www.loria.fr

Laboratoire lorrain de recherche en informatique et ses applications

Data Replication and Consistency
Data Consistency

Operational Transformation Approaches
Conflict-free Replicated Data Structures

Gérald Oster,
Associate Professor, TELECOM Nancy, Université de Lorraine
COAST Team, LORIA, Université de Lorraine, Inria, CNRS

TELECOM Nancy — Ingénieurs du numérique • Inspiring your digital future
École affiliée IMT

CAP Theorem [GL02]

- A system without network partition, can achieve consistency + availability
  - Client and storage system are part of the same environment
- In a large-scale distributed systems, network partitions exist

- Consistency and availability cannot be both achieved
  - Relax consistency, maintain availability
  - Maintain consistency, tolerate unavailability under certain conditions

Loria

Optimistic Replication

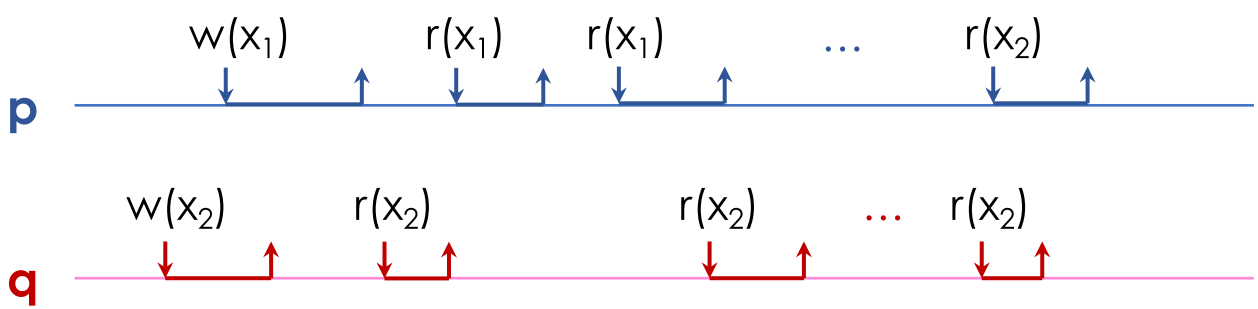
- To sidestep CAP theorem [GL02], a first solution is to avoid synchronizing replicas. Replicas simply reconcile their copies in the background.

- Trade-off between consistency and availability
  - allows replicas to diverge
- This approach is named lazy [LLS90], or optimistic replication [SS05]

- Examples: Bayou, Amazon S3, etc.

Consistency in Distributed Systems
Eventual Consistency

- **Definition** (*eventual consistency*)
  A history h is eventually consistent (EC) when for every object x if there is a bounded amount of write operations on x in h, then eventually all the read operation observer the same state.



Optimistic Replication
Strong Eventual Consistency

- Strong Eventual Consistency
  - Eventual delivery: « *An update executed at some correct replica eventually executes at all correct replicas* »
  - Strong convergence = correct replicas that have executed the same updates **have** equivalent state
  - No consensus in background, no need to rollback

Operational Transformation (OT)
Operation-based approach model

- *n* copies of an object hosted at *n* sites
- An object is modified by applying operations

- Each operation is
  - generated at a site (local execution), and applied immediately on the local copy
  - broadcasted to other sites
  - integrated at those sites (remote execution)

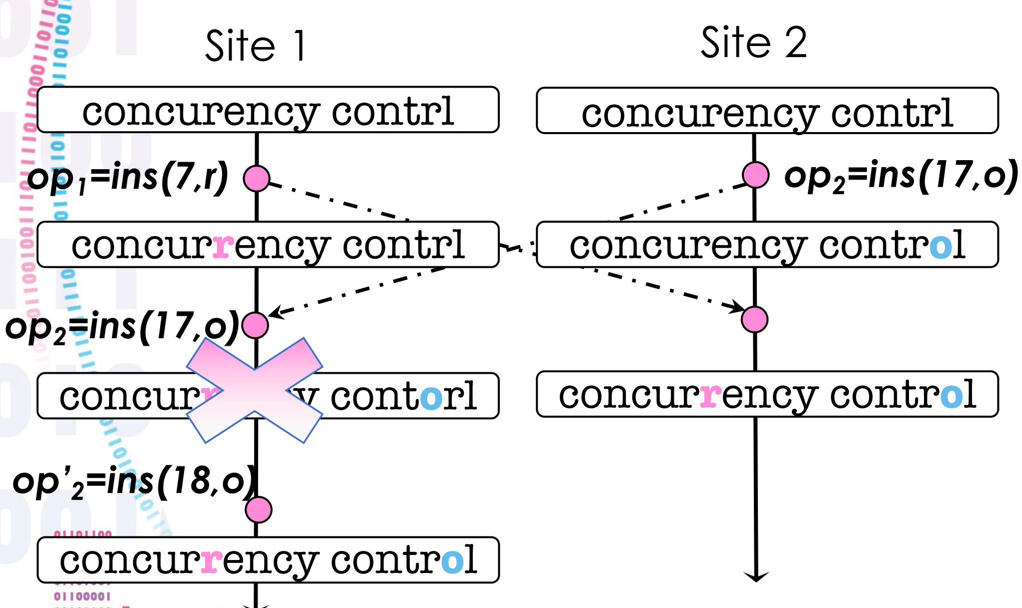
- System is correct if when it is idle all copies are identical (SEC)

Operational Transformation [EG89]
General Architecture

- 2 components:
  - An integration algorithm (diffusion, integration)
  - A set of transformation functions (conflict resolution)
- 3 main issues:
  - Convergence (~ EC)
  - Causality violation
  - (Intention violation)

Operational Transformation
Running Example

- Textual document = sequence of characters
- Operations:
  - Ins(p, c) – inserts character 'c' at position 'p'
  - Del(p) – removes character at position 'p'



T(Ins(p1,c1), Ins(p2,c2)) :-
if (p1<p2) **return** Ins(p1,c1)
else return Ins(p1+1,c1)
endif

Operational Transformation
Example of Transformation Functions

T(Ins(p1,c1), Ins(p2,c2)) :-
    if (p1<p2) **return** *Ins(p1,c1)*
    else return *Ins(p1+1,c1)*

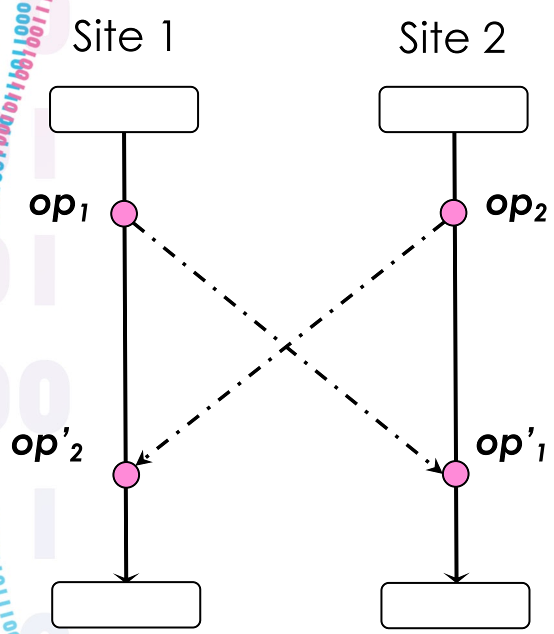
T(Ins(p1,c1), Del(p2)) :-
    if (p1≤p2) **return** *Ins(p1,c1)*
    else return *Ins(p1-1,c1)*
    endif

T(Del(p1), Ins(p2,c2)) :-
    if (p1<p2) **return** *Del(p1)*
    else return *Del(p1+1)*

T(Del(p1), Del(p2)) :-
    if (p1<p2) **return** *Del(p1)*
    else if (p1>p2) **return** *Del(p1-1)*
    else return *Id()*

Operational Transformation
Correctness [EG89]

(TP1) $op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$



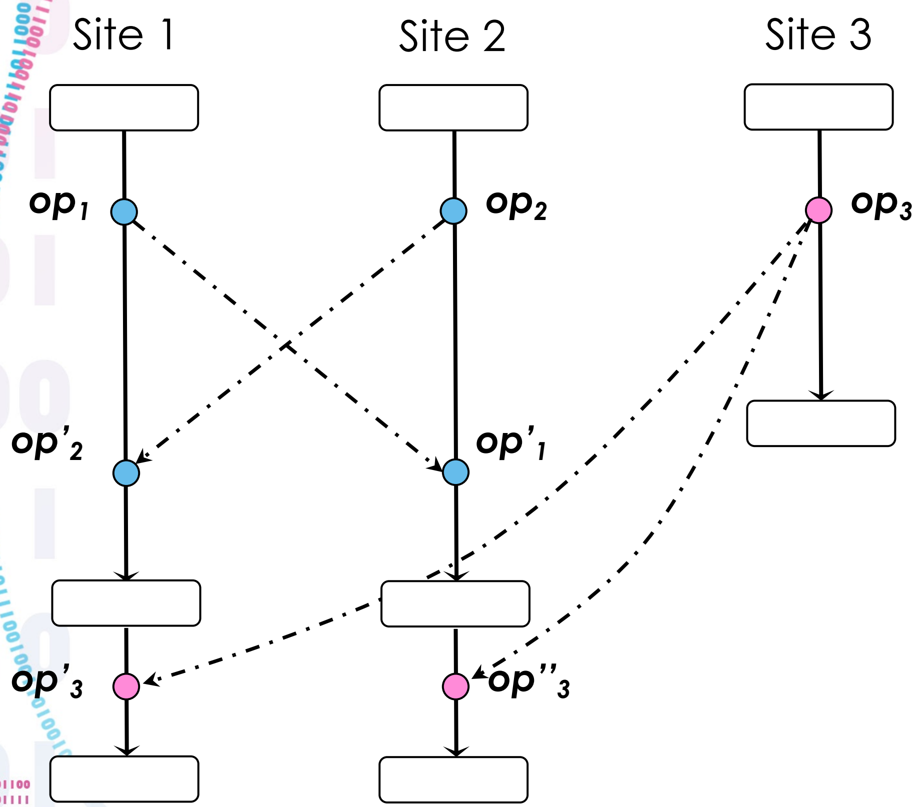
T(op_2: operation, op_1: operation) = op'_2

- op_1 and op_2 concurrent, defined on a state S
- op'_2 same effects as op_2, defined on S.op_1

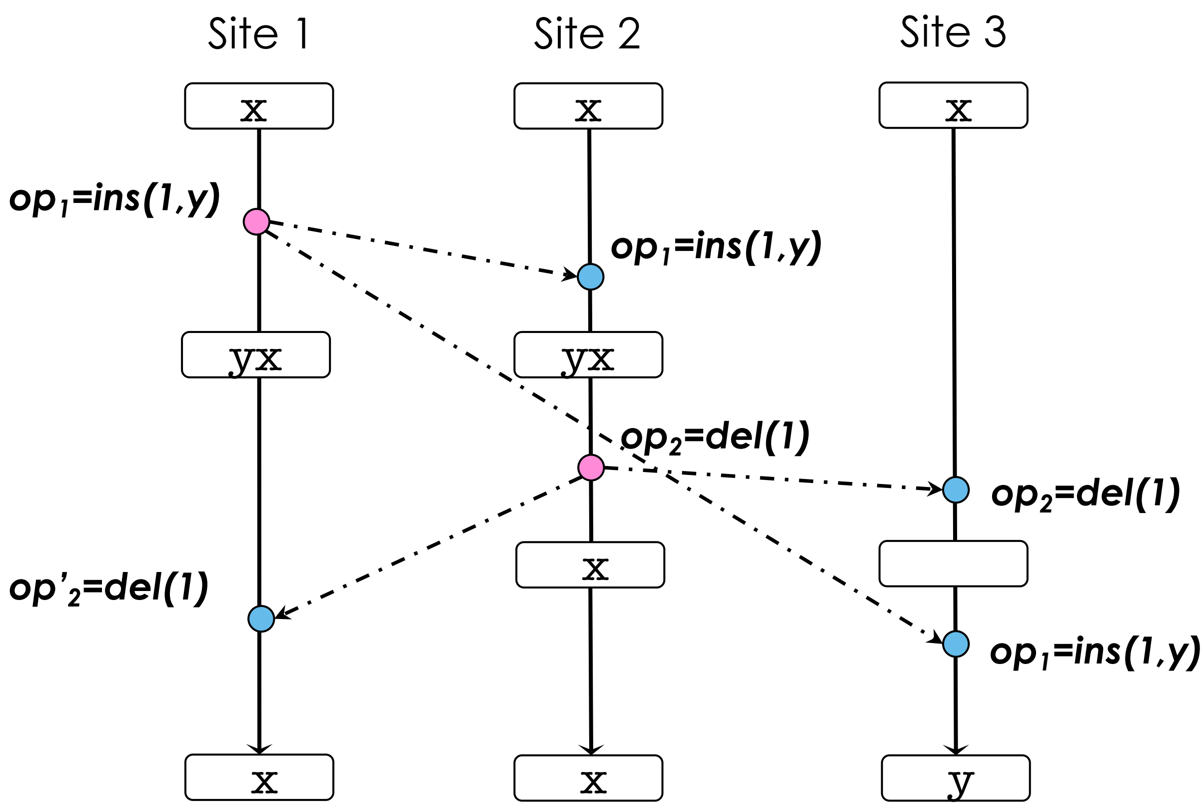
Operational Transformation

Correctness [RNG96]

