Performance vs. programmability in geo-replicated systems

Marc Shapiro

Joint work with:
- Marek Zawirski
- Masoud Saeida-Ardekani
- Nuno Preguiça

Survey of our work (and others)
Explore consistency spectrum
- Weakest strong consistency
- Strongest weak consistency
Survey of our work (and others)
Explore consistency spectrum
  • Weakest strong consistency
  • Strongest weak consistency
Cloud: powerful compute centre, parallel, fast
$10^3 - 10^6$ transactions/second
Ex. app: the set of employees of some company
Updates:
Create Bob
Remove Martin
Update Nellie (e.g. increase salary)
Distributed company:
- slow network ⇒ replicate data
- fast reads
- but synchro for updates is slow; wasting parallelism and compute power
(Synch = 100 ms ⇒ 10 transactions/second!!)

Synch is acceptable within a DC, not in geo-replication
Invariant

Total order of reads and writes = replicas are identical (FT)
Read from any replica, write to all
Synchronisation between each commit / start
  • Implem may be less strict, but optimisation opportunities are limited
Maintaining sequential invariant: trivial
Parallelise: Consistency Models

Remove synchronisation points
  • \(\implies\) Propagate asynchronously

Pros:
  • More implementation, execution choices
  • Performance
  • Availability

Cons
  • Invariants \(\implies\) Programmability

Acceptable parallelism \(\implies\) what invariants are important
  • depends on application
  • e.g. “stock \(\geq 0\)”
Survey of our work (and others)
Simplified view of consistency spectrum. +/− = properties gained/lost going up
Explore consistency spectrum; our work:
• Weakest strong consistency
• Strongest weak consistency

“Strong” = concurrent writes are not permitted = total ordered writes
Performance domains

2×–40× latency/throughput variation

[Performance vs. programmability in large-scale distributed systems]
Parallelise: Partitioned DB

Early versions of Facebook: per university

Disjoint parts:
- full parallelism
- no cross-part guarantees

OK for some apps but defeats the purpose

Now Facebook allows anyone to be friends
Snapshot Isolation (SI)

Parallelise: overlap reads and writes
Minimal sync: T2 commits iff 
\[ ws(T1) \cap ws(T2) = \emptyset \]

Read-only transactions do not wait or abort
Read from a cache
Write-skew: “x<y” problematic

2 things:
• parallelise by overlap
• decrease synch footprint with minimal commit sync
Decouple reads from writes
• read-only does not synchronise: good because reads >> writes
• decoupled: read from a cache: good for performance
Genuine partial replication

Full replication: / replicas ⇒ / synch, work
Partial replication: spread the load
GPR: a process only receives messages about objects that it replicates
- Decrease synchronisation footprint
- Not possible with SI ⇒ Weaker model

(GPR: a.k.a. Distributed Access Parallelism)

Full replication:
- every replica receives every update
- more replicas ⇒ more synchronisation, more network load, no perf. benefit

Genuine Partial Replication
- If I don't replicate X, I never receive a message about X (no work related to X)
- Does not affect the model, but incompatible with SI ⇒ apply to weaker model
Most parallel total-order model: NMSI

Scalability properties:
- Wait-Free Queries
- Forward Freshness
- Mini. Commitment Synchronisation
- Genuine Partial Replication

Partial replication: x, y, z
Read from consistent snapshot ≠ PSI:
- forward freshness: decreases aborts
- GPR: decreases footprint
More parallelism: allow concurrent updates \( \Rightarrow \) Availability

Lots of replicas \( \Rightarrow \) total order doesn't scale, not available

More availability:
- Tolerate network failures
- Mobile computing

Strongest weak consistency
"Weak" = concurrent writes are permitted = replicas may diverge (but re-converge)
Overlap updates: Asynchronous propagation

How about propagating updates asynchronously, rather than synchronising them?
A data centre makes progress without waiting for the other one
Concurrent updates (Martin || Bob)
Concurrent updates to Nellie: conflict????
   California: increase Nellie salary
   Paris: change Nellie’s department
Merging concurrent updates

Converges? Principles?

