IPL 2025 Workshop on Foundations of Software

March 22, 2025

Masato Takeichi

"Coordination-free Collaborative Replication based on Operational Transformation"

Masato Takeichi. "Coordination-free Collaborative Replication based on Operational Transformation", September 16, 2024, Last Revised December 15 (version 3) arXiv:2409.09934v3. https://doi.org/10.48550/arXiv.2409.09934

How can we keep distributed replicated data consistent?

A Short Story

- Consider a real-world example of distributed data sharing: Peers P and Q have their local data D_P and D_Q to be appropriately "replicated".
  - P and Q have replicas D_P and D_Q as instances of the common set data.
  - P and Q update D_P and D_Q respectively whether or not the network connection is alive, and they try to get the new common states D'_P and D'_Q during the connection is alive.
- Each peer adds element x into its local replica (written as $\oplus x$) and removes element x from the replica (written as $\ominus x$), and sends these operations to the partner peer. This is the basic updating process of the peers.
- The peer receives remote operations sent from the partner peer and puts them on its local replica D so that it becomes same as that of the partner peer.
- What happens in the events:
  1. Start with D_P=D_Q = {A, B}.
  2. Connection fails.
  3. P does $\oplus$ C and then $\ominus$ C.
  4. Q does $\ominus$ B and $\oplus$ C.
  5. Connection is restored.

- How D_P and D_Q are replicated into D'_P and D'_Q?
- What is the result after step 5?
  - Is D'_P = {A, B}, or D'_P = {A, C}?
  - Is D'_Q = {A, C}, or D'_Q = {A, B, C}?
- Is it pertinent and appropriate by sound reasoning?

① Start with $D_P = D_Q = \{A, B\}$.
② Connection fails.
③ P does ⊕ C and then ⊖ C.
④ Q does ⊖ B and ⊕ C.
⑤ Connection is restored.

Ask CCRAgent for Consistent Replication

- What is the result after step 5?
  - Is $D'_P = \{A, B\}$, or $D'_P = \{A, C\}$?
  - Is $D'_Q = \{A, C\}$, or $D'_Q = \{A, B, C\}$?
- Is it pertinent and appropriate?

```
*** CCRAgent Ver6.3 for ESET_String (2025/01/13) ***

* Agent Started on Port 9001
#1: Conn 9000
* Connection $9000 started

> Received Ops [Add $0 "A",Add $0 "B"] from $9000

#2: Show
Replica {
 Data= {"A","B"}
 ...}
Connections= [$9000+:(2..2](2..2]]

#3: Delay 30
* Agent delays 30 sec. before accepts Patch ②

#4: Rem "B", Add "C" ④

< Sent Ops [Rem $1 "B",Add $1 "C"] to $9000
> Received Ops [Add $0 "C",Rem $0 "C"] from $9000 ⑤

< Sent Ops [Rem $1 "B",Add $1 "C",Rem $0 "C"] to $9000
> Received Ops [Add $0 "C",Rem $0 "C",Rem $1 "B"]
   from $9000

#5: Show
Replica {
 Data= {"A"}
 Revs= 5
 UpdLog= [Add $0 "A",Add $0 "B",Rem $1 "B",Add $1 "C",
                Rem $0 "C"]
 ...}
Connections= [$9000+:(5..5](5..5]]
```

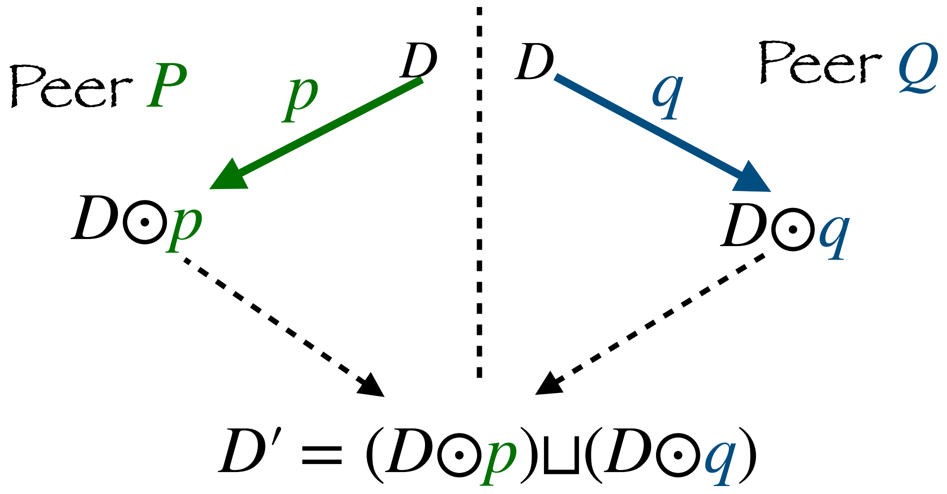
①

Why not {A, C} ?

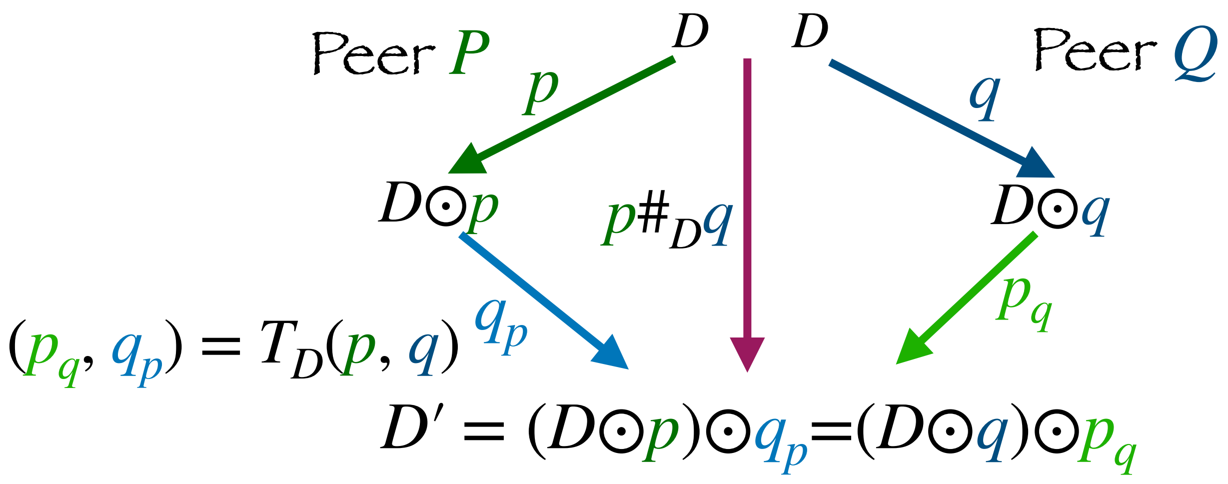
```
**** CCRAgent Ver6.3 for ESET_String (2025/01/13) ***

* Agent Started on Port 9000
#1: Conn 9001
* Connection $9001 started

#2: Add "A", Add "B"

< Sent Ops [Add $0 "A",Add $0 "B"] to $9001

#3: Show
Replica {
 Data= {"A","B"}
 ...}

#4: Delay 30
* Agent delays 30 sec. before accepts Patch ②

#5: Add "C", Rem "C"  ③

< Sent Ops [Add $0 "C",Rem $0 "C"] to $9001
> Received Ops [Rem $1 "B",Add $1 "C"] from $9001 ⑤

< Sent Ops [Add $0 "C",Rem $0 "C",Rem $1 "B"] to $9001
> Received Ops [Rem $1 "B",Add $1 "C",Rem $0 "C"]
   from $9001

#6: Show
Replica {
 Data= {"A"}
 Revs= 5
 UpdLog= [Add $0 "A",Add $0 "B",Add $0 "C",Rem $0 "C",
                Rem $1 "B"]
 ...}
Connections= [$9001+:(5..5](5..5]]
```

Problems to be solved for Consistent Replication

- Local replicated data $D_P \in \mathscr{D}$ in P and $D_Q \in \mathscr{D}$ in Q may be concurrently updated by p to produce $D_P \odot p \in \mathscr{D}$ and by q to produce $D_Q \odot q \in \mathscr{D}$.
- Given the Lastly Replicated Common State $D \in \mathscr{D}$ as the "baseline", how can we make $D \odot p$ and $D \odot q$ consistent for the next baseline $D' \in \mathscr{D}$?



CRDT Solution



OT-based Solution

- Predefined "merge" $\sqcup$ on $\mathscr{D}$ produces the next baseline $D' \in \mathscr{D}$ from $D \in \mathscr{D}$.
- p and q must be conformable to $\sqcup$ defined on partially ordered set $\mathscr{D}$ with element-wise operation $\odot x$.
  - Set $\mathscr{D}$: $\odot x = \cup \{x\}$ for *insert* x into D.
  - Counter $\mathscr{D}$: $\odot x = +x$ for *add* x to
  - Max $\mathscr{D}$: $\odot x = \uparrow x$ for *max* of x and D.

- $T_D(p, q)$ produces (p_q, q_p) for generating the "confluent" operation $p \#_D q$ which makes $D \in \mathscr{D}$ into the next baseline $D' \in \mathscr{D}$.
  - $p \odot q_p$ and $q \odot p_q$ are concrete representations of $p \#_D q$.
- $T_D(p, q)$ must satisfy TP1 and TP2 properties for Consistency and Coordination Avoidance.

Shopping Cart Problem in CRDT implementation

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

Shadaj Laddad, et.al. "Keep CALM and CRDT On". PVLDB, 16(4): 856-863, 2022

Current Trend in Replicated Data Sharing

- Intended Contents of the 2P-Set = $A - R$
- Cannot effectively add items into the Cart (Set) if they have been once added and then removed.