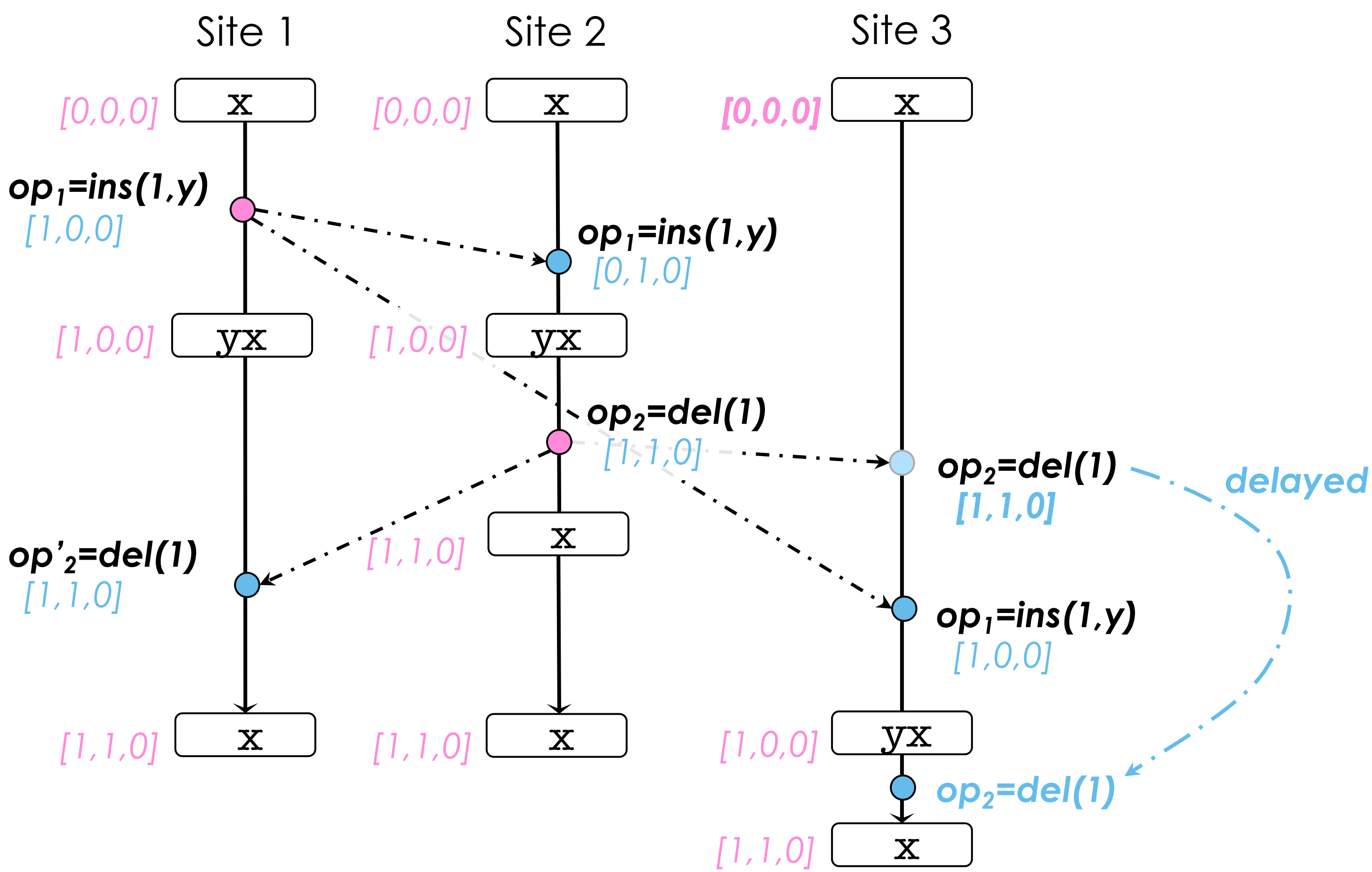
(TP2) $T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$



Operational Transformation
Causality Violation



Operational Transformation
Causality Preservation



Operational Transformation (OT)
Operation Intention

- Intention of an operation is the observed effect as result of its execution on its generation state

- Passing from an initial state ‘ab’ to a final state ‘aXb’ could be observed as:
  - ins(2,x)
  - ins($a \prec X \prec b$)
  - ins($a \prec X$)
  - ins($X \prec b$)

Operational Transformation (OT)

Intention Preservation [SJZYC98]

- For any operation *op* the effects of executing *op* at all sites should be the same as the intention of op
- The effect of executing *op* does not change the effects of independent operations.

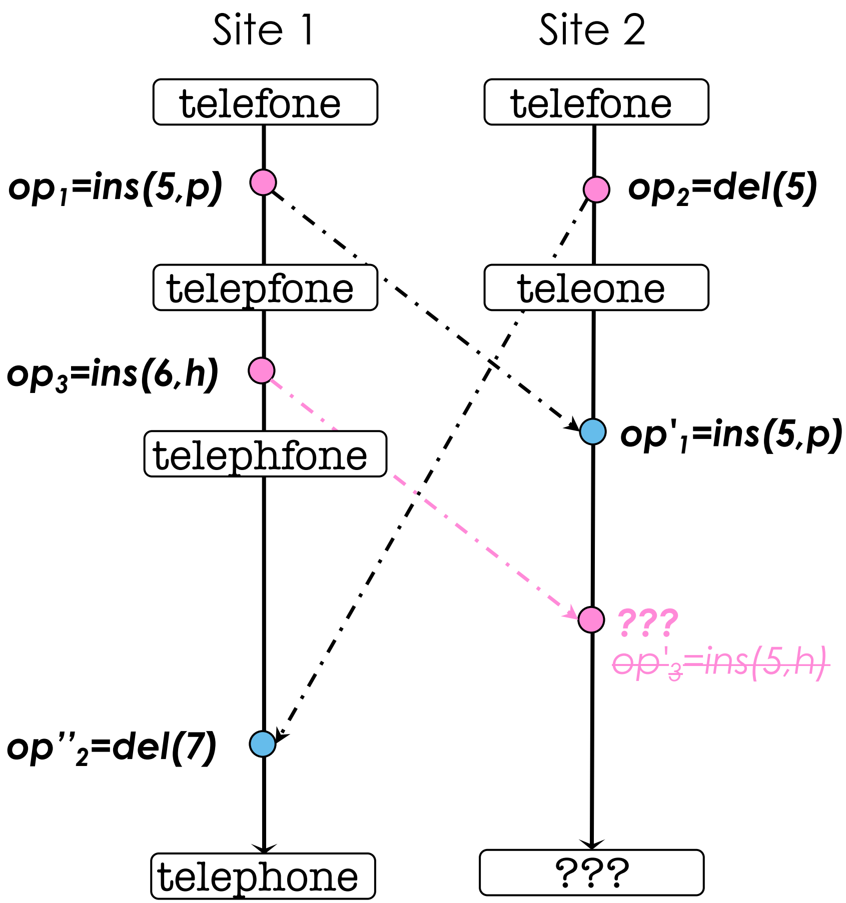
Operational Transformation (OT)
Existing Approaches

- Two main families:
  - Transformation functions satisfying both TP1 and TP2
  - Control algorithms avoiding (needs of) TP2

Operational Transformation (OT)
Existing Approaches

- Design and verify transformation functions T
- T also known as forward transposition (*transpose_fd)*
- Verification of conditions TP1 and TP2
  - Combinatorial explosion (>*100* cases for a string)
  - Iterative process
  - Repetitive and error prone task

Operational Transformation (OT)
Partial Concurrency



$op'_2=T(op_2,op_1)=del(6)$
$op''_2=T(op'_2,op_3)=del(7)$

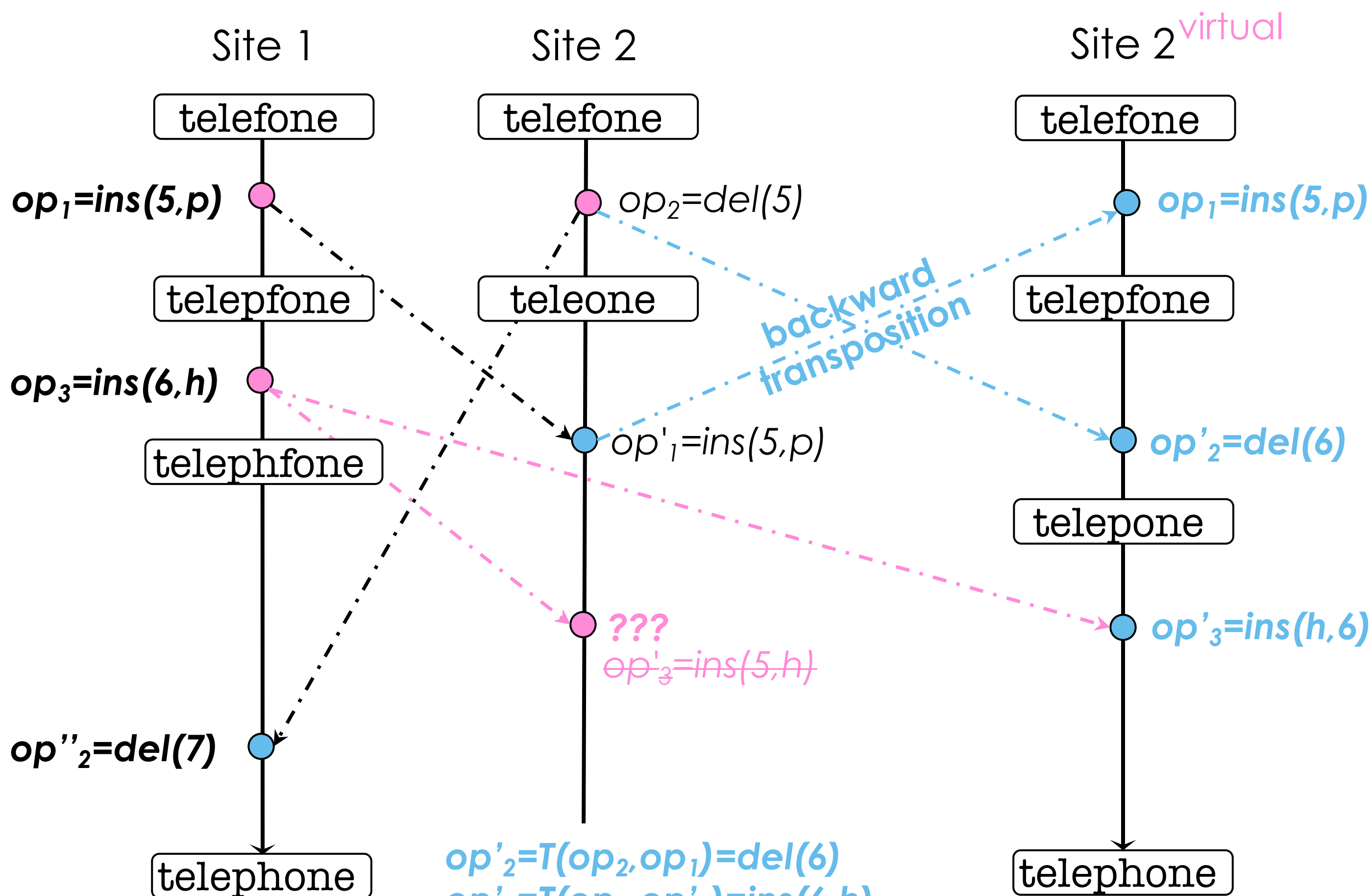
$op'_1=T(op_1,op_2)=ins(5)$

$T(op_3,op_2)$
is not allowed to be performed

Nor $T(op_3,op'_1)$

Operational Transformation (OT)

Partial Concurrency



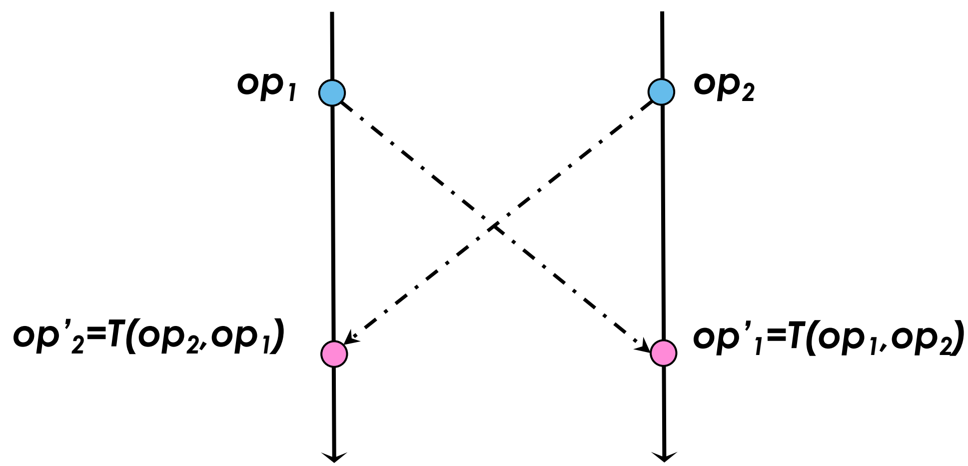
$op'_2=T(op_2,op_1)=del(6)$
$op'_3=T(op_3,op'_2)=ins(6,h)$

Operational Transformation (OT)

Backward Transposition [SJZYC98]

- Backward transposition (*transpose_bk*)

- transpose_bk(op_1, op'_2) = (op_2, op'_1)
  - $op'_2 = T(op_2, op_1)$
  - $op'_1 = T(op_1, op_2)$



- Note:
  - transpose_bk(op_a, op_b) = ($T^{-1}(op_b,op_a)$, $T(op_a, T^{-1}(op_b,op_a))$)

Operational Transformation

Example of Transformation Functions [RNG96]

T(Ins(p1,c1,u1), Ins(p2,c2,u2)) :-
    if (p1<p2) or (p1=p2 and u1<u2) **return** *Ins(p1,c1,u1)*
    else return *Ins(p1+1,c1,u1)*

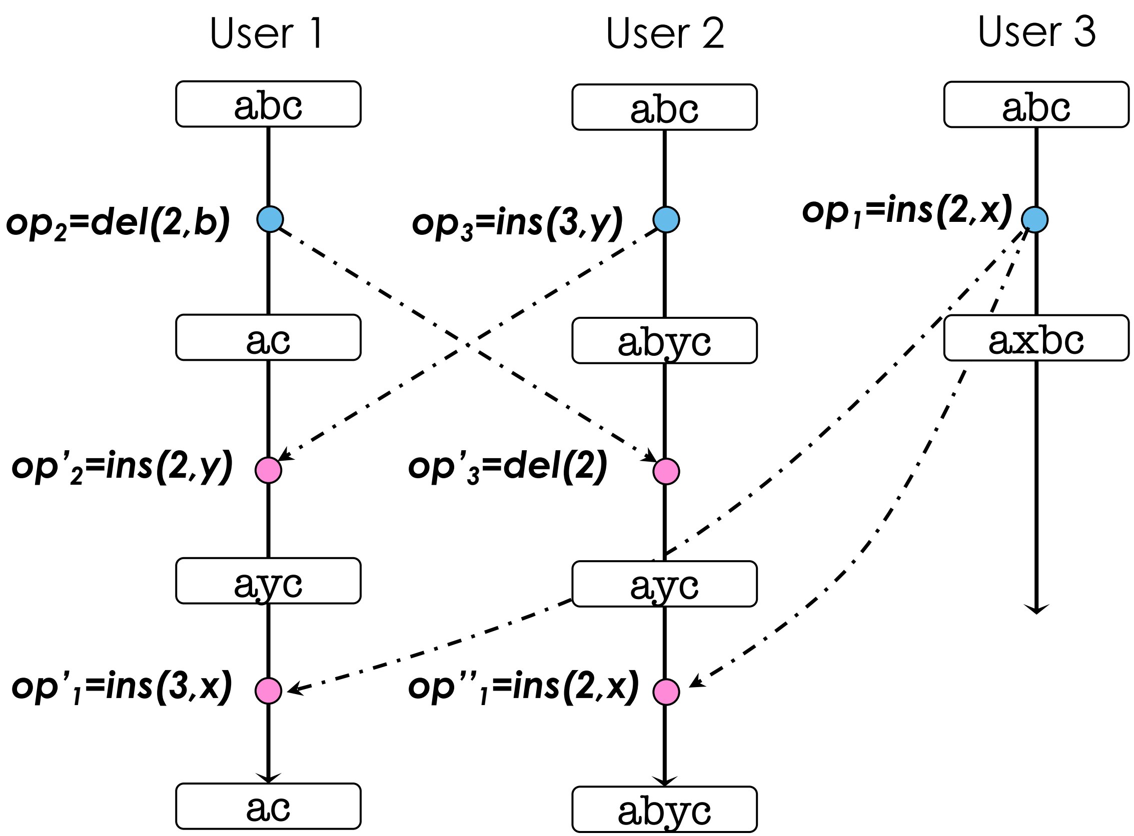
T(Ins(p1,c1,u1), Del(p2,u2)) :-
    if (p1≤p2) **return** *Ins(p1,c1,u1)*
    else return *Ins(p1-1,c1,u1)*
    endif

Satisfy TP1 but not TP2!