EC:
- asynch propagation
- merge concurrent updates

Very tricky, ad-hoc so far.
But how? Up to the user? Synchronise to ensure convergence? Depends on semantics?
Causal consistency

\[ S^0 \]

- \( u \) observed effects of \( u \) at source
- \( \implies \) \( v \) to be delivered after \( u \)
- Strongest always-available consistency

Constraint on the ordering of updates at downstream replicas
- Concurrent ops can be delivered in any order
- \( u \rightarrow v \implies (u \text{ observed before } v \text{ at every replica}) \)

Example: email, photo
Merging & convergence

Convergence conditions:
• Deterministic
• Dependent only on delivery set
• Not on delivery order, local info

Formally:
• State space is a semi-lattice
• Updates are monotonic
• Concurrent updates merged by join (least upper bound)

Ensures convergence by construction
Decentralised resolution:

**Highest timest. wins (LWW)**

Update: assign, overwrite file

- Transmit file + timestamp (*unique, monotonic*)
- Highest timestamp wins

Widely used

---

a.k.a. Thomas Write Rule

[Paul R. Johnson, Robert H. Thomas, The maintenance of duplicate databases, 1976, RFC 677]

1st, a protocol widely used, e.g. in replicated file systems (Microsoft Client-Side Caching) or in Active Directory, called Timestamped Writes or Last Writer Wins. It is very simple and most importantly does not require a consensus algorithm – but it has serious drawbacks.

Here we have Bob and Suzy independently writing values into file “x”. Every time someone changes the file, this updates the file’s timestamp. If two people write to the same file, LWW keeps the latest version. So every write action is timestamped; if the timestamp of a write is lower than the current timestamp of the file, the write has no effect – it reduces to a no-op.

This is great because it’s simple and somewhat intuitive. As long as sites eventually communicate, file “x” converges. No consensus is needed! The reason is that writes effectively commute. The final value doesn’t depend on the order in which the writes were received and applied.

The big drawback is that LWW does not maintain any invariants (other than timestamps).

- “If x>10 then x++” may be violated: (1) the condition may be false when write is applied; (2) x++ is not applied, just the value
- Does not maintain dependence between x and y (or even between successive updates to x)
Operations: assign (write), read
In a sequential (or h-b) execution, the usual register semantics apply
If there are concurrent assignments, subsequent reads return the set of assigned values.
  • The expectation is that the application will read this set and write out a reconciled value (in this example: sum).
Notes:
  • Not a set (no add, rm ops) ⇒ anomalies
  • Concurrent semantics ≠ seq. interleaving!
Set

Operations:
• \textit{add} (element \(e\))
• \textit{rmv} (element \(e\))
• \textit{lookup} (element \(e\)) : boolean

No duplicates
Extending the Set seq. spec.

Sequential specification of Set:
- \{true\} \text{add}(e) \quad \{e \in S\}
- \{true\} \text{rmv}(e) \quad \{e \not\in S\}

Commutative \((e \neq f)\):
- \{true\} \text{add}(e) \parallel \text{add}(e) \quad \{e \in S\}
- \{true\} \text{rmv}(e) \parallel \text{rmv}(e) \quad \{e \not\in S\}
- \{true\} \text{add}(e) \parallel \text{add}(f) \quad \{e, f \in S\}
- \{true\} \text{rmv}(e) \parallel \text{rmv}(f) \quad \{e, f \not\in S\}
- \{true\} \text{add}(e) \parallel \text{rmv}(f) \quad \{e \in S, f \not\in S\}

Ambiguous:
- \{true\} \text{add}(e) \parallel \text{rmv}(e) \quad \{????\}

[Performance vs. programmability in large-scale distributed systems]
Extending the Set seq. spec.

Sequential specification of Set:

- $\{true\} add(e) \{e \in S\}$
- $\{true\} rmv(e) \{e \not\in S\}$

Commutative ($e \neq f$):

- $\{true\} add(e) ; add(e) \{e \in S\}$
- $\{true\} rmv(e) ; rmv(e) \{e \not\in S\}$
- $\{true\} add(e) ; add(f) \{e, f \in S\}$
- $\{true\} rmv(e) ; rmv(f) \{e, f \not\in S\}$
- $\{true\} add(e) ; rmv(f) \{e \in S, f \not\in S\}$

What about:

- $\{true\} add(e) || rmv(e) \{??\}$

“Principle of

All sequential composition have same effect → let parallel composition have same effect”
Extending the Set seq. spec.