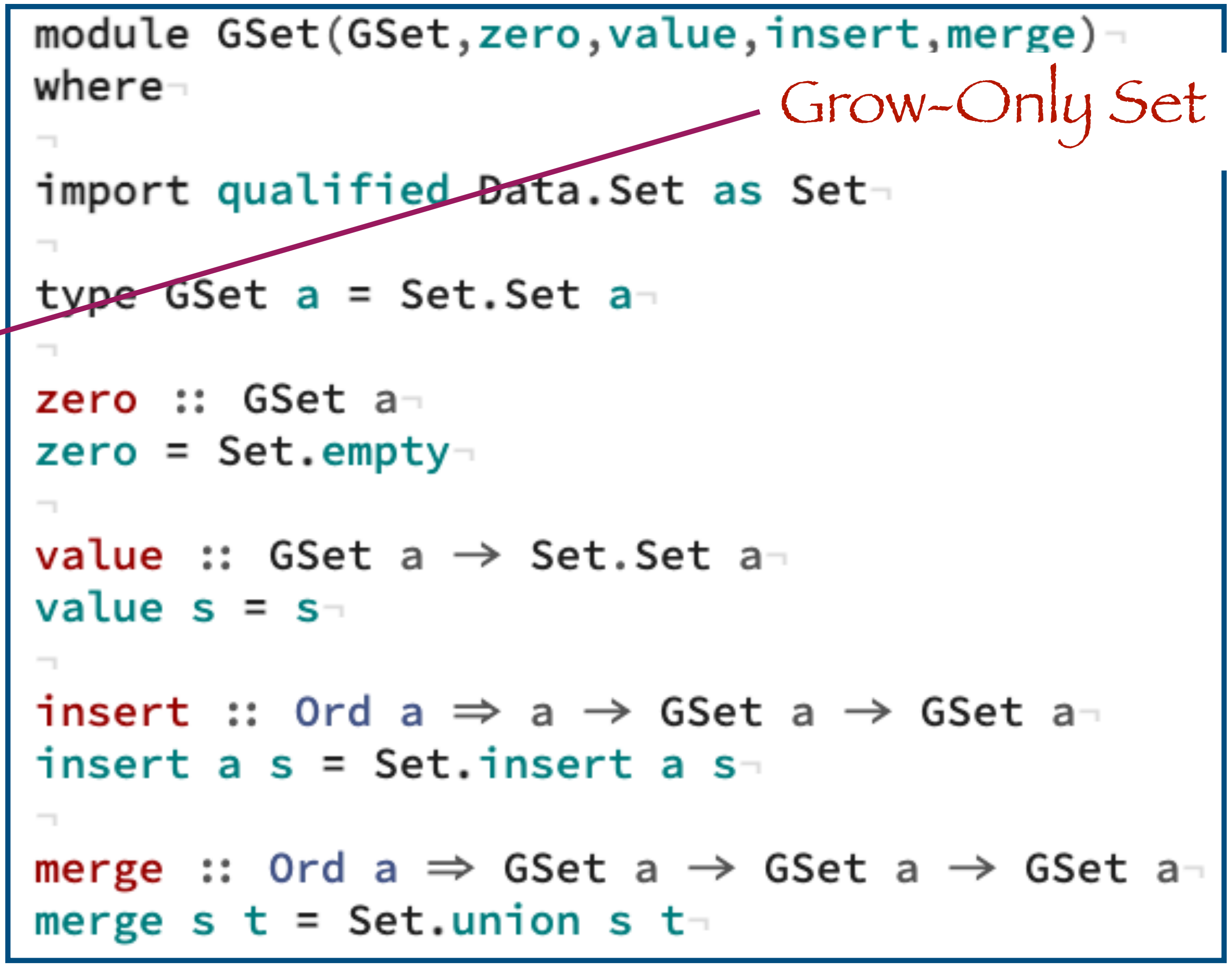
Example 1 (The Potato and the Ferrari, a.k.a. Early Read). *A canonical CRDT is the Two-Phase Set (2P-Set) [51], which is a pair of sets* (A, R) *that track items to be added (A) and removed (R). The merge function for two 2P-Sets is defined simply as the pairwise union,* $(A_1 \cup A_2, R_1 \cup R_2)$ *and is patently ACI. This scheme was used in the well-known Amazon Dynamo shopping cart example [11].*

tombstone set

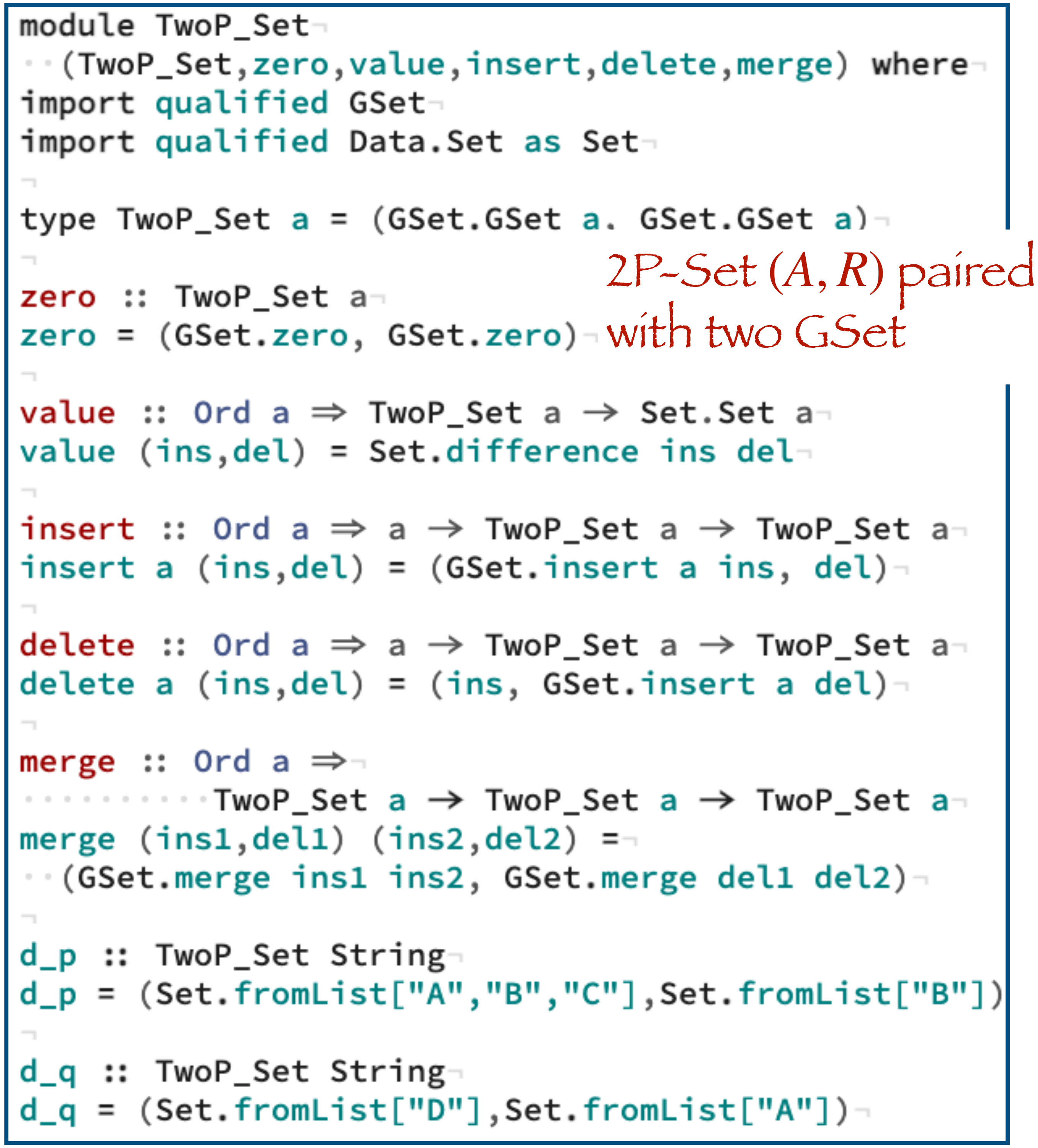
Demonstration of Two-Phase Set CRDT in Haskell

CRDT can represent grow-able data only!

Grow-Only Set



```haskell
module GSet(GSet,zero,value,insert,merge)
where

import qualified Data.Set as Set

type GSet a = Set.Set a

zero  :: GSet a
zero = Set.empty

value  :: GSet a → Set.Set a
value s = s

insert  :: Ord a ⇒ a → GSet a → GSet a
insert a s = Set.insert a s

merge  :: Ord a ⇒ GSet a → GSet a → GSet a
merge s t = Set.union s t
```



```haskell
module TwoP_Set
  (TwoP_Set,zero,value,insert,delete,merge) where
import qualified GSet
import qualified Data.Set as Set

type TwoP_Set a = (GSet.GSet a. GSet.GSet a)

zero  :: TwoP_Set a
zero = (GSet.zero, GSet.zero)

value  :: Ord a ⇒ TwoP_Set a → Set.Set a
value (ins,del) = Set.difference ins del

insert  :: Ord a ⇒ a → TwoP_Set a → TwoP_Set a
insert a (ins,del) = (GSet.insert a ins, del)

delete  :: Ord a ⇒ a → TwoP_Set a → TwoP_Set a
delete a (ins,del) = (ins, GSet.insert a del)

merge  :: Ord a ⇒
          TwoP_Set a → TwoP_Set a → TwoP_Set a
merge (ins1,del1) (ins2,del2) =
  (GSet.merge ins1 ins2, GSet.merge del1 del2)

d_p  :: TwoP_Set String
d_p = (Set.fromList["A","B","C"],Set.fromList["B"])

d_q  :: TwoP_Set String
d_q = (Set.fromList["D"],Set.fromList["A"])
```

2P-Set (A, R) paired with two GSet

```
ghci> d_p
(fromList ["A","B","C"],fromList ["B"])
ghci> d_q
(fromList ["D"],fromList ["A"])
ghci> value d_p
fromList ["A","C"]
ghci> value d_q
fromList ["D"]
ghci> merge d_p d_q
(fromList ["A","B","C","D"],fromList ["A","B"])
ghci> value it
fromList ["C","D"]
ghci> insert "B" d_p
(fromList ["A","B","C"],fromList ["B"])
ghci> value d_p
fromList ["A","C"]
```

