A principled approach to programming distributed systems

Marc Shapiro

Summer School on Distributed and Replicated Environments
DARE 2023

CRDTs: not a silver bullet

CRDT:

- Extend sequential data type (commutative or not)
- Merge concurrent updates: commutative, associative, idempotent
- Guarantee availability + convergence

Designing distributed applications remains challenging

- Multiple CRDTs
- Correct interactions?

Experiments: calendar, student registration, game tournaments, auction, ticket reservations, file system, etc.

Real applications: LWW, Bet365, text editor, ???

CAP trade-offs

Strong consistency

- High system cost
  - One-off
- Low programmer cost
- Sequential bottleneck
- partition ⟹ ¬available

Too strong: slow

“Weak” consistency

- Responsive
- High programmer cost
  - Recurring
- Asynchronous
- Available

Too weak: buggy

“As fast as possible, as strong as necessary”

Strong enough; no stronger

Consistent ≜ maintains invariant

Invariant ≜ predicate over the state of the system

- Credit cards: $hash(n) = 0$
- Seat reservations: $\mathit{remaining\_seats} \geq 0$
- Social network: $\mathit{friend}\ (A,B) \Longleftrightarrow \mathit{friend}\ (B,A)$
- Bank: $\Sigma\ \mathit{accounts} = \mathit{constant}$
- File system: $\mathit{tree} \wedge \ldots$
- Student registration: $\mathit{attends}\ (\mathit{student, course}) \Longrightarrow \mathit{registered}\ (\mathit{student})$
- Storage backend: $\mathit{journal.low} \leq \mathit{checkpoint.high}$

Safe system ≜ invariant is true in every observable state

System guarantee vs. application-level effort

Sequential safety

Precondition, postcondition

$$\frac{Pre(\sigma) \qquad\qquad Post(\sigma')}{\sigma \xrightarrow{U} \sigma'}$$

Update *U*

Transitions the state from *σ to σ'*

Assuming *Pre(σ)=true*

then *U* ensures *Post(σ')*

$$\frac{x=0 \qquad\qquad x'=1}{x' = x+1}$$

weakest precondition

$$\frac{\qquad\qquad e \in S'}{S' = S \cup \{e\}}$$

$$\frac{t > 0 \qquad\qquad x' > x}{x' = x+t}$$

$$\frac{x \geq t \geq 0 \qquad\qquad x' \geq 0}{x' = x-t}$$

Precondition, postcondition

$$\frac{Pre(\sigma) \qquad\qquad Post(\sigma')}{\sigma \xrightarrow{U} \sigma'}$$

Update *U*

Transitions the state from *σ to σ'*

To ensure *Post(σ')*,

it must be that *Pre(σ)=true*

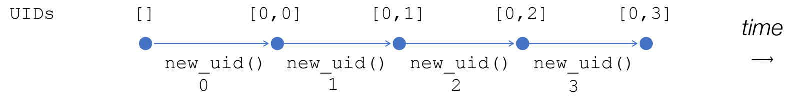
$$\frac{x=0 \qquad\qquad x'=1}{x' = x+1}$$

$$\frac{true \qquad\qquad e \in S'}{S' = S \cup \{e\}}$$

$$\frac{t > 0 \qquad\qquad x' > x}{x' = x+t}$$

$$\frac{x \geq t \geq 0 \qquad\qquad x' \geq 0}{x' = x-t}$$

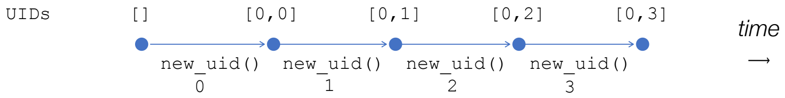
Ex.: Generate unique identifier



$$\frac{M \text{ is a set} \qquad n \notin M \qquad M' \text{ is a set}}{M' = M \cup \{n\}}$$

```
Inv() = { IDs = [0, last] }
static last = -1;
new_uid() = {
  precondition Inv();      // multiset is a set, max=last
  last++;                  // ∉ IDs
  return last;             // is unique
  postcondition Inv();
}
```

∀ update safe ⟹ seq. system safe



Individually Safe "C"

Safe state ≙ satisfies invariant

In a sequential system, if:

- The initial state satisfies *Inv*
- Every update *U* has precondition *wp(U, Inv)*

Then the system is safe

$$\forall U \qquad \frac{Inv(\sigma) \qquad wp(U,Inv) \qquad Inv(\sigma')}{\sigma \xrightarrow{U} \sigma'} \quad \Longrightarrow \text{safe}$$

Examples

Issue credit card

$$\frac{hash\ (n) = 0}{S' = S \cup \{n\}}$$

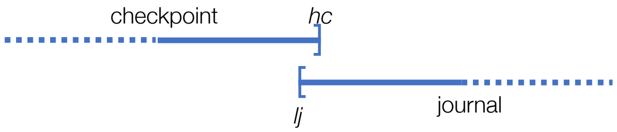
Deliver medication

- *Inv(r) { r ≥ 0}* *// r = remaining number of boxes*

Increase allocation: $\dfrac{Inv(r) \qquad\qquad Inv(r')}{r' = r+1}$

Deliver a box: $\dfrac{Inv(r) \qquad r \geq 1 \qquad Inv(r')}{r' = r-1}$

Ex.: maintain checkpoint



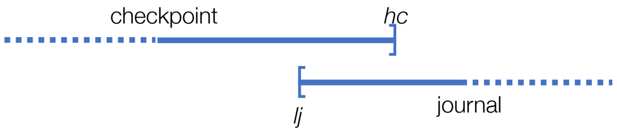
Inv(σ) = { σ.lj ≤ σ.hc} *// no gap*

Advance checkpoint: $\dfrac{Inv(\sigma) \wedge d \geq 0 \qquad\qquad Inv(\sigma')}{\sigma'.hc = \sigma.hc + d}$

Advance journal: $\dfrac{Inv(\sigma) \wedge d \geq 0 \qquad \sigma.lj + d \leq \sigma.hc \qquad Inv(\sigma')}{\sigma'.lj = \sigma.lj + d}$

Advance both: $\dfrac{Inv(\sigma) \wedge d \geq 0 \qquad\qquad Inv(\sigma'')}{\sigma'.hc = \sigma.hc + d;\ \sigma''.lj = \sigma'.lj + d}$

