INF-5360 Presentation
Optimistic Replication

Ali Ahmad

April 29, 2013
Data replication is a key technology in distributed systems that enables higher availability and performance. This article surveys optimistic replication algorithms. They allow replica contents to diverge in the short term to support concurrent work practices and tolerate failures in low-quality communication links. The importance of such techniques is increasing as collaboration through wide-area and mobile networks becomes popular.

Optimistic replication deploys algorithms not seen in traditional "pessimistic" systems. Instead of synchronous replica coordination, an optimistic algorithm propagates changes in the background, discovers conflicts after they happen, and reaches agreement on the final contents incrementally.

We explore the solution space for optimistic replication algorithms. This article identifies key challenges facing optimistic replication systems—ordering operations, detecting and resolving conflicts, propagating changes efficiently, and bounding replica divergence—and provides a comprehensive survey of techniques developed for addressing these challenges.
Structure of presentation

- Pessimistic and optimistic replication
- Elements of Optimistic replication
- Eventual consistency
  - Scheduling
  - Conflict detection
  - Conflict resolution
  - Commitment protocols
- State-transfer systems
  - Delete/update ambiguity.
- Propagation
Why replication?

- increased data availability
- improved performance
- increased throughput
What is pessimistic replication?

- Single-copy replication is pessimistic
- What is missing? why do we need more?

- Why optimistic replication?
  - performance and availability in issues WANs.
  - Mobile communication
  - Scalability issues
    - Frequent updates
    - Many replicas.
  - Multiple writers
## Difference between optimistic and pessimistic replication

<table>
<thead>
<tr>
<th>Operation</th>
<th>Optimistic</th>
<th>Pessimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>WANs</td>
<td>Works well</td>
<td>Slows down the system</td>
</tr>
<tr>
<td>Flexible and dynamic networks</td>
<td>Yes</td>
<td>Nope!</td>
</tr>
<tr>
<td>Scalability</td>
<td>Better</td>
<td>Good</td>
</tr>
<tr>
<td>Autonomy</td>
<td>Pretty much</td>
<td>Not so much</td>
</tr>
<tr>
<td>Communication</td>
<td>Background propagation</td>
<td>Frequently</td>
</tr>
<tr>
<td>Consistency guarantees</td>
<td>Eventual consistency with possible bounded divergence</td>
<td>Strong or possibly single-copy consistency</td>
</tr>
</tbody>
</table>
What is optimistic replication?

A replication technique where:
- Consistency is relaxed
  - Less ACID, more like "base"
- More autonomy for each replica
- Higher degree on Asynchrony
- Requires less frequent communication.
- Background propagation of updates
Challenges of optimistic replication

- The key difference is in concurrency control
- Trade-off between availability and consistency
- The challenges are:
  - Divergence of replicas
  - Conflict detection & resolution
- Applications.
  - DNS
  - Usenet
  - Personal digital assistance (PDA)
  - Bayou mobile database system
  - CVS, software version control
Elements of optimistic replication?

- Operation submission
- Scheduling
- Conflict detection and resolution
- Propagation
- Commitment
Some Definitions

Most definitions were self-explanatory

- Operations
- Vector clocks
- Master replica
- Tentative operation
Types of optimistic replication

- Single vs. multiple writers
- State-transfer vs Operation-transfer
- Semantic vs Syntactic scheduling
- Semantic vs syntactic conflict handling
Writers

Single writer is when a (master) replica submit, schedules and propagates the updates

- Limited availability when updates are frequent or large
- Encounter $O(M)$ concurrent updates.

Multiple writers enables multiple replicas to submit updates

- Higher availability, but more complex
- Scheduling issues
- Conflict management
  - Does not scale well when conflicts rate is high.
  - Encounters $O(M^2)$ concurrent updates (worst case)
Transfer types

In state-transfer solutions, the entire newest object is sent and overwritten.

▶ Simple
▶ Not so flexible for conflict management.

In operation-transfer systems, only the update-operations are sent.

▶ Each replica must maintain a list of operations and their order.
▶ More flexible for conflict management
▶ Saves bandwidth
▶ Efficient when updates are large
Schedulers

Because updates are propagates in background, replicas might receive updates in different order.

- A scheduler orders these operations for satisfy
  - Replica’s states are the same
  - Replica’s states satisfy user’s expectation. ”Read your writes”
- Operations can be committed *tentatively* until scheduled.
- Syntactic, orders all updates according to a rule.
  - Who, When and where updates were submitted
  - Simple, but can cause unnecessary conflicts.
  - E.g. Time-stamps
Semantic scheduler

- Used only in operation-transfer
- Can re-order operations

Exploits properties such as:

**Commutativity**

In mathematics, a binary operation is commutative if changing the order of the operands does not change the result.\(^1\)

Basically, independent updates can be ordered and executed in any order.

---

\(^1\)Wikipedia ”Commutative property”
Semantic scheduler cont.

- Idempotence

**Idempotence**

*is the property of certain operations ... that can be applied multiple times without changing the result beyond the initial application.*[^2]

Operation that do not change the value of an object, when repeated.