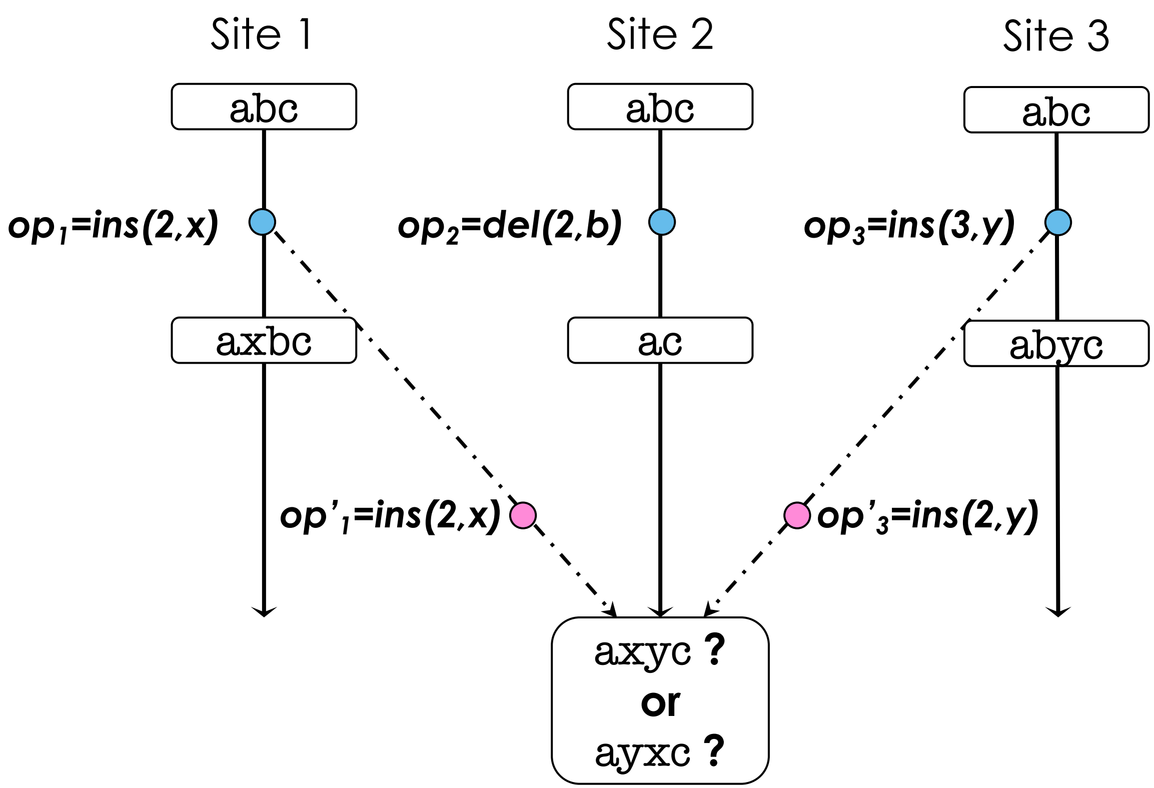
T(Del(p1,u1), Ins(p2,c2,u2)) :-
    if (p1<p2) **return** *Del(p1,u1)*
    else return *Del(p1+1,u1)*

T(Del(p1,u1), Del(p2,u2)) :-
    if (p1<p2) **return** *Del(p1,u1)*
    else if (p1>p2) **return** *Del(p1-1,u1)*
    else return *Id()*

Operational Transformation
Example of Violation of TP2



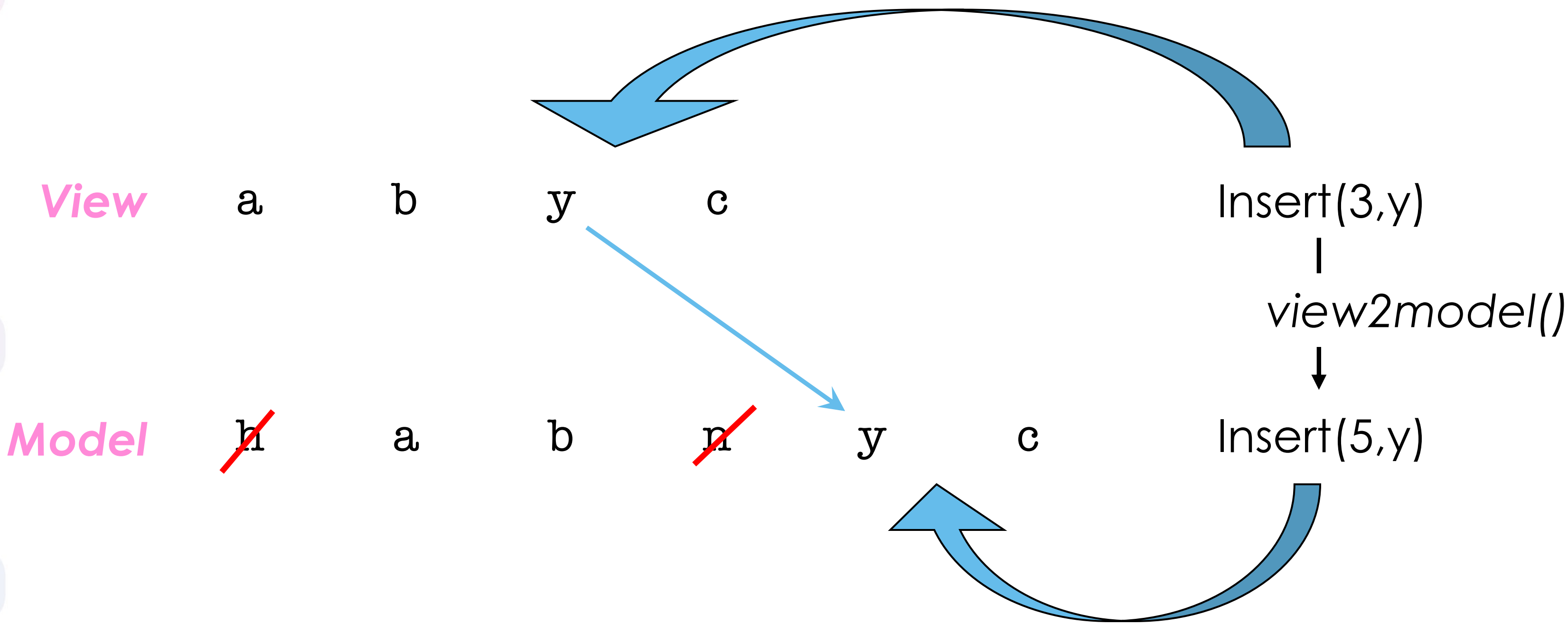
Operational Transformation (OT)
False-Tie Problem



Operational Transformation (OT)

Tombstone Transformation Functions [OUMI06]

- Keep 'tombstones' of deleted elements



Operational Transformation (OT)
Tombstone Transformation Functions

T(Ins(p1,c1,s1), Ins(p2,c2,s2)) :-
    if (p1<p2) or (p1=p2 and s1<s2) **return** *Ins(p1,c1,s1)*
    else return *Ins(p1+1,c1,s1)*

T(Ins(p1,c1,s1), Del(p2,s2)) :-
    return *Ins(p1,c1,s1)*

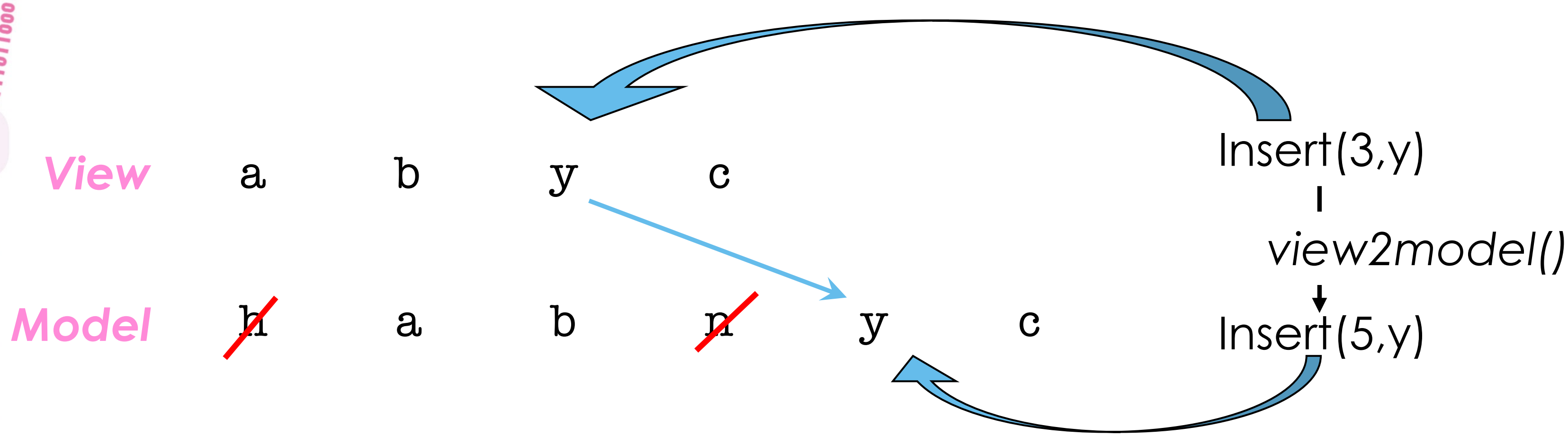
T(Del(p1,s1), Ins(p2,c2,s2)) :-
    if (p1<p2) **return** *Del(p1,s1)*
    else return *Del(p1+1,s1)*

T(Del(p1,s1), Del(p2,s2)) :-
    return *Del(p1,s1)*

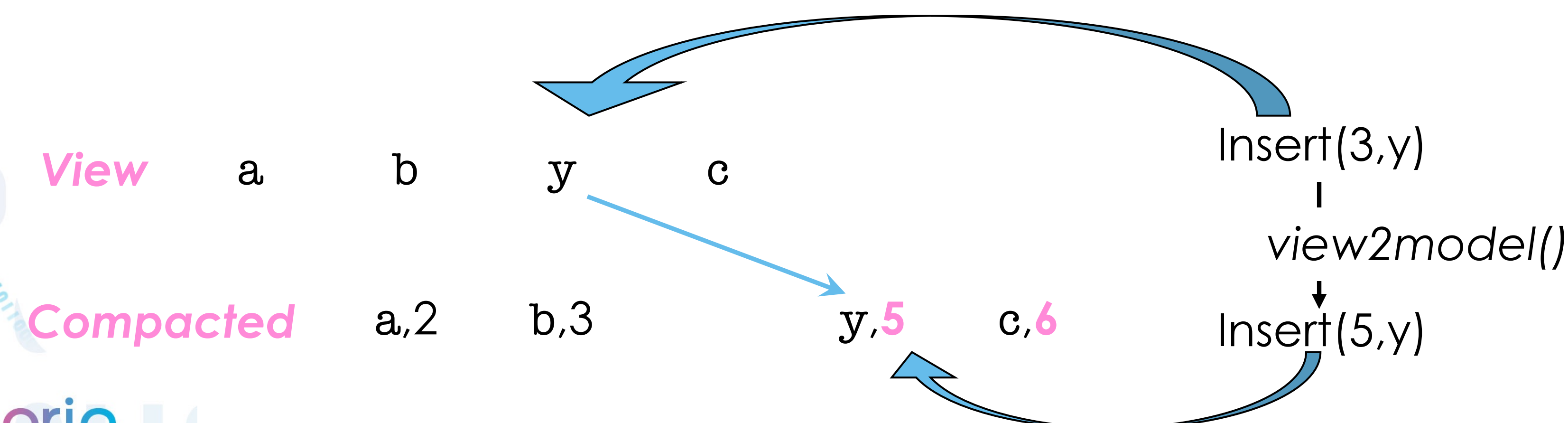
Loria

Operational Transformation (OT)

Tombstone Transformation Functions – Compact Model



Compacted model = sequence of (character, absolute-position)



Operational Transformation (OT)
Tombstone Transformation Functions – Delta Model

View a b y c

Insert(3,y)

view2model()

Insert(5,y)

Compacted a,2 b,3 y,**5** c,**6**

Delta model = sequence of (character, offset)

View a b y c

Insert(3,y)

view2model()

Insert(5,y)