$D_P = (\{A, B, C\}, \{B\})$

$D_Q = (\{D\}, \{A\})$

2P-Set D_P represents Set $\{A, C\}$

2P-Set D_Q represents Set $\{D\}$

$D_P \sqcup D_Q = (\{A, B, C, D\}, \{A, B\})$

$D_P \sqcup D_Q$ represents $\{C, D\}$

B is added again into D_P

But B has not been added to Set

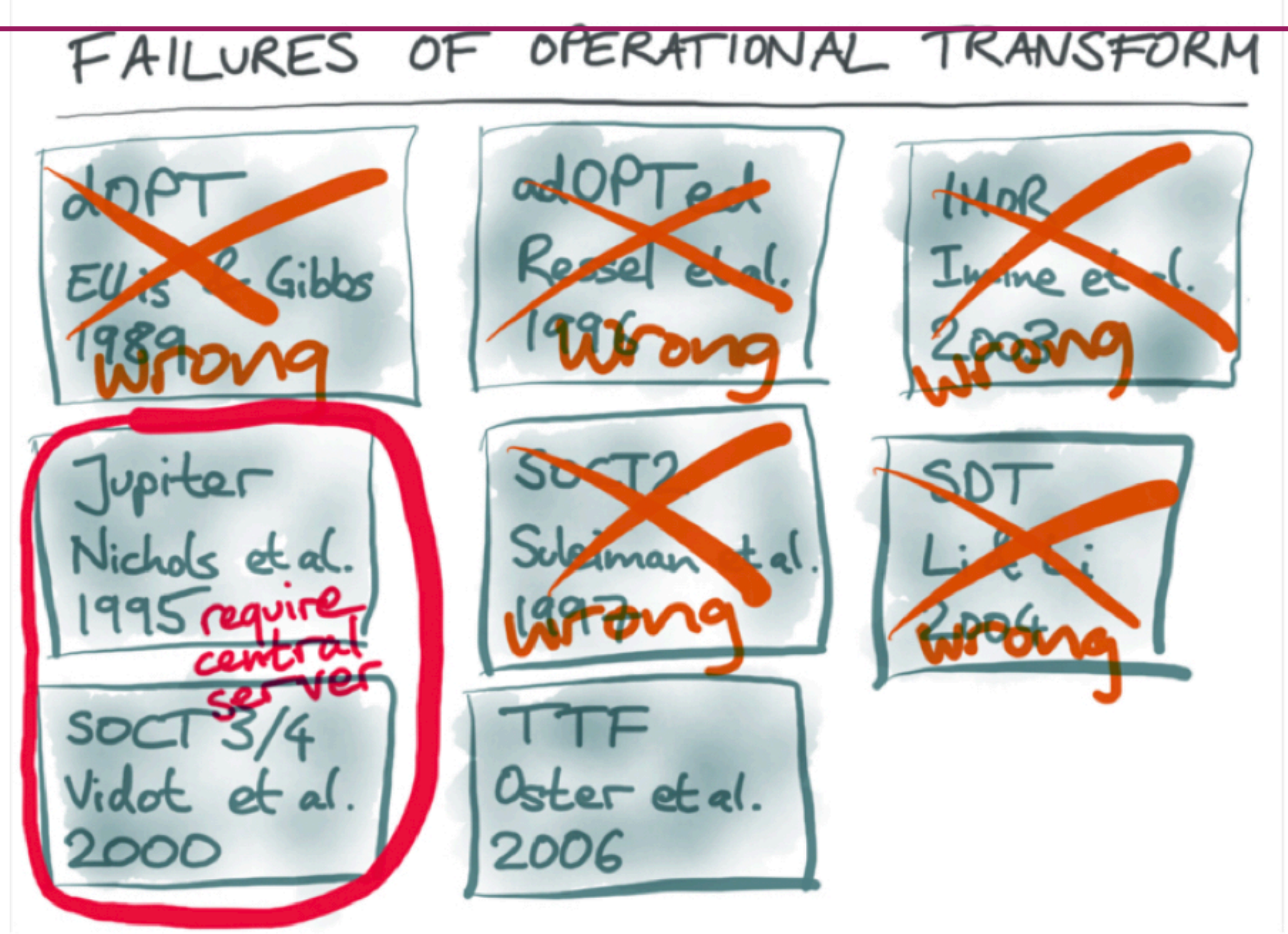
Rebirth of Operational Transformation in Replicated Data Sharing

- OT, originally proposed in 1989, has been kept away from replicated data sharing since most of algorithms were proved wrong.
- In OT-based replication,

① Updating process is broken into a patch of operations transmitted between sites.

② In each site, incoming operations are transformed to get the local operations to be performed since the baseline (last common state).

③ They are then applied locally to get the new baseline.



Martin Kleppmann, CRDTs and the Quest for Distributed Consistency

A talk at QCon London, London, UK, 05 Mar 2018

https://martin.kleppmann.com/2018/03/05/qcon-london.html

- For complex RTCE (Realtime Collaborative Editor) operations, transformation have edge cases difficult to ensure producing the confluent baseline. However, it is not difficult in carefully selected operations with the assumption of collaboration.
- CCR (Coordination-free Collaborative Replication) is a challenge against the trends.

A New Approach to Collaborative Replication

Masato Takeichi. “Coordination-free Collaborative Replication based on Operational Transformation”, September 16, 2024, Last Revised December 15 (version 3) arXiv:2409.09934v3. https://doi.org/10.48550/arXiv.2409.09934

Abstract. We introduce Coordination-free Collaborative Replication (CCR), a new method for maintaining consistency across replicas in distributed systems without requiring explicit coordination messages. CCR automates conflict resolution, contrasting with traditional data sharing systems that typically involve centralized update management or predefined consistency rules.

Conflict-free Replicated Data Type (CRDT), like Two-Phase Sets (2P-Sets), guarantees eventual consistency by allowing commutative and associative operations but often result in counterintuitive behaviors, such as failing to re-add an item to a shopping cart once removed.

In contrast, CCR employs a more intuitive approach to replication. It allows for straightforward updates and conflict resolution based on the current data state, enhancing clarity and usability compared to CRDTs. Furthermore, CCR addresses inefficiencies in messaging by developing a versatile protocol based on data stream confluence, thus providing a more efficient and practical solution for collaborative data sharing in distributed systems.

CCRAgent for ESET_String : Adding and Removing Strings to/from Set

```
type ElemType = String
type ReplicaData = [ElemType]

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision index
  , replicaDelay :: Int  -- Delay d*1000000 sec.
  , replicaVerbose :: Bool
  }
initReplica = Replica
  { replicaData = []
  , replicaPatch = []
  , replicaRev = 0
  , replicaDelay = 0
  , replicaVerbose = False
  }
```

"effectful" means that the operation effectively updates the state

Updating operations are Add, Rem and None

```
data Op
  = Add ReplicaID ElemType
  | Rem ReplicaID ElemType
  | None
  deriving (Eq)
```

non-effectful update None for "no-op"

```
effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op@(Add _ x) d =
  if List.elem x d then None else op
effectfulOp op@(Rem _ x) d =
  if List.elem x d then op else None
effectfulOp None d = None

transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  let p' = effectfulOp p d
      q' = effectfulOp q d
  in
    if p'==q' then (None, None) else (p’,q')
```

OT produces additional ops p' and q' to be performed

```
takeichi@Bowmore ESET_String % ./ESET_String 9000

* Agent Started on Port 9000
Listening on http://127.0.0.1:8000
* CCRAgent Started for ESET_String (2025/01/13)
...
#1: Add "X"
#2: Add "Y"
#3: Show
Replica {
 Data= {"X","Y"}
 Revs= 2
 UpdLog= [Add $0 "X",Add $0 “Y"]
 Delay= 0
 Verbose= False}
Connections= []
#4: Rem "X"
#5: Show
Replica {
 Data= {"Y"}
 Revs= 3
 UpdLog= [Add $0 "X",Add $0 "Y",Rem $0 "X"]
 Delay= 0
 Verbose= False}
Connections= []
#6: Rem "W"
#7: Show
Replica {
 Data= {"Y"}
 Revs= 3
 UpdLog= [Add $0 "X",Add $0 "Y",Rem $0 "X"]
 Delay= 0
 Verbose= False}
Connections= []
```

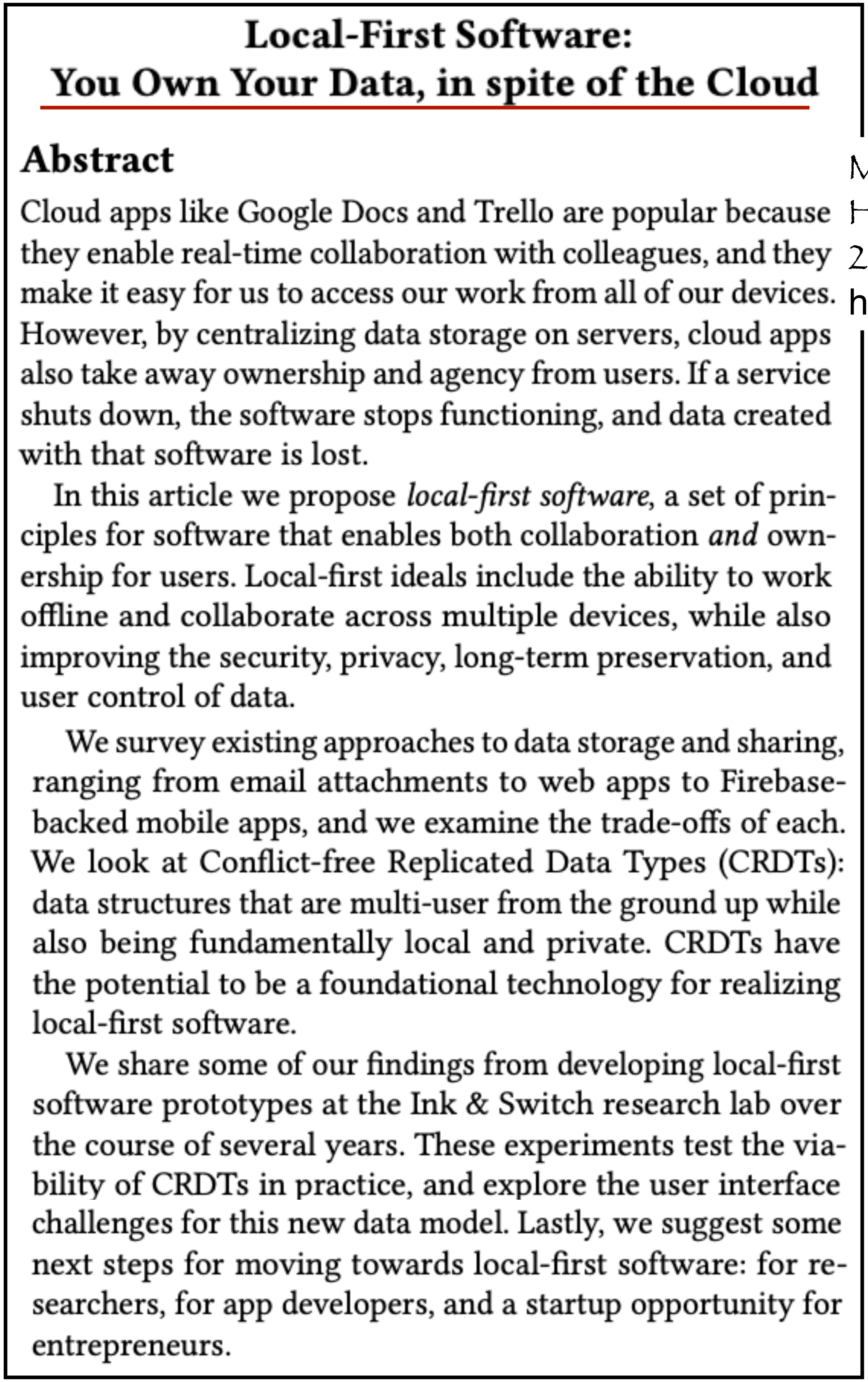
Add "Y" after Add "X" is effectful

Rem "X" is effectful

"X" has been removed

Rem "W" is non-effectful

Data Sharing in “Local-First” Software



Local-First Software:
You Own Your Data, in spite of the Cloud

Abstract

Cloud apps like Google Docs and Trello are popular because they enable real-time collaboration with colleagues, and they make it easy for us to access our work from all of our devices. However, by centralizing data storage on servers, cloud apps also take away ownership and agency from users. If a service shuts down, the software stops functioning, and data created with that software is lost.

In this article we propose *local-first software*, a set of principles for software that enables both collaboration *and* ownership for users. Local-first ideals include the ability to work offline and collaborate across multiple devices, while also improving the security, privacy, long-term preservation, and user control of data.

