From Linearizability to Eventual Consistency

Radha Jagadeesan James Riely,
College of CDM, DePaul University, Chicago

Dec 5, 2016. Compositionality workshop

Organization of talk

Context of problem: Distributed data structures.

Problem: Correctness.

Compositionality and abstraction.

DePaul CDM Tech Report, 2016. "From Linearizability to Eventual Consistency".

Sequential interfaces.

eg. Integer Set.

Mutators: +0 [Add] and -0 [remove]. Return type VOID.

Accessor: ✓1,✗1. Returns a boolean. Do not alter the state of the object

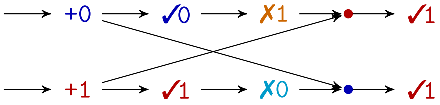
Example traces.

✗0 +0 ✓0 ✗1

+0 +1 ✓0 ✓1 -1 ✓0 ✗1

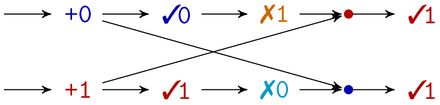
Distributed (implementation of) Set.

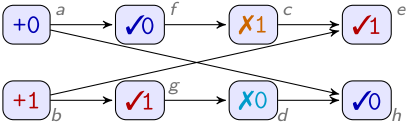
$add(0);\ ?0;\ ?1;\ ?1 \parallel add(1);\ ?1;\ ?0;\ ?0$



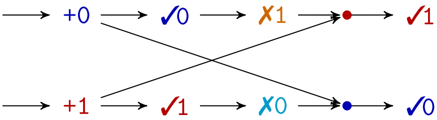
Distributed (implementation of) Set.

$add\,(0);\ ?0;\ ?1;\ ?1 \parallel add\,(1);\ ?1;\ ?0;\ ?0$





No global ordering



Serialization affects performance and scalability

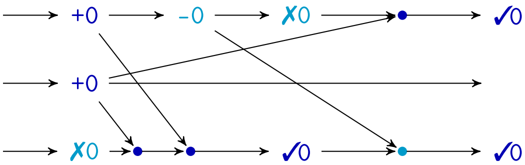
cap *theorem* : can't have all three [Gilbert and Lynch 2002]

Consistency	Every read receives the most recent write or an error
Availability	Every request receives a response
Partition tolerance	The system operates despite arbitrary messages loss

Convergent and Commutative Replicated Data Types

[Shapiro, Pregui, Baquero, Zawirski 2011]

Resolving conflicts among mutators.
Observed Remove Set. or-set: "Add wins"



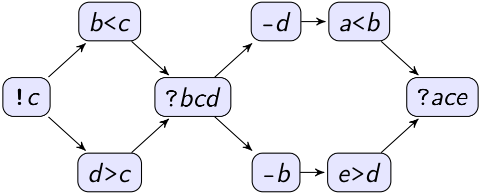
$Specification : +0 - 0 + 0$

Short digresssion. Distributed text editors

[Attiya, Burckhardt, Gotsman, Morrison, Yang, and Zawirski, 2016]

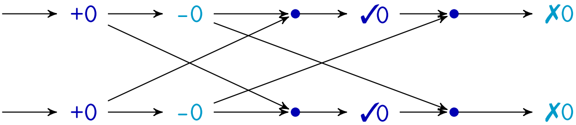
Mutators: $!a$, $a{<}b$, $a{>}b$, $-a$ Accessors: $?a_1 \cdots a_n$

'Deletion wins" (compare to ORSET)