[^2]: Wikipedia "Idempotence"
Happened-before problem

Consider two updates $A$ and $B$ that were submitted to site $i$ and $j$ respectively.

**Definition**

*Update $A$ happened before $B$ if:*

1. *Site $i = j$ and $A$ was submitted before $B$*
2. *Site $i \neq j$ and $B$ was submitted after operation $A$ was received and executed*
3. *Site $i \neq j$ and for some operation $C$, $A$ happened before $C$, and $C$ happened before $B$*

If $A$ did not happen before $B$ or vice versa, they are considered concurrent.
Happened-before Solutions

- Explicit representation
  - Operations are sent with a history of predecessors.
- Vector clocks
- Real-time and logical clocks
Handling conflicts

- Pessimistic technique use:
  - Blocking or aborting to avoid conflicts.
  - Overwriting

- Optimistic techniques used:
  - Syntactically (simpler, but more false positives)
  - Semantically (complex, but more flexible and application specific)
Handling conflicts

- Pessimistic technique use:
  - Blocking or aborting to avoid conflicts.
  - Overwriting

- Optimistic techniques used:
  - Syntactically (simpler, but more false positives)
  - Semantically (complex, but more flexible and application specific)
Eventual Consistency

defined as a system where:

1. There is a set of ordered and committed operation that are the same for all replicas.
2. These lists grow monotonically.
3. The committed operations satisfy the pre-conditions
4. For every submitted operation, they are either committed and executed, or only committed.

The latter is used for conflict resolution
Scheduling

All schedulers order the operation to preserve "happened-before" relationship.

- **Syntactic**
  - Scalar clock
  - Vector clock
  - Logs

- **Semantic**
  - Canonical ordering
  - Operation transformation
  - Optimization approach
Syntactic Scheduling

Typical properties used are:

- Scalar (TSAE, Active directory, Usenet)
- Vector clocks (Locus, Coda)
- Logs used as FIFO-lists
  - Operations are appended
  - The log position is used as logical clock
Canonical ordering

- All possible concurrent operations have a rule for the orders of updates.
- For instance, file "/mydir/myfile/" was modified, then "/mydir" was removed
Operational transformation

- For all possible concurrent operation, a re-write rule exists to guarantee convergence
- Operations are re-written.
Optimization approach

Uses a constraint concept between operations:

<table>
<thead>
<tr>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Choice, either operation A or B</td>
</tr>
<tr>
<td>2. Implication, if A happens so does B</td>
</tr>
<tr>
<td>3. Dependence, A executes only after B</td>
</tr>
</tbody>
</table>

The goal is to find the ”best” order of operation within constraints. The order with least conflict is chosen as best.
Conflict Detection

Definition

If the pre-conditions of an operation are not satisfied, they are said to be in conflict.

- Thomas’s write rule (replace with newest update)
- Syntactic detectors use “happened-before” to flag conflicts.
- Semantic detectors use operation-semantics
  - More expressive, some applications let users write preconditions
Conflict Resolution

- Manual, users inaction required.
- Automatic
  - Mostly application specific routines that take two version, and return one.
  - For instance, concurrent update to *.0 files, can be resolves by re-compiling.
  - Bayou has a *merge procedure* for each *dependency check*.
Commitment Protocols

Ensure:

1. Agreement among replicas about ordering.
2. Which operation are "stable" and will not be rolled-back.
3. Which "history" can be deleted.

Different techniques:

- Common knowledge
- Time-Stamped Anti-Entropy
- Time-stamped matrices
- Consensus
Common Knowledge

- Single-master systems
- Thomas’s write rule
- Timestamps
Time-Stamped Anti-Entropy

The concept is to use ACK-vectors to synchronise the progress of updates.

- For instance, Replica\(_i\) has an ACK-vector

\[
AV_i = \{ \text{Last operation received by replica}_0, \ldots, \text{replica}_n \} 
\]

- If \(AV_i[j] = t\), then the replica\(_j\) has received updates up-to time \(t\).
- These vectors are propagates in an Anti-entropy manner.

For liveness and efficiency, the timestamps are loosely based on real-time.
Time-stamped Matrices

Similar to TSAE.

- Replica_i has an N x M matrix
- Row_r, is i’s estimate of replica r’s vector clock.
- Exchanged similarly to TSAE.
- Both techniques only agree on the order of operations
Consensus

- Single-writer will increase the commit sequence number
- Voting algorithms
- In two-phase commit procedures each replica will:
  - Check if operation conflicts and vote
  - Send a commit notice, if majority is reached.
State-transfer Systems

Replicas converge by receiving the entire newest content.

- Thomas’s write rule
- Two-timestamp algorithm
- Modified bit algorithm
- Vector clock variations
- Hash histories
- Tombstones
Thomas’s write rule

The idea is:

- Replicas keep a time-stamp that represent the "newness’ of their content.
- When exchanging TS, the newer "wins"
- The "loser" requests the newer content and discards it’s own.
Update/delete ambiguity

Issue with Thomas’s write rule. Consider the following:

- Replica$_i$ receives a remove operation for an object.
- Replica$_k$ updates that same object later.
- Replicas$_i$ has to time-stamp to compare to.