Ex.: maintain checkpoint



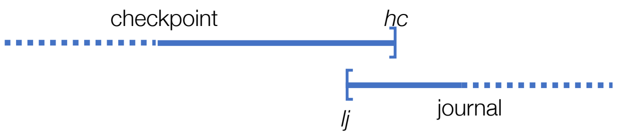
Inv(σ) = { σ.lj ≤ σ.hc} *// no gap*

Advance checkpoint: $\dfrac{Inv(\sigma) \wedge d \geq 0 \qquad\qquad Inv(\sigma')}{\sigma'.hc = \sigma.hc + d}$

Advance journal: $\dfrac{Inv(\sigma) \wedge d \geq 0 \qquad \sigma.lj + d \leq \sigma.hc \qquad Inv(\sigma')}{\sigma'.lj = \sigma.lj + d}$

Advance both: $\dfrac{Inv(\sigma) \wedge d \geq 0 \qquad\qquad Inv(\sigma'')}{\sigma'.hc = \sigma.hc + d;\ \sigma''.lj = \sigma'.lj + d}$

Ex.: maintain checkpoint



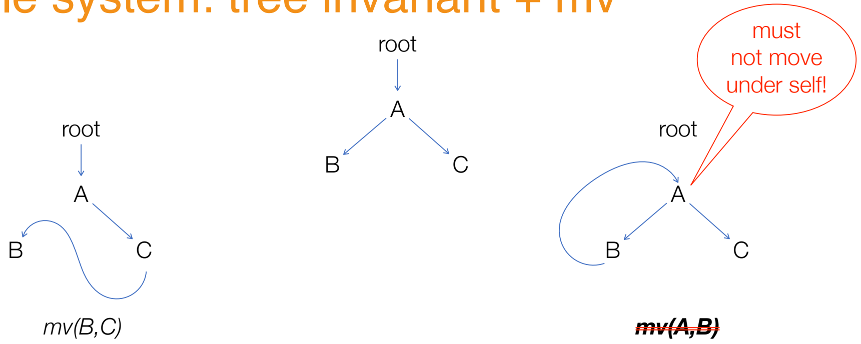
Inv(σ) = { σ.lj ≤ σ.hc} *// no gap*

Advance checkpoint:
$$\frac{Inv(\sigma)\wedge d \ge 0 \qquad\qquad Inv(\sigma')}{\sigma'.hc = \sigma.hc + d}$$

Advance journal:
$$\frac{Inv(\sigma)\wedge d \ge 0 \qquad \sigma.lj + d \le \sigma.hc \qquad Inv(\sigma')}{\sigma'.lj = \sigma.lj + d}$$

Advance both:
$$\frac{Inv(\sigma)\wedge d \ge 0 \qquad\qquad Inv(\sigma'')}{\sigma'.hc = \sigma.hc + d;\ \sigma''.lj = \sigma'.lj + d}$$

File system: tree invariant + mv



$$\frac{Inv(\sigma) \qquad \neg\ \sigma.path(c, np) \qquad Inv(\sigma')}{\sigma \xrightarrow{mv\ (c,\ np)} \sigma'}$$

Inv(σ) = *// Tree invariant*
- Every node is reachable from root
- A node has a single parent (but root has none)
- Names unique per directory
- Acyclic graph

Sequential safety: summary

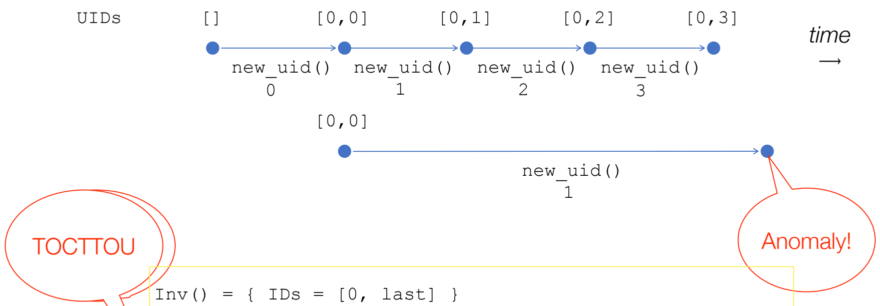
Safe state ≜ satisfies invariant *Inv*

In a sequential system, if:
- The initial state is safe
- Every update *U* is individually safe
  - i.e., has a precondition *Pre(U)* such that *Inv* remains true after *U*
  - i.e., *Pre(U)* ⟹ *wp(U, Inv)*

then every state of the system is safe

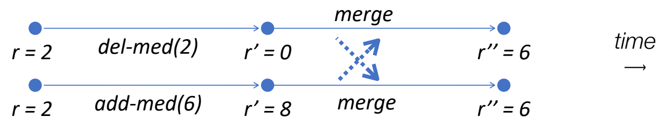
Concurrency anomalies

Concurrent generate UID



```
Inv() = { IDs = [0, last] }
static last = -1;
new_uid() = {
 precondition Inv();      // multiset is a set, max=last
 last++;                  // ∉ IDs
 return last;             // is unique
 postcondition Inv();
}
```

Concurrent delivery of medications (1)

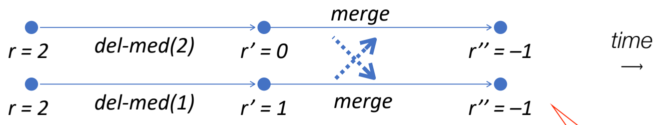


$$\textit{add-med}(n)\ \ \frac{r \geq 0 \qquad n \geq 0 \qquad r' \geq 0}{r' = r+n}$$

$$\textit{del-med}(n)\ \ \frac{r \geq 0 \qquad r \geq n \geq 0 \qquad r' \geq 0}{r' = r-n}$$

harmless TOCTTOU

Concurrent delivery of medications (2)



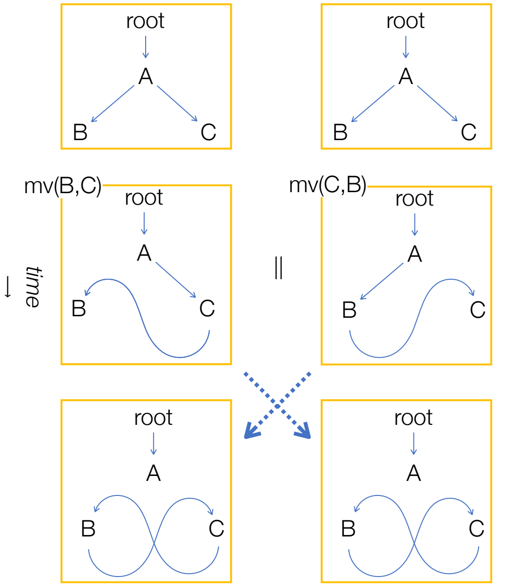
Anomaly!

harmful TOCTTOU

$$\textit{add-}\ \ \frac{\geq 0 \qquad n \geq 0 \qquad r' \geq 0}{r' = r+n}$$

$$\textit{del-med}(n)\ \ \frac{r \geq 0 \qquad r \geq n \geq 0 \qquad r' \geq 0}{r' = r-n}$$

FS: Concurrent move anomaly



TOCTTOU

$$\frac{Inv(\sigma) \qquad \neg\, \sigma.path(c, np) \qquad Inv(\sigma')}{\sigma \xrightarrow{mv\,(c,\,np)} \sigma'}$$

Concurrent moves

Converges

But violates invariant!

