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

Acknowledgments

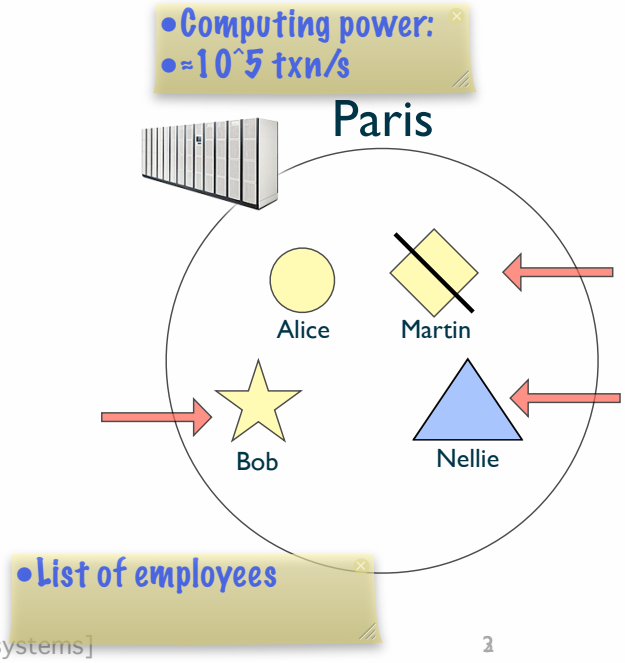
Association for Computing Machinery
Distinguished Speaker Program

SyncFree

ConcoRDanT
PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

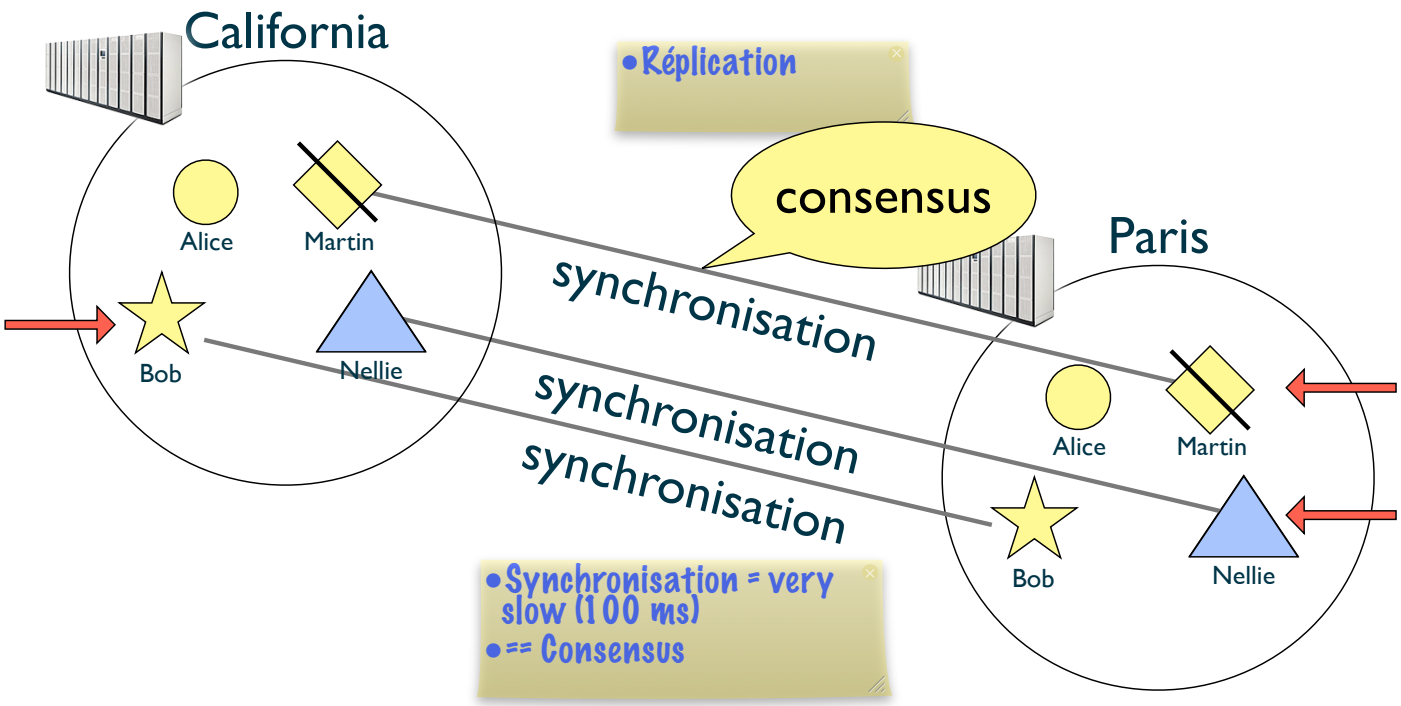
Survey of our work (and others)
Explore consistency spectrum
- Weakest strong consistency
- Strongest weak consistency

1 data centre



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)

