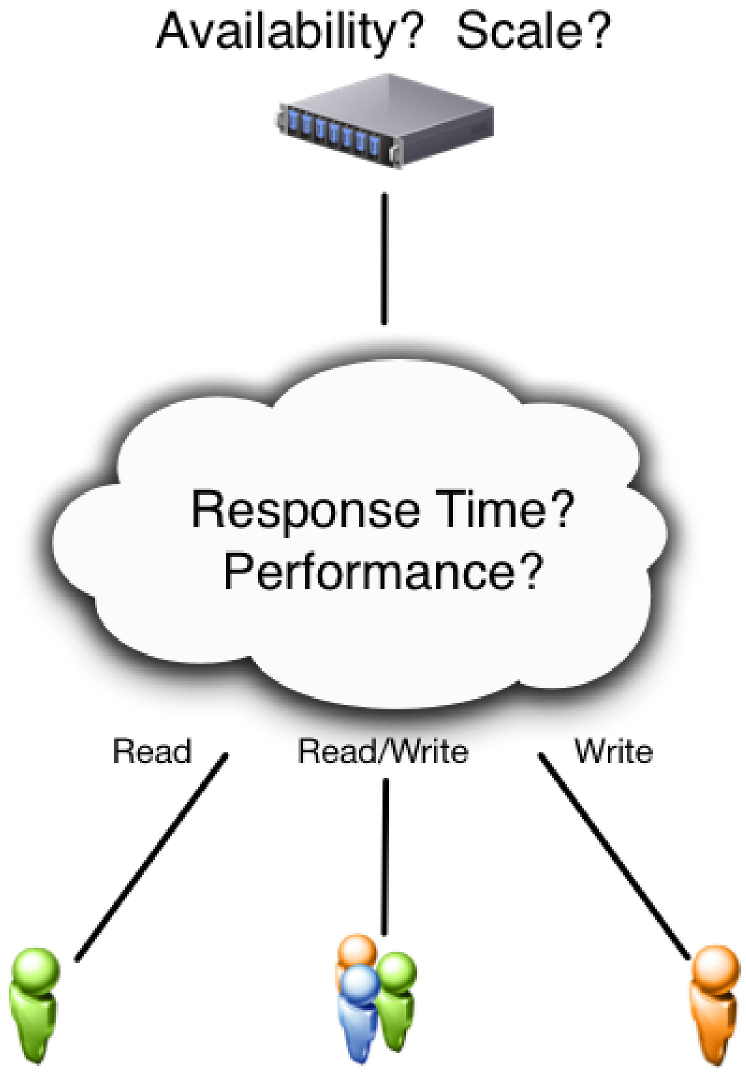
CRDTs
Conflict-free Replicated Data Types