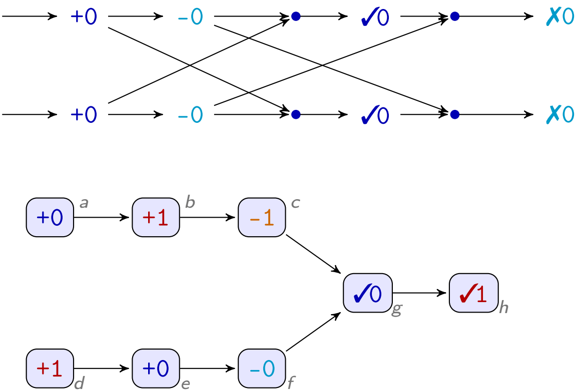


$!c;\ b{<}c;\ d{>}c;\ ?bcd;\ a{<}b;\ e{>}d;\ -b;\ -d;\ ?ace$

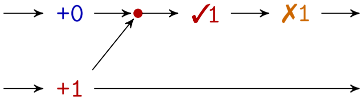
Resume: or-set examples



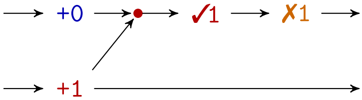
Resume: or-set examples

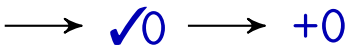


or-set: non-behaviors

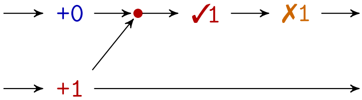


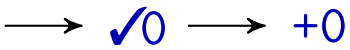
or-set: non-behaviors

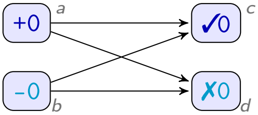




or-set: non-behaviors







In what sense does the or-set implement a Set?

When is implementation(U) valid for a specification (Σ) :

$$U \precsim \Sigma$$

What are the constraints?

Constraint 1: Compositionality (a)

[Herlihy, Wing 1990] Given two separate and independent sets:

$$L_{\Sigma_1} \cap L_{\Sigma_2} = \emptyset.$$

and two implementations, each of which is correct individually:

$$U_1 \sqsubseteq_{\sim} \Sigma_1, U_2 \sqsubseteq_{\sim} \Sigma_2$$

we want:

$$U_1 \parallel\mid U_2 \sqsubseteq_{\sim} \Sigma_1 \parallel\mid \Sigma_2$$

Constraint 2: Compositionality (b)

[Filipovic, O Hearn, Rinetzky, Yang 2009]
Let $\mathscr{P}$ be the graph implementation, which is a client of the two sets (for vertices,edges).
We want:

$$(\mathscr{P} \parallel\!\lceil (\Sigma_1 \parallel\!\mid \Sigma_2)) \backslash (L_{\Sigma_1} \cup L_{\Sigma_2}) \precsim T$$

implies

$$(\mathscr{P} \parallel\!\lceil (U_1 \parallel\!\mid U_2)) \backslash (L_{\Sigma_1} \cup L_{\Sigma_2}) \precsim T.$$

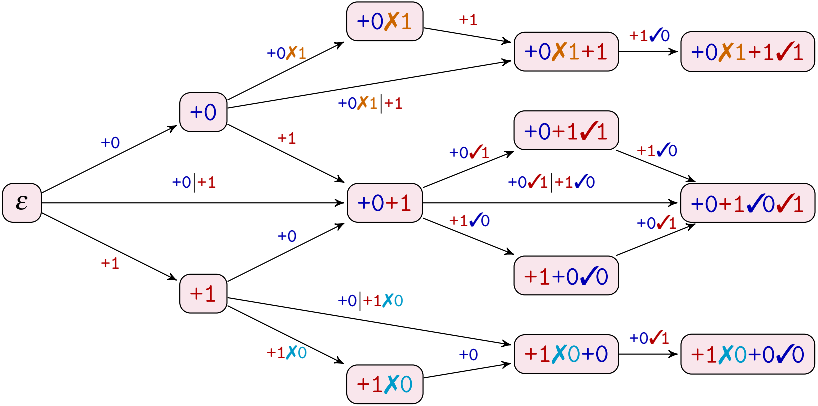
Constraint 3: Coherence with the sequential specification

Single threaded semantics: A correct implementation should behave according to the sequential semantics if accessed at a single replica.