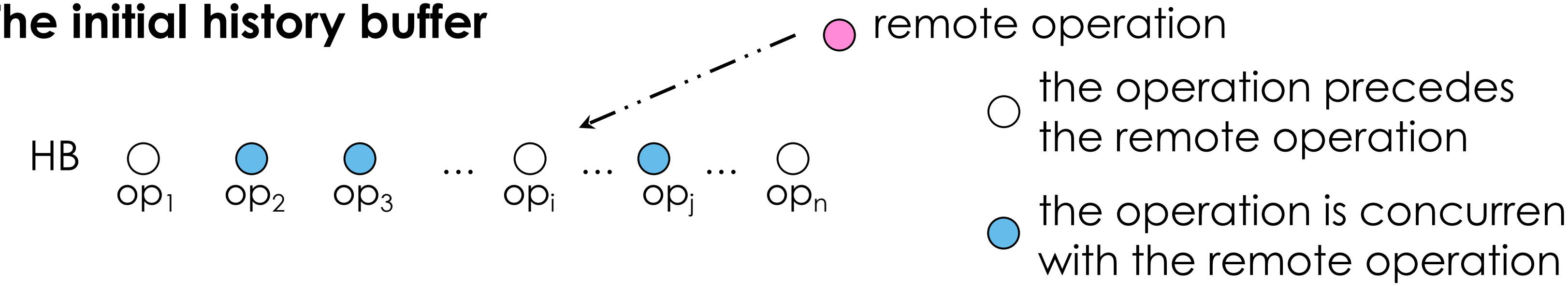
Delta a,+2 b,+1 y,**+2** c,**+1**

Operational Transformation (OT)
Tombstone Transformation Functions – Comparison

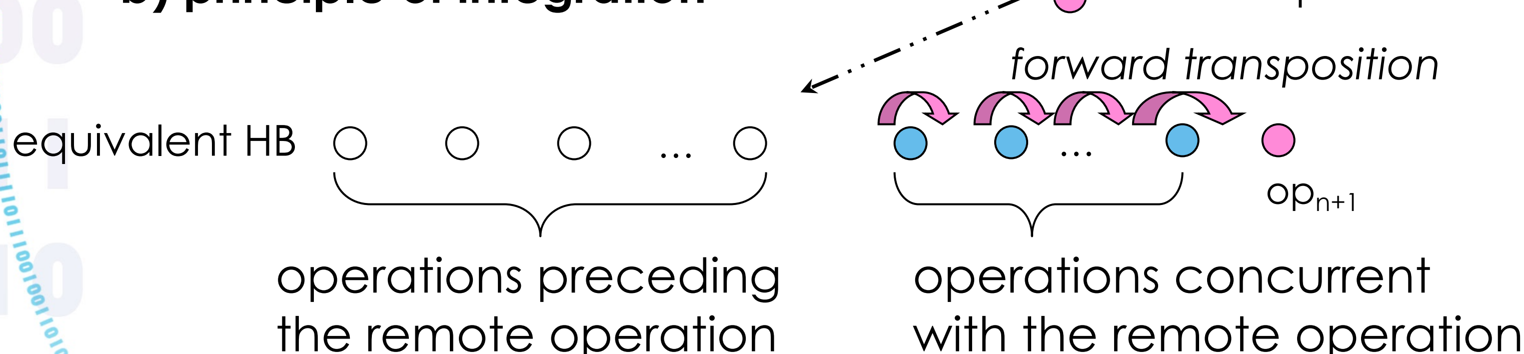
- Basic Model
  - Deleted characters are kept
  - Size of the model is growing infinitely
- Compacted Model
  - Update absolute position of all characters located after the effect position
- Delta Model
  - Update the offset of next character
- Observations
  - view2model can be optimized (caret position)
  - Overhead of view2model is not significant

Operational Transformation (OT)
General Control Algorithm – SOCT2 [SCF97]

a) The initial history buffer



b) principle of integration



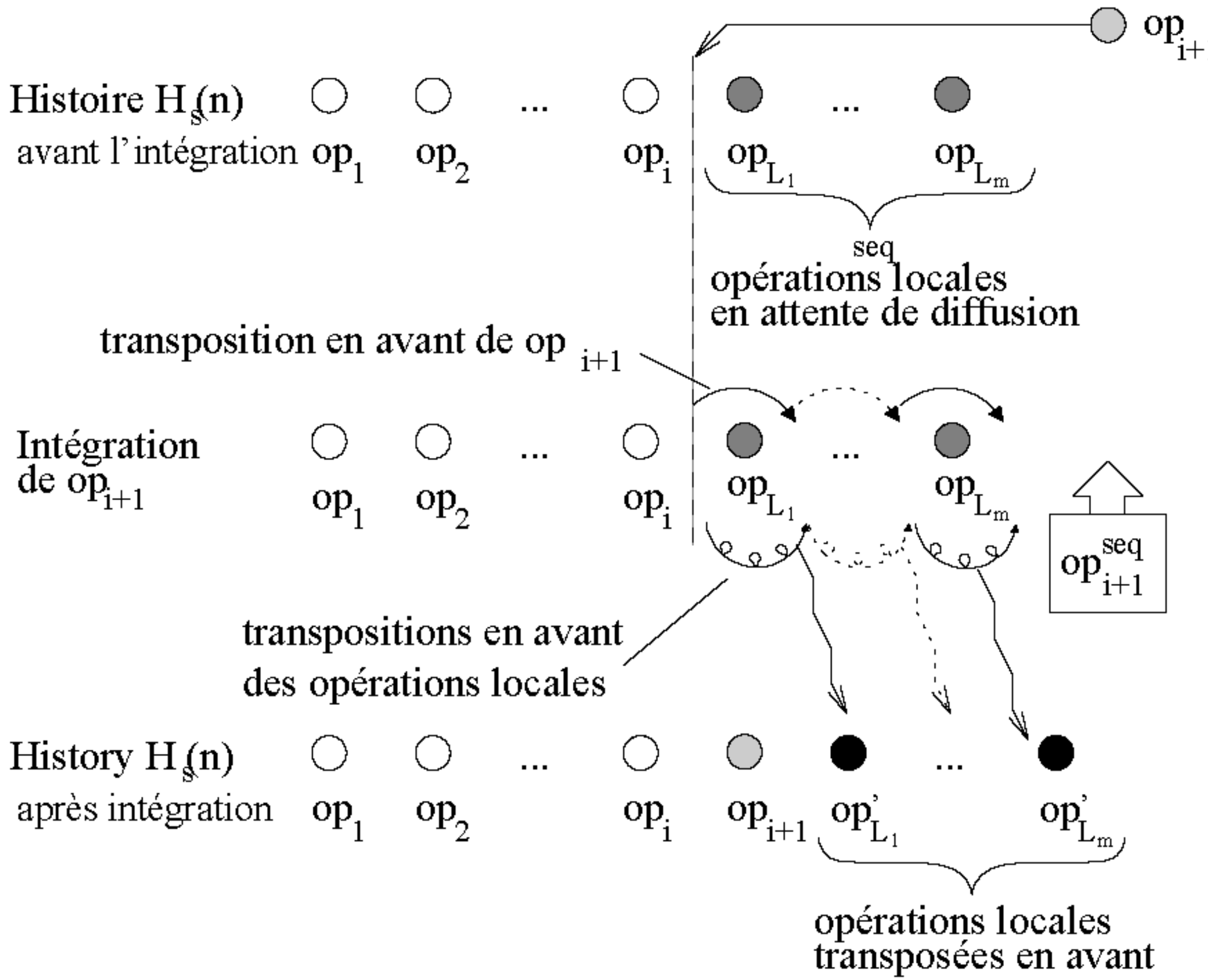
Operational Transformation (OT)
Control Algorithms that avoid TP1 and/or TP2

- GOT algorithm [SJZYC98]
  - Does not need to satisfy TP1 and TP2
  - Requires a global serialization order
    - Sum of state vector components
    - If equality, then priority on sites
  - Requires very costly undo/redo mechanism
    - Does not ensure SEC (only EC)

Operational Transformation (OT)
Control Algorithms that avoid TP1 and/or TP2

- SOCT4 Algorithm [VCFS00]
  - No undo/redo mechanism, no state vectors
  - Eliminates TP2, but requires TP1
  - Global order of operations according to timestamps generated by a sequencer
  - Local operations executed immediately
  - Assigns a timestamp to the operation and transmits it to the other sites
  - Defers broadcast until execution of all preceding operations
  - Transformations performed by each site

Operational Transformation (OT)
SOCT4 Algorithm [VCFS00]



Operational Transformation (OT)

SO6 [MOSI03]– SOCT4 Variant

```
Sync(log, Ns) :-
  while ((op_r = getOp(Ns+1)) != ∅)
    for (i=0; i<log.size(); i++)
      op_l=log[i];
      log[i]=T(op_l, op_i)
      op'_i=T(op_i, op_l);
    endfor
    execute(op'i)
    Ns=Ns+1
  endwhile

  for (i=0; i<log.size(); i++)
    op'_l=log[i];
    if send(op'_l, Ns+1) then
      Ns=Ns+1
    else
      error 'need to synchronize'
    endif
  endfor
```

getOp(ticket) retrieves operation identified by timestamp ticket

send(op,ticket) sends local operation with timestamp ticket. If ticket already exists, returns false

Operational Transformation (OT)
SO6 [MOSI03]– SOCT4 Variant

Site 1, Ns=0	Site 2, Ns=0
op_1	op_3
op_2	op_4
s_1=synchronize()	
	s_2=synchronize()
s_3=synchronize()	

- **At step s_1**
  - sync($[op_1,op_2]$,0)
  - Send op_1, op_2
  - Ns=2
- **At step s_2**
  - sync($[op_3,op_4]$,0)
  - $op'_1=T(op_1,op_3)$
  - $op'_3=T(op_3,op_1)$
  - $op''_1=T(op'_1,op_4)$
  - $op'_4=T(op_4,op'_1)$
  - $op'_2=T(op_2,op'_3)$
  - $op''_3=T(op'_3,op_2)$
  - $op''_2=T(op'_2,op'_4)$
  - $op''_4=T(op'_4,op'_2)$
  - op''_1,op''_2 are executed
  - op''_3,op''_4 are sent
  - Ns=4
- **At step s_3**
  - sync([],2)
  - op''_3 and op''_4 are executed
  - Ns=4

Operational Transformation (OT)
SO6 [MOSI03]– SOCT4 Variant

Site 1, Ns=0	Site 2, Ns=0
op_1	op_3
op_2	op_4
s_1=synchronize()	
	s_2=synchronize()
s_3=synchronize()	

- **At site 1**
  - op_1
  - op_2
  - $op''_3=T(T(op_3, op_1), op_2)$
  - $op''_4=T(T(op_4, op'_1), op'_2)$
- **At site 2**
  - op_3
  - op_4
  - $op''_1=T(T(op_1, op_3), op_4)$
  - $op''_2=T(T(op_2, op'_3), op'_4)$

Operational Transformation (OT)
Control Algorithms that avoid TP1 and/or TP2

- Jupiter algorithm [NCDL95]
  - Used in GoogleDocs
  - Requires a central server
  - Eliminates TP2, but requires TP1
  - Does not need state vectors
  - Transformations done on the server + client side

Operational Transformation (OT)
Jupiter algorithm [NCDL95]

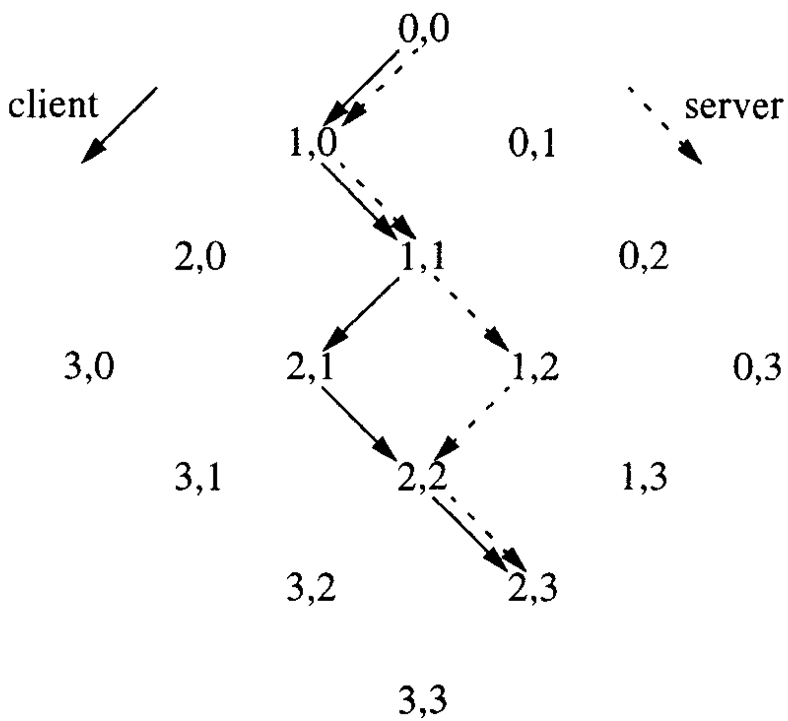
- GoogleWave

… now in GoogleDocs

Loria

Operational Transformation (OT)