We survey existing approaches to data storage and sharing, ranging from email attachments to web apps to Firebase-backed mobile apps, and we examine the trade-offs of each. We look at Conflict-free Replicated Data Types (CRDTs): data structures that are multi-user from the ground up while also being fundamentally local and private. CRDTs have the potential to be a foundational technology for realizing local-first software.

We share some of our findings from developing local-first software prototypes at the Ink & Switch research lab over the course of several years. These experiments test the viability of CRDTs in practice, and explore the user interface challenges for this new data model. Lastly, we suggest some next steps for moving towards local-first software: for researchers, for app developers, and a startup opportunity for entrepreneurs.

Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Proc. 2019 ACM Onward’19.
https://doi.org/10.1145/3359591.3359737

Development of Local-First Data Sharing

Christian Kuessner, et.al., “Algebraic Replicated Data Types: Programming Secure Local-First Software”. ACM ECOOP 2023.
https://doi.org/10.1145/3359591.3359737

Development of Local-First Data Sharing
- Designing the application state with replication-awareness
- Efficient messaging in given target network topology
- Security of exchanged data

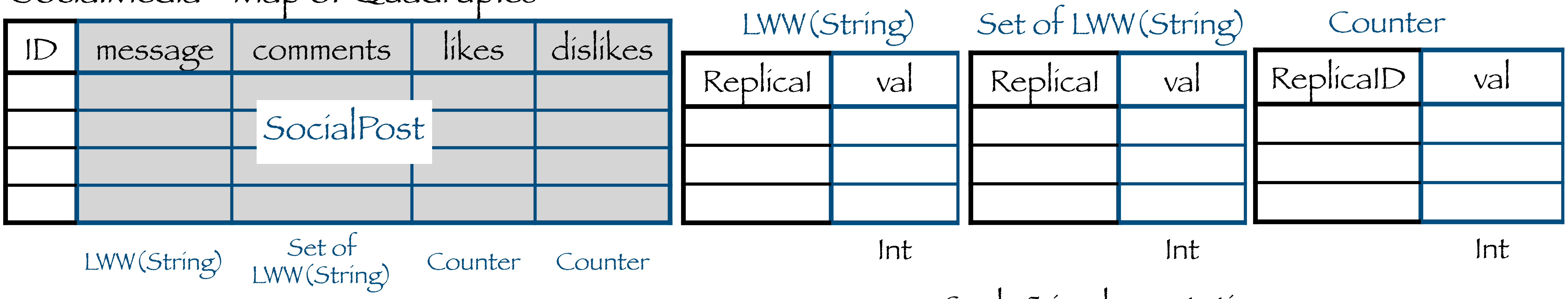
More Realistic Example follows …

More Realistic Example: Local-First Social Media Application with CRDT

Used by a group of friends in a peer-to-peer network to share messages, comments, likes, and dislikes.

Christian Kuessner, et.al., "Algebraic Replicated Data Types: Programming Secure Local-First Software". ACM ECOOP 2023.
https://doi.org/10.1145/3359591.3359737

SocialMedia - Map of Quadruples



ID	message	comments	likes	dislikes
		SocialPost		
	LWW(String)	Set of LWW(String)	Counter	Counter

LWW(String)

ReplicaI	val
	Int

Set of LWW(String)

ReplicaI	val
	Int

Counter

ReplicaID	val
	Int

Scala 3 implementation

```scala
case class SocialMedia(sm: Map[ID, SocialPost]):
  def like(post: ID, replica: ReplicaID): SocialMedia =
      val increment = sm(post).likes.inc(replica)
      SocialMedia(Map(post -> SocialPost(likes = increment)))

case class SocialPost(message: LWW[String], comments:
    Set[LWW[String]], likes: Counter, dislikes: Counter)
```

```scala
case class Counter(c: Map[ReplicaID, Int]):
  def value: Int = c.values.sum
  def inc(id: ReplicaID): Counter =
    Counter(Map(id -> (c.getOrElse(id, 0) + 1))

object Counter: // object for static methods
  def zero: Counter = Counter(Map.empty)
```

In CRDT, along with these type definitions, building composite semi-lattice from component ones is required.

CCR implementation shown later

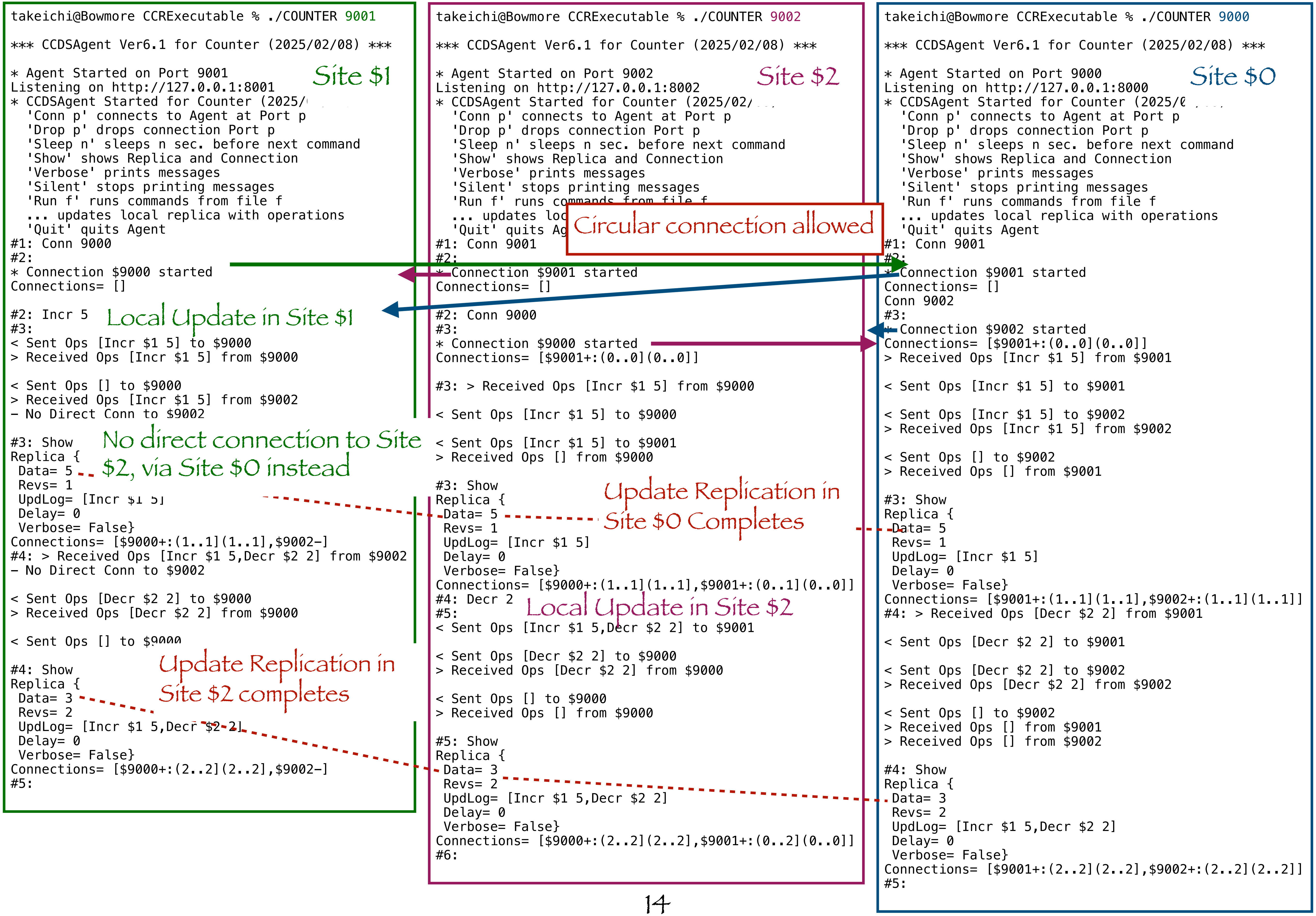
Comparison of CRDT and CCR for Replication

	CRDT Conflict-free Replicated Data Type	CCR Coordination-free Collaborative Replication
Replica Data Representation	Ordered Set (Semi-lattice) indirect from the Data Value	Any Data Type
Updating Operation	Monotonic Operations only	Any Operations
Query for the Value of the Replica	Monotonic Query	Straight Query for the Data Value
Conflict-free Confluence by …	Pre-defined **Merge** to get least upper bounds (LUB) of Semi-lattice	Operational Transformation (OT) with TP-1 Compositional Property
Eventual Consistency by …	Convergence over Semi-lattice	TP2-Confluence with Idempotence, Associativity and commutativity
Additional data for Replica	Metadata required for reasoning about causal relations on the Representation	No metadata required
Coordination-free Asynchronous messaging by …	CvRDT (State-based): Replica Data CmRDT (Operation-based): Reliable Causal Broadcast (RCB) for commutative and just-once messaging	Exchange operation sequences (patch) since the last common replication in any order; allows duplicated messaging and over circular networks
Structured Data	Build structured semi-lattices from components' semi-lattice	As structured algebraic data type with definition of OT using components' OTs

Implementation of CCRAgent
for
Coordination-free Collaborative Data Sharing

⇓

Running CCRAgent in 3 Sites with Network Connections



Site $1

```
takeichi@Bowmore CCRExecutable % ./COUNTER 9001

*** CCDSAgent Ver6.1 for Counter (2025/02/08) ***

* Agent Started on Port 9001
Listening on http://127.0.0.1:8001
* CCDSAgent Started for Counter (2025/
  'Conn p' connects to Agent at Port p
  'Drop p' drops connection Port p
  'Sleep n' sleeps n sec. before next command
  'Show' shows Replica and Connection
  'Verbose' prints messages
  'Silent' stops printing messages
  'Run f' runs commands from file f
  ... updates local replica with operations
  'Quit' quits Agent
#1: Conn 9000
#2:
* Connection $9000 started
Connections= []

#2: Incr 5
#3:
< Sent Ops [Incr $1 5] to $9000
> Received Ops [Incr $1 5] from $9000

< Sent Ops [] to $9000
> Received Ops [Incr $1 5] from $9002
- No Direct Conn to $9002

#3: Show
Replica {
 Data= 5
 Revs= 1
 UpdLog= [Incr $1 5]
 Delay= 0
 Verbose= False}
Connections= [$9000+:(1..1](1..1],$9002-]
#4: > Received Ops [Incr $1 5,Decr $2 2] from $9002
- No Direct Conn to $9002

< Sent Ops [Decr $2 2] to $9000
> Received Ops [Decr $2 2] from $9000

< Sent Ops [] to $9000

#4: Show
Replica {
 Data= 3
 Revs= 2
 UpdLog= [Incr $1 5,Decr $2 2]
 Delay= 0
 Verbose= False}
Connections= [$9000+:(2..2](2..2],$9002-]
#5:
```