Sequential specification of Set:
- \{true\}  \text{add}(e) \{e \in S\}
- \{true\}  \text{rmv}(e) \{e \notin S\}

Commutative \((e \neq f)\):
- \{true\}  \text{add}(e) \parallel \text{add}(e) \{e \in S\}
- \{true\}  \text{rmv}(e) \parallel \text{rmv}(e) \{e \not\in S\}
- \{true\}  \text{add}(e) \parallel \text{add}(f) \{e,f \in S\}
- \{true\}  \text{rmv}(e) \parallel \text{rmv}(f) \{e, f \not\in S\}
- \{true\}  \text{add}(e) \parallel \text{rmv}(f) \{e \in S, f \not\in S\}

What about:
- \{true\}  \text{add}(e) \parallel \text{rmv}(e) \{??\}
add(e) || rmv(e)

\{true\} add(e) || rmv(e) \{????\}

• sequential consistency, linearisable?
• last writer wins? \{ add(e) < rmv(e) \Rightarrow e \not\in S \\
\land \ rmv(e) < add(e) \Rightarrow e \in S \} 
• error state? \{ \bot \not\in S \}
• add wins? \{ e \in S \}
• remove wins? \{ e \not\in S \}

Deterministic
• Independent of order of delivery
• Independent of local state
• No synchronisation

Not an interleaving semantics

Any of these is an acceptable semantics, i.e., ensures convergence
Satisfies the Monotonic Semi-Lattice property
Which one you choose depends on application semantics
Better than previous app-dependent approaches:
encapsulated in a data type
no synchronisation
guaranteed correct
Add-Wins Set (OR-Set)

\[
\{a\}.\text{add}(a) = \{a, a\} \quad \text{unique}
\]

\[
\{a, b\}.\text{rmv}(a) = \{a, b\} \quad \text{mark visible instances of } a
\]

\[
\{a, a\}.\text{contains}(a) \mid \text{true} \quad \exists \text{ non-marked instance of } a
\]

\[
\{a, b\}.\text{merge}(\{a, c\}) = \{a, b, c\} \quad \text{union + marker}
\]

\[
\{\text{true}\} \text{ add}(e) \parallel \text{rmv}(e) \{e \in S\}
\]

In case of add+remove, OR-Set gives precedence to add
## Conflict-free Replicated Data Types (CRDTs)

Encapsulate replication & resolution
Re-usable data types
Correct by construction

<table>
<thead>
<tr>
<th>Register</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Last-Writer Wins</td>
<td>• Unlimited</td>
</tr>
<tr>
<td>• Multi-Value</td>
<td>• Restricted non-negative</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Grow-Only</td>
<td>• Directed</td>
</tr>
<tr>
<td>• 2P</td>
<td>• Monotonic DAG</td>
</tr>
<tr>
<td>• Observed-Remove</td>
<td>• Edit graph</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Map</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Power of consensus: any sequential specification
What is the power of asynchronous systems?
SwiftCloud

Client-side replicas: hundreds (millions?)
Transactional causal consistency

Challenge:
• Fast writes
• Causal delivery at scale, FT
• Small metadata
• Avoid waits (even more overlap)

Overlapped reads: slightly stale
Hybrid system

- Asynch: As fast and available as possible
- Synch: Strongly consistent when necessary

*Delivering an asynchronous update must never violate the invariant*

Examples:

- $x \leq y$
- $A \subseteq B$
- $P \implies Q$
- decrease LHS, increase RHS = asynchronous

Express application invariants + update pre/postconditions
Explored the consistency spectrum. Focused on “weakest strong” and “strongest weak”.
Summary

**Weakening total-order** updates:
- Wait-Free Queries
- Forward Freshness
- Mini. Commitment Synchronisation
- Genuine Partial Replication

**Strengthening partial-order** updates:
- Causal order
- Transactions
- Deterministic merge

**CRDTs**: correct, convergent by construction
- Register, Counter, Set, Map, Graph, Monotonic DAG, Sequence

**Ongoing work:**
- Based on application semantics
- Total order + partial order