Jupiter Algorithm



xform(c,s)={c',s'}

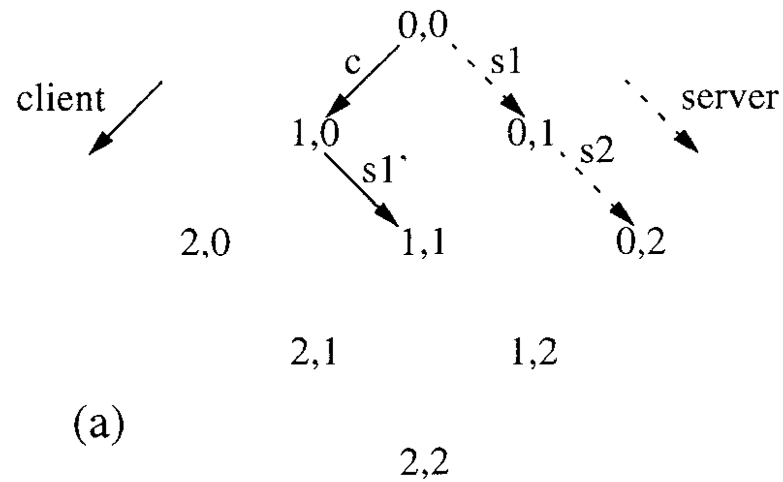
xform(del x, del y)=

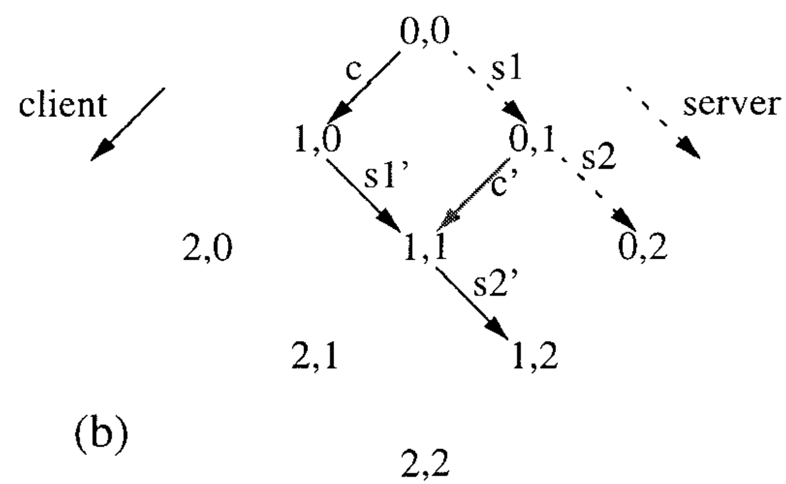
{del x-1, del y} if x>y

{del x, del y-1} if x<y

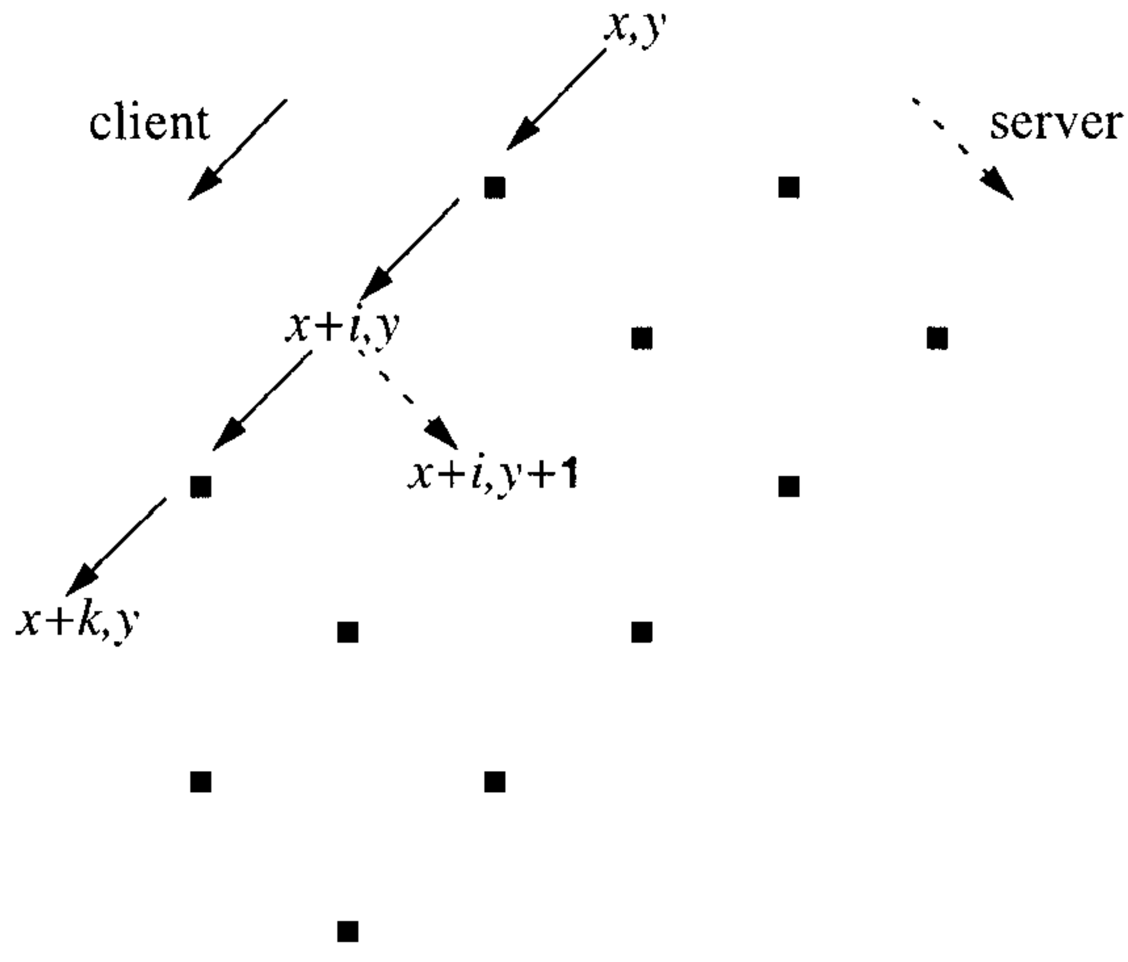
{no-op, no-op} if x=y

Operational Transformation (OT)
Jupiter Algorithm





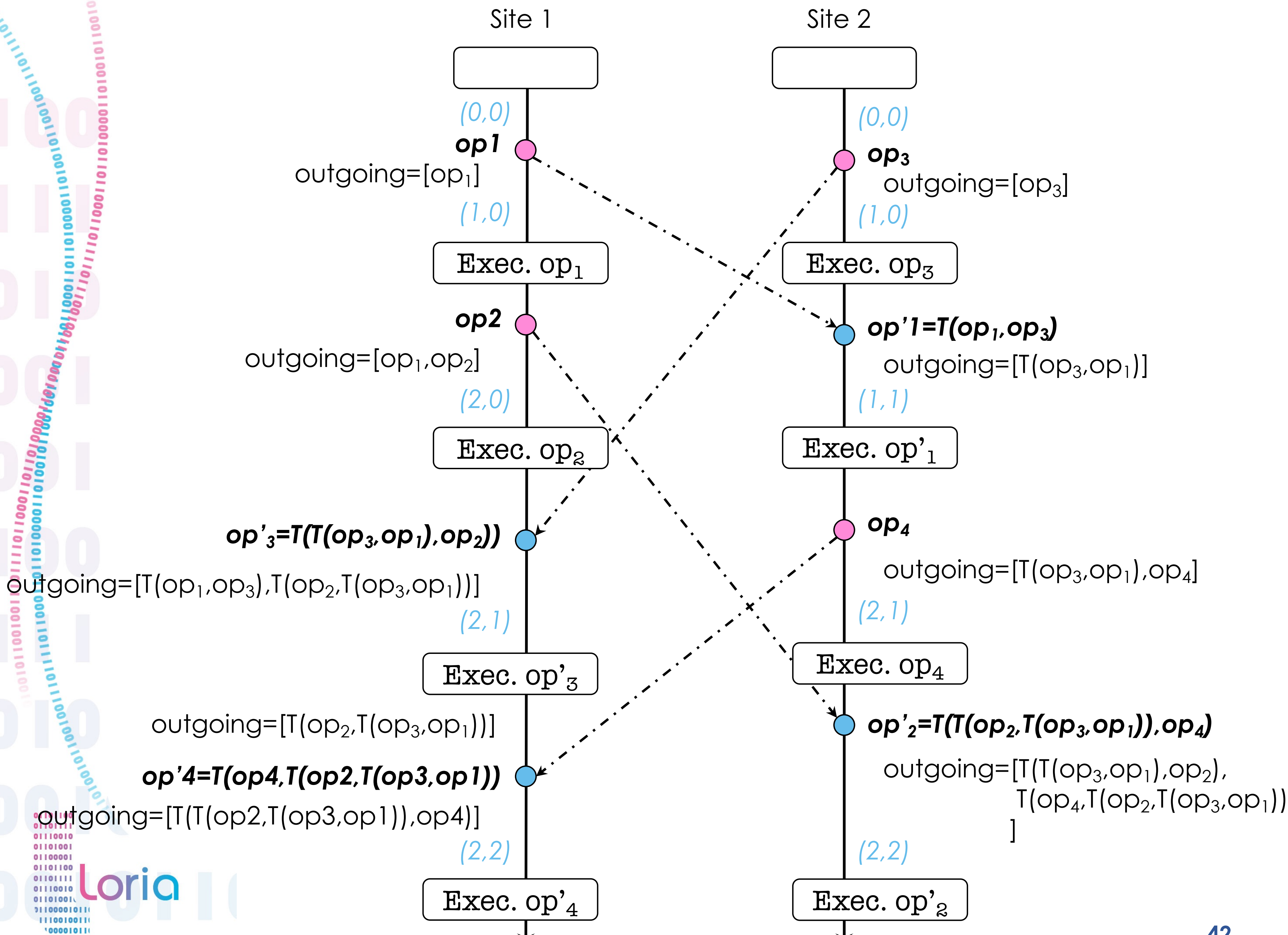
Operational Transformation (OT)
Jupiter Algorithm



Operational Transformation (OT)
Jupiter Algorithm

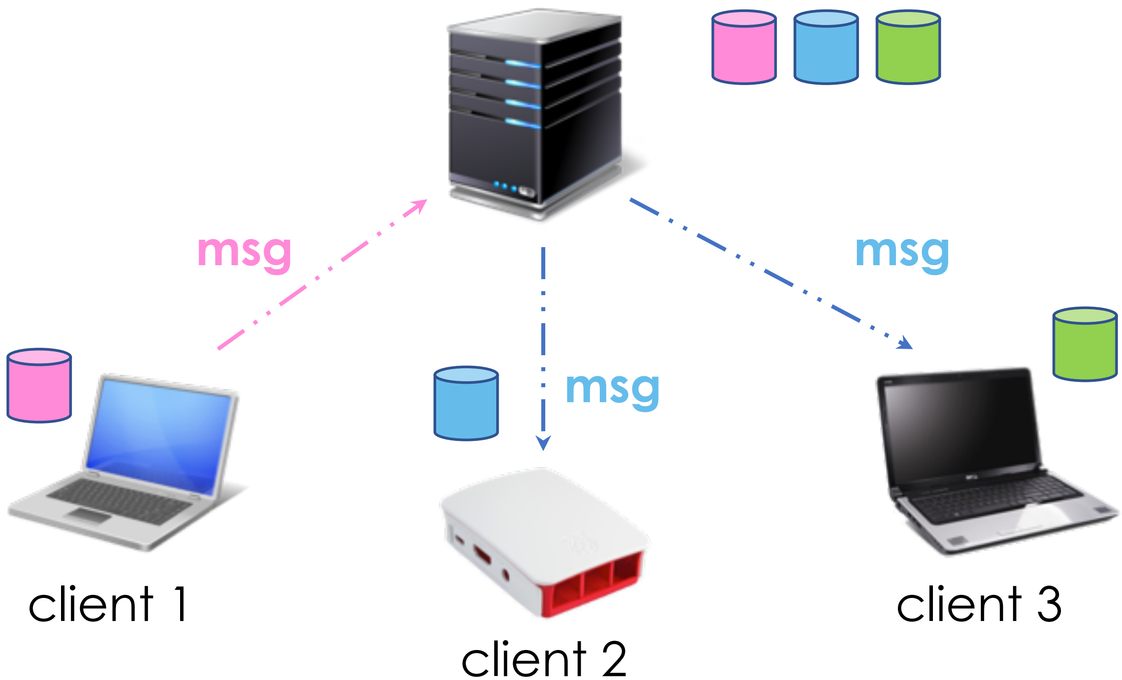
```
int myMsgs = 0; /* number of messages generated */
int otherMsgs = 0; /*number of messages received */

generate(op) {
  apply op locally;
  send(op, myMsgs, otherMsgs);
  add(op, myMsgs) to outgoing;
  myMsgs = myMsgs + 1;
}
receive(msg) {
  /* discard acknowledged messages */
  for m in (outgoing) {
    if (m.myMsgs < msg.otherMsgs)
      remove m from outgoing;
  }
  /* ASSERT: msg.myMsgs == otherMsgs */
  for i in [1..length(outgoing)] {
    /* transform new messages and the ones in the queue */
    { msg, outgoing[i] } = xform(msg, outgoing[i]);
  }
  apply msg.op locally;
  otherMsgs = otherMsgs + 1;
}
```


Site 1
Site 2
(0,0)
op1
outgoing=[op$_1$]
(1,0)
Exec. op$_1$
op2
outgoing=[op$_1$,op$_2$]
(2,0)
Exec. op$_2$
op'$_3$=T(T(op$_3$,op$_1$),op$_2$))
outgoing=[T(op$_1$,op$_3$),T(op$_2$,T(op$_3$,op$_1$))]
(2,1)
Exec. op'$_3$
outgoing=[T(op$_2$,T(op$_3$,op$_1$))]
op'4=T(op4,T(op2,T(op3,op1))
outgoing=[T(T(op2,T(op3,op1)),op4)]
(2,2)
Exec. op'$_4$
(0,0)
op$_3$
outgoing=[op$_3$]
(1,0)
Exec. op$_3$
op'1=T(op$_1$,op$_3$)
outgoing=[T(op$_3$,op$_1$)]
(1,1)
Exec. op'$_1$
op$_4$
outgoing=[T(op$_3$,op$_1$),op$_4$]
(2,1)
Exec. op$_4$
op'$_2$=T(T(op$_2$,T(op$_3$,op$_1$)),op$_4$)
outgoing=[T(T(op$_3$,op$_1$),op$_2$),
T(op$_4$,T(op$_2$,T(op$_3$,op$_1$)))
]
(2,2)
Exec. op'$_2$

Operational Transformation (OT)

Jupiter Algorithm – Generalization to n clients [Z01]



