10. Replication

CSEP 545 Transaction Processing
Philip A. Bernstein
Sameh Elnikety

Copyright ©2012 Philip A. Bernstein
Outline

1. Introduction
2. Primary-Copy Replication
3. Multi-Master Replication
4. Other Approaches
5. Products
1. Introduction

• Replication - using multiple copies of a server or resource for better availability and performance.
  – Replica and Copy are synonyms

• If you’re not careful, replication can lead to
  – worse performance - updates must be applied to all replicas and synchronized
  – worse availability - some algorithms require multiple replicas to be operational for any of them to be used
Read-only Database

- $T_1 = \{ r[x] \}$
Update-only Database

- $T_1 = \{ w[x=1] \}$
- $T_2 = \{ w[x=2] \}$
Update-only Database

- $T_1 = \{ w[x=1] \}$
- $T_2 = \{ w[y=1] \}$
Replicated Database

- **Objective**
  - Availability
  - Performance

- **Transparency**
  - 1 copy serializability

- **Challenge**
  - Propagating and synchronizing updates
Replicated Server

- Can replicate servers on a common resource
  - Data sharing - DB servers communicate with shared disk

  ![Diagram of a replicated server system](image.png)

- Helps availability for process (not resource) failure
- Requires a replica cache coherence mechanism, so this helps performance only if
  - little conflict between transactions at different servers or
  - loose coherence guarantees (e.g. read committed)
Replicated Resource

• To get more improvement in availability, replicate the resources (too)
• Also increases potential throughput
• This is what’s usually meant by replication
• It’s the scenario we’ll focus on
Synchronous Replication

- Replicas function just like a non-replicated resource
  - Txn writes data item x. System writes all replicas of x.
  - Synchronous – replicas are written within the update txn
  - Asynchronous – One replica is updated immediately. Other replicas are updated later

- Problems with synchronous replication
  - Too expensive for most applications, due to heavy distributed transaction load (2-phase commit)
  - Can’t control when updates are applied to replicas
Synchronous Replication - Issues

• If you just use transactions, availability suffers.
• For high-availability, the algorithms are complex and expensive, because they require heavy-duty synchronization of failures.
• … of failures? How do you synchronize failures?
• Assume replicas $x_A$, $x_B$ of $x$ and $y_C$, $y_D$ of $y$

\[
\begin{align*}
  r_1[x_A] & \rightarrow y_D \text{ fails} & \rightarrow w_1[y_C] \\
  r_2[y_D] & \rightarrow x_A \text{ fails} & \rightarrow w_2[x_B]
\end{align*}
\]

Not equivalent to a one-copy execution, even if $x_A$ and $y_D$ never recover!

• DBMS products support it only in special situations
Atomicity & Isolation Goal

• One-copy serializability (abbr. \textit{ISR})
  – An execution of transactions on the replicated database has the same effect as a serial execution on a one-copy database.

• \textit{Readset} (resp. \textit{writeset}) - the set of data items (not copies) that a transaction reads (resp. writes).

• 1SR Intuition: the execution is SR \textit{and} in an equivalent serial execution, for each txn T and each data item x in readset(T), T reads from the most recent txn that wrote into \textbf{any} copy of x.

• To check for 1SR, first check for SR (using SG), then see if there’s equivalent serial history with the above property
Atomicity & Isolation (cont’d)

• Previous example was not 1SR. It is equivalent to
  - $r_1[x_A] w_1[y_C] r_2[y_D] w_2[x_B]$ and
  - $r_2[y_D] w_2[x_B] r_1[x_A] w_1[y_C]$
  - but in both cases, the second transaction does not read its input from the previous transaction that wrote that input.

• These are 1SR
  - $r_1[x_A] w_1[y_D] r_2[y_D] w_2[x_B]$
  - $r_1[x_A] w_1[y_C] w_1[y_D] r_2[y_D] w_2[x_A] w_2[x_B]$

• The previous history is the one you would expect
  - Each transaction reads one copy of its readset and writes into all copies of its writeset

• But it may not always be feasible, because some copies may be unavailable.
Asynchronous Replication

- Asynchronous replication
  - Each transaction updates one replica.
  - Updates are propagated later to other replicas.