Local Update in Site $1

No direct connection to Site $2, via Site $0 instead

Update Replication in Site $2 completes

Site $2

```
takeichi@Bowmore CCRExecutable % ./COUNTER 9002

*** CCDSAgent Ver6.1 for Counter (2025/02/08) ***

* Agent Started on Port 9002
Listening on http://127.0.0.1:8002
* CCDSAgent Started for Counter (2025/02/
  'Conn p' connects to Agent at Port p
  'Drop p' drops connection Port p
  'Sleep n' sleeps n sec. before next command
  'Show' shows Replica and Connection
  'Verbose' prints messages
  'Silent' stops printing messages
  'Run f' runs commands from file f
  ... updates loc
  'Quit' quits Ag
#1: Conn 9001
#2:
* Connection $9001 started
Connections= []

#2: Conn 9000
#3:
* Connection $9000 started
Connections= [$9001+:(0..0](0..0]]

#3: > Received Ops [Incr $1 5] from $9000

< Sent Ops [Incr $1 5] to $9000

< Sent Ops [Incr $1 5] to $9001
> Received Ops [] from $9000

#3: Show
Replica {
 Data= 5
 Revs= 1
 UpdLog= [Incr $1 5]
 Delay= 0
 Verbose= False}
Connections= [$9000+:(1..1](1..1],$9001+:(0..1](0..0]]
#4: Decr 2
#5:
< Sent Ops [Incr $1 5,Decr $2 2] to $9001

< Sent Ops [Decr $2 2] to $9000
> Received Ops [Decr $2 2] from $9000

< Sent Ops [] to $9000
> Received Ops [] from $9000

#5: Show
Replica {
 Data= 3
 Revs= 2
 UpdLog= [Incr $1 5,Decr $2 2]
 Delay= 0
 Verbose= False}
Connections= [$9000+:(2..2](2..2],$9001+:(0..2](0..0]]
#6:
```

Circular connection allowed

Update Replication in Site $0 Completes

Local Update in Site $2

Site $0

```
takeichi@Bowmore CCRExecutable % ./COUNTER 9000

*** CCDSAgent Ver6.1 for Counter (2025/02/08) ***

* Agent Started on Port 9000
Listening on http://127.0.0.1:8000
* CCDSAgent Started for Counter (2025/0
  'Conn p' connects to Agent at Port p
  'Drop p' drops connection Port p
  'Sleep n' sleeps n sec. before next command
  'Show' shows Replica and Connection
  'Verbose' prints messages
  'Silent' stops printing messages
  'Run f' runs commands from file f
  ... updates local replica with operations
  'Quit' quits Agent
#1: Conn 9001
#2:
* Connection $9001 started
Connections= []
Conn 9002
#3:
* Connection $9002 started
Connections= [$9001+:(0..0](0..0]]
> Received Ops [Incr $1 5] from $9001

< Sent Ops [Incr $1 5] to $9001

< Sent Ops [Incr $1 5] to $9002
> Received Ops [Incr $1 5] from $9002

< Sent Ops [] to $9002
> Received Ops [] from $9001

#3: Show
Replica {
 Data= 5
 Revs= 1
 UpdLog= [Incr $1 5]
 Delay= 0
 Verbose= False}
Connections= [$9001+:(1..1](1..1],$9002+:(1..1](1..1]]
#4: > Received Ops [Decr $2 2] from $9001

< Sent Ops [Decr $2 2] to $9001

< Sent Ops [Decr $2 2] to $9002
> Received Ops [Decr $2 2] from $9002

< Sent Ops [] to $9002
> Received Ops [] from $9001
> Received Ops [] from $9002

#4: Show
Replica {
 Data= 3
 Revs= 2
 UpdLog= [Incr $1 5,Decr $2 2]
 Delay= 0
 Verbose= False}
Connections= [$9001+:(2..2](2..2],$9002+:(2..2](2..2]]
#5:
```

CCR Implementation of Realtime Collaborative Editor (1)

```
type ReplicaData = String

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision Index
  , replicaDelay :: Int  -- Delay d*1000000 sec.
  , replicaVerbose :: Bool
  }
initReplica = Replica
  { replicaData = ""
  , replicaPatch = []
  , replicaRev = 0
  , replicaDelay = 0
  , replicaVerbose = False
  }

data Op
  = Ins ReplicaID Int String
  | Del ReplicaID Int Int
  | None
  deriving (Eq)

effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op@(Ins _ k s) d =
  if (k>=0)&&(k<=length d) then op else None
effectfulOp op@(Del _ k n) d =
  if (k>=0)&&(k+n<=length d) then op else None
effectfulOp None d = None

transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  let p'= effectfulOp p d
      q'= effectfulOp q d
  in
    if p'==q' then (None, None)
    else trans' p' q'
```

Updating Op: **Ins** (Insert) and **Del** (Delete)

Specify position and text

Specify position and number of chars

```
takeichi@Bowmore RTCE % ./RTCE 9000

*** CCDSAgent Ver6.1 for RTCE Text Editing (2024/10/07) ***

* Agent Started on Port 9000
Listening on http://127.0.0.1:8000
* CCDSAgent Started for RTCE Text Editing (2024/10/07)
  'Conn p' connects to Agent at Port p
  'Drop p' drops connection Port p
  'Sleep n' sleeps n sec. before next command
  'Show' shows Replica and Connection
  'Verbose' prints messages
  'Silent' stops printing messages
  'Run f' runs commands from file f
  ... updates local replica with operations
  'Quit' quits Agent
#1: Ins 0 "XY", Ins 1 "AB"
#2: Show
Replica {
 Data= "XABY"
 Revs= 2
 UpdLog= [Ins $0 0 "XY",Ins $0 1 "AB"]
 Delay= 0
 Verbose= False}
Connections= []
#3: Del 2 2
#4: Show
Replica {
 Data= "XA"
 Revs= 3
 UpdLog= [Ins $0 0 "XY",Ins $0 1 "AB",Del $0 2 2]
 Delay= 0
 Verbose= False}
Connections= []
```

Successive insertions

Delete 2 chars from position 2

CCR Implementation of Realtime Collaborative Editor (2)

```
trans' p@(Ins i_p k_p t_p) q@(Ins i_q k_q t_q) =
 if p==q then (None,None)
 else
  if k_p==k_q then
   if t_p==t_q then
    if i_p>i_q then
     (Ins i_p (k_p+length t_q) t_p, Ins i_q k_q t_q)
    else
     (Ins i_p k_p t_p, Ins i_q (k_q+length t_p) t_q)
   else
    if t_p>t_q then
     (Ins i_p (k_p+length t_q) t_p, Ins i_q k_q t_q)
    else
     (Ins i_p k_p t_p, Ins i_q (k_q+length t_p) t_q)
  else
   if (k_p > k_q)
    then
     (Ins i_p (k_p+length t_q) t_p, Ins i_q k_q t_q)
    else
     (Ins i_p k_p t_p, Ins i_q (k_q+length t_p) t_q)
trans' p@(Del i_p k_p n_p) q@(Del i_q k_q n_q)
 | k_p==k_q && n_p==n_q
 = (None,None)
 | k_q >= k_p+n_p
 = (Del i_p k_p n_p, Del i_q (k_q-n_p) n_q)
 | k_p >= k_q+n_q
 = (Del i_p (k_p-n_q) n_p, Del i_q k_q n_q)
 | k_p >= k_q && k_p+n_p <= k_q+n_q
 = (Del i_p k_q 0, Del i_q k_q (n_q-n_p))
 | k_q >= k_p && k_q+n_q <= k_p+n_p
 = (Del i_p k_p (n_p-n_q), Del i_q k_p 0)
 | k_p >= k_q
 = let d = k_q+n_q-k_p
  in (Del i_p k_q (n_p-d), Del i_q k_q (n_q-d))
 | otherwise
 = let d = k_p+n_p-k_q
  in (Del i_p k_p (n_p-d), Del i_q k_p (n_q-d))
```

- OT for RTCE is rather complex and tedious to proof that this has the TP1 and TP2 with compositional and confluence properties.
- Currently, the proof has not yet been done!

  Instead, it has been checked by execution to confirm no exceptions reported through more than 1000 randomly generated operations.

```
trans' (Ins i_p k_p t_p) (Del i_q k_q n_q)
 | k_p >= k_q && k_p < k_q+n_q
 = (Ins i_p k_q "", Del i_q k_q (n_q+length t_p))
 | k_p < k_q
 = (Ins i_p k_p t_p, Del i_q (k_q+length t_p) n_q)
 | otherwise
 = (Ins i_p (k_p-n_q) t_p, Del i_q k_q n_q)
trans' d@Del{} i@Ins{} =
 let (p',q') = trans' i d in (q',p')
trans' None q = (None,q)
trans' p None = (p,None)
```

CCR Implementation of Composite Structured Replica - QUAD for Social Media

```haskell
import qualified ReplicaLWW_String  -- message
import qualified ReplicaESET_String -- comments
import qualified ReplicaCOUNTER   -- likes and dislikes

data ReplicaData =
  Quad ReplicaLWW_String.ReplicaData  -- message Replica
       ReplicaESET_String.ReplicaData  -- comments Replica
       ReplicaCOUNTER.ReplicaData  -- likes Replica
       ReplicaCOUNTER.ReplicaData  -- dislikes Replica
initReplica = Replica
  { replicaData =
      Quad (ReplicaLWW_String.replicaData
              ReplicaLWW_String.initReplica)
             (ReplicaESET_String.replicaData
              ReplicaESET_String.initReplica)
             (ReplicaCOUNTER.replicaData
              ReplicaCOUNTER.initReplica)
             (ReplicaCOUNTER.replicaData
              ReplicaCOUNTER.
  , replicaPatch = []
  , replicaRev = 0
  , replicaDelay = 0
  , replicaVerbose = False
  }

data Op
  = QuadOp ReplicaID
      ReplicaLWW_String.Patch
      ReplicaESET_String.Patch
      ReplicaCOUNTER.Patch
      ReplicaCOUNTER.Patch
  | None
  deriving (Eq)
```

Replica QUAD composed of 4 primitive of which Replica definitions imported

```haskell
transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp (Quad message comments likes dislikes) p q =
  if p==q then (None, None) else
  if p==None then (None, q) else
  if q==None then (p, None) else
    let (QuadOp replicaID_p ms_p cs_p ls_p ds_p) = p
        (QuadOp replicaID_q ms_q cs_q ls_q ds_q) = q
        (ms_p',ms_q') =
          ReplicaLWW_String.transPatch message ms_p ms_q
        (cs_p',cs_q') =
          ReplicaESET_String.transPatch comments cs_p cs_q
        (ls_p',ls_q') =
          ReplicaCOUNTER.transPatch likes ls_p ls_q
        (ds_p',ds_q') =
          ReplicaCOUNTER.transPatch dislikes ds_p ds_q
    in (QuadOp replicaID_p ms_p' cs_p' ls_p' ds_p',
        QuadOp replicaID_q ms_q' cs_q' ls_q' ds_q')
```

OT transOp transforms ops using OT for component CCRs

```
takeichi@Bowmore QUAD % ./QUAD 9000
*** CCDSAgent Ver6.3 for Social Post Quadruples (2024/08/18) ***
#1: (Write "XYZ";Add "AB";Incr 6;Decr 2)
#2: Show
Replica {
 Data= ({"XYZ"},{"AB"},6,-2)
 Revs= 1
 UpdLog= [QuadOp $0 [Write $0 {"XYZ"}] [Add $0 "AB"] [Incr $0 6] [Decr $0 2]]
 Delay= 0
 Verbose= False}
Connections= []
```

Operations given component-wise in parentheses delimited with ';'.