Permutation equivalence: "If all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states.

Client-server linearizability: Any execution of a correct implementation on a client-server system should be linearizable.

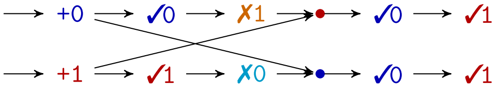
Linearizability: Linear time, Atomic/Instantaneous methods



An implementation U **is valid if it is simulated by the above automaton.**

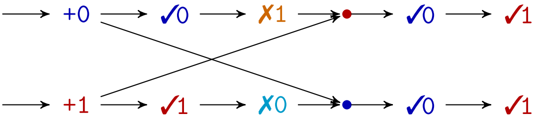
This talk

The linearizability automaton is too restrictive: does not simulate many desired behaviors.



Not *linearizable*: no way to place *both* ✗0, ✗1 in +0 +1 while preserving order.
What is the correct formalization?

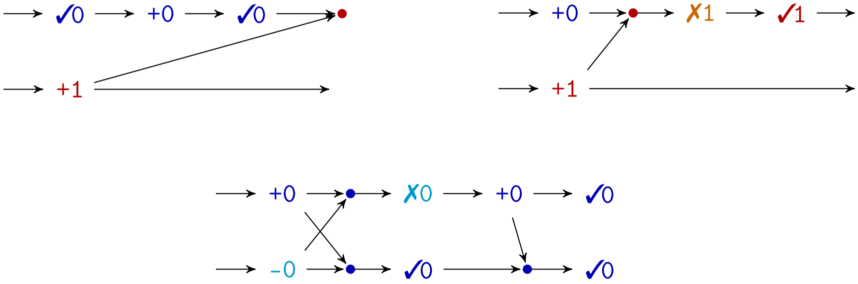
Relaxing linearizability: Eventual consistency



But, states of all the replicas eventually converge when all the messages have been delivered. cf. *quiescent consistency*

Suffices for “shopping cart”.

Consistency at non-quiescent states??



Not enough constraints: eg. "permutation equivalence" not enforced.

Prior Work

Abandon sequential specifications

[Bouajjani, Enea, Hamza 2014]

[Burckhardt, Gotsman, Yang, Zawirski 2014]

☹ Only sequential specifications are canonical

Permutation based

[Burckhardt, Leijen, Fähndrich, Sagiv. 2012]

[Jagadeesan, Riely 2015]

☹ Too restrictive

Our approach: liberalize the linearizability automaton

Two ingredients.

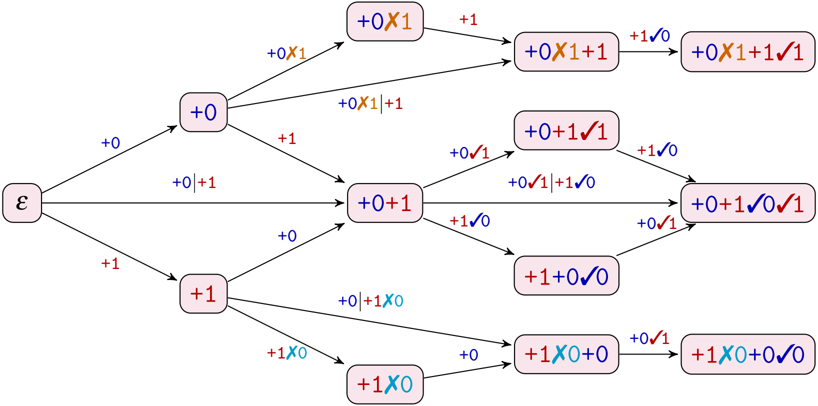
(a) Quotient states under observational equivalence

(b) Time as a partial order

- Prefixes to subsequences
- Explicate and disentangle dependencies

Quotient states under observational equivalence

In linearizability state machine, states are sequences of methods.