Strongly consistent replication

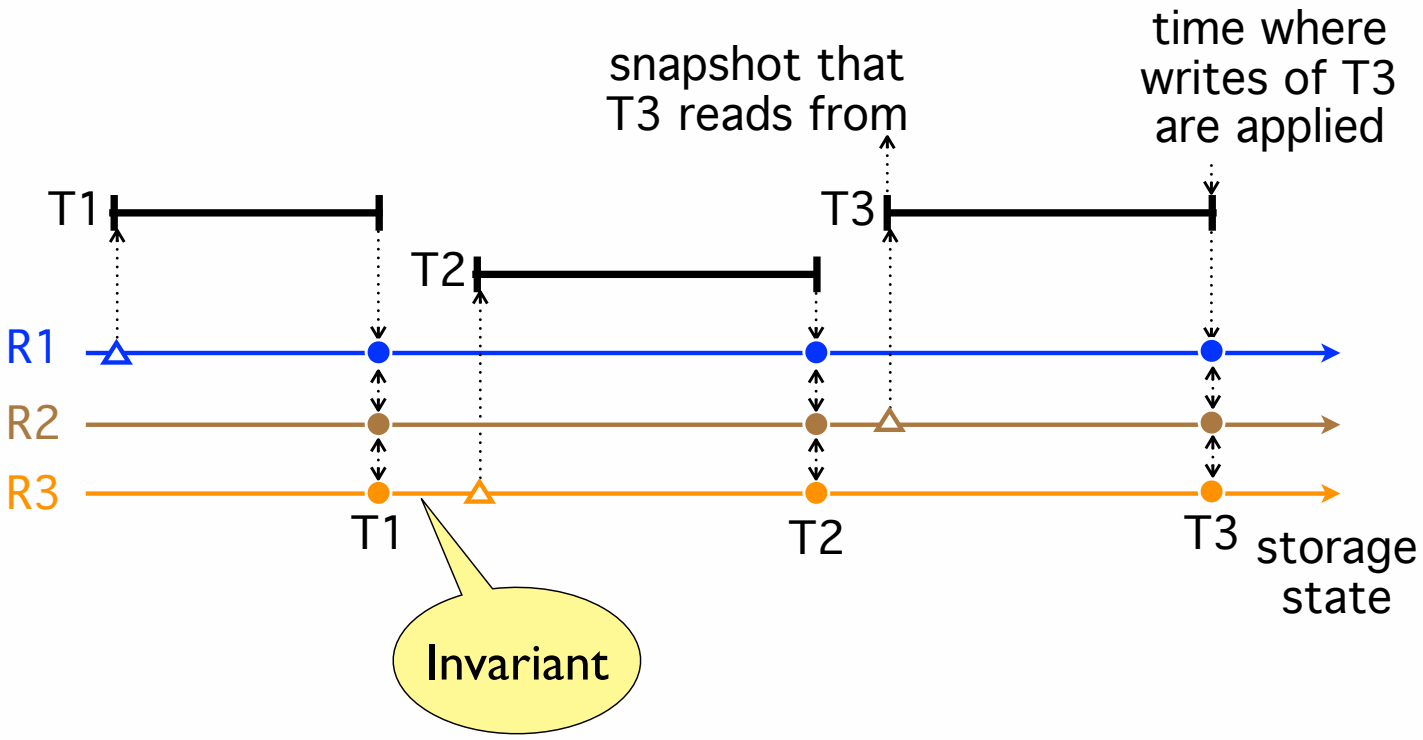


Distributed company:
• slow network ⇒ replicate data

• 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

Serialisable model



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

- $\Longrightarrow$ Propagate asynchronously

Pros:

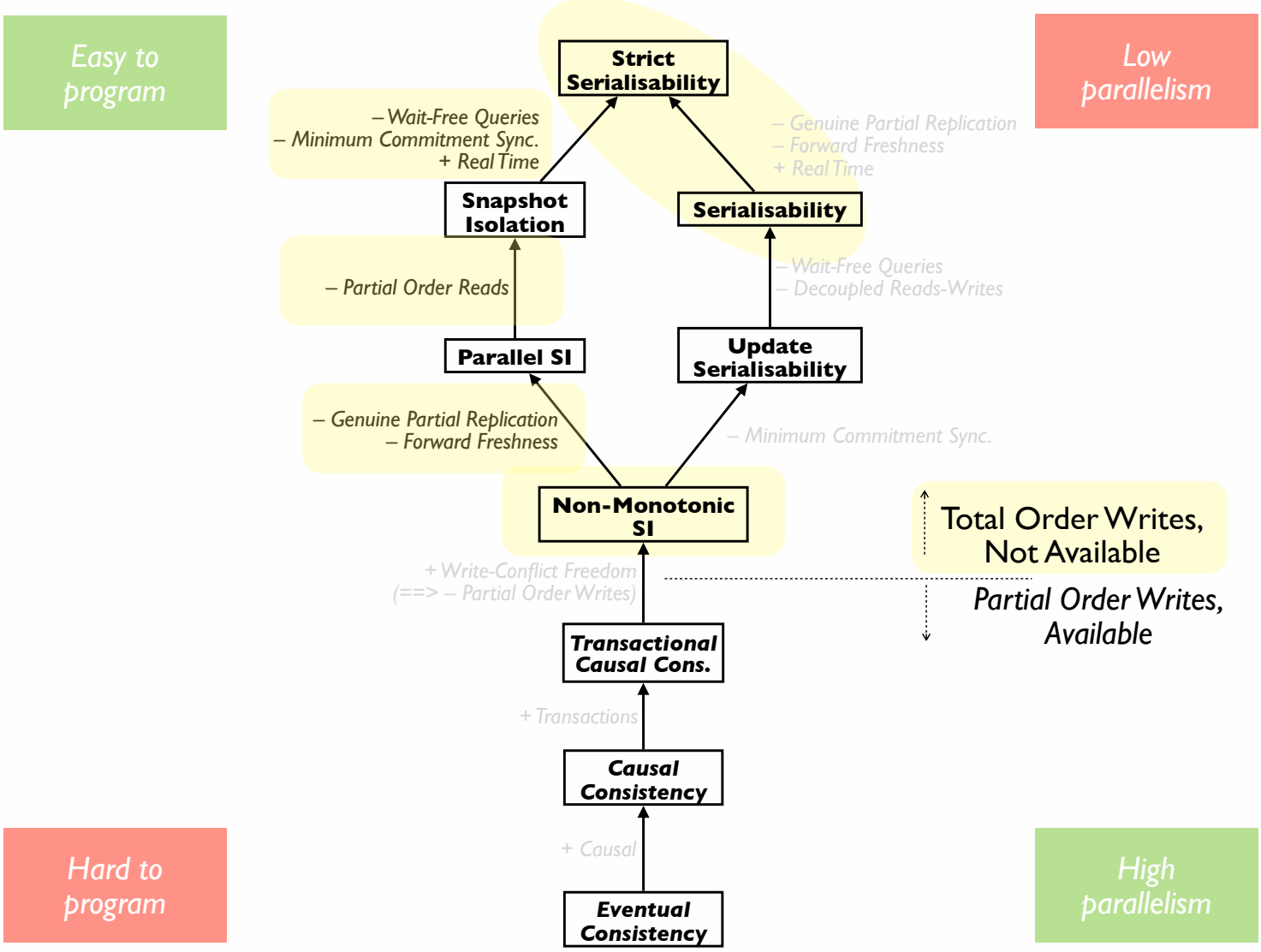
- More implementation, execution choices
- Performance
- Availability