Mohammad Ghasemisharif
John Kristoff
Sepideh Roghanchi

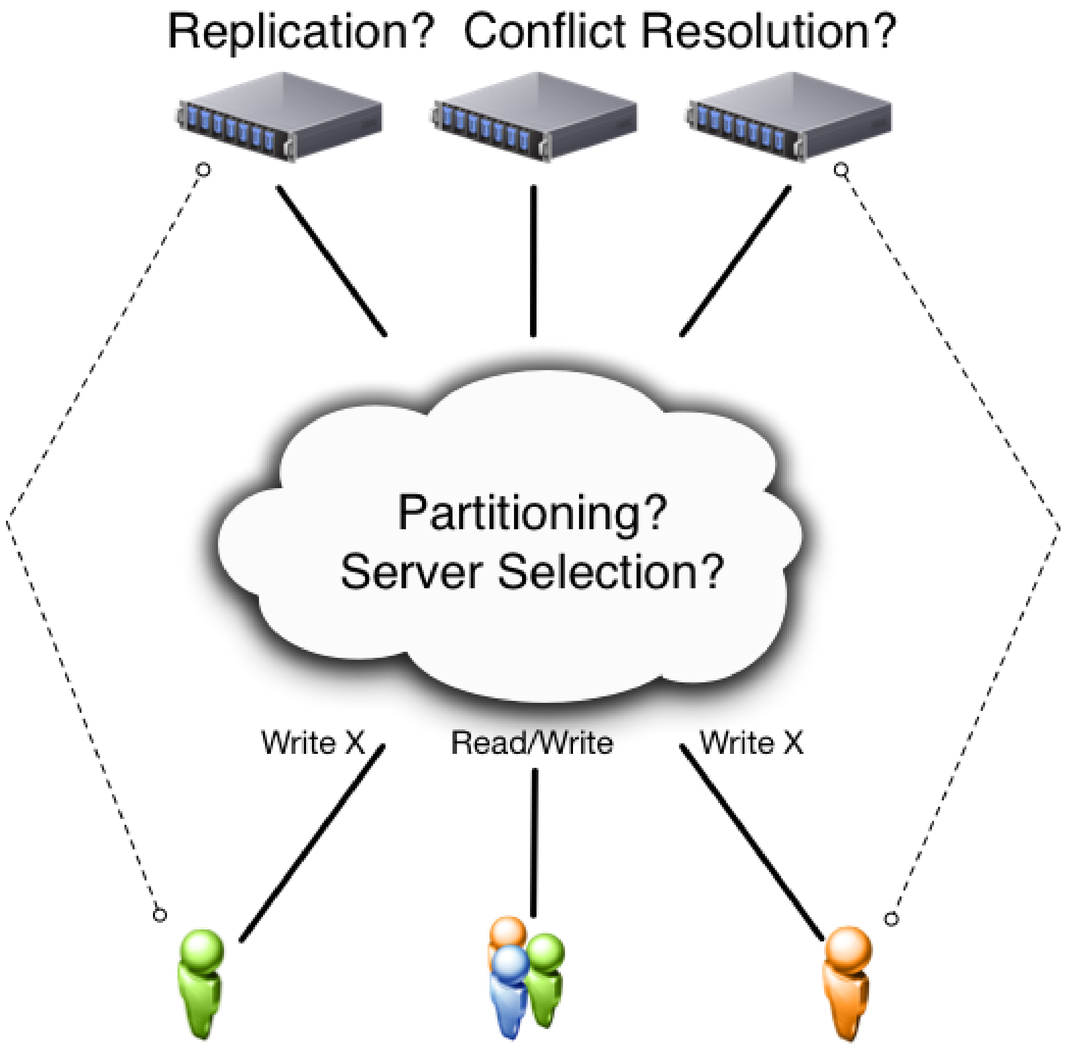
Agenda

- Background and problem definition
- Methods of synchronization
  - Operation-based consistency
  - State-based consistency
- Algorithms, pseudo-code, and examples
  - Data types (graph, set, counter)
- Real-world usage and limitations

Simple Client/Server



Distributed System



Conflicts and Resolution

- Partition A receives update(X_a)
- Partition B receives update(X_b)
- How to merge competing updates?
  - Timestamp?
  - Last-writer-wins?
  - Solution may be imperfect (i.e. a loser)
- Some data-types are conflict-free!

Replication

- Availability is desirable
- Fault-tolerance is desirable
- Low latency (high-performance) is desirable
- Maintaining consistency can be difficult

Key Concepts to Review

- Strong consistency
- Eventual consistency
- Conflict arbitration
  - Consensus and rollback
- CAP problem (pick 2)
  - Consistency
  - Availability
  - Partition Tolerance

strong consistency

- Everyone knows about every update immediately
- There is an order for all operations
- Everyone sees the same order
- Bottlenecks:
  - Consensus problem
    - Makes the system sequential
      - Slow, not scalable
    - Tolerates < n/2 failures

- Unfortunately, the CAP theorem tells us that no system satisfying those three desirable properties together exists:
  - Availability is often dropped!
- Drop strong consistency for a weaker form of consistency:
  - Eventual consistency

eventual consistency

- Update local + propagate
- All updates eventually take effect at all replicas, asynchronously and possibly in different orders
- Concurrent updates may conflict
- Still needs consensus
  - Conflict -> reconcile
  - Moved consensus off the critical path (background)
- Better performance, more complex
- May come at the cost of availability when synchronizing!