What went wrong?

File system: the DFS-R bug

2003: DFS-R: Windows NTFS replication layer

- large industrial customers
- unexplained data loss

2007: model checking exposes move anomaly: completely unexpected

2021: continues to bite developers

- Google Drive diverges
- Dropbox duplicates

Reasoning about concurrency is hard!

Concurrent anomalies: summary

Concurrency allows anomalies

- Incorrect behavior ≜ violates invariant
- Violation does not occur in a sequential execution

Anomalies seem to be caused by TOCTTOU race on precondition

Design objectives:

- Minimise remote coordination
- Allow benign TOCTTOUs
- Avoid harmful TOCTTOUs

But concurrency reasoning is too hard!

CISE: a sound approach to safe distributed systems

’Cause I’m Strong Enough (CISE)

Distributed system with CRDTs

Systematic method to prove whether two updates may execute concurrently without harm

- Concurrency not provably harmless is assumed harmful

Fully formalized, proven sound

- Generalises sequential correctness
- An application of the rely-guarantee logic

My presentation is informal

Gotsman et al., *‘Cause I'm strong enough: Reasoning about consistency choices in distributed systems*, POPL 2016, DOI 10.1145/2837614.2837625

CISE conditions

For all updates *U, U'*:

(1) Sequentially safe:

- Initial state satisfies invariant *Inv*
- Precondition of *U* satisfies the weakest-precondition of the invariant *wp(U,Inv)*

Individually Safe

(2) Convergent:

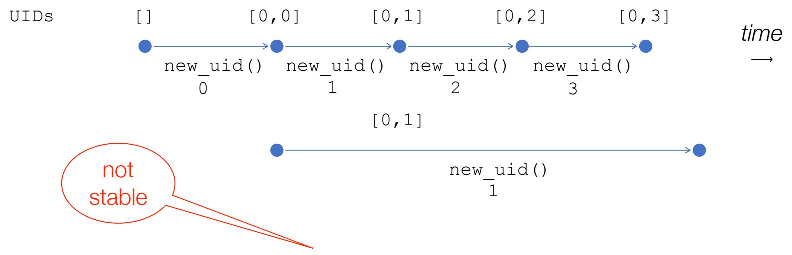
- When *U* concurrent to *U'*
- replace *U* || *U'* with *merge(U,U')*
- *merge(U,U')* commutative, associative, idempotent
- *merge(U,U')* preserves *Inv*

(3) Stable precondition:

- When *U* concurrent to *U'*
- the precondition of *U* is not made false by *U'*

harmless TOCTTOU

CISE Concurrent generate UID

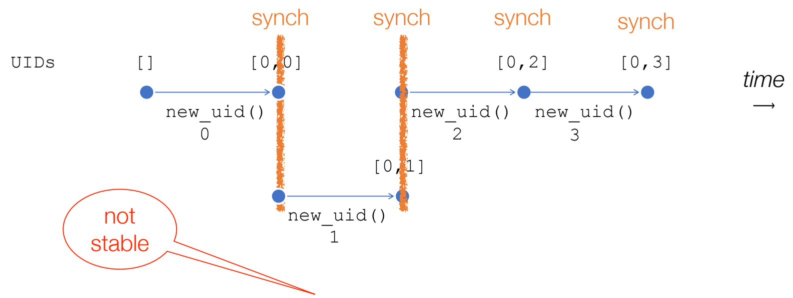


not stable

$$\frac{M \text{ is a set} \qquad n \notin M \qquad M' \text{ is a set}}{M' = M \cup \{n\}}$$

Solution: serialise ⟹ consensus

CISE Concurrent generate UID

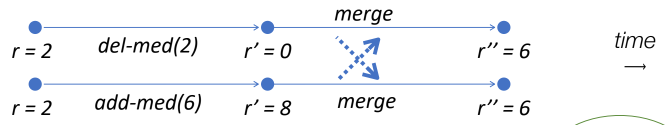


not stable

$$\frac{M \text{ is a set} \qquad n \notin M \qquad M' \text{ is a set}}{M' = M \cup \{n\}}$$

Solution: serialise ⟹ consensus

CISE Concurrent delivery of medications (1)



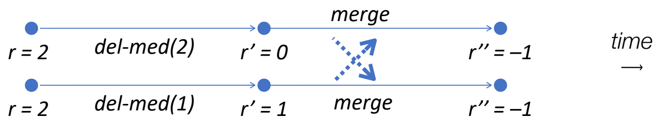
Stable w.r.t. add, deliver

$$\textit{add-med(n)} \quad \frac{r \ge 0 \qquad n \ge 0 \qquad r' \ge 0}{r' = r+n}$$

$$\textit{del-med(n)} \quad \frac{r \ge 0 \qquad r \ge n \ge 0 \qquad r' \ge 0}{r' = r-n}$$

- *add-med* || *add-med* OK
- *del-med* || *add-med* OK

CISE Concurrent delivery of medications (2)



$$\textit{add-med}(n)\quad \frac{r \geq 0 \qquad n \geq 0 \qquad r' \geq 0}{r' = r+n}$$

$$\textit{del-med}(n)\quad \frac{r \geq 0 \qquad r \geq n \geq 0 \qquad r' \geq 0}{r' = r-n}$$

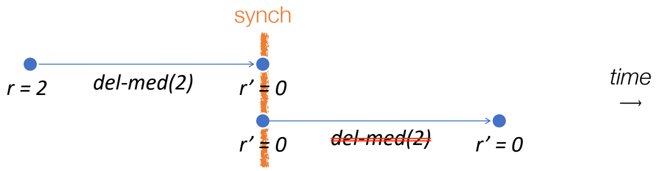
Not stable w.r.t. deliver

del-med design alternatives:

- Allow over-delivery ⟶ possibly punish after the fact
- Synchronise *del-med; del-med* ⟶ lose availability

weaken invariant, compensate

CISE Concurrent delivery of medications (3)



$$\textit{add-med}(n)\quad \frac{r \geq 0 \qquad n \geq 0 \qquad r' \geq 0}{r' = r+n}$$