Cons

- Invariants $\Longrightarrow$ Programmability

Acceptable parallelism $\Longrightarrow$ what invariants are important

- depends on application
- e.g. "*stock* $\geq$ *0*"

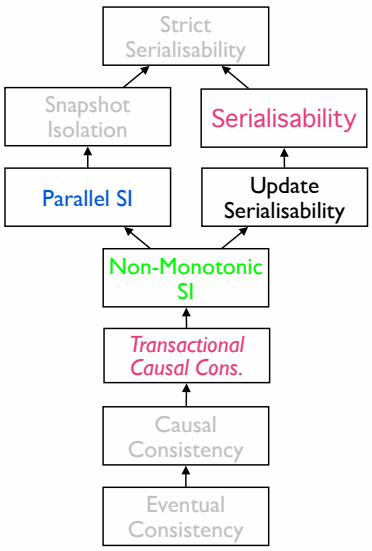


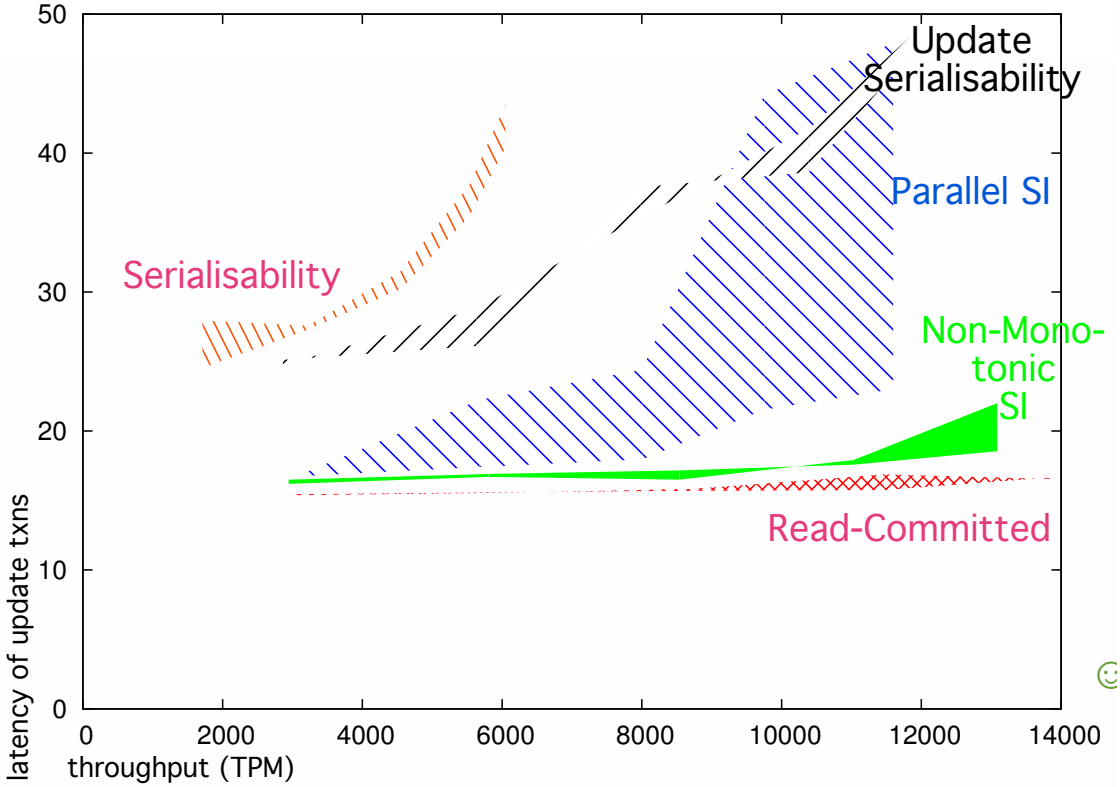
Survey of our work (and others)
Simplified view of consistency spectrum. +/– = properties gained/lost going up
Explore consistency spectrum; our work:

- Weakest strong consistency
- Strongest weak consistency

"Strong" = concurrent writes are not permitted = total ordered writes

Performance domains





2×–40× latency/throughput variation

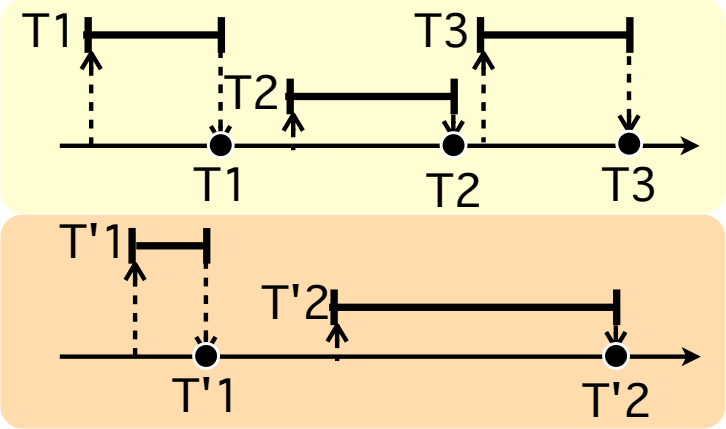
Parallelise: Partitioned DB

Early versions of Facebook: per university

Disjoint parts:

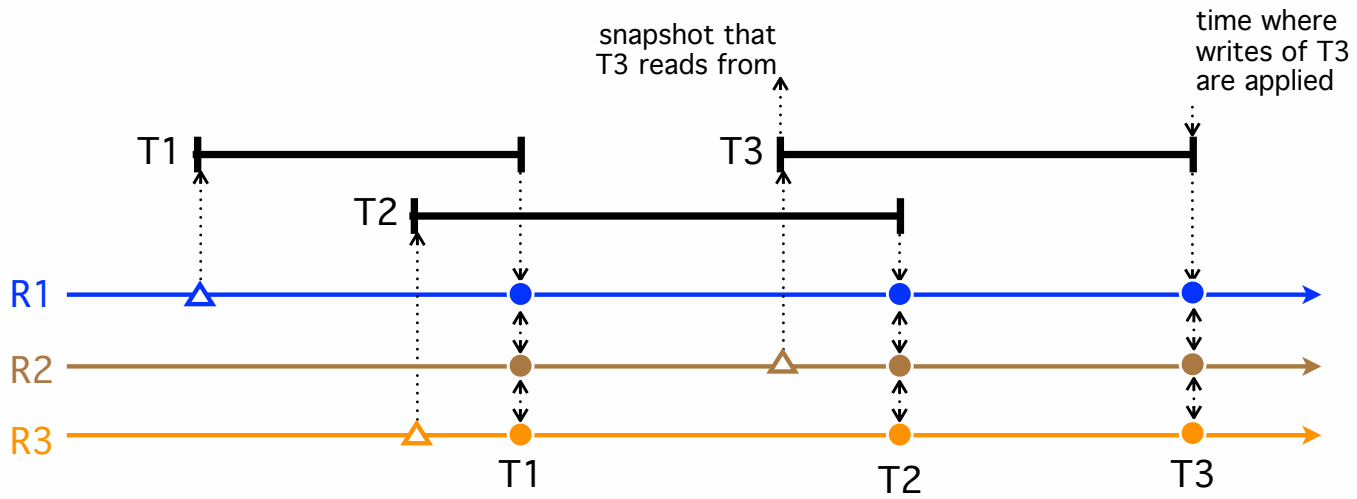
- full parallelism
- no cross-part guarantees

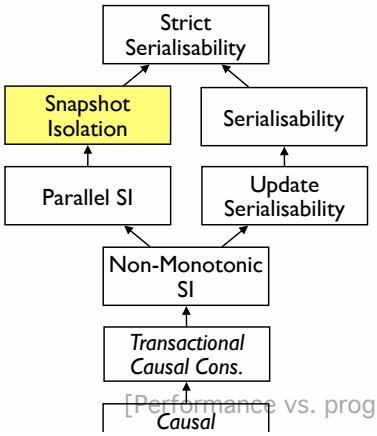
OK for some apps but defeats the purpose



Now Facebook allows anyone to be friends

Snapshot Isolation (SI)





Parallelise: overlap reads and writes

Minimal sync: T2 commits iff
$ws(T1) \cap ws(T2) = \varnothing$

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: $\nearrow$ replicas $\Longrightarrow$ $\nearrow$ 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 $\Longrightarrow$ Weaker model

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

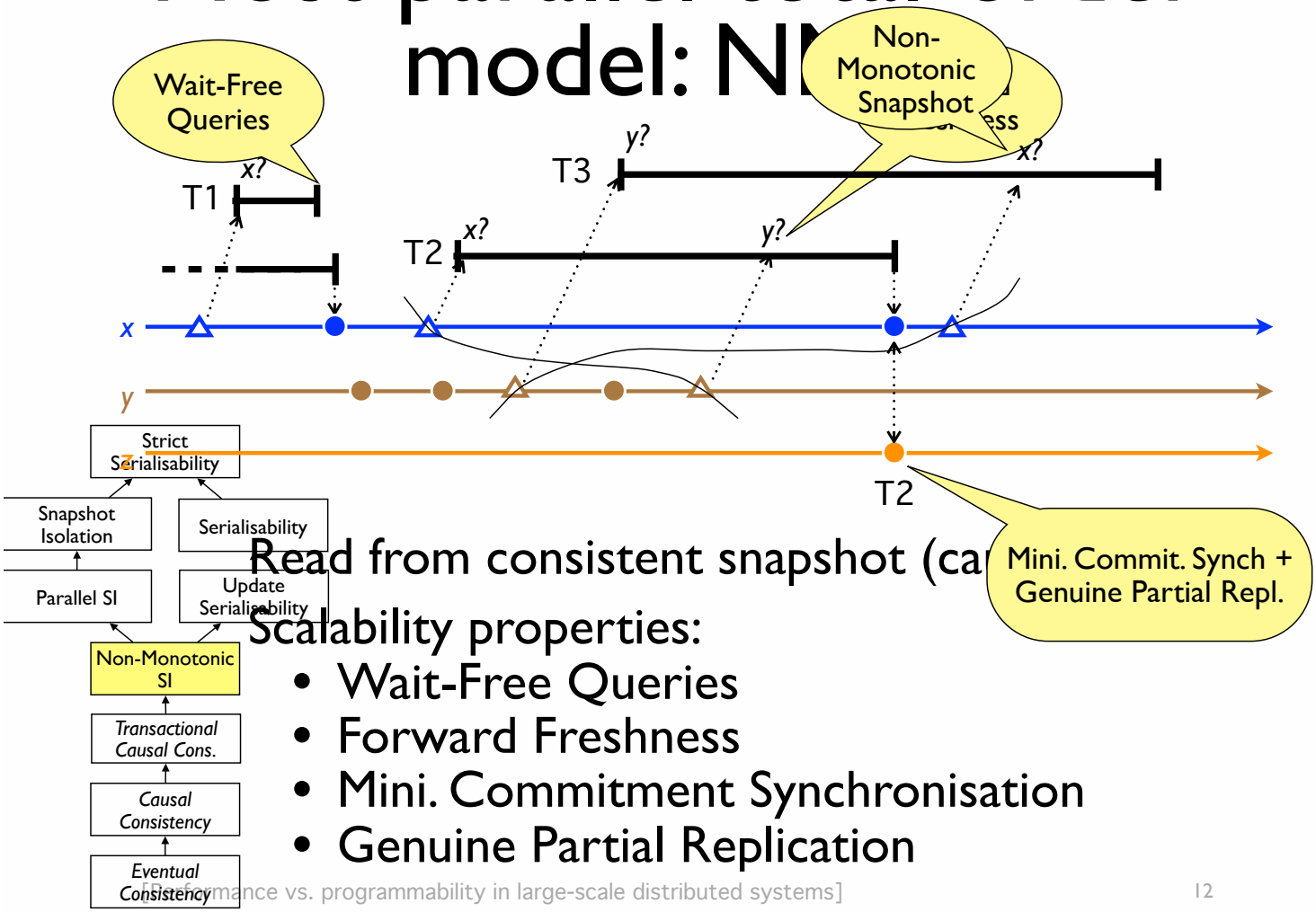
Full replication:

- every replica receives every update
- more replicas $\Longrightarrow$ 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 $\Longrightarrow$ apply to weaker model

Most parallel total-order model: NM



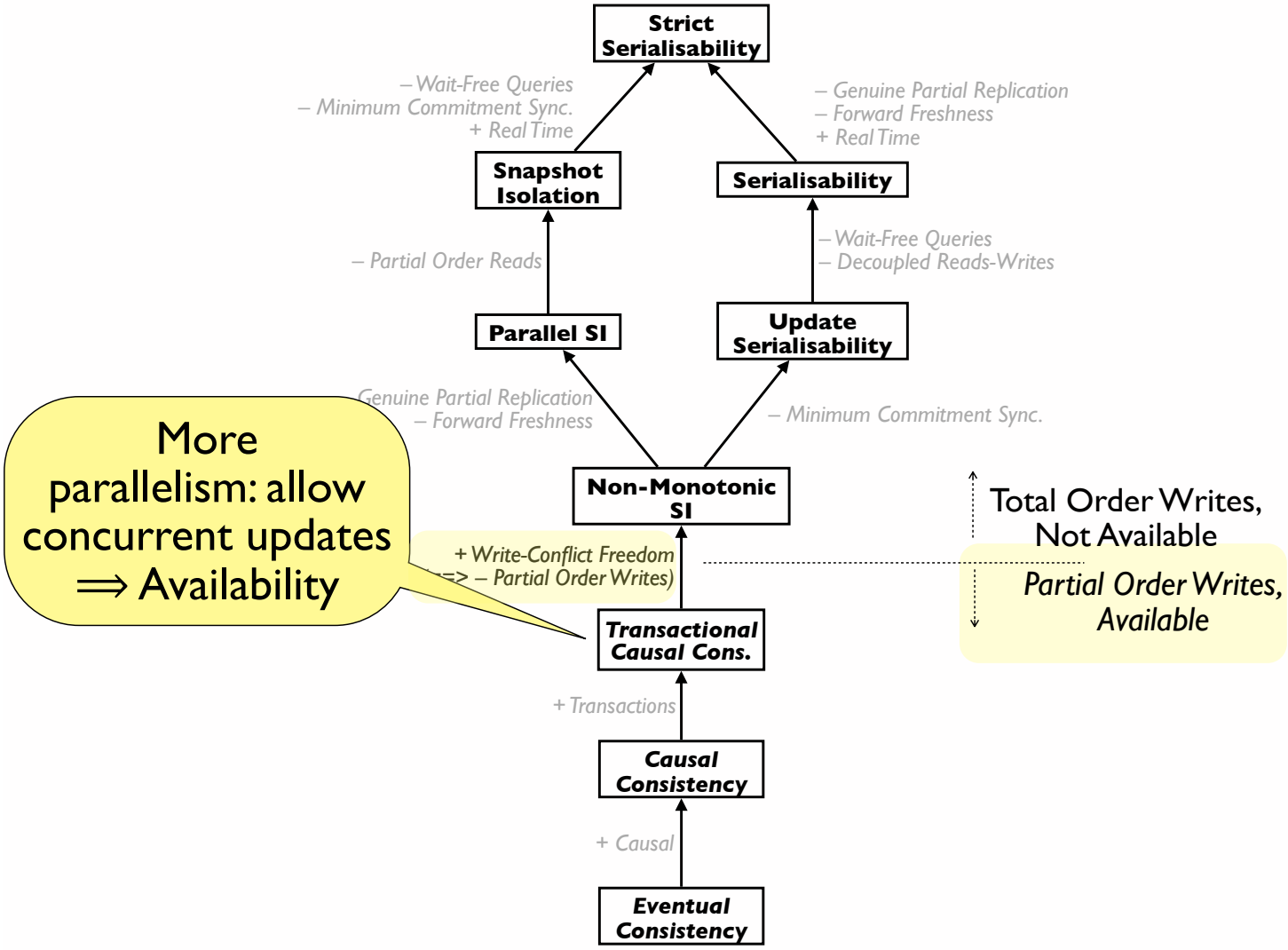
Read from consistent snapshot (ca

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

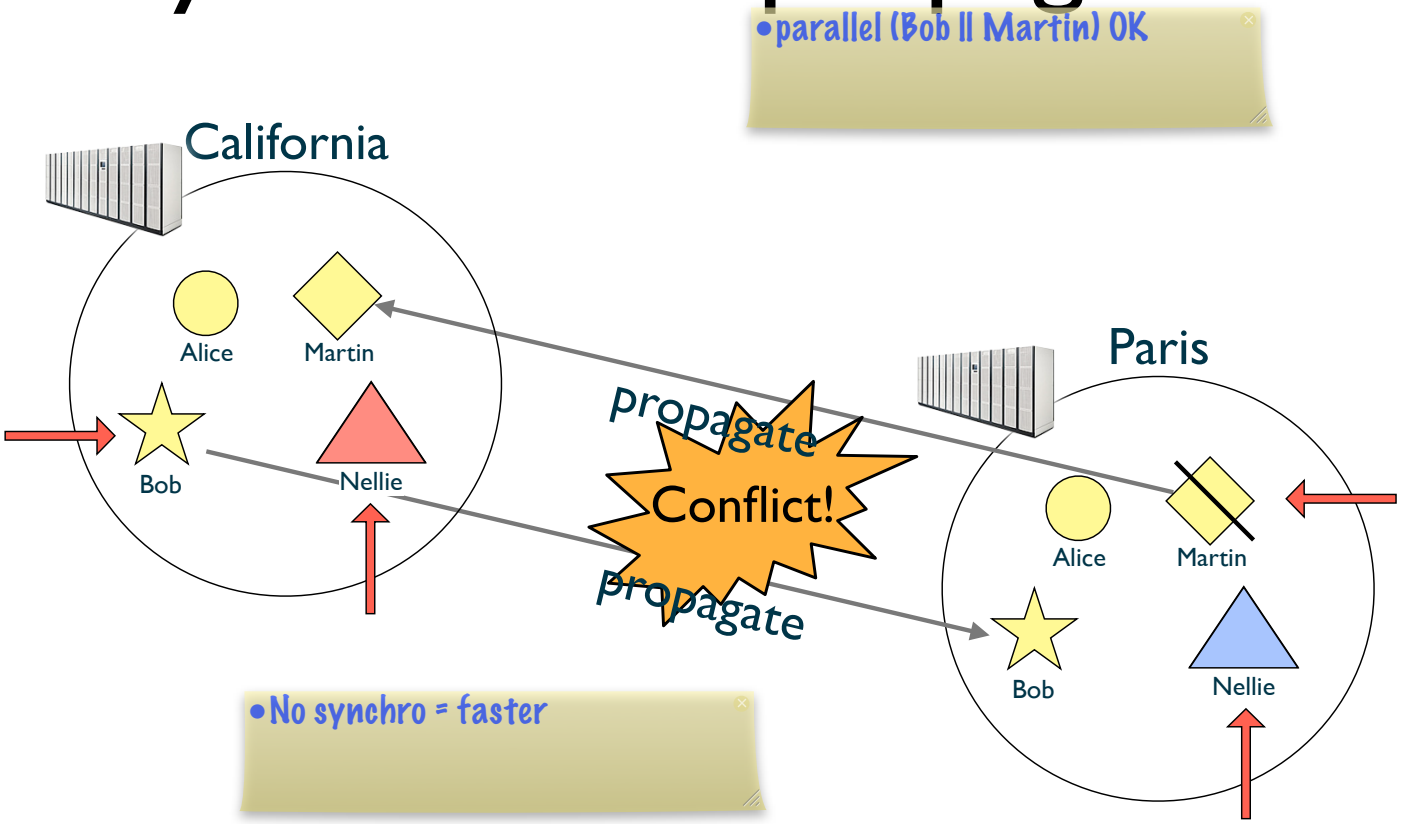


Lots of replicas ⟹ total order doesn't scale, not available