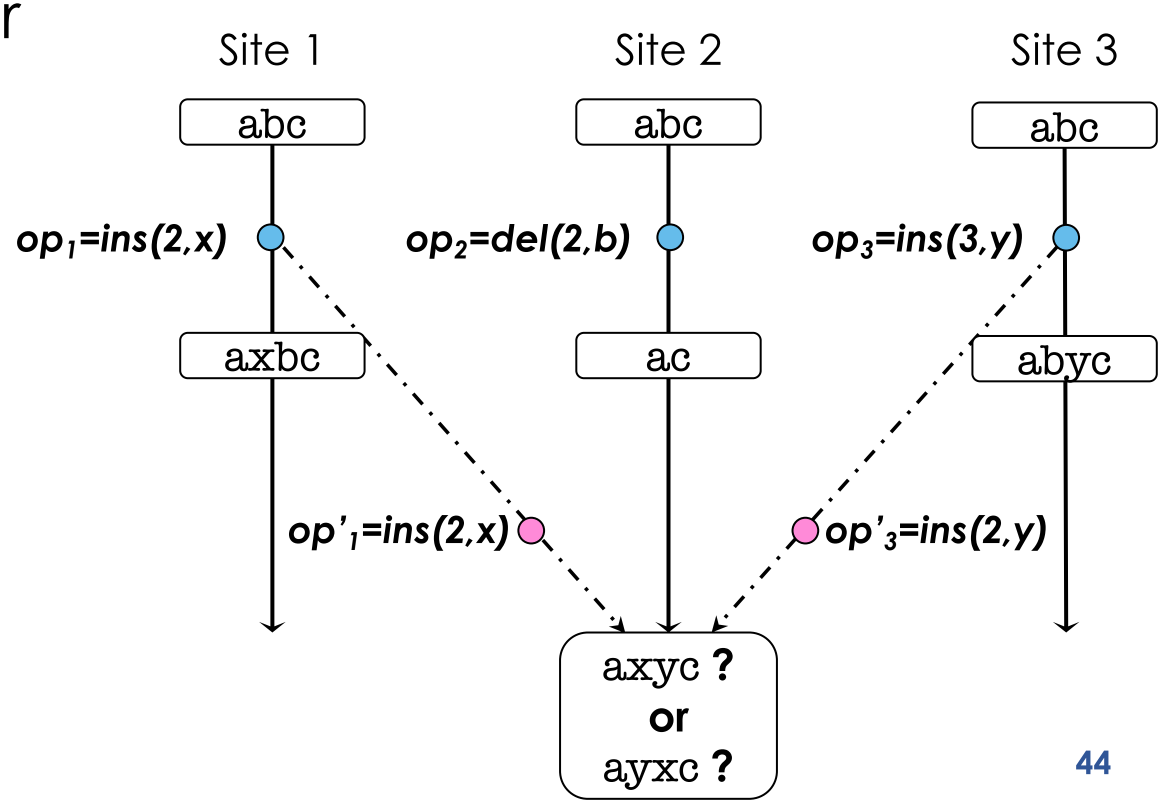
apply msg.op locally;

Algorithm changes
at server side

apply msg.op locally;
for (c in client list) {
  if (c != client)
    send(c, msg);
}

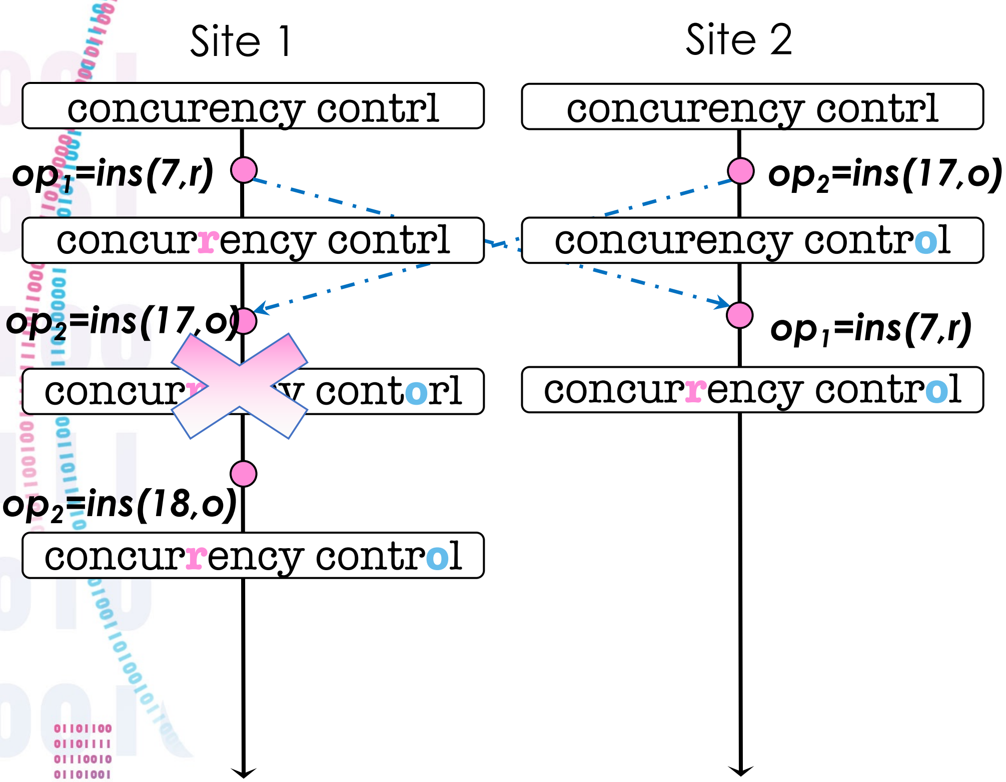
Operational Transformation (OT)
Jupiter Algorithm – Summary

- Requires a server that performs transformations
- Not suitable for P2P environments
- False tie scenario gives different results according to integration order



Operational Transformation (OT)

Summary



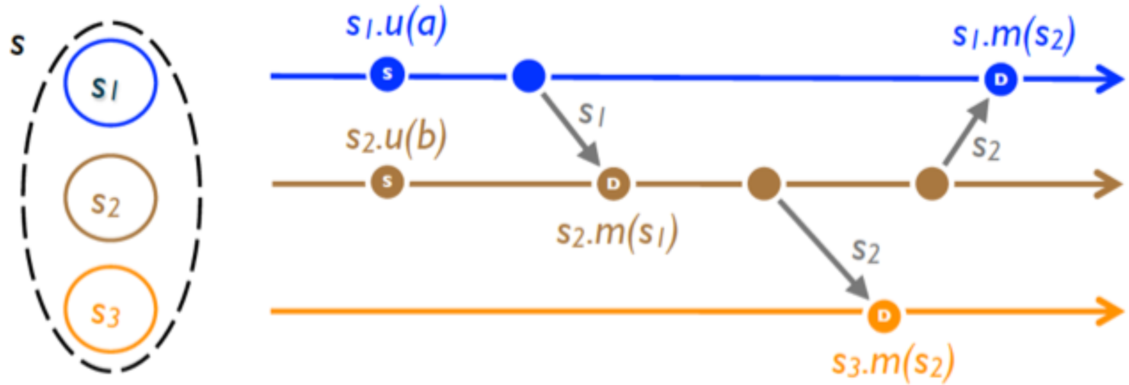
- Transforms non commuting operations to make them commute
- Genericity
- Time complexity
  Average: O(H.c)
  Worst case: O(H^2)
- Difficult to write correct transformation functions
- State vectors used for detecting concurrency ⇒ scalability limitations
- **Not very suitable for large scale peer-to-peer collaboration**

Conflict-free Replicated Data Types (CRDT) [SPBZ09]

- Data structures
  - Same semantics (w/o concurrency) as classical data-structures
  - **Designs operations to be commutative**
- Replicated
  - At multiple sites
- Available
  - Replica is updated without coordination
  - Convergence is (formally) guarantee
  - Decentralized / Peer-to-peer

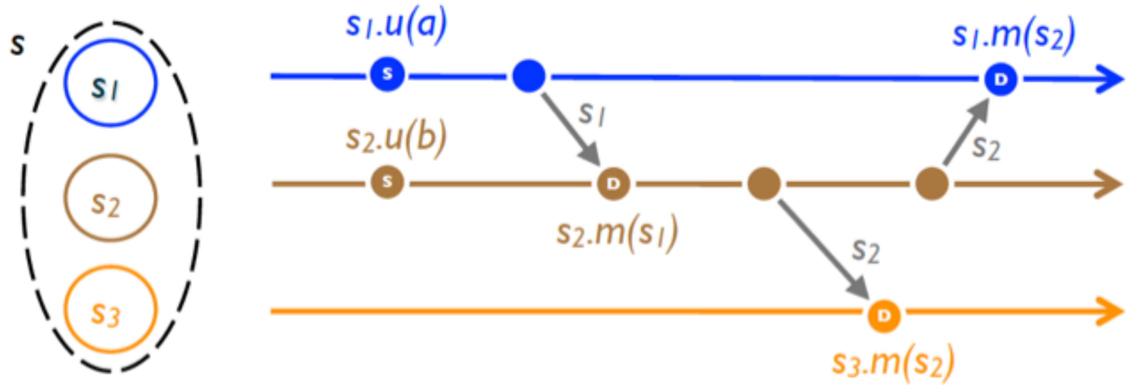
- State-based and Operation-based approaches

Conflict-free Replicated Data Types
State-based Replication



- Replicated object: a tuple (S, s_0, q, u, m)
  - S: state domain
  - Replica at process p_i has state $s_i \in S$
  - s_0: initial state
- Each replica can execute one of following commands
  - q: query object's state
  - u: update object's state
  - m: merge state from a remote replica

Conflict-free Replicated Data Types
State-based Replication



- Algorithm
  - Periodically, replica at p_i sends its current state to p_j
  - Replica p_j merges received state into its local state by executing m
- After receiving all updates (irrespective of order), each replica will have same state

Conflict-free Replicated Data Types
Semi-lattice

- Partial order $\leq$ set S with a least upper bound (LUB), denoted $\sqcup$
  - $m = x \sqcup y$ is a LUB of $\{x, y\}$ under $\leq$ if and only if $\forall m', x \leq m' \wedge y \leq m' \Rightarrow x \leq m \wedge y \leq m \wedge m \leq m'$

- It follows that $\sqcup$ is:
  - commutative: $x \sqcup y = y \sqcup x$
  - idempotent: $x \sqcup x = x$
  - associative: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

Conflict-free Replicated Data Types

Semi-lattice – Example on integers

- Partial order $\leq$ on set of integers
- Least upper bound $\sqcup$: max (maximum function)

- Therefore, we have:
  - commutative: max(x, y) = max(y, x)
  - idempotent: max(x, x) = x
  - associative: max(max(x, y), z) = max(x, max(y, z))

Loria

Conflict-free Replicated Data Types

Semi-lattice – Example on sets

- Partial order $\subseteq$ on sets
- Least upper bound $\sqcup$: $\cup$ (set union)

- Therefore, we have:
  - commutative: $A \cup B = B \cup A$
  - idempotent: $A \cup A = A$
  - associative: $(A \cup B) \cup C = A \cup (B \cup C)$

Conflict-free Replicated Data Types
Monotonic Semi-lattice Object

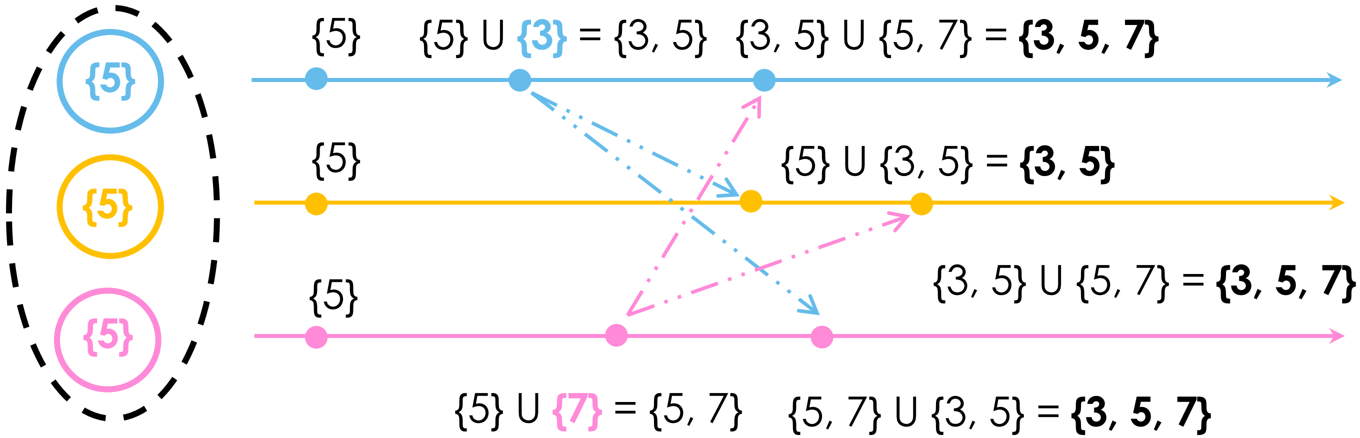
- A state-based object with partial order $\leq$, noted $(S, \leq, s_0, q, u, m)$, that has following properties, is called a monotonic semi-lattice:
  1. Set S of values forms a semi-lattice ordered by $\leq$
  2. Merging state s with remote state s' computes the LUB of the two states, i.e., $s \bullet m(s') = s \sqcup s'$
     (**delivery order is not important**)
  3. State is monotonically non-decreasing across updates, i.e., $s \leq s \bullet u$
     (**updates have effect, no rollback**)

Conflict-free Replicated Data Types
Convergent Replicated Data Type (CvRDT)

- **Theorem:** Assuming eventual delivery and termination, any replicated state-based object that satisfies the monotonic semi-lattice property is SEC

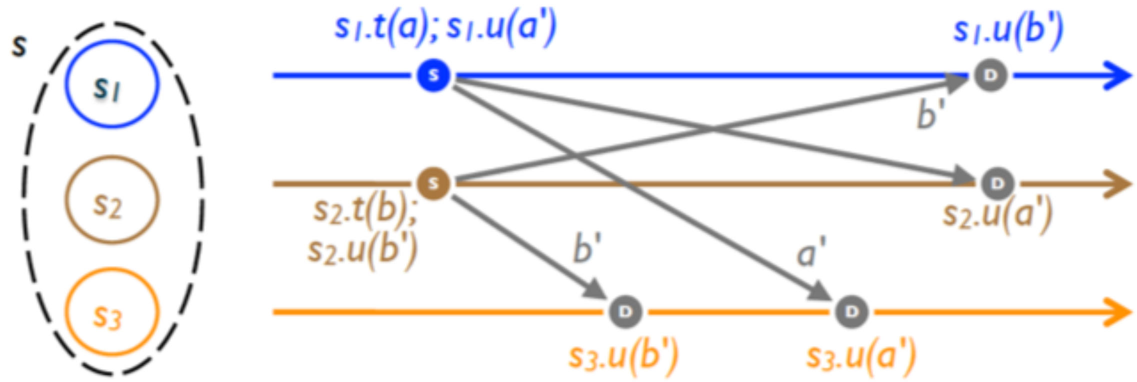
- Since:
  - Merge is both **commutative** and **associative**
    - We do not care about order
  - Merge is **idempotent**
    - We do not care about delivering more than once