Motivation

- provide a data structure distributed over a large network and manipulated by a large base of users around the world
- Having multiple replicas of the data structure:
  - good for fault tolerance and read latency
  - Problem with updates :
    - Synchronize -> slow
    - Don’t synchronize -> conflicts
- The provided data structure should follow the CAP properties:
  - Consistency
  - Availability
  - partition-tolerance

Strong eventual consistency

- Update local + propagate
- A replica of the shared data structure is coherent with other replicas that *have observed* the same operations
- No synchronization (no consensus)
- Deterministic outcome for every conflict
- Allow any number of failure
- ✉ solves CAP problem

Origin Story

- 2011 publication by Marc Shapiro, et al.
- Strong consistency does not scale
- Eventual consistency is better, but…
- Concurrent conflict resolution is hard
- Therefore, strong eventual consistency

How to do it? Need data types to support it…

- CRDT: a simple, theoretically sound approach to eventual consistency
- replicas of any CRDT converge to a common state that is equivalent to some correct sequential execution
- Properties:
  - no synchronization
  - update executes immediately
  - unaffected by network latency, faults, or disconnection
  - extremely scalable
  - fault-tolerant
  - does not require much mechanism

CRDT

- Conflict-free
- Replicated
- Data Type

Data-type (in CRDTs)

- Data structures that ease consistency
- Eliminates complexity of consensus
- Data or operations must be commutative

Definitions

- EC:
  - **Eventual delivery:** *An update delivered at some correct replica is eventually delivered to all correct replicas:* $\forall i,j : f \in c_i \Rightarrow \Diamond f \in c_j$
  - **Termination:** *All method executions terminate*
  - **Convergence:** *Correct replicas that have delivered the same updates eventually reach equivalent state:* $\forall i,j : c_i = c_j \Rightarrow \Diamond s_i \equiv s_j$
- SCE
  - **Strong Convergence:** *Correct replicas that have delivered the same updates have equivalent state:* $\forall i,j : c_i = c_j \Rightarrow s_i \equiv s_j$

Op versus State-based

- Operation: add 5, subtract 6
- State: send(x_i), send(x_{i+1})

Commutative

- Data that commutes is a key property
  - e.g. add A, add B == add B, add A
- Not all ops or states are commutative
  - e.g. multiple 5, subtract 6 != subtract 6, multiply 5

Operation-based replication

- Local replica sends operation
- Concurrent operations must be commutative

op-based object

- Tuple: (S, s^0, q, t, u, P)
- S, s^0 and q same as state-based object
- Replica at process p_i has state $s_i \in S$
- Initial state is s^0
- t = side-effect-free *prepare-update*
- u = *effect-update*
- P = delivery precondition

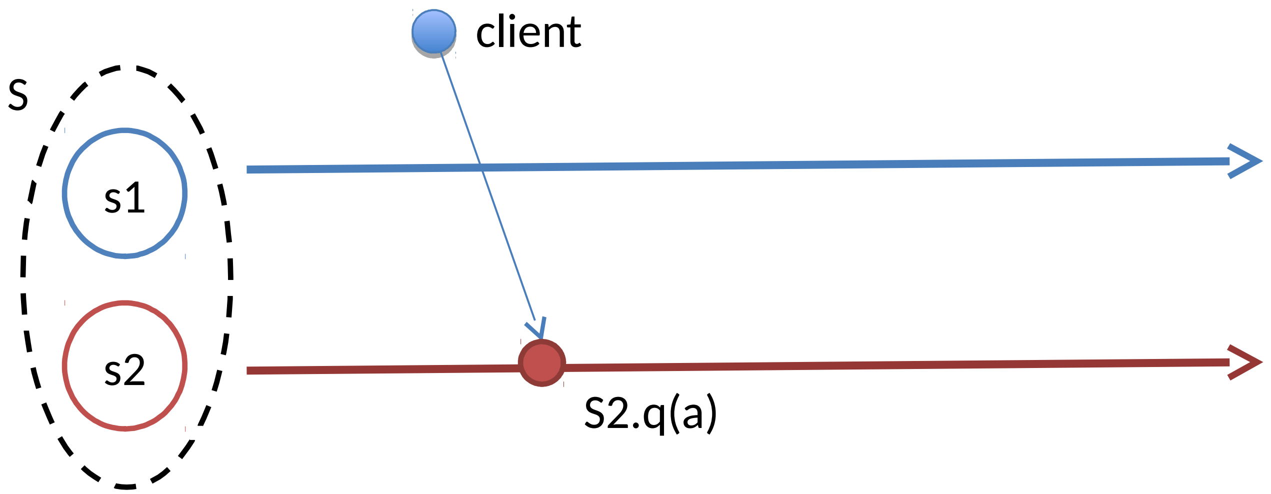
State-based replication

- Local replica (p1)
  - performs computation
  - updates local state
  - Periodically send(p1_state)
- Receiver replica(s)
  - perform merge(p1_state, p2_state)