More availability:

- Tolerate network failures
- Mobile computing

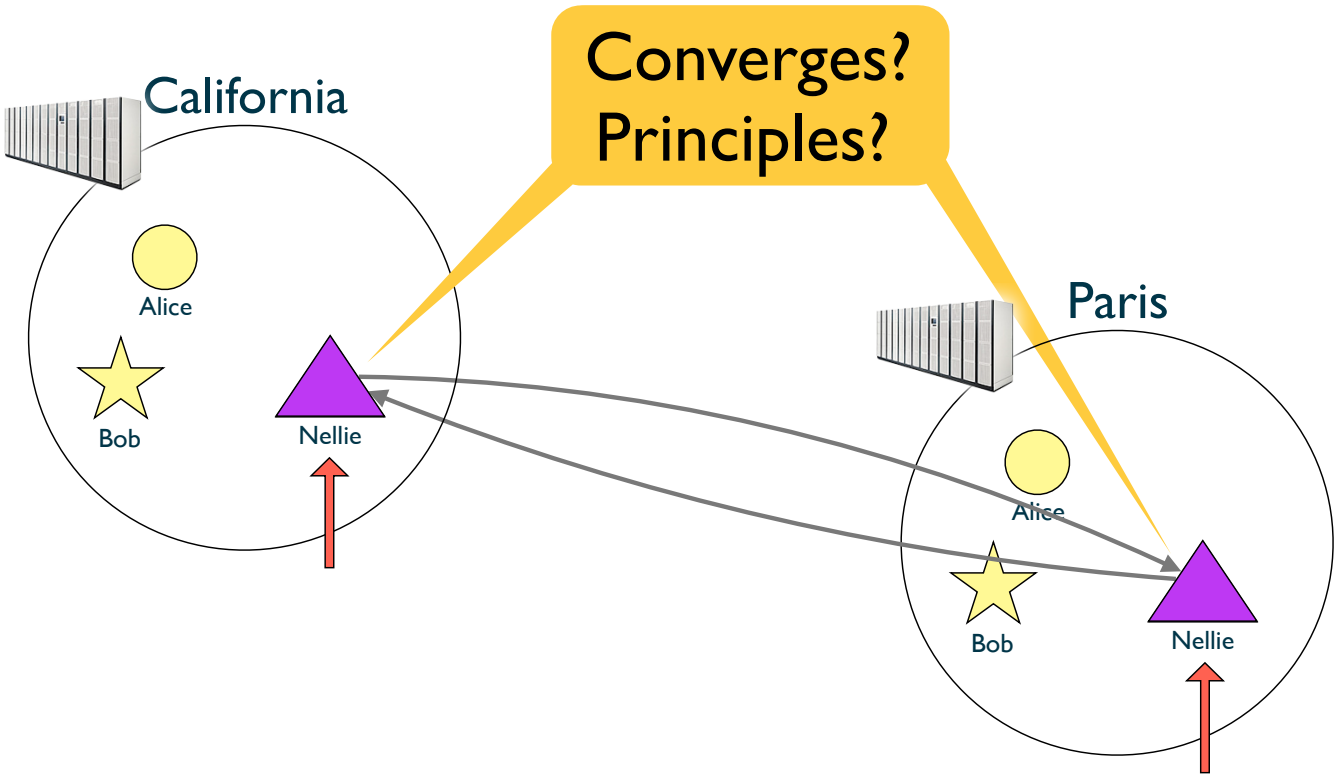
Strongest weak consistency
“Weak” = concurrent writes are permitted = replicas may diverge (but re-converge)

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



EC:

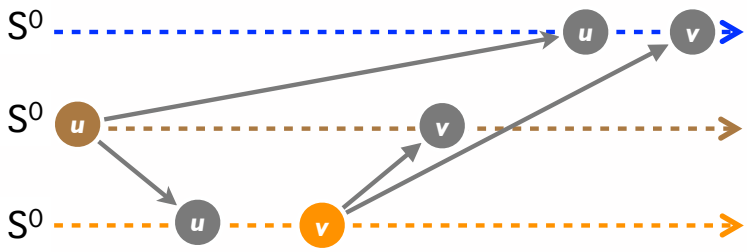
asynch propagation

merge concurrent updates

Very tricky, ad-hoc so far.