ε
+0
+1
+0|+1
+0✗1
+0+1
+1✗0
+0✗1|+1
+0|+1✗0
+0✗1+1
+0+1✓1
+1+0✓0
+1✗0+0
+1✓0
+0✓1
+0✓1|+1✓0
+0✗1+1✓1
+0+1✓0✓1
+1✗0+0✓0

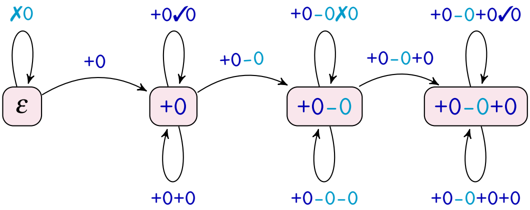
Quotient automaton states under observational equivalence

[Brookes 96]: Two sequences are equivalent if they yield the same sequence of states of the data structure, upto stuttering. In set :

$$+0+0 \sim +0 \qquad\qquad +0✓0 \sim +0$$

and the equivalence classes for a set over one element 0 are:

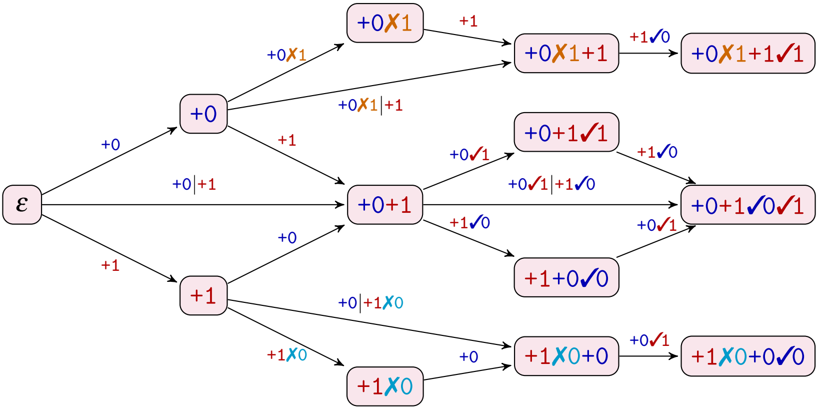
$$+0, +0-0, +0-0+0, +0-0+0-0, \ldots$$



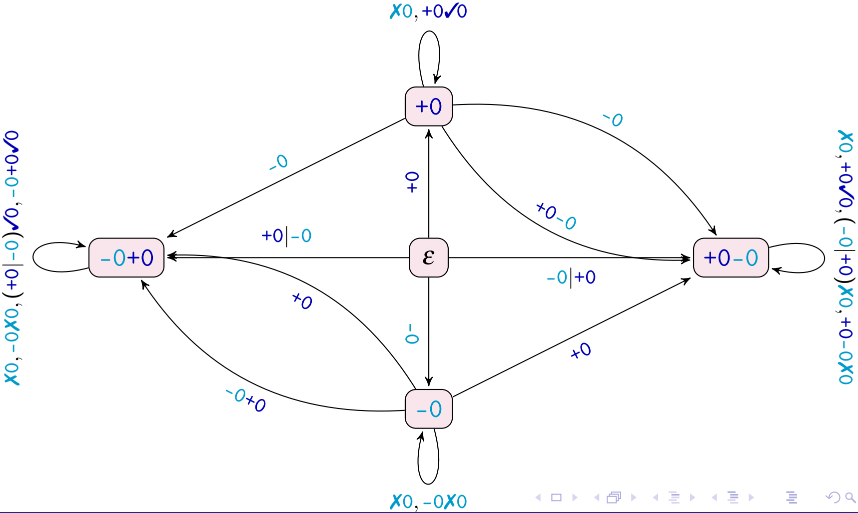
Time as a partial order.

In the linearizability automaton, time is linear.

Strict prefix ordering



Time as a partial order: prefixes to subsequence



Time as a partial order. Disentangling dependencies

The linearizability automaton is insensitive to independence.