state-based object

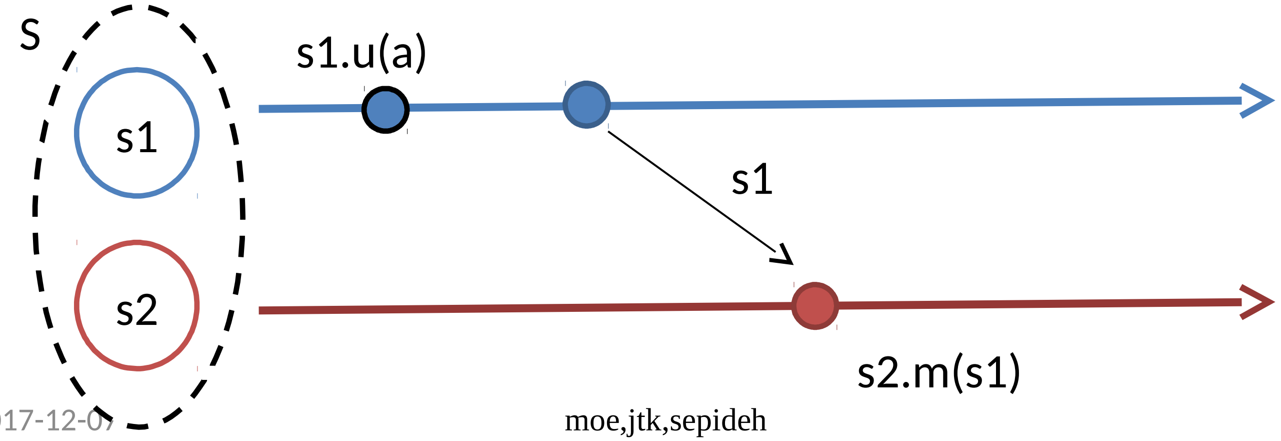
- Tuple: (S, s^0, q, u, m)
- Replica at process p_i has state $s_i \in S$
- Initial state is s^0
- q = query
- u = update
- m = merge

- Clients send query to read replica's state
  - Read only -> easy
- Updates:
  - State-based
  - Operation-based



State-based approach

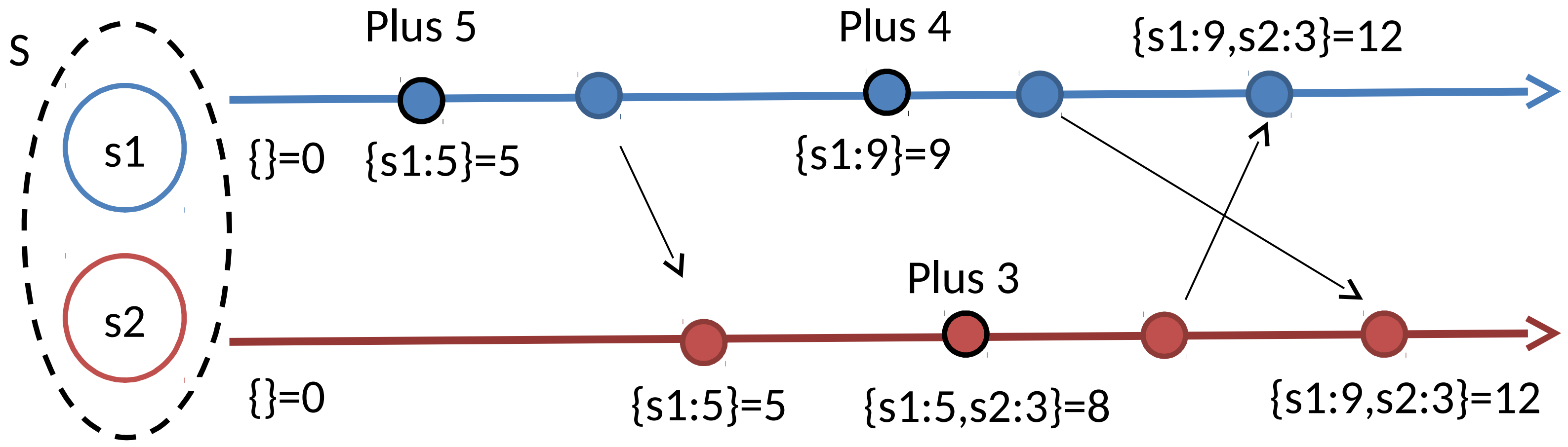
- State-based:
  - Local queries, local updates at source
  - Send full state every once in a while
  - On receive, merge
  - File systems (NFS, Unison, Dynamo)



