Rethinking Eventual Consistency
A survey of synchronization techniques for replicated distributed databases

Phil Bernstein & Sudipto Das
Microsoft Research
June, 2013

Copyright © 2013 Microsoft Corp.
In a replicated database, updates arrive in different orders at different copies of a data item, but eventually the copies converge to the same value.
Eventual Consistency is All the Rage

- Was used in Grapevine (PARC, early 1980’s) and in numerous systems since then.
- Doug Terry et al. coined the term in a 1994 Bayou paper
- Werner Vogels at Amazon promoted it in Dynamo (2007)
- Cover topic of February 2012 IEEE Computer
Despite today’s hype

Most of what we’ll say was known in 1995

There are many published surveys
  But this talk has a rather different spin

We’ll often cite old references to remind you where the ideas came from
Ideally, replication is transparent

In the world of transactions:

- **One-Copy Serializability** - The system behaves like a serial processor of transactions on a one-copy database
  [Attar, Bernstein, & Goodman, IEEE TSE 10(6), 1984]

In the world of operations:

- **Linearizability** - A system behaves like a serial processor of operations on a one-copy database
  [Herlihy & Wing, ACM TOPLAS 12(3), 1990]
But you can’t in many practical situations

Let’s review the three main types of solutions

- Primary Copy
- Multi-Master
- Consensus Algorithms
Only the primary copy is updatable by clients

Updates to the primary flow downstream to secondaries

What if there’s a network partition?

Clients that can only access secondaries can’t run updates

[Alsberg & Day, ICSE 1976] [Stonebraker & Neuhold, Berkeley Workshop 1979]
Multi-Master

- Copies are independently updatable
- Conflicting updates on different copies are allowed
- Doesn’t naturally support 1SR.

To ensure eventual consistency or linearizability of copies:

- Either updates are designed to be commutative
- Or conflicting updates are detected and merged

- Popularized by Lotus Notes, 1989
Copies can be a replicated-state machine
- Essentially, a serial processor of operations
- Can be primary-copy or multi-master

Uses quorum consensus to achieve 1SR or linearizability.
- Ensures conflicting ops access at least one copy in common

Each downstream update is applied to a quorum of secondaries
The CAP Theorem

You can have only two of Consistency-of-Replicas, Availability, and Partition-Tolerance

- Can get C & A, if there’s no partition
- Can get C & P but only one partition can accept updates
- Can get A & P, but copies in different partitions won’t be consistent

Conjecture by [Brewer, PODC 2000]
Proved by [Gilbert & Lynch, SIGACT News 33(3) 2002]
“Partitioning - When communication failures break all connections between two or more active segments of the network ... each isolated segment will continue ... processing updates, but there is no way for the separate pieces to coordinate their activities. Hence ... the database ... will become inconsistent. This divergence is unavoidable if the segments are permitted to continue general updating operations and in many situations it is essential that these updates proceed.”

[Rothnie & Goodman, VLDB 1977]

So the CAP theorem isn’t new, but it does focus attention on the necessary tradeoff.
Can we do better than Eventual Consistency?

- There have been many attempts at defining stronger but feasible consistency
  - Parallel snapshot isolation
  - Consistent prefix
  - Monotonic reads
  - Timeline consistency
  - Linearizability
  - Eventually consistent transactions
  - Causal consistency
  - Causal+ consistency
  - Bounded staleness
  - Monotonic writes
  - Read-your-writes
  - Strong consistency

- There have been many attempts at defining stronger but feasible consistency:
We’ll try to eliminate the confusion by:

- Characterizing consistency criteria
- Describing mechanisms that support each one
- And summarizing their strengths and weaknesses
There are many excellent surveys of replication

We don’t claim ours is better, just different


B. Kemme, G. Alonso: Database Replication: a Tale of Research across Communities. PVLDB 3(1), 2010


There’s a huge literature on replication. Please tell us if we missed something important.

We’ll cover replication mechanisms in database systems, distributed systems, programming languages, and computer-supported cooperative work.

We won’t cover mechanisms in computer architecture.
Multi-master is designed to handle partitions

With primary copy, during a partition

- Majority quorum(x) = partition with a quorum of x’s copies
- Majority quorum can run updates and satisfy all correctness criteria
- Minority quorum can run reads but not updates, unless you give up on consistency

So an updatable minority quorum is just like multi-master
Eventual consistency – there are many good ways to achieve it

For isolation and session goals, the solution space is more complex

-Strengthens consistency, but complicates programming model
-Improves availability, but not clear by how much
-If a client rebinds to another server, ensuring these goals entails more expense, if they’re attainable at all.
-No clear winner
App needs to cope with arbitrary states during a partition

Offer a range of isolation and session guarantees and let the app developer choose among them
- Possibly worthwhile for distributed systems experts
- Need something simpler for “ordinary programmers”

Encapsulate solutions that offer good isolation for common scenarios
- Use data types with commutative operations
- Convergent merges of non-commutative operations
- Scenario-specific classes
We’ll start with the world of operations, and then look at the world of transactions
Can Have One Writable Partition

The partition with a quorum of replicas can run writes

Start here

Partition?