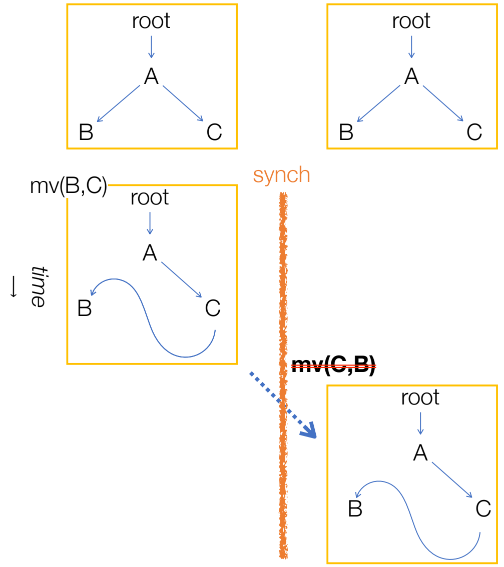
$$\textit{del-med}(n)\quad \frac{r \geq 0 \qquad r \geq n \geq 0 \qquad r' \geq 0}{r' = r-n}$$

Not stable w.r.t. deliver

del-med design alternatives:

- Allow over-delivery ⟶ possibly punish after the fact
- Synchronise *del-med; del-med* ⟶ lose availability

CISE Tree: serialise move



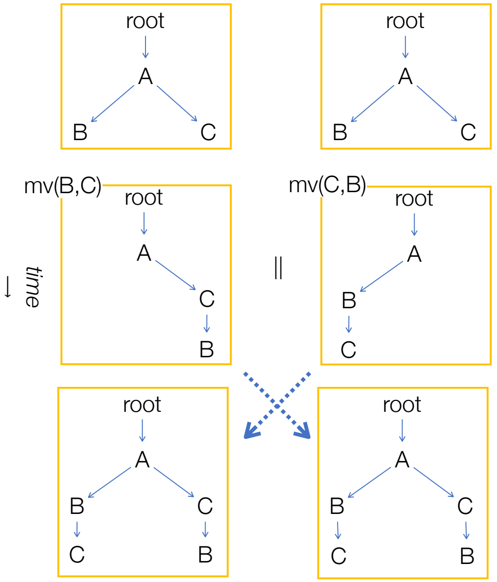
Not stable w.r.t. concurrent mv

$$\frac{Inv(\sigma) \qquad \neg\, \sigma.path(c, np) \qquad Inv(\sigma')}{\sigma \xrightarrow{mv\,(c,\, np)} \sigma'}$$

Impossible to satisfy both moves without violating the invariant!

- serialise conc. moves

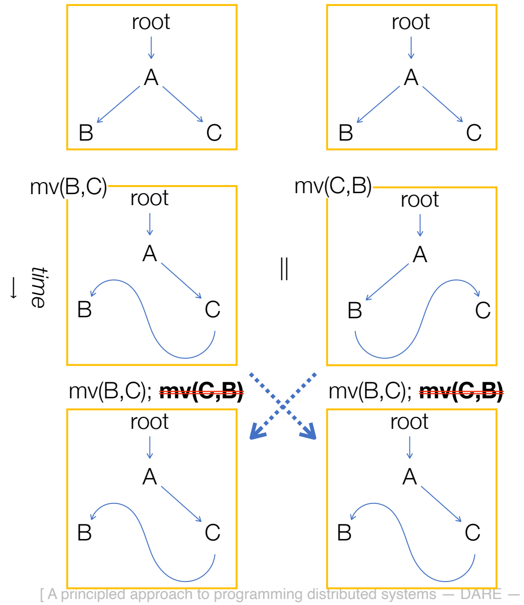
Dropbox: *mv* as cut & paste



Impossible to satisfy both moves without violating the invariant!

- implement move as cut & paste

Tree + mv: Kleppmann’s approach



$$\frac{Inv(\sigma) \qquad \neg\, \sigma.path(c, np) \qquad Inv(\sigma')}{\sigma \xrightarrow{mv\,(c,\, np)} \sigma'}$$

Impossible to satisfy both moves without violating the invariant!

If concurrent moves cause cycle:

- roll back
- serialise

Not “monotonic reads”

Summary: Tree + mv

Sequentially correct: sequential moves are OK

- Weakest precondition: not mv under self

CISE: precondition not stable under concurrent *mv*

Known design options:

- no *mv* op (XML)
- no *mv* op, copy-paste ⟹ duplicates (Dropbox)
- *up-mv* vs. *down-mv* [Nair 2021]
  - *up-mv* || *up-mv* stable
  - *up-mv* || *down-mv* stable
  - *down-mv* || *down-mv* not stable
- serialise *a priori:* lock [Najafzadeh 2018]
- serialise *a posteriori*: non-monotonic [Kleppmann 2022]

CISE conditions (again!)

For all updates *U, U':*

(1) Sequentially safe:

- Initial state satisfies invariant *Inv*
- Precondition of *U* satisfies the weakest-precondition of the invariant *wp(U,Inv)*

(2) Convergent:

- When *U* concurrent to *U'*
- *U* || *U’* convergent
  - *merge(U,U')* preserves *Inv*

(3) Stable precondition:

- When *U* concurrent to *U'*
- the precondition of *U* is not made false by *U'*

Using CISE in practice

Manual:

1. Individually correct: manual, testing
2. Convergence: CRDT library
3. Stability: consider all pairs of possibly concurrent updates

Tools

- Library: Bounded Counter [Balegas SRDS 2015]
- Stand-alone: specification language + SMT solver
  - CISE Tool [Najafzadeh 2015]
  - Soteria [Nair ESOP 2020]
  - BLOOM [Alvaro CIDR 2011]
- Integrated language/compiler
  - Conflict-Aware Replicated Data Types [arXiv 1802.08733]
  - LoRe [Haas ECOOP 2023]

Summary: As fast as possible

U, U' individually safe ∧ convergent ∧ mutually stable

- May execute concurrently
- Availability
- Perfect scalability

Summary: As strong as necessary

U, U' must not execute concurrently if:
    not commutative ∧ not convergent
∨ not mutually stable