But how? Up to the user? Synchronise to ensure convergence? Depends on semantics?

Causal consistency



v observed effects of *u* at source
$\Longrightarrow$ *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
- $u \longrightarrow v \Longrightarrow$ (u observed before v at every replica)

Example: email, photo

Merging & convergence

Convergence conditions:

- Deterministic
- Dependent only on delivery set
- Not on delivery order, local info

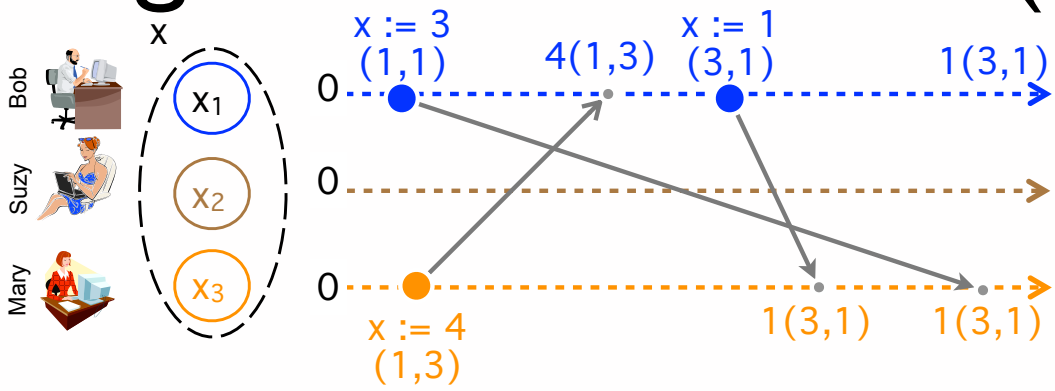
Formally:

- 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

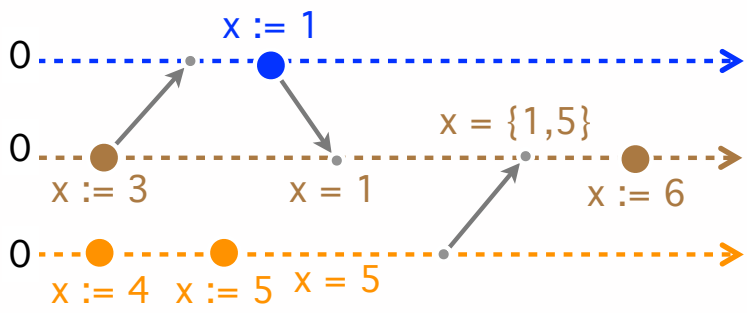
a.k.a. Thomas Write Rule
[Paul R. Johnson, Robert H. Thomas, The maintenance of duplicate databases, 1976, RFC 677]
1st, a protocol widely used, e.g. in replicated file systems (Microsoft Client-Side Caching) or in Active Directory, called Timestamped Writes or Last Writer Wins. It is very simple and 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)

Decentralised resolution:

Multi-Value Register



Not concurrent: usual register semantics

Concurrent assignments: return *set* of values

Concurrent semantics ≠ seq. interleaving!

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:

- *add* (element e)
- *rmv* (element e)
- *lookup* (element e) : boolean

No duplicates

- union and intersection commute
- not set difference

Extending the Set seq. spec.

Sequential specification of Set:

- $\{true\}$ add(e) $\{e \in S\}$
- $\{true\}$ rmv(e) $\{e \notin S\}$

Commutative ($e \neq f$):

- $\{true\}$ add(e) || add(e) $\{e \in S\}$
- $\{true\}$ rmv(e) || rmv(e) $\{e \notin S\}$
- $\{true\}$ add(e) || add(f) $\{e, f \in S\}$
- $\{true\}$ rmv(e) || rmv(f) $\{e, f \notin S\}$
- $\{true\}$ add(e) || rmv(f) $\{e \in S, f \notin S\}$

Ambiguous:

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

Extending the Set seq. spec.

Sequential specification of Set:

- $\{true\}$ add(e) $\{e \in S\}$
- $\{true\}$ rmv(e) $\{e \notin S\}$

Commutative ($e \neq f$):

- $\{true\}$ add(e) ; add(e) $\{e \in S\}$
- $\{true\}$ rmv(e) ; rmv(e) $\{e \notin S\}$
- $\{true\}$ add(e) ; add(f) $\{e, f \in S\}$
- $\{true\}$ rmv(e) ; rmv(f) $\{e, f \notin S\}$
- $\{true\}$ add(e) ; rmv(f) $\{e \in S, f \notin S\}$

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:

- $\{true\}$ add(e) $\{e \in S\}$
- $\{true\}$ rmv(e) $\{e \notin S\}$

Commutative ($e \neq f$):

- $\{true\}$ add(e) || add(e) $\{e \in S\}$
- $\{true\}$ rmv(e) || rmv(e) $\{e \notin S\}$
- $\{true\}$ add(e) || add(f) $\{e, f \in S\}$
- $\{true\}$ rmv(e) || rmv(f) $\{e, f \notin S\}$
- $\{true\}$ add(e) || rmv(f) $\{e \in S, f \notin S\}$

What about:

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

add(e) || rmv(e)

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

- ~~sequential consistency, linearisable?~~
- last writer wins? $\{\ add(e) < rmv(e) \Rightarrow e \notin S \ \wedge\ rmv(e) < add(e) \Rightarrow e \in S\ \}$
- error state? $\{\bot_e \in S\}$
- add wins? $\{e \in S\}$
- remove wins? $\{e \notin S\}$