Display current Replica in short format

CCR Implementation of Local-First Social Media Application (1)

- MAP_QUAD maps Int keys to quadruples values
- Quadruple values consist of four component values of Replica Type representing Social Media information
- Updates of the elements of the Map are processed in component-wise of quadruples

```
type ElemType = ReplicaQUAD.ReplicaData
type KeyType = Int

type ReplicaData = IntMap.IntMap ElemType

data Replica = Replica
  { replicaData :: ReplicaData
  , replicaPatch :: Patch
  , replicaRev :: RevIndex  -- Revision Index
  , replicaDelay :: Int  -- Delay d*1000000 sec.
  , replicaVerbose :: Bool
  }
```

Updating Operations of the Map

```
data Op
  = Upd ReplicaID KeyType ReplicaQUAD.Op
  | Del ReplicaID KeyType
  | None
  deriving (Eq)

applyOp :: Op -> ReplicaData -> (ReplicaData,Op)
applyOp op@(Upd _ k quadOp) d =
  let v =
        case IntMap.lookup k d of
          Nothing ->
            ReplicaQUAD.replicaData
              ReplicaQUAD.initReplica
          Just v -> v
      (v',_) = ReplicaQUAD.applyOp quadOp v
  in (IntMap.insert k v' d, op)
applyOp op@(Del _ k) d =
  (IntMap.delete k d, op)
applyOp None d = (d,None)
```

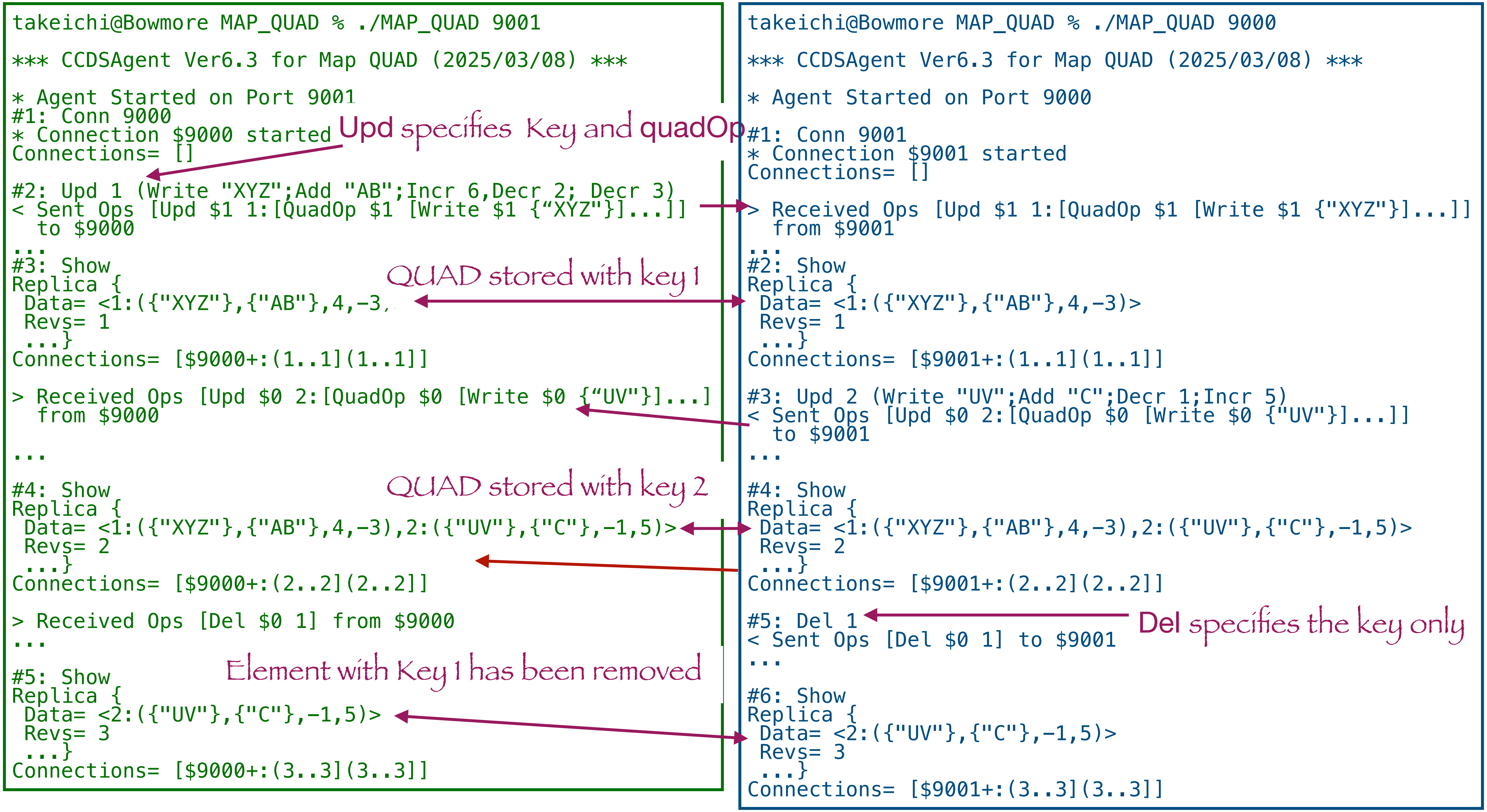
Initial QUAD value installed when no element with key found in the Map

When Upd and Del given with the same key, Del is preferred

```
effectfulOp :: Op -> ReplicaData -> Op
effectfulOp op@(Upd _ k quadOp) d = op
effectfulOp op@(Del _ k) d = op
effectfulOp None d = None
transOp :: ReplicaData -> Op -> Op -> (Op, Op)
transOp d p q =
  let p'= effectfulOp p d
      q'= effectfulOp q d
  in if p'==q' then (None, None)
     else trans' p' q'
  where
  trans' p@(Upd i_p k_p ops_p) q@(Upd i_q k_q ops_q)=
    if k_p == k_q then
          v = case IntMap.lookup k_p d of
                Nothing ->
                  ReplicaQUAD.replicaData
                    ReplicaQUAD.initReplica
                Just v -> v
          (ops_p',ops_q')=
            ReplicaQUAD.transPatch v ops_p ops_q
      in (Upd i_p k_p ops_p', Upd i_q k_q ops_q')
    else (p,q)
  trans' p@(Del i_p k_p) q@(Del i_q k_q) =
    if k_p == k_q then (None,None) else (p,q)
  -- Del is preferred to Upd
  trans' p@(Upd i_p k_p ops_p) q@(Del i_q k_q) =
    if k_p == k_q then (None,q) else (p,q)
  trans' p@(Del i_p k_p) q@(Upd i_q k_q ops_q) =
    if k_p == k_q then (p,None) else (p,q)
  trans' None q = (None,q)
  trans' p None = (p,None)
```

CCR Implementation of Local-First Social Media Application (2)

- Updating operations of the Map are **Upd** and **Del** :
  - **Upd** *k* **quadOp** updates the quadruple with the Map key *k* by **quadOp** for quadruples.
  - **Del** *k* deletes the quadruple with *k* as the Map key



```
takeichi@Bowmore MAP_QUAD % ./MAP_QUAD 9001

*** CCDSAgent Ver6.3 for Map QUAD (2025/03/08) ***

* Agent Started on Port 9001
#1: Conn 9000
* Connection $9000 started
Connections= []

#2: Upd 1 (Write "XYZ";Add "AB";Incr 6,Decr 2; Decr 3)
< Sent Ops [Upd $1 1:[QuadOp $1 [Write $1 {“XYZ”}]...]]
  to $9000
...
#3: Show
Replica {
 Data= <1:({"XYZ"},{"AB"},4,-3,
 Revs= 1
 ...}
Connections= [$9000+:(1..1](1..1]]

> Received Ops [Upd $0 2:[QuadOp $0 [Write $0 {“UV”}]...]
  from $9000

...

#4: Show
Replica {
 Data= <1:({"XYZ"},{"AB"},4,-3),2:({"UV"},{"C"},-1,5)>
 Revs= 2
 ...}
Connections= [$9000+:(2..2](2..2]]

> Received Ops [Del $0 1] from $9000
...

#5: Show
Replica {
 Data= <2:({"UV"},{"C"},-1,5)>
 Revs= 3
 ...}
Connections= [$9000+:(3..3](3..3]]
```

```
takeichi@Bowmore MAP_QUAD % ./MAP_QUAD 9000

*** CCDSAgent Ver6.3 for Map QUAD (2025/03/08) ***

* Agent Started on Port 9000

#1: Conn 9001
* Connection $9001 started
Connections= []

> Received Ops [Upd $1 1:[QuadOp $1 [Write $1 {"XYZ"}]...]]
  from $9001
...
#2: Show
Replica {
 Data= <1:({"XYZ"},{"AB"},4,-3)>
 Revs= 1
 ...}
Connections= [$9001+:(1..1](1..1]]

#3: Upd 2 (Write "UV";Add "C";Decr 1;Incr 5)
< Sent Ops [Upd $0 2:[QuadOp $0 [Write $0 {"UV"}]...]]
  to $9001
...

#4: Show
Replica {
 Data= <1:({"XYZ"},{"AB"},4,-3),2:({"UV"},{"C"},-1,5)>
 Revs= 2
 ...}
Connections= [$9001+:(2..2](2..2]]

#5: Del 1
< Sent Ops [Del $0 1] to $9001
...

#6: Show
Replica {
 Data= <2:({"UV"},{"C"},-1,5)>
 Revs= 3
 ...}
Connections= [$9001+:(3..3](3..3]]
```