Convergent Replicated Data Types
Example



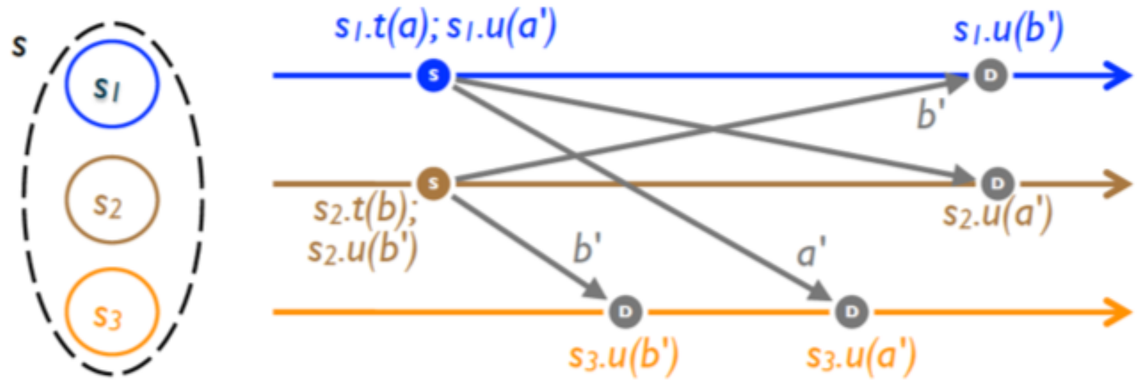
- Each replica can execute one of following commands
  - query q: returns entire set
  - update u: adds new element (e, α)to local set
  - merge m: compute unions between local set and remote set

Conflict-free Replicated Data Types
Operation-based Replication



- Replicated object: a tuple (S, s_0, q, t, u, P).
  - S: state domain
  - Replica at process p_i has state $s_i \in S$
  - s_0: initial state
- Each replica can execute one of following commands
  - q: query object's state
  - t: side-effect-free prepare-update method (at local replica)
  - u: effect-free update method (at all replicas)
  - P: delivery precondition

Conflict-free Replicated Data Types
Operation-based Replication



- Algorithm
  - Updates are delivered to all replicas
  - Use causally-ordered broadcast communication protocol, i.e., deliver every message to every node exactly once, w.r.t. happen-before order
  - Happen-before: updates from same replica are delivered in the order they happened to all recipients (effectively delivery precondition, P)
  - Note: concurrent updates can be delivered in any order

Conflict-free Replicated Data Types
Commutativity Property

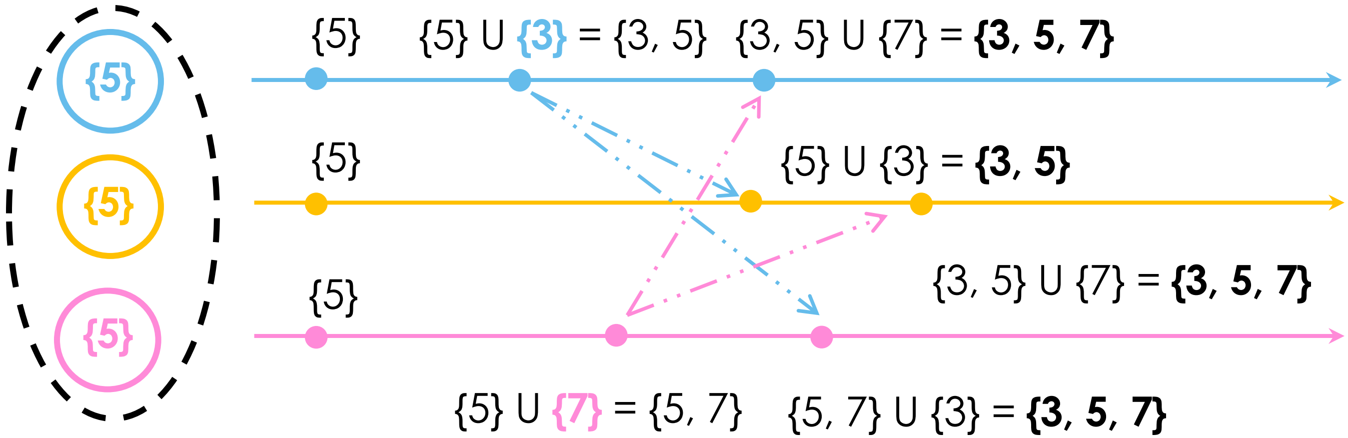
- Updates (t, u) and (t', u') commute, if and only if for any reachable replica state s where both u and u' are enabled:
  - u (resp. u') remains enabled in state $s \bullet u'$ (resp. $s \bullet u$)
  - $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$

- Commutativity holds for concurrent updates

Conflict-free Replicated Data Types
Commutative Replicated Data Type (CmRDT)

- **Theorem:** Assuming causal delivery of updates and method termination, any replicated op-based object that satisfies the commutativity property for all concurrent updates is SEC

Commutative Replicated Data Types
Example



- query q: returns entire set
- prepare method t: adds new element (e,α) to local set
- update u: add delta to any remote replica

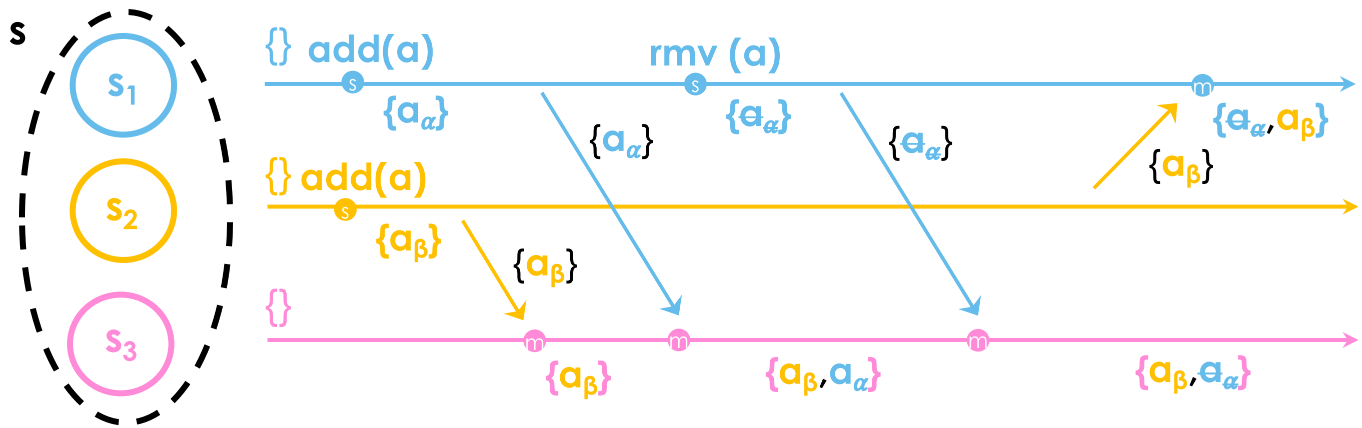
Consistency Maintenance
Conflict-free Replicated Data Types (CRDT)

- Register
  - Last-Writer Wins
  - Multi-Value
- Set
  - Grow-Only
  - 2-Phase
  - **Observed-Remove**
  - Observed-Update-Remove
- Map
- **Counter**
- Graph
  - Directed
  - Monotonic DAG
  - Edit graph
- **Sequence**

basho
bet365
ROVIO
Riot Games
League of Legends
SCALITY
riak
TomTom
TRIFORK. ...think software
Tapjoy
SOUNDCLOUD
HELIUM
akka
PROJECTRIO
ATOM

Loria

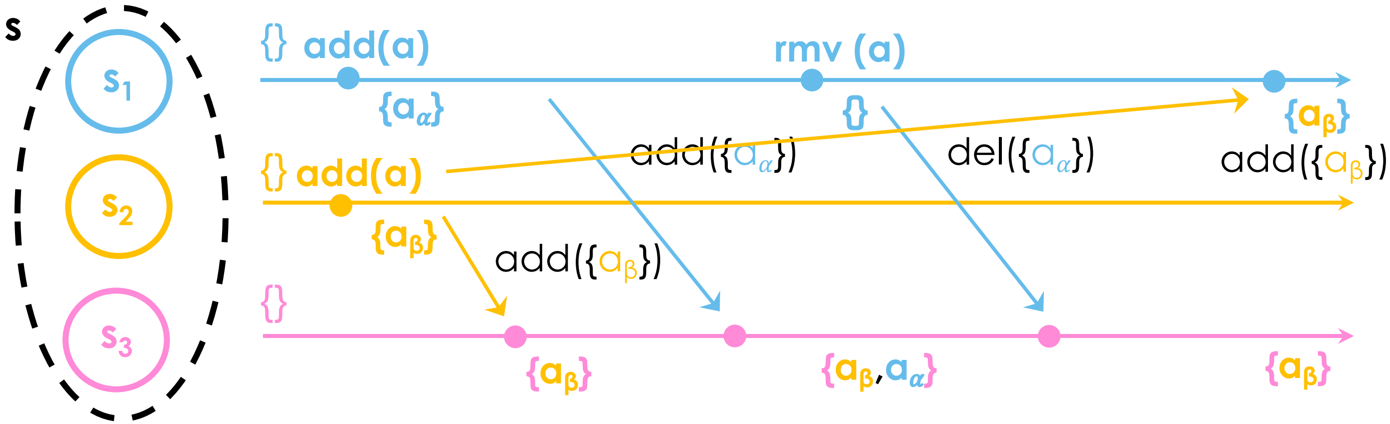
Conflict-free Replicated Data Types
Observed-Remove Set (CvRDT)



- Payload: (A,R) - added/removed sets of *(element, unique-token)*
- Operations:
  - *add(e): A := A U {(e, α)}*
  - *remove(e): R := R U {(e,-)∈A}* **remove all unique elements observed**
  - *lookup(e): ∃(e,-)∈A\R*
  - merge(S, S'): *(A U A', R U R')*
- *{true} add(e) ǁ remove(e) {e∈S}*

Optimised OR-set
- summarise received *adds* in a vector
- check vector during merge

Conflict-free Replicated Data Types
Observed-Remove Set (CmRDT)



- Payload:
  $S = \{(e,\alpha), (e,\beta), (e,\gamma), \ldots\}$ where α, β, γ,… are unique token
- Operations:
  add(e): $S := S \cup \{(e,\alpha)\}$ where α is a fresh unique token
  lookup(e): $\exists \alpha: (e,\alpha) \in S$
  remove(e): $R := \{(e,\alpha) \mid \exists \alpha\ (e,\alpha) \in S\}$ (at source) no tombstones
  $S := S \setminus R$
  $\{true\}\ add(e) \parallel remove(e)\ \{e \in S\}$

Conflict-free Replicated Data Types
P-Counter (CvRDT)

- Payload:
  - P = [int, int, …]
- Operations:
  - value(): $\sum_i P[i]$
  - increment(): P[MyID]++
  - merge(S,S'): $S \sqcup S' = [\ldots,\max(s.P[i],s'.P[i]),\ldots]_i$
- Positive

Conflict-free Replicated Data Types
PN-Counter (CvRDT)

- Payload:
  - P = [int, int, …], N = [int, int, …]
- Operations:
  - value(): $\sum_i P[i] - \sum_i N[i]$
  - increment(): P[MyID]++
  - decrement(): N[MyID]++
  - merge(S,S'): $S \sqcup S' = ([\ldots,\max(s.P[i],s'.P[i]),\ldots]_i, [\ldots,\max(s.N[i],s'.N[i]),\ldots]_i)$
- Positive or negative

Conflict-free Replicated Data Types
CvRDT vs. CmRDT

- Both approaches are equivalent
  - A state-based object can emulate an operation-based object, and vice-versa

- Operation-based:
  - More efficient since you only ship small updates
  - But require causally-ordered broadcast
- State-based:
  - Only require reliable broadcast

Conflict-free Replicated Data Types (CRDT)

(Text) Sequence [PMSL09] [WUM09]

- Document = linear sequence of elements
  - Each element has a unique identifier
  - Identifier constant for the lifetime of the document
  - Dense total order of identifiers consistent with element order:
    - $\forall\, id_x , id_y : id_x < id_y \Rightarrow \exists\, id_z : id_x < id_z < id_y$

- Different approaches for generating identifiers:
  - TreeDoc, Logoot, LogootSplit, …

Conflict-free Replicated Data Types (CRDT)

Logoot [WUM09]

Logoot identifiers: $<p_1,s_1,h_1><p_2,s_2,h_2> \cdots <p_k,s_k,h_k>$

p_i integer

s_i site identifier

h_i logical clock at site s_i

<1,2,1>	c
<1,2,2>	o
<2,1,2>	n
<3,1,3>	c
<3,1,3><8,4,5>	u
<3,2,5>	r
<4,1,7>	e
<4,1,7><9,2,6>	n
<7,2,8>	c
<9,1,7>	y
<10,2,8>	
<12,3,1>	c
<12,3,1><6,5,1>	o
<12,3,1><7,8,2>	n
<12,3,1><7,8,2><12,3,5>	t
<12,3,1><7,8,2><13,3,6>	r
<12,3,1><7,8,2><14,3,7>	l

ins(<3,2,5><13,1,7>, r)

ins(<12,3,1><7,8,2><13,3,6><7,2,9>, o)

- Time complexity
  Average: O(k.log(n))
  Worst case: O(H.log(H))
- No need for concurrency detection
- Identifiers storage cost
- New design for each data type
- **Suitable for large-scale collaboration**

Conflict-free Replicated Data Types (CRDT)
LogootSplit [AMOI13]

LogootSplit identifiers

Base: p_1 … p_n *site_id* *clock*
Interval: begin end

p_1	…	p_n	*site_id*	*clock*	begin	end

1,1,[0,16]	concurency contrl

Insert r between "concur" and "ency contrl"

1,1,[0,5]	concur
1,1,5,2,1,[0,0]	r
1,1, [6,16]	ency contrl

Insert o between "ency contr" and "l"

1,1,[0,5]	concur
1,1,5,2,1,[0,0]	r
1,1, [6,15]	ency contr
1,1,15,3,1,[0,0]	o
1,1,[16,16]	l

Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]