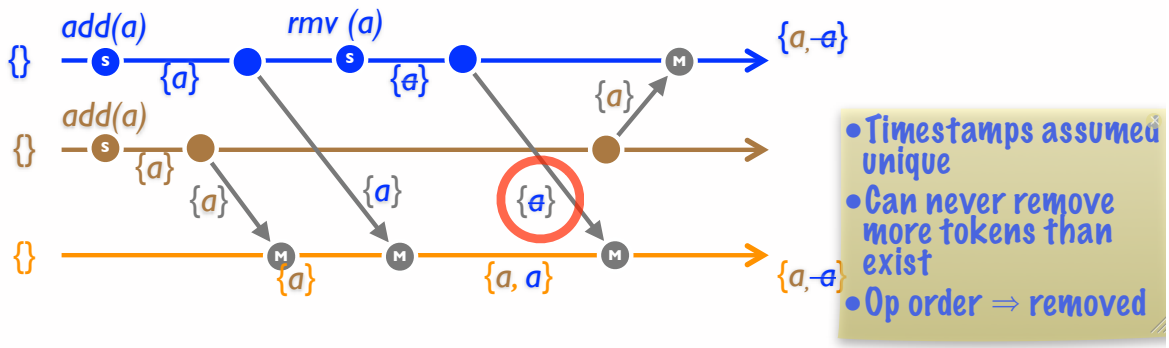
Deterministic

- 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)



$\{a\}.add(a) = \{a, a\}$	*unique*
$\{a, b\}.rmv(a) = \{\cancel{a}, b\}$	*mark visible instances of a*
$\{\cancel{a}, a\}.contains(a) \mid true$	*∃ non-marked instance of a*
$\{a, b\}.merge(\{\cancel{a}, c\}) = \{\cancel{a}, b, c\}$	*union + marker*

$\{true\}$ add(e) || 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

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- Observed-Remove

Map

Counter
- Unlimited
- Restricted non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence

[Performance vs. programmability in large-scale distributed systems]

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

Systems day talk

Current work: Invariant-based protocol design

Hybrid system

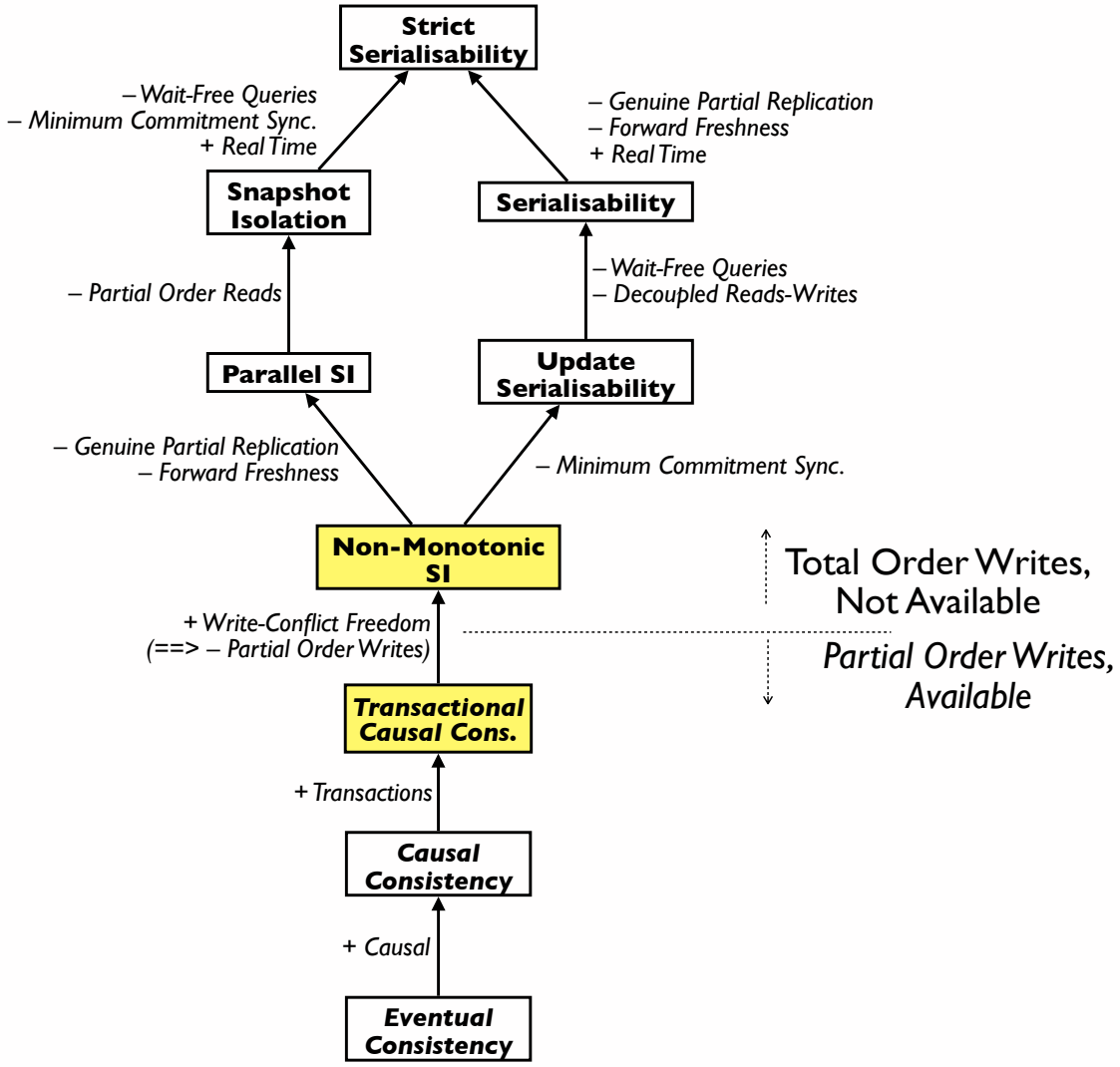
- Asynch: As fast and available as possible
- Synch: Strongly consistent when necessary

Delivering an asynchronous update must never violate the invariant

Examples:

- $x \leq y$
- $A \subseteq B$
- $P \Longrightarrow Q$
- 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