Upd specifies Key and **quadOp**

QUAD stored with key 1

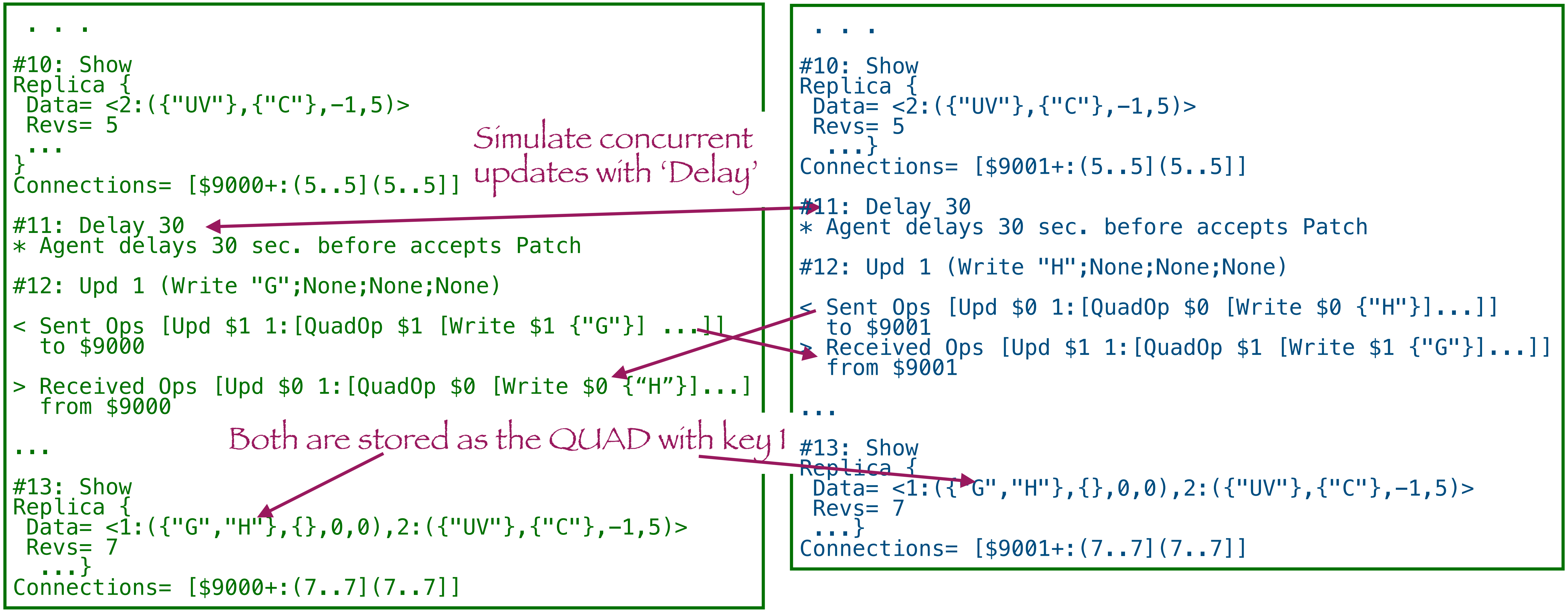
QUAD stored with key 2

Del specifies the key only

Element with Key 1 has been removed

CCR Implementation of Local-First Social Media Application (3)

- The first component of QUAD is of Replica type LWW_String , which keeps both values as set elements when which of the concurrent updates cannot be determined as the Last-Writer.



```
 . . .

#10: Show
Replica {
 Data= <2:({"UV"},{"C"},-1,5)>
 Revs= 5
 ...
}
Connections= [$9000+:(5..5](5..5]]

#11: Delay 30
* Agent delays 30 sec. before accepts Patch

#12: Upd 1 (Write "G";None;None;None)

< Sent Ops [Upd $1 1:[QuadOp $1 [Write $1 {"G"}] ...]]
  to $9000

> Received Ops [Upd $0 1:[QuadOp $0 [Write $0 {“H”}]...]
  from $9000

...

#13: Show
Replica {
 Data= <1:({"G","H"},{},0,0),2:({"UV"},{"C"},-1,5)>
 Revs= 7
  ...}
Connections= [$9000+:(7..7](7..7]]
```

```
 . . .

#10: Show
Replica {
 Data= <2:({"UV"},{"C"},-1,5)>
 Revs= 5
  ...}
Connections= [$9001+:(5..5](5..5]]

#11: Delay 30
* Agent delays 30 sec. before accepts Patch

#12: Upd 1 (Write "H";None;None;None)

< Sent Ops [Upd $0 1:[QuadOp $0 [Write $0 {"H"}]...]]
  to $9001
> Received Ops [Upd $1 1:[QuadOp $1 [Write $1 {"G"}]...]]
  from $9001

...

#13: Show
Replica {
 Data= <1:({"G","H"},{},0,0),2:({"UV"},{"C"},-1,5)>
 Revs= 7
  ...}
Connections= [$9001+:(7..7](7..7]]
```

Simulate concurrent updates with ‘Delay’

Both are stored as the QUAD with key 1

More to do for
Coordination-free Collaborative Data Sharing

- Proving TP1+TP2 Properties of OT and Reasoning about “Confluence”
- Putting forward the claim on “Monotonicity is not, but Confluence is”
- Developing Privacy-Preserving Local-First Software with Dejima Architecture

⇑

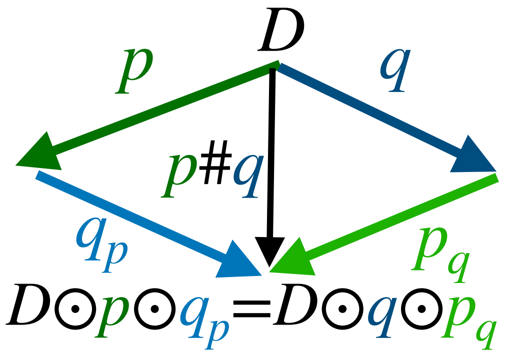
More to do for
Coordination-free Collaborative Data Sharing

Proving TP1+TP2 Properties of OT
and
Reasoning about "Confluence"

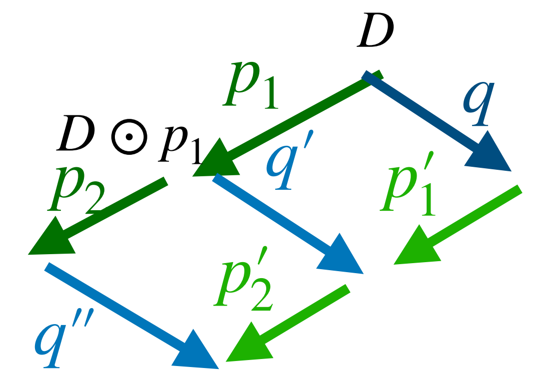
OT for Coordination-free Collaborative Replication

How to Replicate Collaboratively with Conflict resolution

=> Replicate local updates with conflict resolution



$(p_q, q_p) = T_D(p, q)$

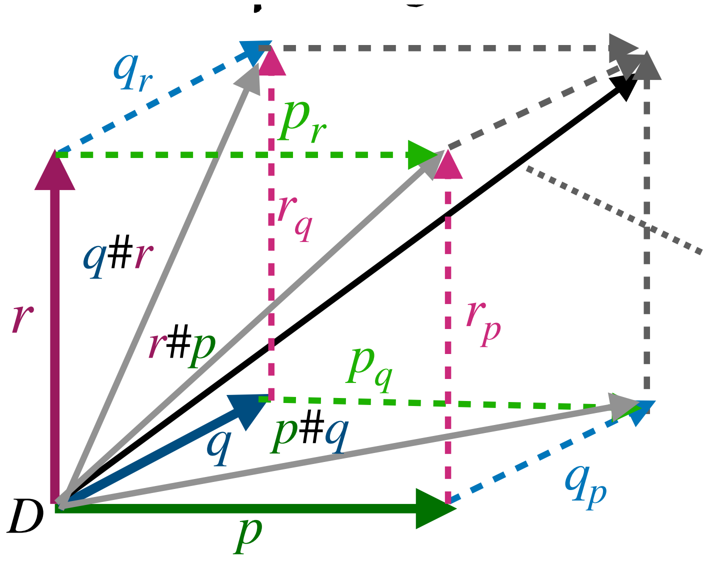


$(\,p'_1\odot p'_2,\ q''_1\,) = T_D(p_1\odot p_2,\ q_1\,)$

Operational Transformation T_D with Compositional TP1-Confluence defines collaborative operator $\#_D$

How to Replicate Collaboratively without Coordination

=> Replicate confluent updating operations for coordination avoidance



$$(p \# q) \# r =_D (q \# r) \# p =_D (r \# p) \# q$$

TP2-Confluence guarantees #'s Idempotence, Associativity and Commutativity

Building up Confluence with OT

$T_D :: O \times O \to O \times O$, $(p_q, q_p) = T_D(p, q)$ for p in site P and q in Q fulfills *TP1-Confluence* $p \odot q_p \leftrightarrow_D q \odot p_q$, written as $p \#_D q$, also as $p \# q$ when D is obvious.

- Idempotence When P and Q share p applied to D to get the confluent state $D \odot p$ and hence $p \#_D p = p$.
- Commutativity It is straightforward that $p \#_D q = q \#_D p$ for p and q on D.
- Associativity To establish the confluence property of T_D applied in two steps for updates on D by three sites, the relations should hold regardless of the application order:
  $(p \#_D q) \#_D r = (p \#_D r) \#_D q$, $(q \#_D r) \#_D p = (q \#_D p) \#_D r$, and $(r \#_D p) \#_D q = (r \#_D q) \#_D p$.

Algebraic Structure of Confluence Property

- **Identity element** $p \#_D ! = \ ! \#_D p =_D p$.
- **Idempotence** $p \#_D p = p$, which comes from the Minimal Property of T_D
- **Commutativity** $p \#_D q = q \#_D p$, which comes from the TP1-Property of T_D
- **Associativity** $(p \#_D q) \#_D r = p \#_D (q \#_D r)$, which is required for the TP2-Property of T_D

The Algebraic Structure of Confluence Property suggests us to put the TP2-Confluence Property into practical coordination-free replication by sending updated operations on the common replicated data in any order to others.

More to do for
Coordination-free Collaborative Data Sharing

Putting forward the claim on
"Monotonicity is not, but Confluence is"

Confluence for Coordination-freeness

- Monotonicity is not the only golden rule for coordination-free collaborative replication, while the CALM theorem lays stress on this as in

  J. M. Hellerstein and P. Alvaro. Keeping CALM: when distributed consistency is easy. Communications of the ACM, 63(9):72–81, 2020.

- The proof sketch states that the confluence property of the operation is a generalization of commutativity, that is, the order of its operands makes no difference to the result.
  - An operation is confluent if it produces the same outputs for any nondeterministic ordering of a set of inputs.
  - If the outputs of one confluent operation are consumed by another confluent operator as inputs, the resulting composite operation is confluent.
  - Hence, if we build programs by composing confluent operations, our programs are confluent by construction, despite orderings of messages or execution steps within and across distributed sites.
- CCR actually realizes the confluence property by implementing confluent operations with OT and therefore “CALM” should be superseded by “Consistency By Constructing Confluent Operations”.