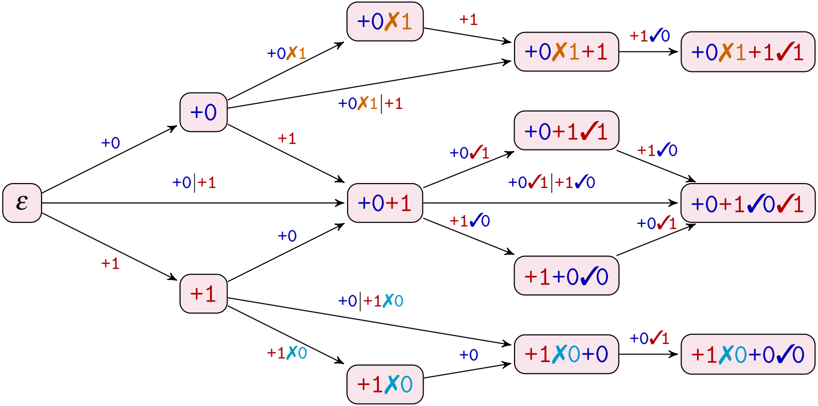
In binary set [+0, -0, ✓0, ✗0, ✗1, ✓1], the two values are independent, i.e a trace for a binary set is valid iff its projection to 0 (resp. 1) is valid.

More generally, enrich specification with notion of conflict: #

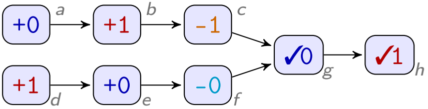
set : +0 # ✓0, +0 # ✗0, +1 # ✓1, +1 # ✗1, ✓0 # ✗0, ✓1 # ✗1 . . . . . .

Distributed text editors: Two labels from this alphabet are in conflict iff they mention overlapping sets of text identifiers, or if one is a query and the other is a remove.

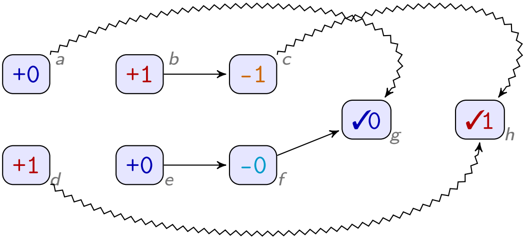
b<*c* # *a*<*b*, ?*ce* # -*a* . . . . . .


ε
+0
+1
+0|+1
+0✗1
+0+1
+1✗0
+1
+0✗1|+1
+0|+1✗0
+0✗1+1
+0+1✓1
+1+0✓0
+1✗0+0
+1✓0
+0✓1
+0✓1|+1✓0
+0✗1+1✓1
+0+1✓0✓1
+1✗0+0✓0

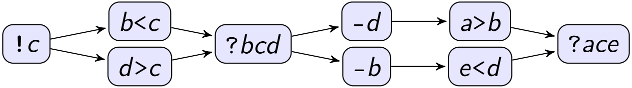
Disentangling dependencies: set



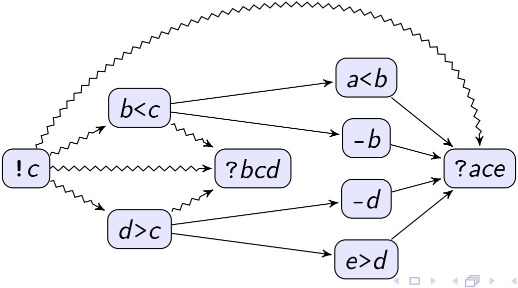
Specification: (+0-0+0) ||| (+1-1+1)



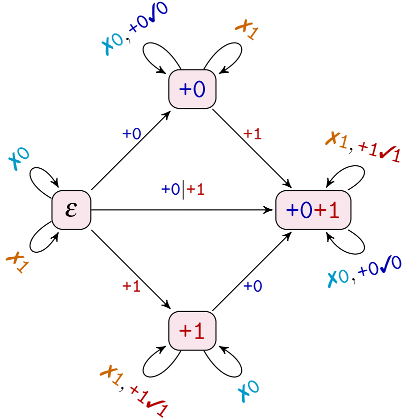
Disentangling dependencies: Distributed text editor



Specification : !*c*; *b*<*c*; *d*>*c*; ?*bcd*; *a*<*b*; *e*>*d*; -*b*; -*d*; ?*ace*



Disentangling dependencies: the set automaton.