- Primary copy: Each data item has a primary copy
  - All transactions update the primary copy
  - Other copies are for queries and failure handling

- Multi-master: Transactions update different copies
  - Useful for disconnected operation, partitioned network

- Both approaches ensure that
  - Updates propagate to all replicas
  - If new updates stop, replicas converge to the same state

- Primary copy ensures serializability, and often 1SR
  - Multi-master does not.
2. Primary-Copy Replication

- Designate one replica as the primary copy (publisher)
- Transactions may update only the primary copy
- Updates to the primary are sent later to secondary replicas (subscribers) in the order they were applied to the primary

T1: Start
... Write(x1) ...
Commit

T2
...

Tn

Primary Copy

x1

Secondary

x2

... 

xm
Update Propagation

• Collect updates at the primary using triggers or by post-processing the log
  – Triggers: on every update at the primary, a trigger fires to store the update in the update propagation table.
  – Log post-processing: “sniff” the log to generate update propagations

• Log post-processing (log sniffing)
  – Saves triggered update overhead during on-line txn.
  – But R/W log synchronization has a (small) cost

• Optionally identify updated fields to compress log
• Most DB systems support this today.
Update Processing 1/2

- At the replica, for each tx T in the propagation stream, execute a refresh tx that applies T’s updates to replica.

- Process the stream serially
  - Otherwise, conflicting transactions may run in a different order at the replica than at the primary.
  - Suppose log contains \(w_1[x] c_1 w_2[x] c_2\). Obviously, \(T_1\) must run before \(T_2\) at the replica.
  - So the execution of update transactions is serial.

- Optimizations
  - Batching: \(\{w(x)\} \{w(y)\} \rightarrow \{w(x), w(y)\}\)
  - “Concurrent” execution
Update Processing 2/2

• To get a 1SR execution at the replica
  – Refresh transactions and read-only queries use an atomic and isolated mechanism (e.g., 2PL)

• Why this works
  – The execution is serializable
  – Each state in the serial execution is one that occurred at the primary copy
  – Each query reads one of those states

• Client view
  – Session consistency
Request Propagation

- An alternative to propagating updates is to propagate procedure calls (e.g., a DB stored procedure call).

- Or propagate requests (e.g. txn-bracketed stored proc calls)

- Requirements
  - Must ensure same order at primary and replicas
  - Determinism

- This is often a txn middleware (not DB) feature.
Failure & Recovery Handling 1/3

- Secondary failure - nothing to do till it recovers
  - At recovery, apply the updates it missed while down
  - Needs to determine which updates it missed, just like non-replicated log-based recovery
  - If down for too long, may be faster to get a whole copy

- Primary failure
  - Normally, secondaries wait till the primary recovers
  - Can get higher availability by electing a new primary
  - A secondary that detects primary’s failure starts a new election by broadcasting its unique replica identifier
  - Other secondaries reply with their replica identifier
  - The largest replica identifier wins
Failure & Recovery Handling 2/3

• Primary failure (cont’d)
  – All replicas must now check that they have the same updates from the failed primary
  – During the election, each replica reports the id of the last log record it received from the primary
  – The most up-to-date replica sends its latest updates to (at least) the new primary.
Failure & Recovery Handling 3/3

• Primary failure (cont’d)
  – Lost updates
  – Could still lose an update that committed at the primary and wasn’t forwarded before the primary failed … but solving it requires synchronous replication (2-phase commit to propagate updates to replicas)
  – One primary and one backup
    • There is always a window for lost updates.
Communications Failures

- Secondaries can’t distinguish a primary failure from a communication failure that partitions the network.

- If the secondaries elect a new primary and the old primary is still running, there will be a reconciliation problem when they’re reunited. This is multi-master.

- To avoid this, one partition must know it’s the only one that can operate. It can’t communicate with other partitions to figure this out.

- Could make a static decision. E.g., the partition that has the primary wins.

- Dynamic solutions are based on Majority Consensus
Majority Consensus

- Whenever a set of communicating replicas detects a replica failure or recovery, they test if they have a majority (more than half) of the replicas.
- If so, they can elect a primary
- Only one set of replicas can have a majority.
- Doesn’t work with an even number of copies.
  - Useless with 2 copies
- Quorum consensus
  - Give a weight to each replica
  - The replica set that has a majority of the weight wins
  - E.g. 2 replicas, one has weight 1, the other weight 2
3. Multi-Master Replication

- Some systems **must** operate when partitioned.
  - Requires many updatable copies, not just one primary
  - Conflicting updates on different copies are detected late