Site 3

ABCDEF
2,1,**[0,5]**

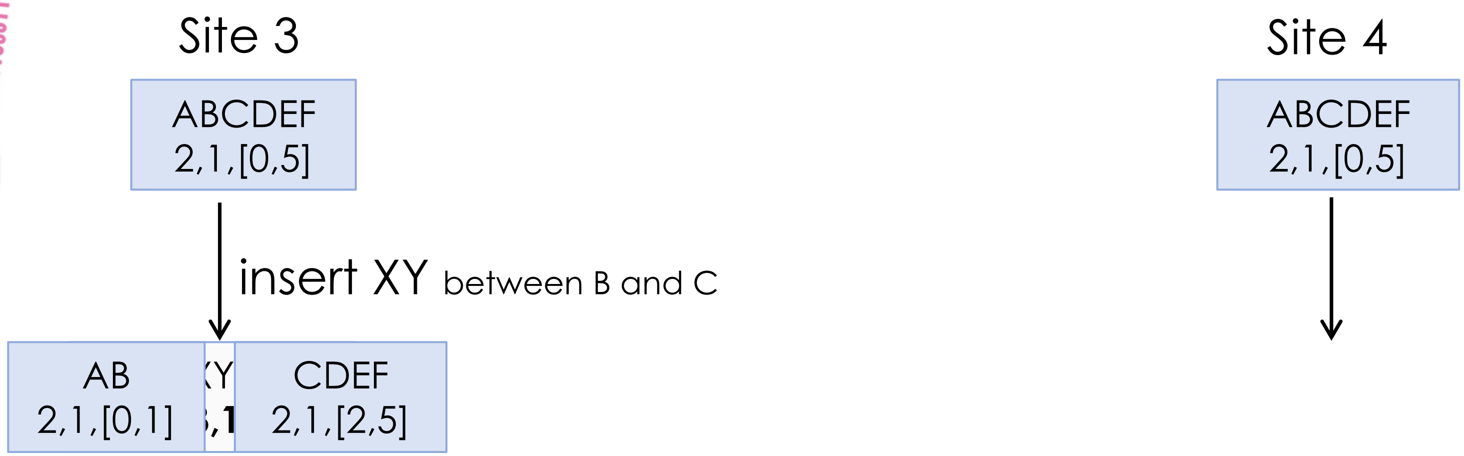
insert XY between B and C

Site 4

ABCDEF
2,1,**[0,5]**

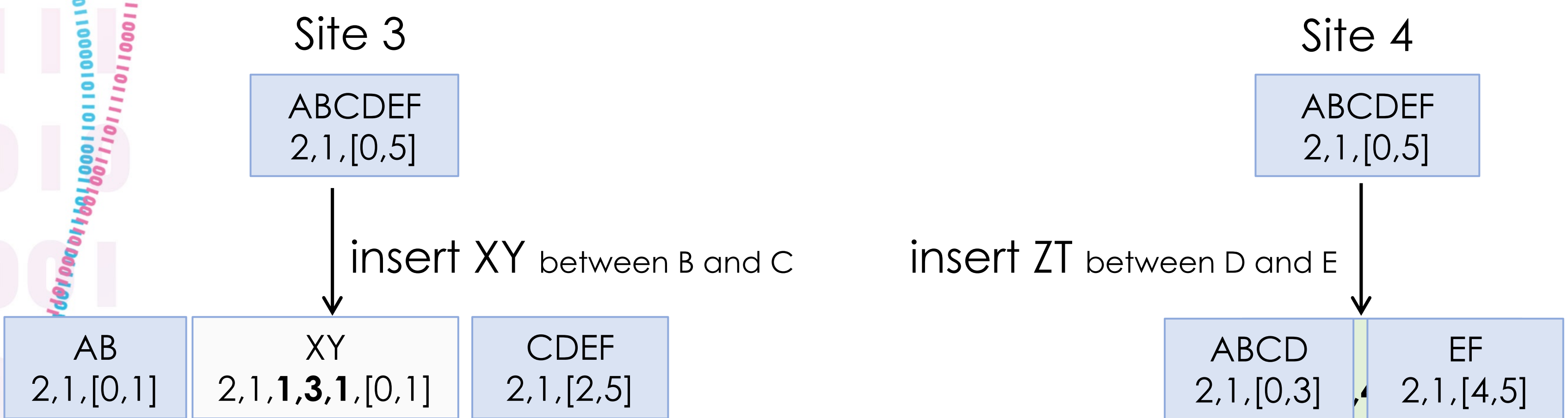
Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]



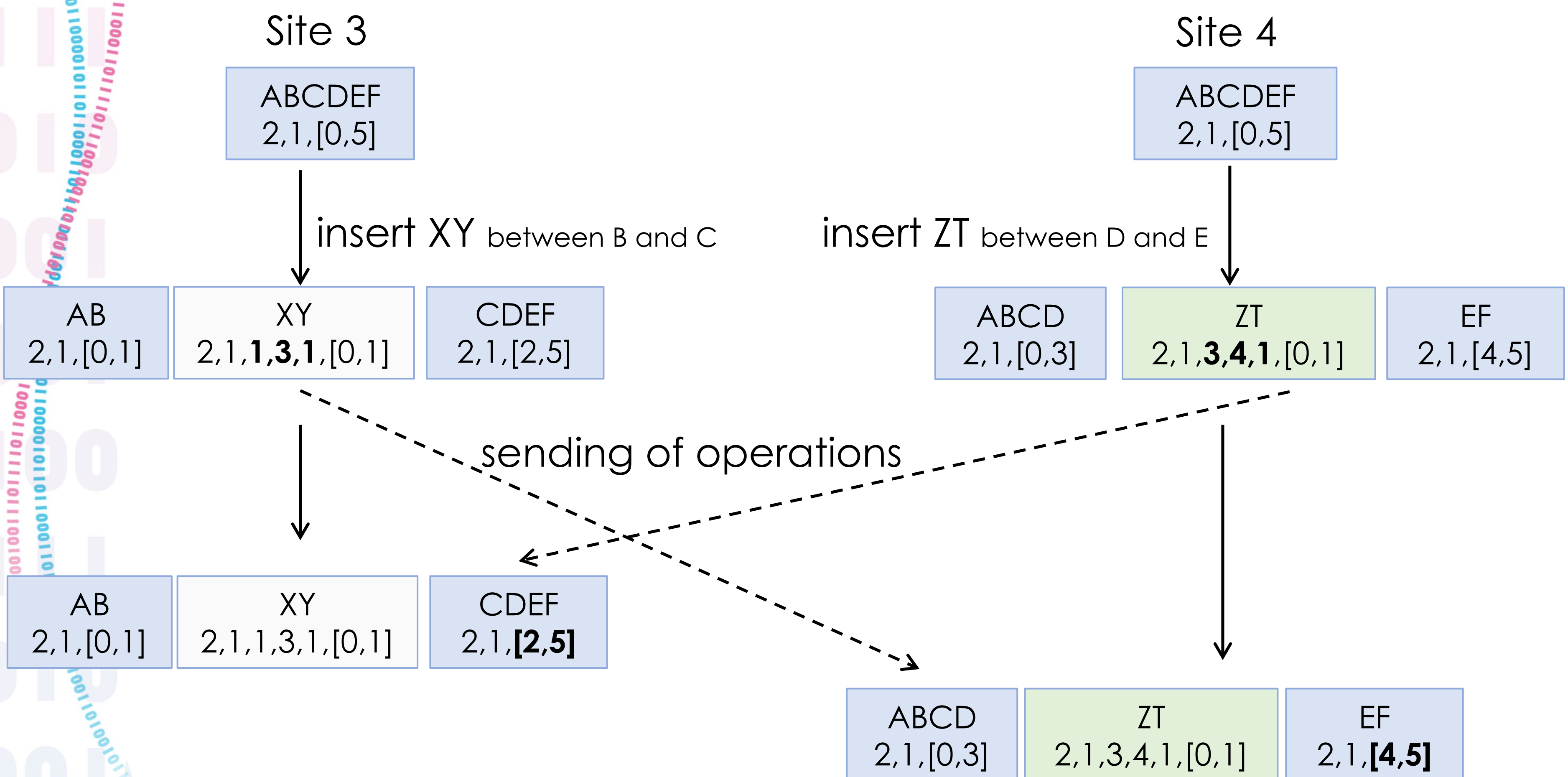
Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]



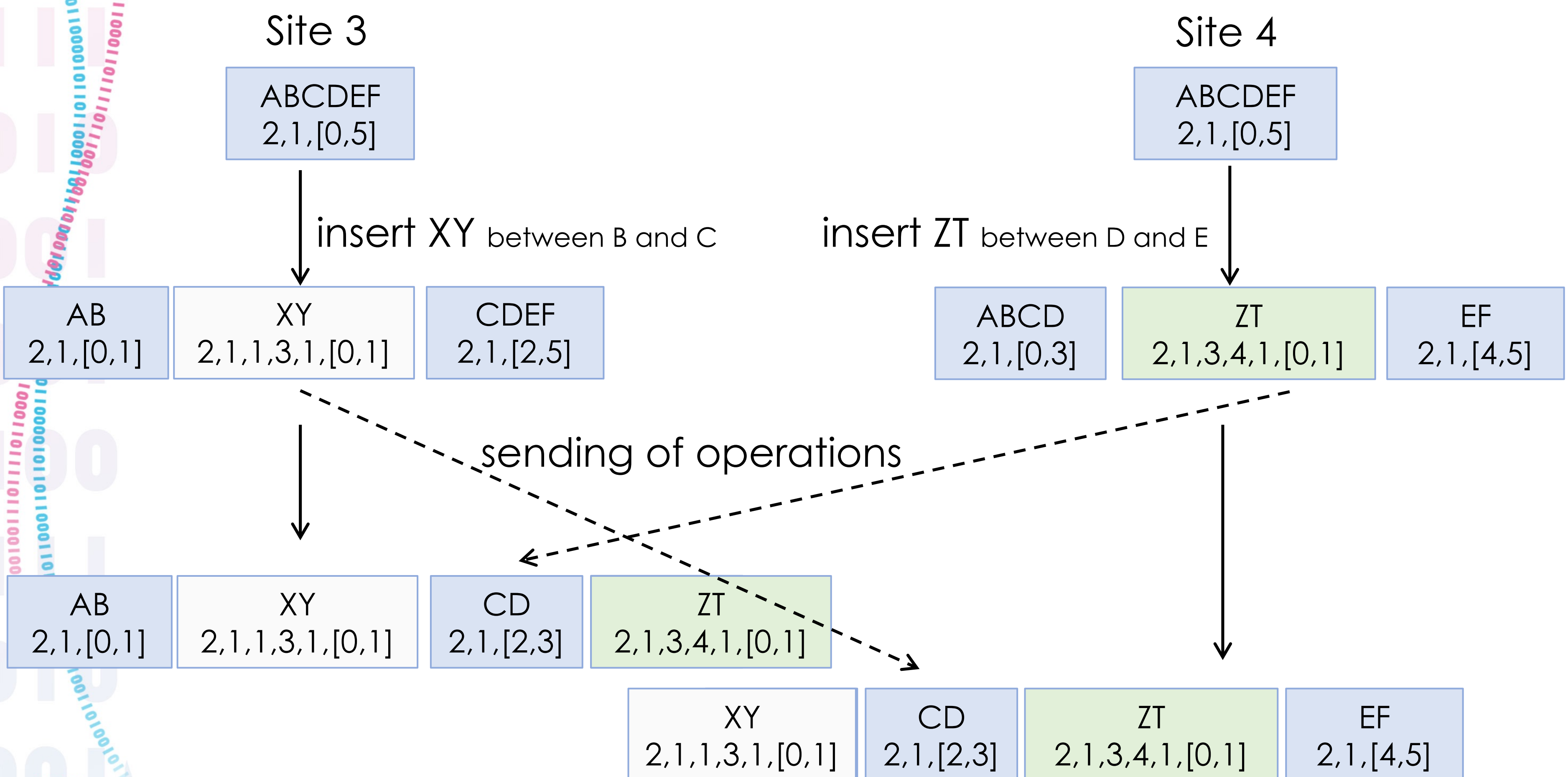
Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]



Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]



Conflict-free Replicated Data Types (CRDT)
Take away

- Strong eventual consistency
  - Parallel and unsynchronized updates
  - Deterministic merge
- State-based approach (CvRDT):
  - Monotonic semi-lattice
  - Reliable broadcast (epidemic propagation)
- Operation-based approach (CmRDT):
  - Commutative concurrent updates
  - Causally-ordered broadcast

References

- **[GL02]** Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, S. Gilbert and N. Lynch, ACM SIGACT news, 2002.
- **[SS05]** Optimistic replication. Y. Saito and M. Shapiro. ACM *Computing Surveys*. 37(1), 2005.
- **[LLS90]** Lazy replication: exploiting the semantics of distributed services, R. Ladin, B. Liskov, L. Shrira, ACM SIGOPS european workshop, 1990.
- **[EG89]** Concurrency Control in GroupWare Systems, C.A. Ellis and S.J. Gibbs. ACM SIGMOD Record 18(2), 1989.
- **[RNG96]** An integrating, transformation-oriented approach to concurrency control and undo in group editors, M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. CSCW 1996.

References

- **[SJZYC98]** Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems, C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. ACM Transactions on Computer-Human Interaction. 5(1), 1998.
- **[OUMI06]** Tombstone transformation functions for ensuring consistency in collaborative editing systems, G. Oster, P. Urso, P. Molli, and A. Imine. CollaborateCom 2006.
- **[SCF97]** Serialization of concurrent operations in a distributed collaborative environment, M. Suleiman, M. Cart, and J. Ferrié. GROUP 1997.
- **[VCFS00]** Copies convergence in a distributed real-time collaborative environment. N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. CSCW 2000.
- **[MOSI03]** Using the Transformational Approach to Build a Safe and Generic Data Synchronizer. P. Molli, G. Oster, H. Skaf-Molli and A. Imine. GROUP 2003.
- **[NCDL95]** High-latency, low-bandwidth windowing in the Jupiter collaboration system, D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. UIST 1995.

References

- **[Z01]** AA. Zafer, NetEdit: A Collaborative Editor. Master's Thesis, VirginiaPolytechnic Institute and State University. 2001.
- **[SPBZ09]** CRDTs: Consistency without concurrency control, M. Shapiro, N. Preguica, C. Baquero, M. Zawirski Research Report, RR-6956, INRIA, 2009
- **[PMSL09]** A commutative replicated data type for cooperative editing, N. Preguica, M. Joan, M. Shapiro, and M. Letia. ICDCS 2009.
- **[WUM09]** Logoot : a Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks, S. Weiss, P. Urso and P. Molli. ICDCS 2009.
- **[AMOI13]** Supporting Adaptable Granularity of Changes for Massive Scale Collaborative Editing, L. André, S. Martin and G. Oster C.-L. Ignat. CollaborateCom 2013.

Materials

Some of the presented materials are coming from:

- “Conflict-free Replicated Data Types (CRDTs) for collaborative environments”, Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. Keynote WETICE 2011
  http://events.telecom-sudparis.eu/wetice/invited_speakers/ShapiroKeyNote.pdf
- “Data Replication and Consistency”, Claudia-Lavinia Ignat. Lectures at Université de Lorraine
  https://members.loria.fr/CIgnat/replication/

Thank you

COAST Team

http://team.inria.fr/coast/

informatics mathematics

UNIVERSITÉ DE LORRAINE