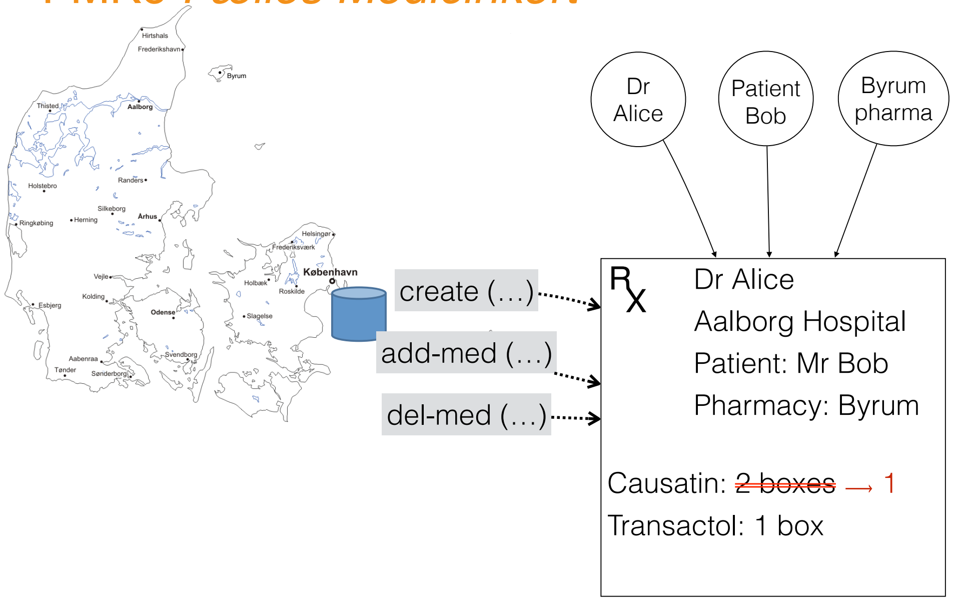
Design options:

- Refine invariant (e.g., bank account number)
- Downgrade invariant
  - from *"require x≥0"* to *"prefer x≥0"*
- Compensate: Weaken invariant, repair
- Serialise: *U; U'* or *U'; U*
  - Lock, single server, social convention, etc.
  - Monotonic, a priori: consensus
  - A posteriori: rollback. Finality?

Analyse again!

Classifying invariants by their coordination protocol

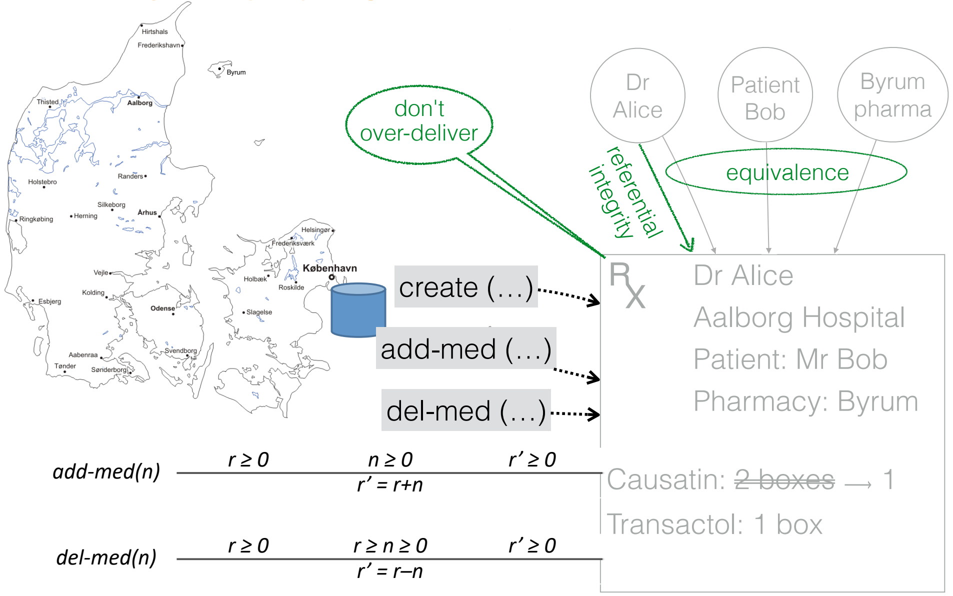
FMKe *Fælles Medicinkort*



Dr Alice
Aalborg Hospital
Patient: Mr Bob
Pharmacy: Byrum

Causatin: ~~2 boxes~~ → 1
Transactol: 1 box

FMKe invariants



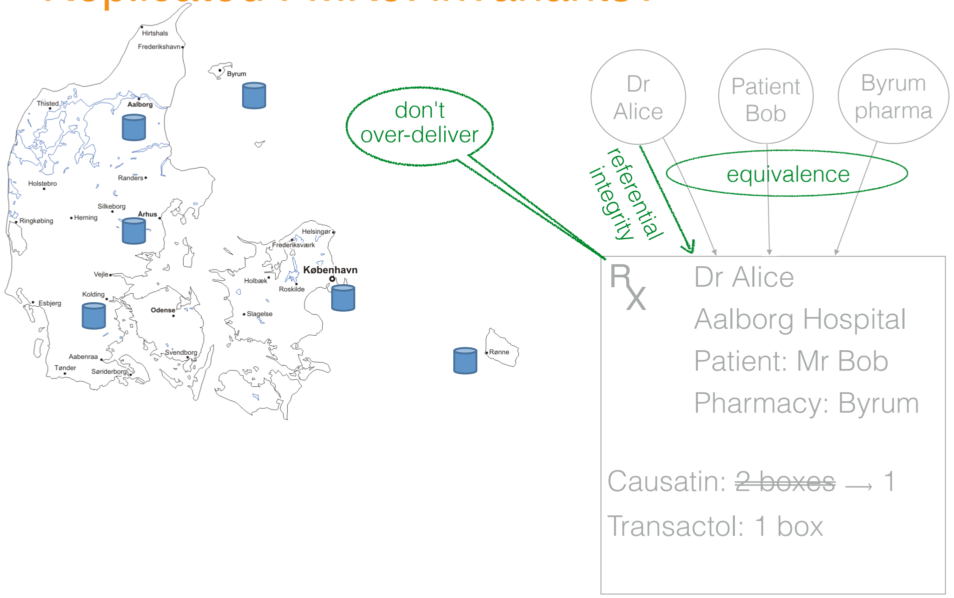
Dr Alice
Aalborg Hospital
Patient: Mr Bob
Pharmacy: Byrum