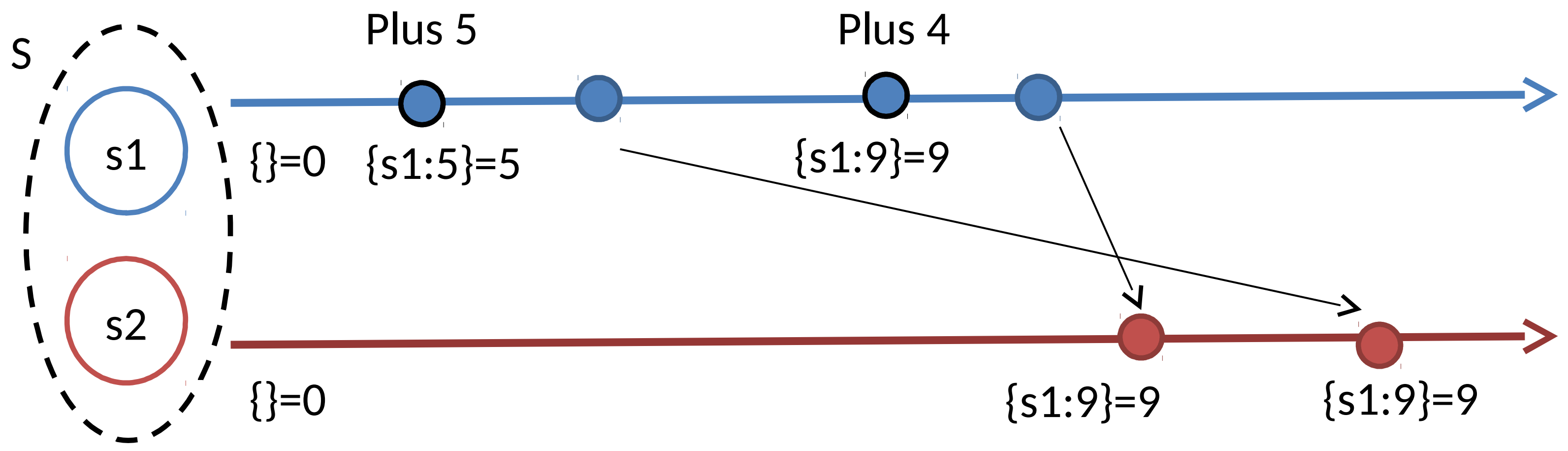
- all the replicas talk to each other directly on indirectly
- if it converges it will satisfy the strong consistency
- What is the sufficient condition for it to converge?

- Semilattice
  - Set with partial order and an operation can take any two values and give you an upper bound on them
- If the payload forms a semilattice (partial order on values, always can take an upper bound)
- If updates are increasing in semilattice
- If merge function computes this upper bound
- -> replicas converge to LUB of last values