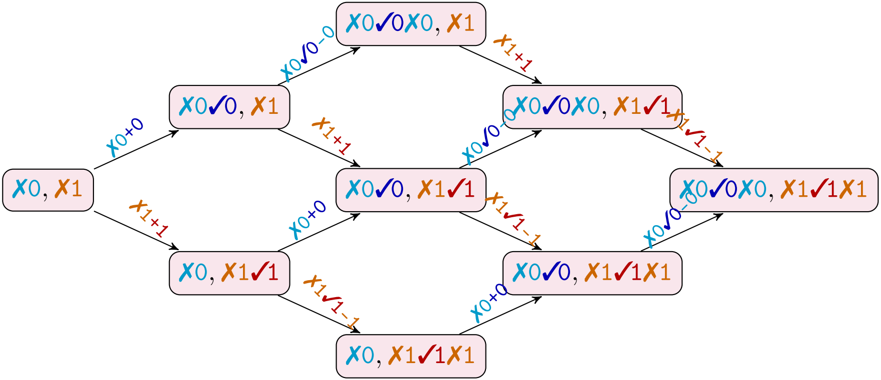
Our correctness criterion: $U \precsim \Sigma$

Σ: Incorporate "quotienting of the label sequences under observational equivalence", and "time as a partial order".

is generated *purely* from the standard sequential specification.

An implementation U is valid if it is simulated by Σ.

The set automaton



Results (1)

Alternative characterization. Via a direct definition.

Coherence with the sequential specification. Single threaded semantics, Permutation equivalence and Client-server linearizability.

Expressiveness. Addresses the CRDT examples.

Results (2)

Composition. Given two separate and independent sets, $L_{\Sigma_1} \cap L_{\Sigma_2} = \emptyset$. and $U_1 \precsim \Sigma_1, U_2 \precsim \Sigma_2$, we have :

$$U_1 \mathbin{|||} U_2 \precsim \Sigma_1 \mathbin{|||} \Sigma_2$$

Abstraction. Let $\mathscr{P}$ be the graph implementation, which is a client of the two sets (for vertices,edges). Then:

$$(\mathscr{P} \parallel\!\lceil\, (\Sigma_1 \mathbin{|||} \Sigma_2)) \backslash (L_{\Sigma_1} \cup L_{\Sigma_2}) \precsim T$$

implies

$$(\mathscr{P} \parallel\!\lceil\, (U_1 \mathbin{|||} U_2)) \backslash (L_{\Sigma_1} \cup L_{\Sigma_2}) \precsim T.$$

Results (3). calm clients can program sequentially

calm: "Consistency as logical monotonicity" . [Hellerstein 2010] The Bloom language. [Conway, Marczak, Alvaro, Hellerstein, Maier] "Monotonic reasoning requires no coordination"

$$path(@Src, Dest) :- path(@Src, X), link(@X, Dest)$$

BUT: "non-monotonic reasoning in general requires global barriers". eg. state change, counting aggregates..

$$\begin{array}{ll} toggle(1) & :- \; state(0) \\ toggle(0) & :- \; state(1) \\ state(X)@next & :- \; toggle(X) \end{array}$$

No "races" between concurrent mutators and mutators/accessors.

QUESTIONS??

For full details refer to:
DePaul CDM Tech Report, 2016. “From Linearizability to Eventual Consistency”.