Causatin: ~~2 boxes~~ ⟶ 1
Transactol: 1 box

add-med(n) $\dfrac{r \geq 0 \quad n \geq 0 \quad r' \geq 0}{r' = r+n}$

del-med(n) $\dfrac{r \geq 0 \quad r \geq n \geq 0 \quad r' \geq 0}{r' = r-n}$

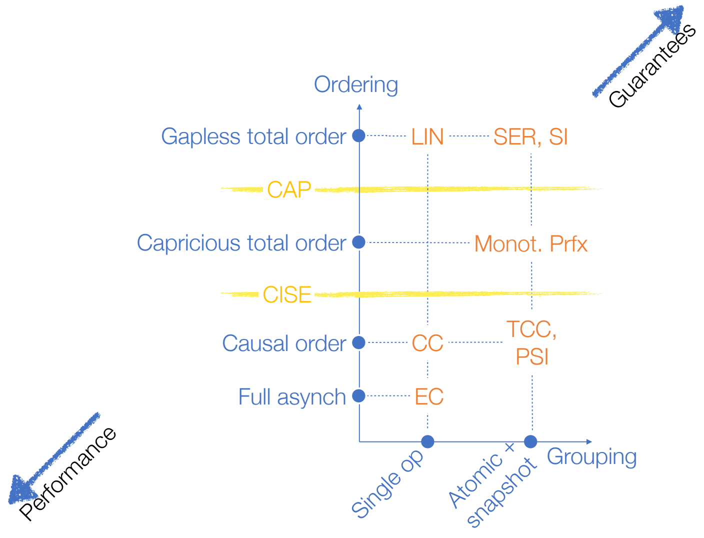
Replicated FMKe: invariants?



Dr Alice
Aalborg Hospital
Patient: Mr Bob
Pharmacy: Byrum

Causatin: ~~2 boxes~~ ⟶ 1
Transactol: 1 box

What protocols for what invariants?



Fully commutative updates

Some examples:

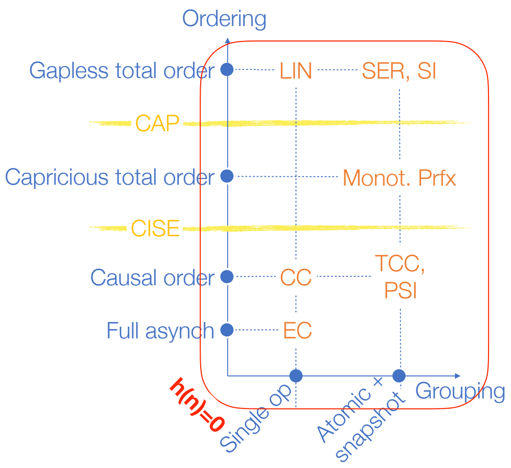
- Non-shared state
- Local blind: e.g., credit card number *hash(n)* = *0*
- Empty invariant + fully-commuting CRDTs
    - LWW
    - Grow-only set
    - PN counter
    - Vector clock

Convergent

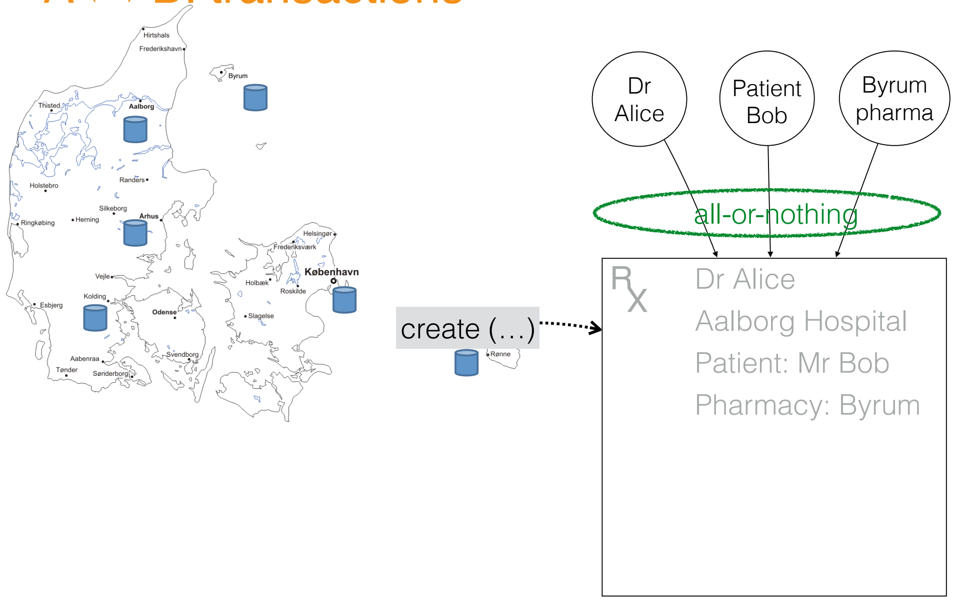
Asynchronous propagation

- Perfect scalability
- Perfect availability under partition

What protocols for fully commutative?



A ⟺ B: transactions



Transaction: Atomic writes + snapshot reads

create-p updates doctor, patient & pharmacy record

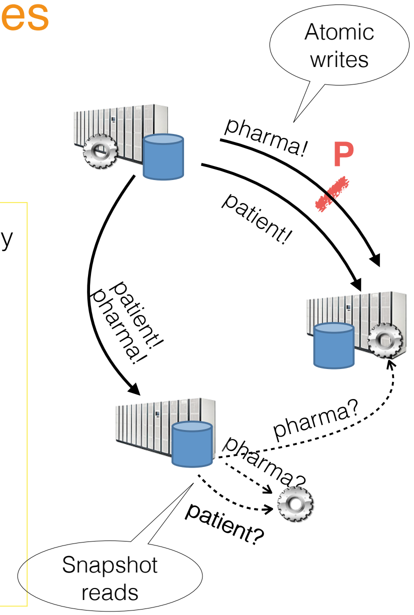
Atomic:

- = All-or-Nothing (A of ACID)
- Transmit joint updates together
- asynchronous

Snapshot: single database state

- multi-version concurrency control
- asynchronous

Asynch: Available under partition

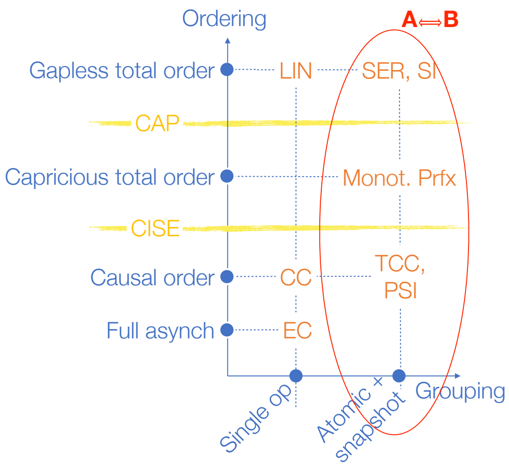


"A ⟺ B" style invariants

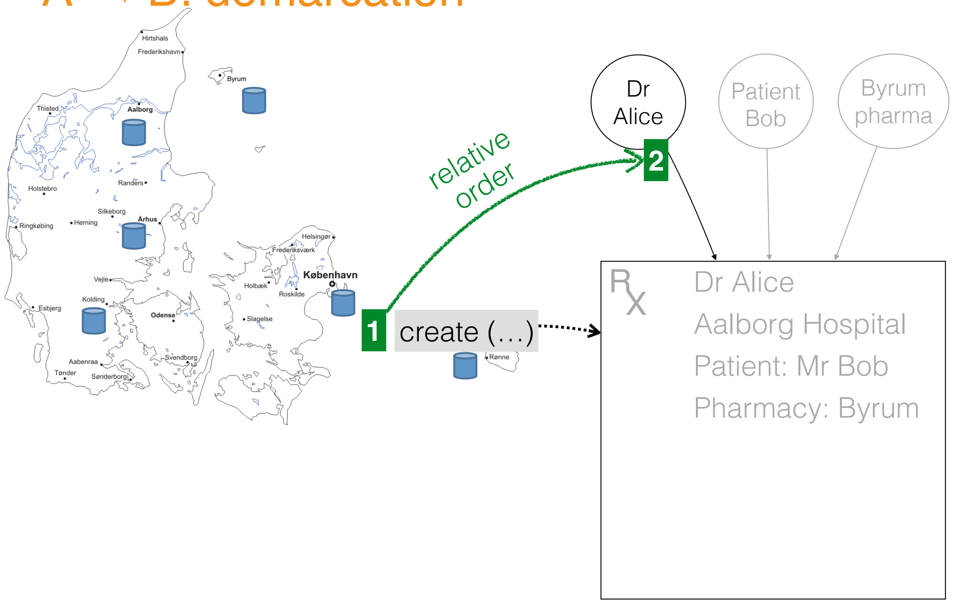
Some example cases:

- *A=B*
- *A = ¬B*
- *friend(x,y)* ⟺ *friend(y,x)*
- *x≤y: <x++;y++>* (one actor)

What protocols for A ⟺ B?

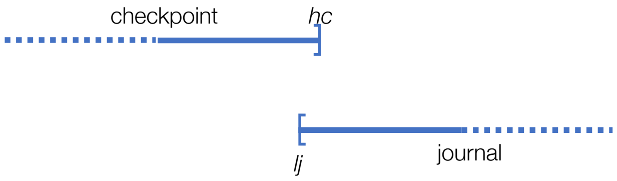


A ⟹ B: demarcation



Dr Alice
Aalborg Hospital
Patient: Mr Bob
Pharmacy: Byrum

Distributed checkpoint: demarcation

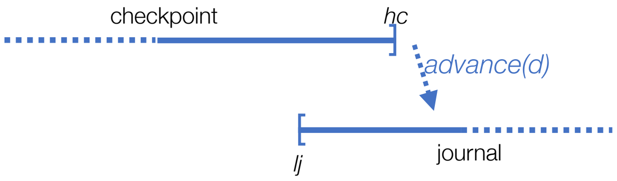


Inv(σ) = { σ.lj ≤ σ.hc}   *// no gap*

Demarcation protocol

- Journal dæmon: lower bound += d
- Journal: Send message to Checkpoint *advance(d)*
- Checkpoint dæmon: upper bound += d

Distributed checkpoint: demarcation

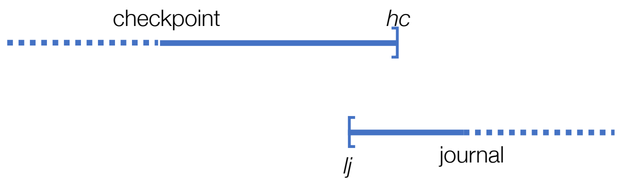


Inv(σ) = { σ.lj ≤ σ.hc}   *// no gap*

Demarcation protocol

- Journal dæmon: lower bound += d
- Journal: Send message to Checkpoint *advance(d)*
- Checkpoint dæmon: upper bound += d

Distributed checkpoint: demarcation



$Inv(\sigma) = \{ \sigma.lj \leq \sigma.hc \}$ *// no gap*

Demarcation protocol

- Journal dæmon: lower bound += d
- Journal: Send message to Checkpoint *advance(d)*
- Checkpoint dæmon: upper bound += d

Non-commuting, demarcation

Non-commuting CRDTs

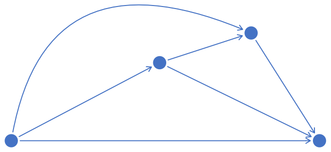
- Empty invariant
  - Set
  - Map
  - MVR
- Universally-stable operations
  - Acyclic graph: add-parallel, remove
  - Sequence: insert-at, remove

2-actor implication invariants

- A ⟹ B
- Referential integrity
- Chicken/fox/grain: *grain* ⟹ ¬*chicken*
- *x≤y*

Preserve order across processes

⟶ Causal Consistency



Causal Consistency

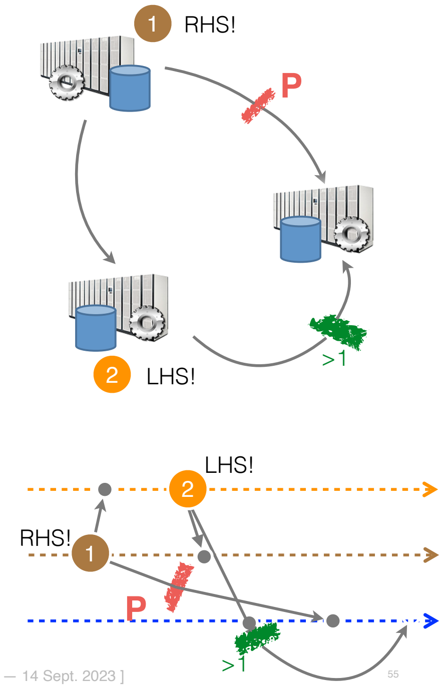
create-p before *add-med*

- “Bob points to Rx ⟹ Rx valid”
  - Referential integrity
- General case: LHS ⟹ RHS
- pattern: RHS!; LHS!