- Classic example - salesperson’s disconnected laptop
  - Customer table (rarely updated) Orders table (insert mostly)
  - Customer log table (append only)
  - So conflicting updates from different salespeople are rare

- Use primary-copy algorithm, with multiple masters
  - Each master exchanges updates (“gossips”) with other replicas when it reconnects to the network
  - Conflicting updates require reconciliation (i.e. merging)

- In Lotus Notes, Access, SQL Server, Oracle, …
Example of Conflicting Updates

• Assume all updates propagate via the primary

<table>
<thead>
<tr>
<th>Time</th>
<th>Replica 1</th>
<th>Primary</th>
<th>Replica 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially x=0</td>
<td>Initially x=0</td>
<td>Initially x=0</td>
<td>Initially x=0</td>
</tr>
<tr>
<td>T₁: X=1</td>
<td>X=1</td>
<td>X=2</td>
<td>X=1</td>
</tr>
<tr>
<td>Send (X=1)</td>
<td>Send (X=1)</td>
<td>Send (X=2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X=2</td>
<td>X=1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Send (X=2)</td>
<td></td>
<td>X=1</td>
</tr>
</tbody>
</table>

• Replicas end up in different states
Thomas’ Write Rule

- To ensure replicas end up in the same state
  - Tag each data item with a timestamp
  - A transaction updates the value and timestamp of data items (timestamps monotonically increase)
  - An update to a replica is applied only if the update’s timestamp is greater than the data item’s timestamp
  - You only need timestamps of data items that were recently updated (where an older update could still be floating around the system)

- All multi-master products use some variation of this

Robert Thomas, *ACM TODS*, June ’79
Thomas Write Rule \(\not\Rightarrow\) Serializability

- Replicas end in the same state, but neither T₁ nor T₂ reads the other’s output, so the execution isn’t serializable.
- This requires reconciliation
Multi-Master Performance

• The longer a replica is disconnected and performing updates, the more likely it will need reconciliation

• The amount of propagation activity increases with more replicas
  – If each replica is performing updates, the effect is quadratic in the number of replicas
Making Multi-Master Work

• Transactions
  – $T_1$: $x++ \{x=1\}$ at replica 1
  – $T_2$: $x++ \{x=1\}$ at replica 2
  – $T_3$: $x++ \{y=1\}$ at replica 3
  – Replica 2 and 3 already exchanged updates

• On replica 1
  – Current state $\{x=1, y=0\}$
  – Receive update from replica 2 $\{x=1, y=1\}$
  – Receive update from replica 3 $\{x=1, y=1\}$
Making Multi-Master Work

• Time in a distributed system
  – Emulate global clock
  – Use local clock
  – Logical clock
  – Vector clock

• Dependency tracking metadata
  – Per data item
  – Per replica
  – This could be bigger than the data
Microsoft Access and SQL Server

• Each row $R$ of a table has 4 additional columns
  – Globally unique id (GUID)
  – Generation number, to determine which updates from other replicas have been applied
  – $\text{Version num} = \text{the number of updates to } R$
  – Array of $[\text{replica}, \text{version num}]$ pairs, identifying the largest version num it got for $R$ from every other replica

• Uses Thomas’ write rule, based on version nums
  – Access uses replica id to break ties.
  – SQL Server 7 uses subscriber priority or custom conflict resolution.
4. Other Approaches (1/2)

- Non-transactional replication using timestamped updates and variations of Thomas’ write rule
  - Directory services are managed this way
- Quorum consensus per-transaction
  - Read and write a quorum of copies
  - Each data item has a version number and timestamp
  - Each read chooses a replica with largest version number
  - Each write increments version number one greater than any one it has seen
  - No special work needed for a failure or recovery
Other Approaches 2/2

• Read-one replica, write-all-available replicas
  – Requires careful management of failures and recoveries
• E.g., Virtual partition algorithm
  – Each node knows the nodes it can communicate with, called its view
  – Txn T can execute if its home node has a view including a quorum of T’s readset and writeset
  – If a node fails or recovers, run a view formation protocol (much like an election protocol)
  – For each data item with a read quorum, read the latest version and update the others with smaller version #.
Summary

• State-of-the-art products have rich functionality.
  – It’s a complicated world for app designers
  – Lots of options to choose from

• Most failover stories are weak
  – Fine for data warehousing
  – For 24×7 TP, need better integration with cluster node failover