Y

N

Y

Consistent & Available

N

Quorum of replicas?

Y

N

Not available for updates
What to do about the bad case?

Start here

Partition?

N

Y

N

Y

Quorum of replicas?

Consistent & Available

To do better, we need to give up on consistency

Not available for updates
Eventual Consistency

- Eventual consistency is one popular proposal
  - The copies will be identical ... someday
  - App still needs to handle arbitrary intermediate states

- How to get it
  - Commutative downstream operations
  - Mergeable operations
  - Vector clocks
Assign a timestamp to each client write operation

- Each copy of x stores timestamp(last-write-applied)
- Apply downstream-write(x) only if downstream-write(x).timestamp > x.timestamp
- So highest-timestamp wins at every copy

Downstream writes arrive in this order:
- W(X=40), TS:1
- W(X=70), TS:5
- W(X=30), TS:3

Final value:
- X=70, TS:5
Pros

- Updates can be applied anywhere, anytime
- Downstream updates can be applied in any order after a partition is repaired

Cons

- Doesn’t solve the problem of ordering reads & updates
- For fairness, requires loosely-synchronized clocks
Convergent & Commutative Replicated Data Types

[Shapiro et al., INRIA Tech. Report, Jan 2011]

Set operations add/remove don’t commute,

\[ \text{[add}(E), \text{add}(E), \text{remove}(E)] \not\equiv \text{[add}(E), \text{remove}(E), \text{add}(E)] \]

But for a counting set, they do commute

- Each element E in set S has an associated count
- Add(set S, element E) increments the count for E in S.
- Remove(S, E) decrements the count
Pros
- Updates can be applied anywhere, anytime
- Downstream updates can be applied in any order after a partition is repaired

Cons
- Constrained, unfamiliar programming model
- Doesn’t solve the problem of ordering reads & updates
- Some app functions need non-commutative updates
Custom Merge Operations

Custom merge procedures for downstream operations whose client operations were not totally ordered.
- Takes two versions of an object and creates a new one
- For eventual consistency, merge must be commutative and associative

Notation: $M(O_2, O_1)$ merges the effect of $O_2$ into $O_1$

Commutative: $O_1 \cdot M(O_2, O_1) \equiv O_2 \cdot M(O_1, O_2)$

Associative: $M(O_3, O_1 \cdot M(O_2, O_1)) \equiv M(M(O_3, O_2) \cdot O_1)$

[Ellis & Gibbs, SIGMOD 1989]
Pros

- Enables concurrent execution of conflicting operations without the synchronization expense of total-ordering

Cons

- Requires application-specific logic that’s hard to generalize
Vector Clocks tell us the merging order

- In multi-master, each copy assigns a monotonically increasing version number to each client update

- **Vector clock** is an array of version numbers, one per copy
  - Identifies the set of updates received or applied

- Use it to identify the state that a client update depends on and hence overwrote
  - If two updates conflict but don’t depend on one another, then merge them.

- [Fischer & Michael, PODS 1982]
- [Parker et al., IEE TSE 1983]
- [Wuu & Bernstein, PODC 1984]
Problem: Discard or Merge?

- \( C_i \)
- \( C_k \)
- \( C_m \)

Update_1[x] \( \rightarrow \) \( w_1[x] \)

Update_2[x] \( \rightarrow \) \( w_2[x] \)

\( w_1[x] \) Discard or Merge?
Vector Clocks (2)

A vector clock can be used to identify the state that a client update depends on ("made-with knowledge")

- If $\text{VC}_1[k] \geq \text{vn}_2$, then $x_2$ was "made from" $x_1$ & should overwrite it
- If $\text{VC}_2[i] \geq \text{vn}_1$, then $x_1$ was "made from" $x_2$, so discard $x_2$
- Else the updates should be reconciled

[Ladin et al., TOCS, 1992]
[Malkhi & Terry, Dist. Comp. 20(3), 2007]
Another Use of Vector Clocks

- A copy can use it to identify the updates it has received
  - When it syncs with another copy, they exchange vector clocks to tell each other which updates they already have.
- Avoids shipping updates the recipient has already seen
- Enables a copy to discard updates that it knows all other copies have seen
In the Operation World

Start here

Partition?  
  Y  
  N

Consistent & Available

Quorum of replicas?  
  Y  
  N

Ops are commutative or mergeable

Eventually Consistent & Available

Not available for updates
Start here

Partition?