Confluence rather than Monotonicity

- While coordination is a "killer" of performance in distributed systems, no coordination may suffer from the consistency of distributed data.
- The CALM (Consistency As Logical Monotonicity) theorem brings about a solution to the question

  > "What is the family of problems that can be consistently computed in a distributed fashion without coordination, and what lies outside that family?"

  as

  > "A program has a consistent, coordination-free distributed implementation if and only if it is monotonic."
- Since confluent operations are the basic constructs of monotonic systems, they can do more than that if collaboration is utilized for establishing confluence of components of distributed systems.
- Thus, we should shatter the CALM of monotonicity to open the door to claim that the same holds for programs composed of confluent operations like CCR.

Confluence Operation for Coordination-freeness

- The proof sketch states that the confluence property of the operation is a generalization of commutativity, that is, the order of its operands makes no difference to the result.
  - An operation is confluent if it produces the same outputs for any nondeterministic ordering of a set of inputs.
  - The CCR replication procedure is confluent since it produces the same replicated data for any nondeterministic ordering of a set of concurrent updates.
  - If the outputs of one confluent operation are consumed by another confluent operator as inputs, the resulting composite operation is confluent.
  - Hence, if we build programs by composing confluent operations, our programs are confluent by construction, despite orderings of messages or execution steps within and across distributed sites.
- Confluence of CCR also satisfies above properties and "CALM" should be superseded by "Consistency By Confluence".

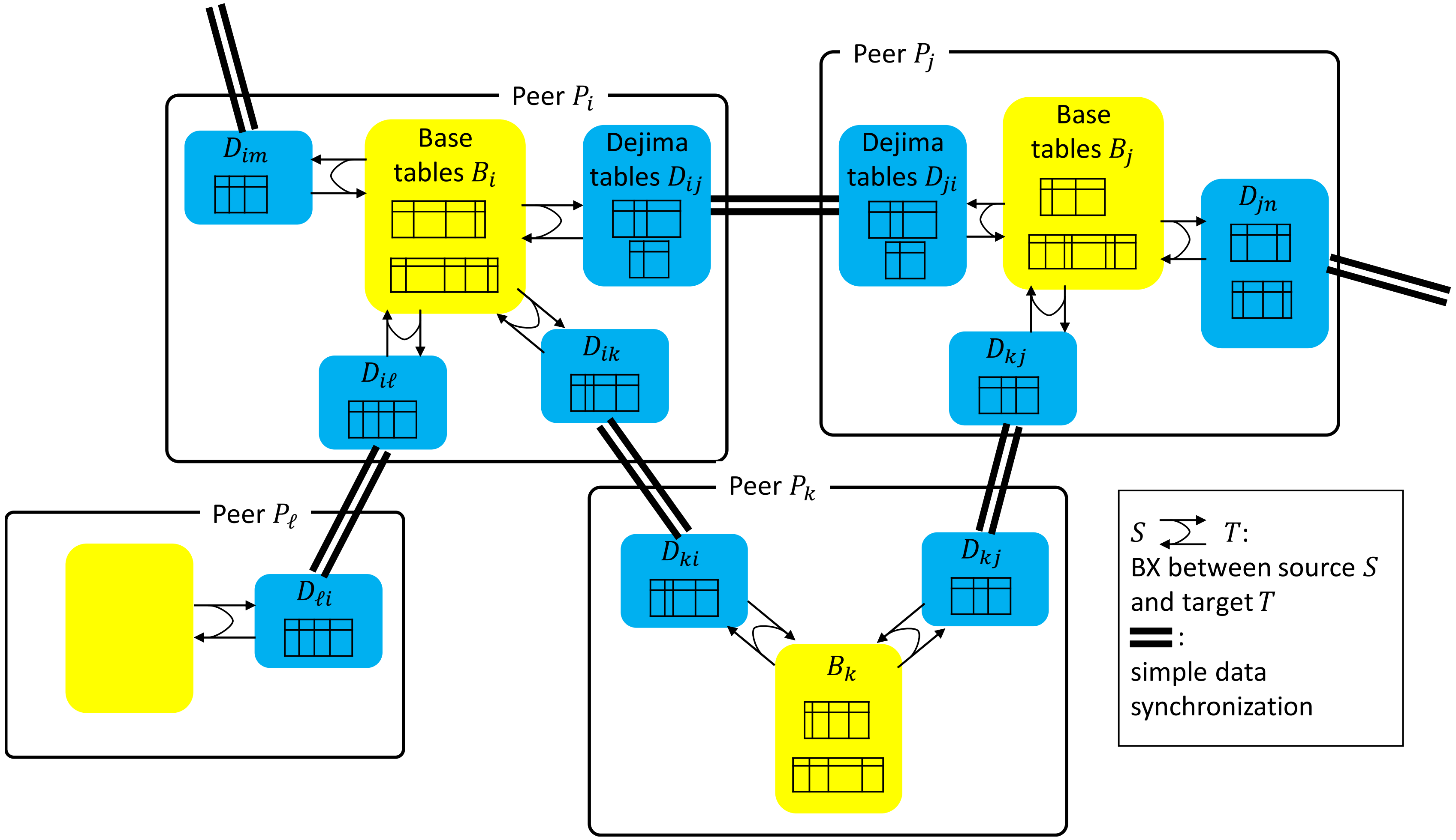
More to do for Coordination-free Collaborative Data Sharing

* * *

Developing Privacy-Preserving Local-First Software with Dejima Architecture

Coordination-free Collaborative Dejima Data Sharing for Privacy-Preserving Local-First Software (1)

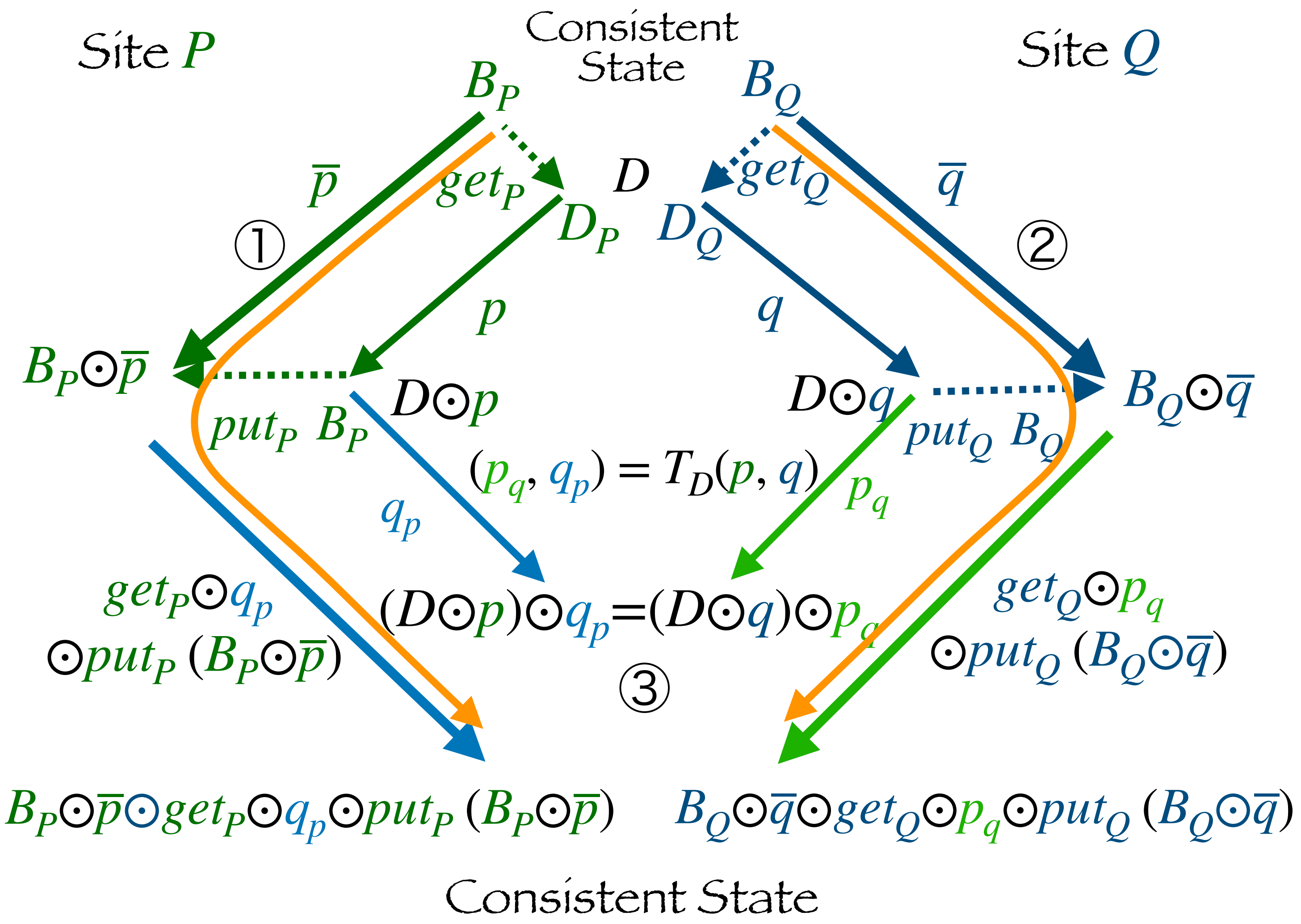
- Dejima Architecture manages selective P2P data sharing using Bidirectional Transformation between the local Base table and shared Dejima tables between distributed Peers.
- Combined with CCR, Dejima architecture strengthens privacy of the Local-First Software.



Ishihara, Y., Kato, H., Nakano, K., Onizuka, M., & Sasaki, Y. (2019). Toward BX-based architecture for controlling and sharing distributed data. In 2019 IEEE BigComp. https://doi.org/10.1109/BIGCOMP.2019.8679145

Coordination-free Collaborative Dejima Data Sharing for Privacy-Preserving Local-First Software (2)

- In Site P, updates $\overline{p}$ on the Base table B_P are transformed by the forward transformation get_P to get each Dejima table D_P that can be replicated with Dejima tables D_Q of other Sites Q using Coordination-free Collaborative Replication.
- The replicated Dejima D'_P is put back to the Base table as B'_P by the backward transformation put_P.


Site P

Consistent State

Site Q

B_P

B_Q

$\overline{p}$

get_P

D

get_Q

$\overline{q}$

①

D_P

D_Q

②

p

q

$B_P \odot \overline{p}$

$D \odot p$

$D \odot q$

$B_Q \odot \overline{q}$

$put_P\ B_P$

$put_Q\ B_Q$

$(p_q, q_p) = T_D(p, q)$

q_p

p_q

$get_P \odot q_p$

$\odot put_P\ (B_P \odot \overline{p})$

$(D \odot p) \odot q_p = (D \odot q) \odot p_q$

$get_Q \odot p_q$

$\odot put_Q\ (B_Q \odot \overline{q})$

③

$B_P \odot \overline{p} \odot get_P \odot q_p \odot put_P\ (B_P \odot \overline{p})$

$B_Q \odot \overline{q} \odot get_Q \odot p_q \odot put_Q\ (B_Q \odot \overline{q})$

Consistent State

Coordination-free Collaborative Dejima Data Sharing