State-based example



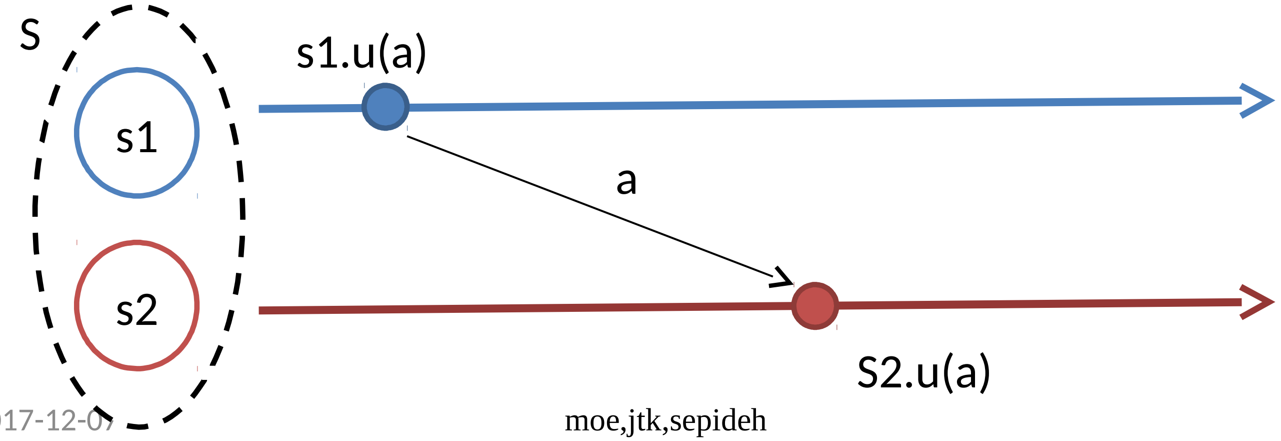
Another state-based example



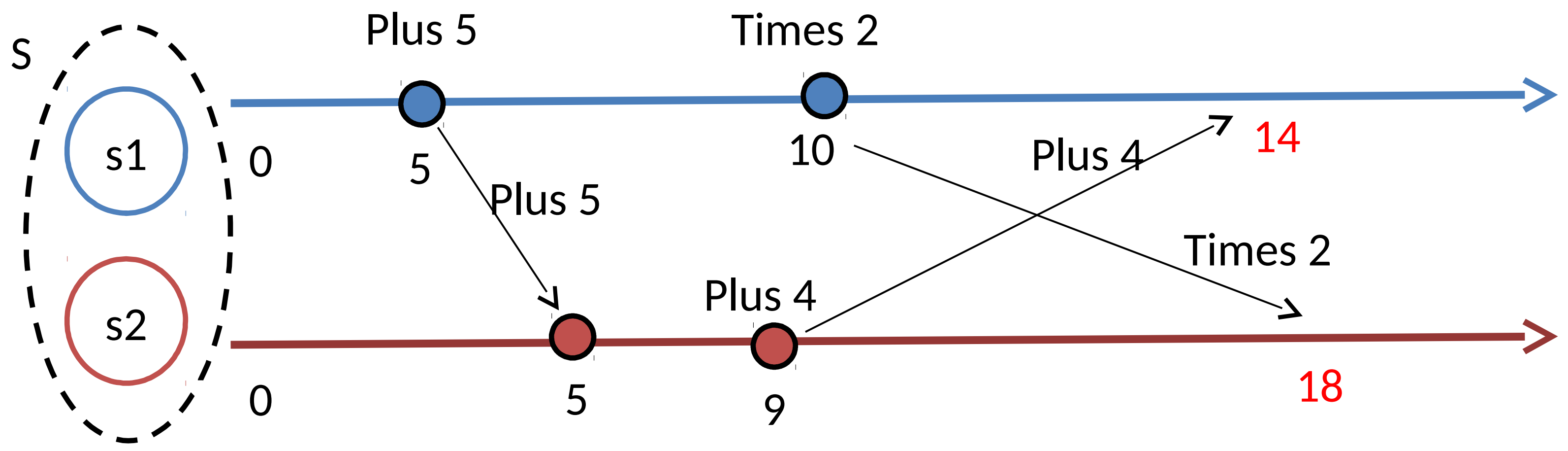
- Out of order delivery

Operation-based approach

- operation-based:
  - Only updates are sent (smaller)
  - Each replica replay the updates
  - Reconcile non-commutative operations
  - Collaborative editing, Bayou, PNUTS
- Need something stronger!
- Make sure that all updates are propagated to all other replicas (downstream replicas)



Op-based example



$$(5 + 4) \times 2 \neq (5 \times 2) + 4$$

- Updates should be commutative operations

Compare!

- state-based:
  - Update and merge
  - Simple data types
  - Not efficient for large objects
- Operation-based:
  - Update
  - More complex
  - More powerful
  - Small messages
- They are equivalent
  - you can take any state based object and emulate it in an op-based model and if one converges the other converges

G-Counter CRDT

- TODO: algorithm and method (see wikipedia page)

PN-Counter CRDT

- TODO: algorithm and method (see wikipedia page)

G-Set CRDT

- TODO: algorithm and method (see wikipedia page)

2P-Set CRDT

- TODO: algorithm and method (see wikipedia page)

LWW-Set CRDT

- TODO: algorithm and method (see wikipedia page)

OR-Set CRDT

- TODO: algorithm and method (see wikipedia page)

Sequence CRDTs

- TODO: algorithm and method (see see wikipedia page)

CRDT in Action

- SoundCloud
- Bet365
- Redis
- Riak
- League of Legends
- orbit-db

References

- https://pages.lip6.fr/Marc.Shapiro/
- https://www.youtube.com/watch?v=ebWVLVhiaiY
- https://www.youtube.com/watch?v=vBU70EjwGfw
- https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type