Y

Quorum of replicas?

Consistent & Available

N

Ops are commutative or mergeable

Y

Eventually Consistent & Available

N

Admissible executions

Causality constraints

Session constraints

Not available for updates

The case we can strengthen
**Causal Consistency**

**Definition** – The sequence of operations on each replica is consistent with session order and reads-from order.

- **Example:** User 1 stores a photo P and a link L to it. If user 2 reads the link, then she’ll see the photo.
- **Causality imposes write-write orders**

**Causal relationships:**

- **WW Session order:** \( w_1[y] \) executes after \( w_0[x] \) in session S
- **WR Session order:** \( w_3[z] \) executes after \( r_2[y] \) in session V
- **Reads-from order:** \( r_2[y] \) in session V reads from \( w_1[y] \) in session S
- **Causality is transitive:** Hence, \( w_0[x] \) causally precedes \( w_3[z] \)

[Lamport, CACM 21(7), 1978]
Causal Consistency (2)

- If all atomic operations preserve database integrity, then causal consistency with eventual consistency may be good enough
  - Store an object, then a pointer to the object
  - Assemble an order and then place it
  - Record a payment (or any atomically-updatable state)

- Scenarios where causal consistency isn’t enough
  - Exchanging items: Purchasing or bartering require each party to be credited and debited atomically
  - Maintaining referential integrity: One session deletes an object O while another inserts a reference to O
Implementing Causal Consistency

Enforce it using dependency tracking and vector clocks

COPS: Causality with convergent merge [Lloyd et al., SOSP 2011]

- Assumes multi-master replication
- Session context (dependency info) = <data item, version#> of the last items read or of the last item written.
- Each downstream write includes its dependent operations.
- A write is applied to a copy after its dependencies are satisfied
- Merge uses version vectors
- With additional dependency info, it can support snapshot reads
- Limitation: No causal consistency if a client rebinds to another replica due to a partition
Session Constraints

- **Read your writes** – a read sees all previous writes
- **Monotonic reads** – reads see progressively later states
- **Monotonic writes** – writes from a session are applied in the same order on all copies
- **Consistent prefix** – a copy’s state only reflects writes that represent a prefix of the entire write history
- **Bounded staleness** – a read gets a version that was current at time $t$ or later

[Terry et al., PDIS 1994]
Client session maintains IDs of reads and writes

- Accurate representation of the constraints
- High overhead per-operation

Client session maintains vector clocks for the last item read or written

- Compact representation of the constraints
- Conservative
The operation world ignores transaction isolation.

To get the benefits of commutative or mergeable operations, need a weaker isolation level.

- The operation world ignores transaction isolation.
- To get the benefits of commutative or mergeable operations, need a weaker isolation level.
Common weaker isolation levels

- Read committed
  - Transaction reads committed values

- Snapshot reads
  - Transaction reads committed values that were produced by a set of committed transactions
  - All of a transaction’s updates must be installed atomically to ensure the writeset is consistent in the minority partition
Is Weaker Isolation Acceptable?

- People do it all the time for better performance
  - Throughput of Read-Committed is 2.5x to 3x that of Serializable

- Weaker isolation produces errors. Why is this OK?

- No one knows, but here are some guesses:
  - DB's are inconsistent for many other reasons.
    - Bad data entry, bugs, duplicate txn requests, disk errors, ....
  - Maybe errors due to weaker isolation levels are infrequent
  - When DB consistency matters a lot, there are external controls.
    - People look closely at their paychecks
    - Financial information is audited
    - Retailers take inventory periodically
In the Transaction World

Start here

Partition?  
N

Consistent & Available

Y

Quorum of replicas?
N

Not available for updates

Y

Ops are commutative or mergeable

N

Eventually Consistent & Available

Read Committed or Snapshot Reads
Y

Consistent & Available

N
Other Admissibility Constraints

- Admissible executions
  - Causality constraints
  - Session constraints
  - Isolation constraints
    - RedBlue Consistency [Li et al., OSDI 2012]
    - 1-SR, Read-committed, Snapshot Isolation
    - Parallel Snapshot Isolation [Sovran et al, SOSP 2011]
    - Concurrent Revisions [Burckhardt et al., ESOP 2012]
RedBlue Consistency

*Blue* operations commute with all other operations and can run in different orders on different copies.

*Red* ones must run in the same order on all copies.

Use a side-effect-free *generator* operation to transform a red operation to a blue one that is valid in all states.

