commit
Problem: non-serializable interleaving

- Concurrent chains ordered inconsistently at different hops

```
X=1
Y=1
X=2
Y=2
```

Server-X: $T_1 < T_2$
Server-Y: $T_2 < T_1$

Not serializable!

- Traditional 2PL+2PC prevents non-serializable interleaving at the cost of high latency
Static Analysis

- Statically analyze all chains to be executed
  - Web applications invoke fixed set of operations

A SC-cycle has both red and blue edges

Serializable if no SC-cycle [Shasha et. al TODS’95]
How Lynx uses chains

- **User chains**: used by programmers to implement application logic
- **System chains**: used internally to maintain
  - Secondary indexes
  - Materialized join views
  - Geo-replicas
Example: secondary index

<table>
<thead>
<tr>
<th>Bidder</th>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Camera</td>
<td>$100</td>
</tr>
<tr>
<td>Bob</td>
<td>iPhone</td>
<td>$20</td>
</tr>
</tbody>
</table>

Bidder | Item  | Price |
--------|-------|-------|
Alice   | Camera| $100  |
Bob     | Car   | $20   |

Alice   | Book  | $20   |
Bob     | Car   | $20   |
Alice   | iPhone| $100  |
Bob     | Camera| $100  |
Example user and system chain

Alice

Datacenter-1

bid on Bob’s camera

Bob

Datacenter-2

Alice  Book  $100

Alice  Camera  $100

Bob  Camera  $100
**Static Analysis**

One solution: execute chain as a distributed transaction
False conflicts in user chains

- Put-bid
  - Insert to *Bids* table
  - Insert to *Bids* table
  - Update *Items* table
  - Update *Items* table

False conflict because $\max(bid, \text{current\_price})$ commutes
Solution: users annotate commutativity
System chains
Solution: chains provide origin-ordering

- Observation: conflicting system chains originate at the same first hop server.

- Origin-ordering: if chains T1 < T2 at same first hop, then T1 < T2 at all subsequent overlapping hops.
  - Can be implemented cheaply \(\rightarrow\) sequence number vectors
Limitations of Lynx/chains

1. Chains are not strictly serializable, only serializable.
2. Programmers can abort only at first hop