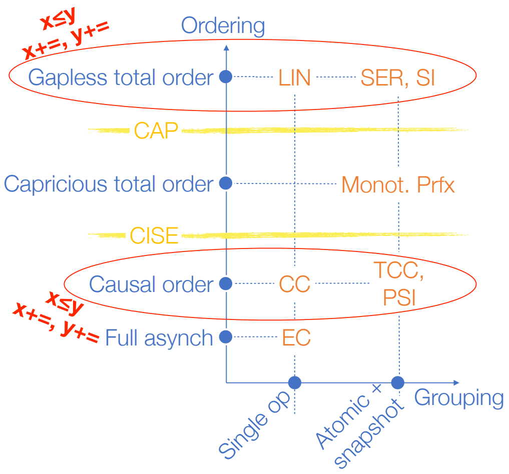
Deliver in the right order: Causal Consistency

Local decision:

- requires metadata
- available



What protocols for demarcation?



x≥0 x—: total/mutual order

u(), v() not mutually stable

- “Conflicting”
- Either *u()* before *v()*, or *v()* before *u()*

Protocols:

- General case: total order, consensus
- 1 lock / set of mutually-conflicting operations
  - Coarser locks OK
- Single server / conflict set (flat combining)
- Social

Bounded Counter

Specific, common case

Shared counter:

- *x ≥ 0*
- *increment (n)*
- *decrement (n)* // precondition *x ≥ n*

Escrow:

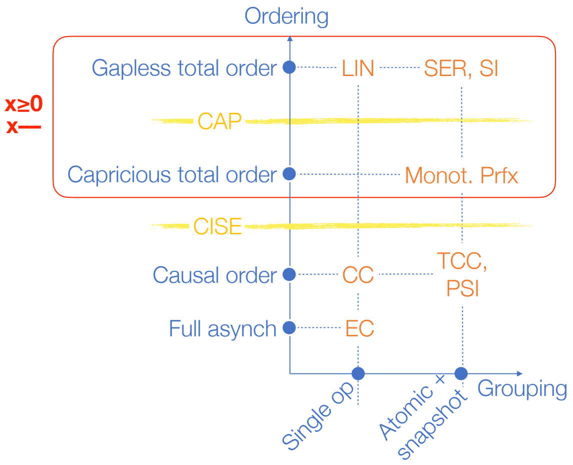
- Local share, decrement *share* −= *n*
- decrement disallowed if *share* < *n*
- Donate share

Mostly AP

Encapsulated, proven correct (CISE)

Causal ordering essential

What protocols for x≥0 x—?



Mutual order: *a posteriori* vs. *a priori*

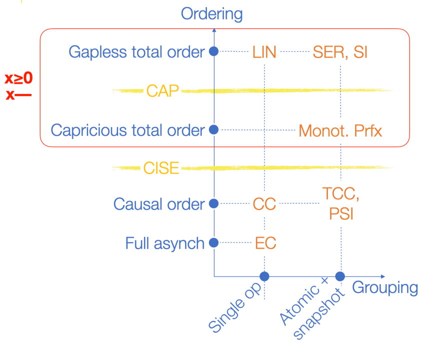
A posteriori, Capricious, Monotonic-Prefix:

- Execute
- Pick a number
- Propagate
- Sort
- Roll back; roll forward
- Iterate

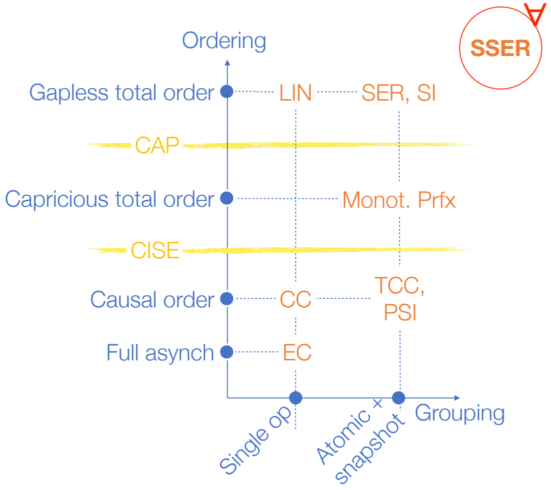
A priori, Gapless, Monotonic Reads:

- Consensus on a number
- Wait for my turn
- Execute & propagate

Capricious + finality ⟹ consensus



What protocols for arbitrary invariants?



Sweet spot: Transactional Causal Consistency + optional consensus

TCC =

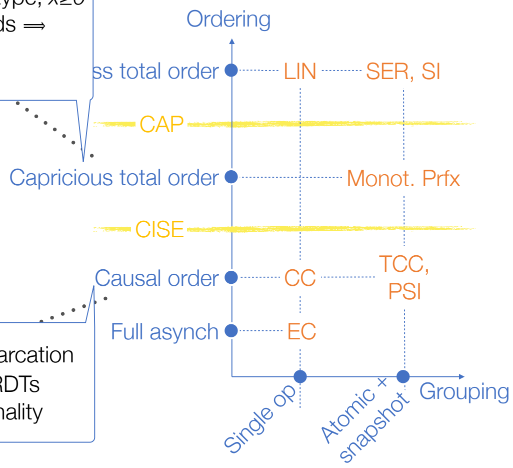
- Causal consistency
  - *x≤y,* demarcation
- Snapshot reads + Atomic writes
  - *A⟺B*
- System: AntidoteDB

Available: not *x≥0*

- Strengthen when necessary
- System: Colony

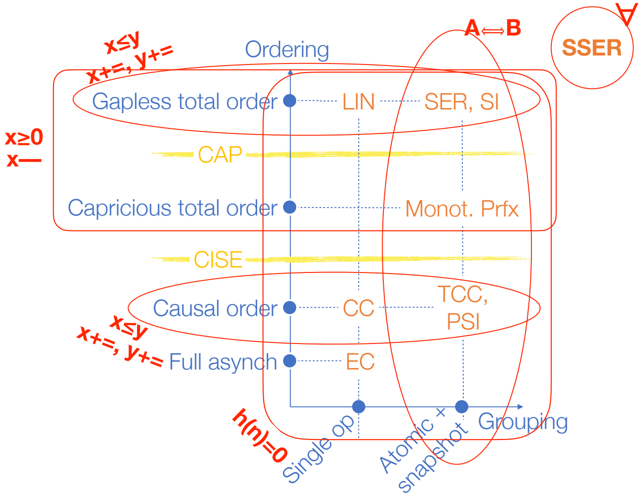
Strongest AP protocol(s)

Total order
Any sequential data type, *x≥0*
Non Monotonic Reads ⟹ rollbacks
No finality



Partial order ⟶ demarcation
Requires Merge / CRDTs
Monotonic Reads; finality

What protocols for what invariants?



Creative Commons Attribution-ShareAlike 4.0 Intl. License

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

for any purpose, even commercially, under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.