Example

- Deposit(acct, amt): acct.total = acct.total + amt
- EarnInterest(acct): acct.total = acct.total * 1.02

Deposit is blue, EarnInterest is red

Transform EarnInterest into:
- Interest = acct.total * 1.02  // runs locally at acct’s copy
- Deposit(acct, Interest)  // blue operation runs at all copies

[Li et al., OSDI 2012]
The history is equivalent to one of this form:

\[
\begin{align*}
&\text{r}_1[\text{readset}_1] \quad \text{w}_1[\text{writeset}_1] \\
&\text{r}_2[\text{readset}_2] \quad \text{w}_2[\text{writeset}_2] \\
&\text{r}_3[\text{readset}_3] \quad \text{w}_3[\text{writeset}_3] \\
\end{align*}
\]

\[
\begin{align*}
&\text{r}_4[\text{readset}_4] \quad \text{w}_4[\text{writeset}_4] \\
&\text{r}_5[\text{readset}_5] \quad \text{w}_5[\text{writeset}_5] \\
&\text{r}_6[\text{readset}_6] \quad \text{w}_6[\text{writeset}_6] \\
\end{align*}
\]

\[
\begin{align*}
&\text{ws}_1 \cap \text{ws}_2 \cap \text{ws}_3 = \emptyset \\
&\text{ws}_4 \cap \text{ws}_5 \cap \text{ws}_6 = \emptyset
\end{align*}
\]

Benefit of SI: Don’t need to test read-write conflicts
Parallel Snapshot Isolation (PSI)

- **Parallel SI** - Execution is equivalent to one that allows parallel threads with non-conflicting writesets running SI
- Allows a transaction to read stale copies

[Sovran, Power, Aguilera, & Li, SOSP 2011]
Example: Parallel SI

- A parallel SI execution may not be equivalent to a serial SI history
- Site 1 and Site 2 are each snapshot isolated.
- But the result is not equivalent to
  
  \[ T_1 \ T_2 \ T_3 \ T_4 \] or
  
  \[ T_3 \ T_4 \ T_1 \ T_2 \] or

  \[ \ldots \]

Site 1 has x’s primary
Site 2 has y’s primary
Concurrent Revisions

- Each arrow is an operation or transaction
- A fork defines a new private snapshot and a branch
- A join causes all updates on the branch to be applied
- Ops are pure reads or pure writes. Writes never fail.

[Burckhardt, et al., ESOP 2012]
In the Transaction World

Start here

Partition? Y N
Consistent & Available

Quorum of replicas? Y N
Ops are commutative or mergeable

Read Committed or Snapshot Reads Y N
Eventually Consistent & Available

Other Isolation Levels Y N
Not available for updates
RETURNING TO CAP ...
If the system guarantees only eventual consistency, then be ready to read nearly arbitrary database states.

Use commutative operations whenever possible.

- System needn’t totally order downstream writes, which reduces latency

Else use convergent merges of non-commutative ops

- Enables updates during partitioned operation and in multi-master systems
If availability and partition-tolerance are required, then consider strengthening eventual consistency with admissibility criteria.

If possible, use consistency-preserving operations, in which case causal consistency is enough.

Hard case for all admissibility criteria is rebinding a session to a different replica.

- Replica might be older or newer than the previous one it connected to.
# Enforcing Admissibility in a Minority Partition

<table>
<thead>
<tr>
<th></th>
<th>Session maintains connection to server</th>
<th>Session migrates to another replica</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Primary Copy or Quorum-based</td>
<td>Multi-master</td>
</tr>
<tr>
<td>Read-Your-Writes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Monotonic Writes</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bounded staleness</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>✓</td>
<td>😞</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Causality</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

?W: Only if the session caches its writes
?R: Only if the session caches its reads

Writes disabled
Encapsulate solutions that offer good isolation for common scenarios

- Commutative Replicated Data Types
- Convergent merges of non-commutative operations
- Research: Scenario-specific design patterns
  - Overbooking with compensations
  - Queued transactions
Does this design space matter?

- Probably not to enterprise developers

- Spanner [OSDI 2012] “Many applications at Google ... use Megastore because of its semi-relational data model and support for synchronous replication, despite its relatively poor write throughput.”

- Mike Stonebraker [blog@ACM, Sept 2010]: “No ACID Equals No Interest” for enterprise users

- Same comment from a friend at Amazon
So Why Bother?

The design space does matter to Einstein-level developers of high-value applications that need huge scale out.

People like you! 😊
Summary

Eventual consistency
- Commutative operations
  - Thomas’ write rule
  - Convergent data types
- Custom merge
  - Vector clocks

Admissible executions
- Causality constraints
- Session constraints
  - Read your writes
  - Monotonic reads
  - Monotonic writes
  - Consistent prefix
  - Bounded staleness

Isolation constraints