Two possible solutions:

- User interaction
- Tombstones
Two-timestamp algorithm

Each replicas stores:

1. A time-stamp for "newness".
2. A "previous" time-stamp.

A conflicts is detected when previous time-stamps are different.

Flaws:

- May detect false conflicts.
- Especially when updates are frequent.
Modified-bit algorithm

Use a set of bits to indicate changes such as:

- Modification
- Removal
- Archived

- Each set bit is indicative of difference that has to be reconciled.
- Work only with previously synchronized entities
Version vectors

When replica $i$ exchanges VVs with replica $j$ then:

- $VV_i = VV_j$, indicates no modification
- $VV_i \neq VV_j$, If $VV_i$ dominates $VV_j$ then $i$ is newer than $j$
- Otherwise, concurrent operations are marked.

More accurate than two-timestamp algorithm
Version Timestamps

Adaptation of Version vectors to create and remove replicas on-the-fly. Three types of operations:

- **Fork**, creates a new replica
- **Join(i,j)**, merges content of $i$ into $j$ and destroying $j$
- **Update($i$)**
Hash histories

Uses hash of content instead of time-stamps to represent state of replicas.
Culling Tombstones

Can be used to resolve the issue with Thomas’s write rule. The idea is:

- Removed objects get tombstones for a long time.
- Can lead to huge overhead in storage.
- Can not be removed in case an updates arrives.
- This is solved by few replicas keeping tombstones indefinitely.
Propagation Axes

- Degree of synchrony (speed and frequency of communication and reconciliation)
- Communication topology (star, mesh, random)
- Generally, quick propagation results in lower delay and less conflicts, but also introduces complexity and overhead.
Propagation

- Using
  - Version vectors
  - Hybrid state-operation transfer
  - Collision resistant hash-functions
  - Self-reconciliation approach.

- Topology

- Transfer techniques
Operation-propagation: Version clocks

Used to show timestamps for replicas last operation.

- The difference is used to show the exact operations needs for synchronization.
- Ensures that no duplicate updates are sent.
Staet-Propagation: Hybrid

The idea is to only send diffs

- If synchronized, replicas can only send diffs as updates.
- If not, the entire state is transferred.
Staet-Propagation: Hierarchical object division and comparison

The idea is to avoid sending parts of an objects that were not changed.

- Intermediate nodes record time-stamps for their children.
- Uses Thomas’s write rule to replace sub-objects.
- Can narrow the conflict down to sub-objects.
- Replicas can maintain a list of modified sub-objects.
State-Propagation: Collision-resistant hash-functions

Divides objects into chunks.

1. Sender sends the hash values for chunks.
2. Receiver requests the chunks missing locally.
3. Does not work well when bytes are inserted or removed from the middle of objects.
4. Objects can be hashed from the opposite side using rsync.
Self-reconciliation approach

The idea is:

- Senders apply a special polynomial functions to its hash values.
- Receivers solve the polynomial equation and discover the exact lacking pieces.
Topology

Remember propagation Axes?

- Two-tiered replicas, core and mobile.
- Pessimistic approach among core replicas and optimistic among mobile replicas.
- This scales better, at the cost of network flexibility.
Push-transfer techniques

- Blind flooding
- Link-state monitoring
- Multicast-based
- Time-stamp matrices.
Controlling replicas divergence

- Making data unavailable.
- Enforcing "read/write" ordering.
- Explicit dependencies.
- Session Guarantees
  - Read your writes
  - Monotonic reads
  - Writes follow reads
  - Monotonic writes
## Conclusion

<table>
<thead>
<tr>
<th></th>
<th>Single master, state transfer</th>
<th>Single master, op transfer</th>
<th>Multi master, state transfer</th>
<th>Multi master, op transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Availability</strong></td>
<td>low: master single point of failure</td>
<td></td>
<td>high</td>
<td></td>
</tr>
<tr>
<td><strong>Conflict resolution flexibility</strong></td>
<td>N/A</td>
<td>inflexible</td>
<td>flexible: semantic operation scheduling</td>
<td></td>
</tr>
<tr>
<td><strong>Algorithmic complexity</strong></td>
<td>very low</td>
<td>low</td>
<td>high: scheduling and commitment.</td>
<td></td>
</tr>
<tr>
<td><strong>Space overhead</strong></td>
<td>low: Tombstones</td>
<td>high: log</td>
<td>low: Tombstones</td>
<td>high: log</td>
</tr>
<tr>
<td><strong>Network overhead</strong></td>
<td>$O(\text{object-size})$</td>
<td>$O(\text{#operations})$</td>
<td>$O(\text{object-size})$</td>
<td>$O(\text{#operations})$</td>
</tr>
</tbody>
</table>
Discussion

▶ Are WANs still as unreliable and slow as then (2005)?
▶ Do we want to let users to resolve conflicts?
▶ Are two-tiered solutions better than complete optimistic approaches?
Discussion

- Are WANs still as unreliable and slow as then (2005)?
- Do we want to let users to resolve conflicts?
- Are two-tiered solutions better than complete optimistic approaches?